University of Pisa

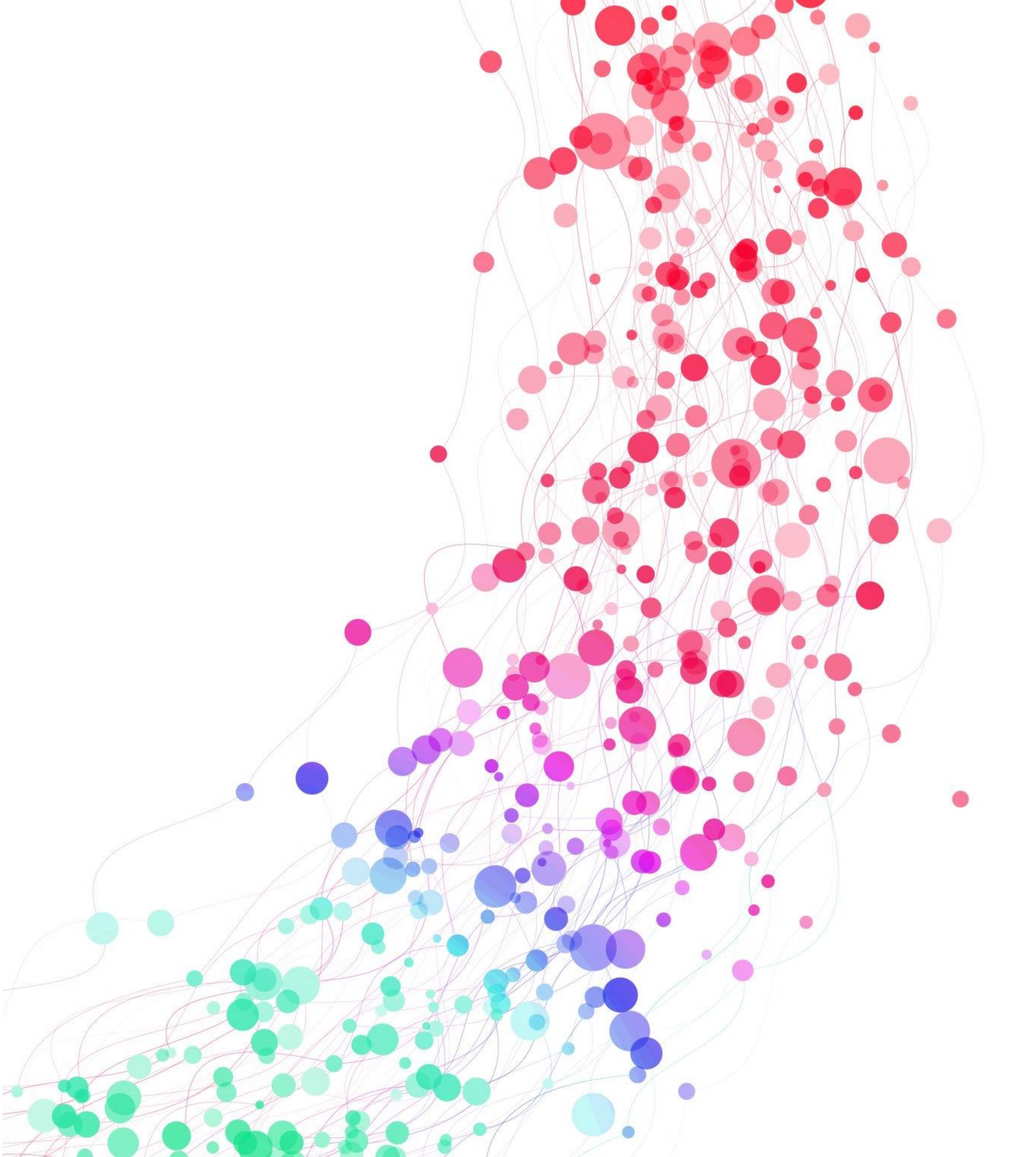
Computer Architecture
2023-24

KMeans parallel implementation

Students

Carlo Pio Pace

Davide Vigna



The Algorithm

Input:

- Dataset of points
- Number of clusters K
- Max # of iterations

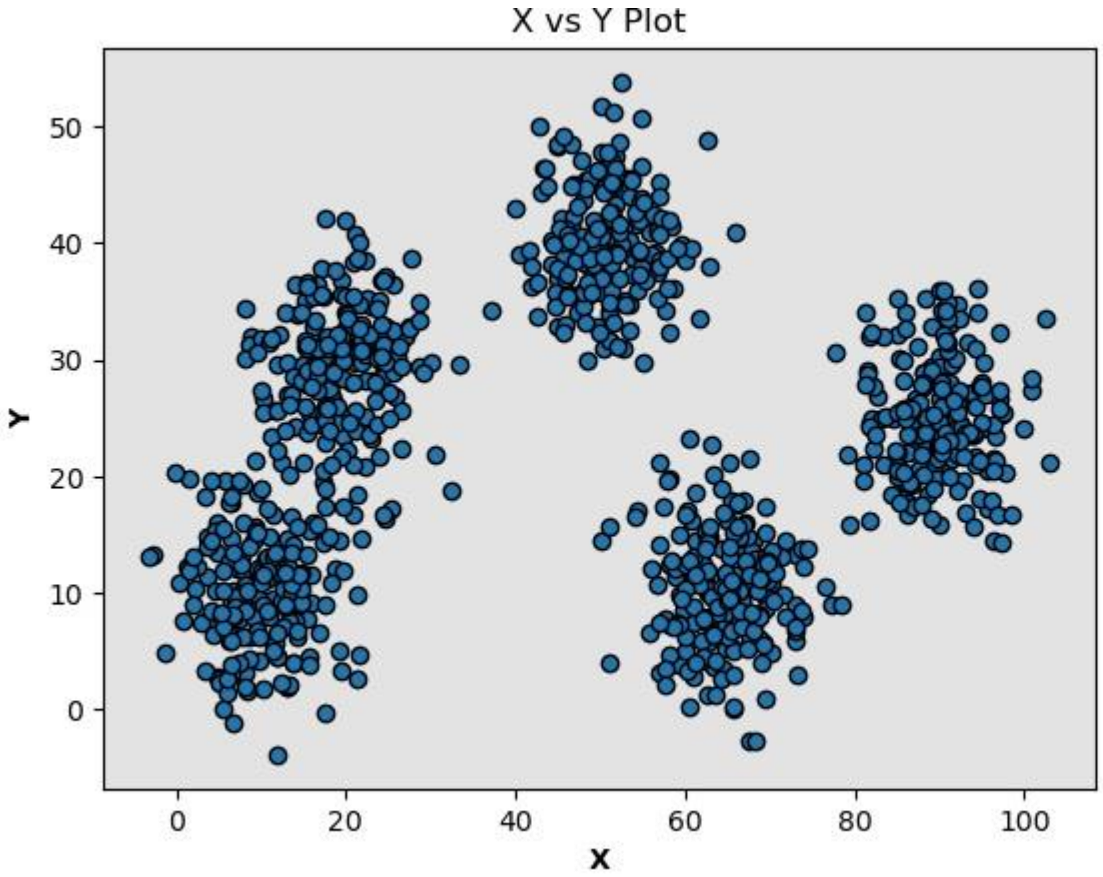
Output:

- A set of K points called Centroids
- The membership of each point in the dataset to one of the K clusters

Parameters:

- K=5
- Max # of iterations = 10

1) Start Execution



The Algorithm

Input:

- Dataset of points
- Number of clusters K
- Max # of iterations

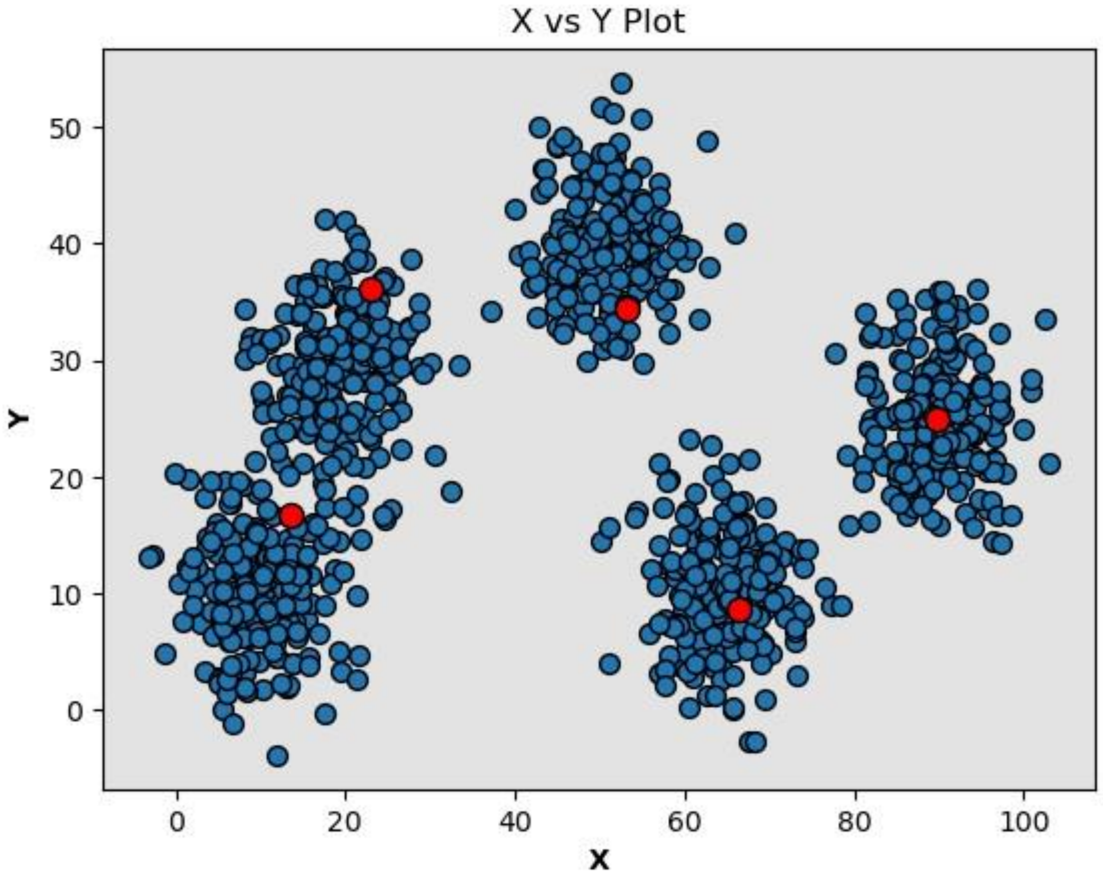
Output:

- A set of K points called Centroids
- The membership of each point in the dataset to one of the K clusters

Parameters:

- K=5
- Max # of iterations = 10

2) Initialization of K random Centroids



The Algorithm

Input:

- Dataset of points
- Number of clusters K
- Max # of iterations

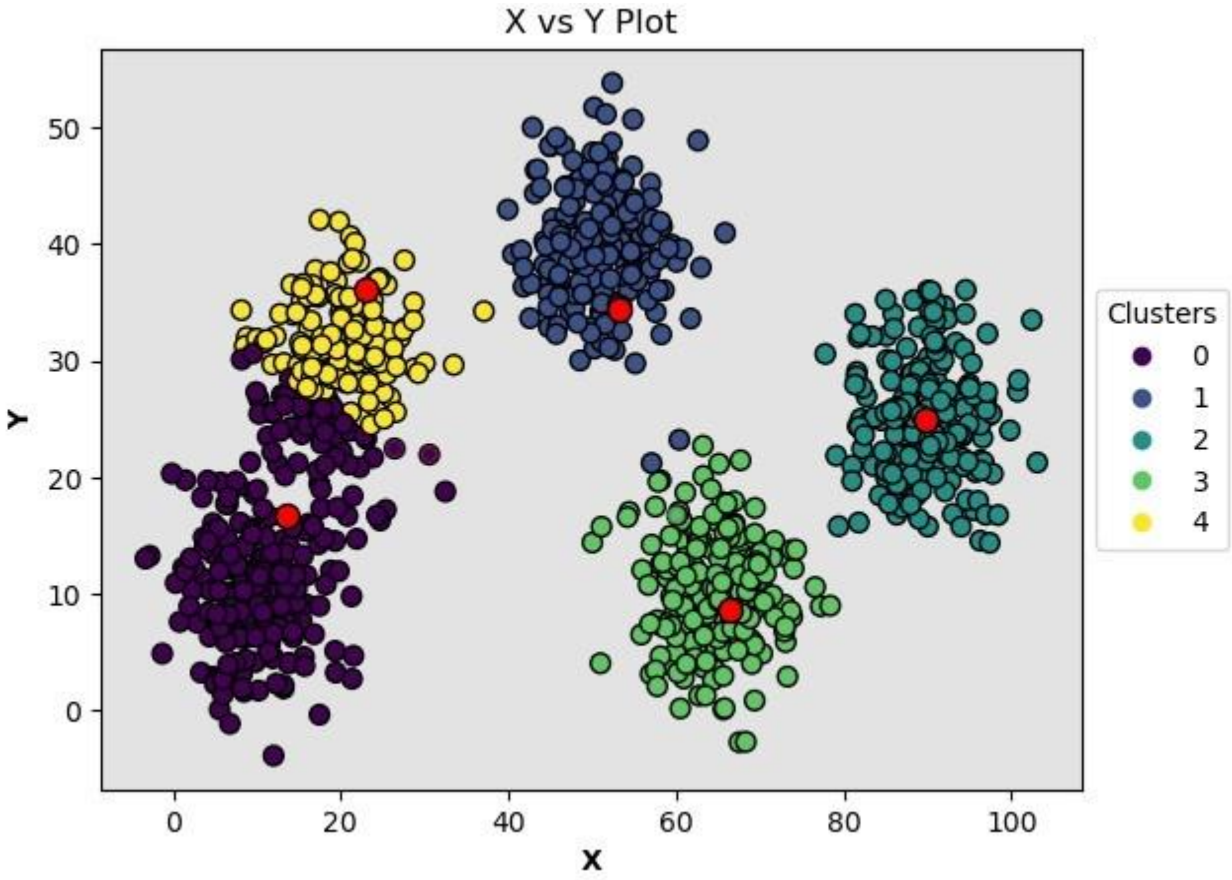
Output:

- A set of K points called Centroids
- The membership of each point in the dataset to one of the K clusters

Parameters:

- K=5
- Max # of iterations = 10

3) Assign points to clusters



The Algorithm

Input:

- Dataset of points
- Number of clusters K
- Max # of iterations

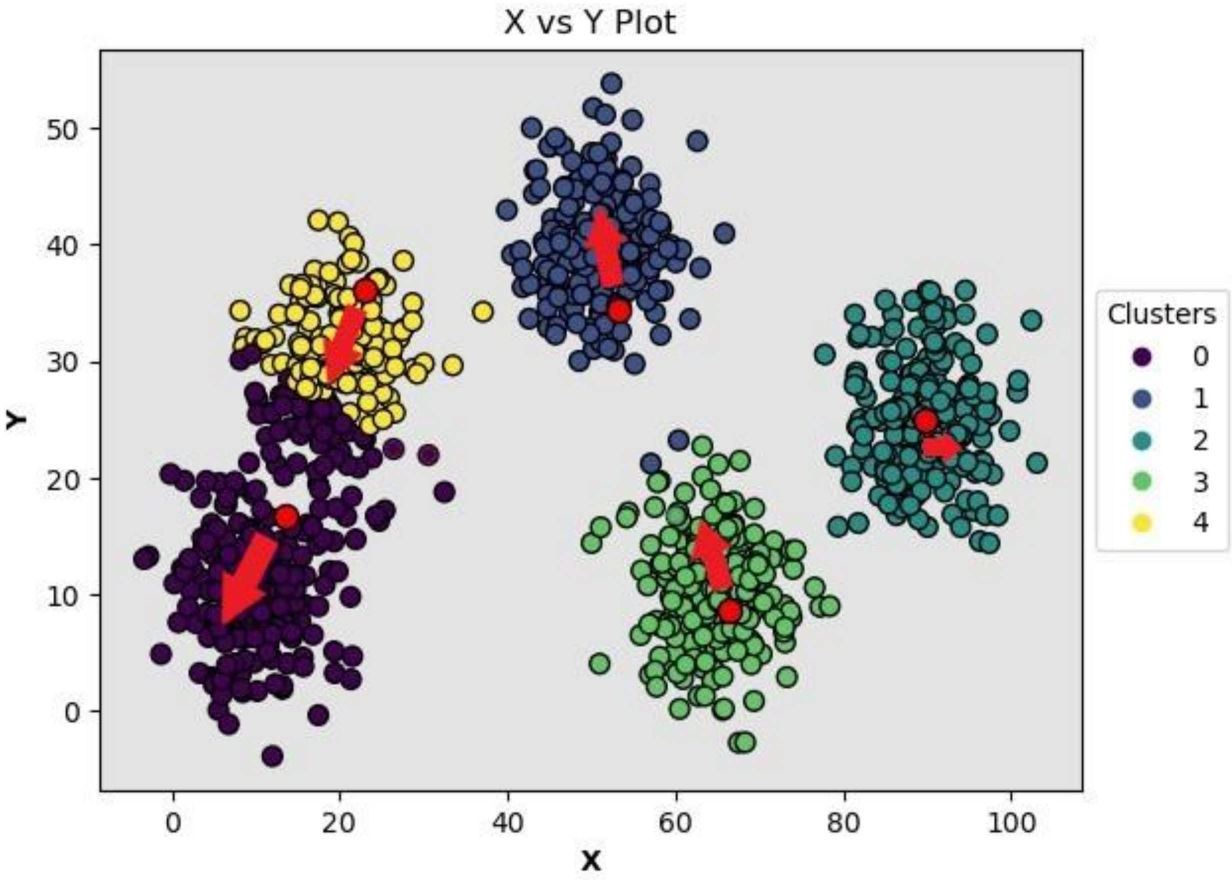
Output:

- A set of K points called Centroids
- The membership of each point in the dataset to one of the K clusters

Parameters:

- K=5
- Max # of iterations = 10

4) Update Centroids



The Algorithm

Input:

- Dataset of points
- Number of clusters K
- Max # of iterations

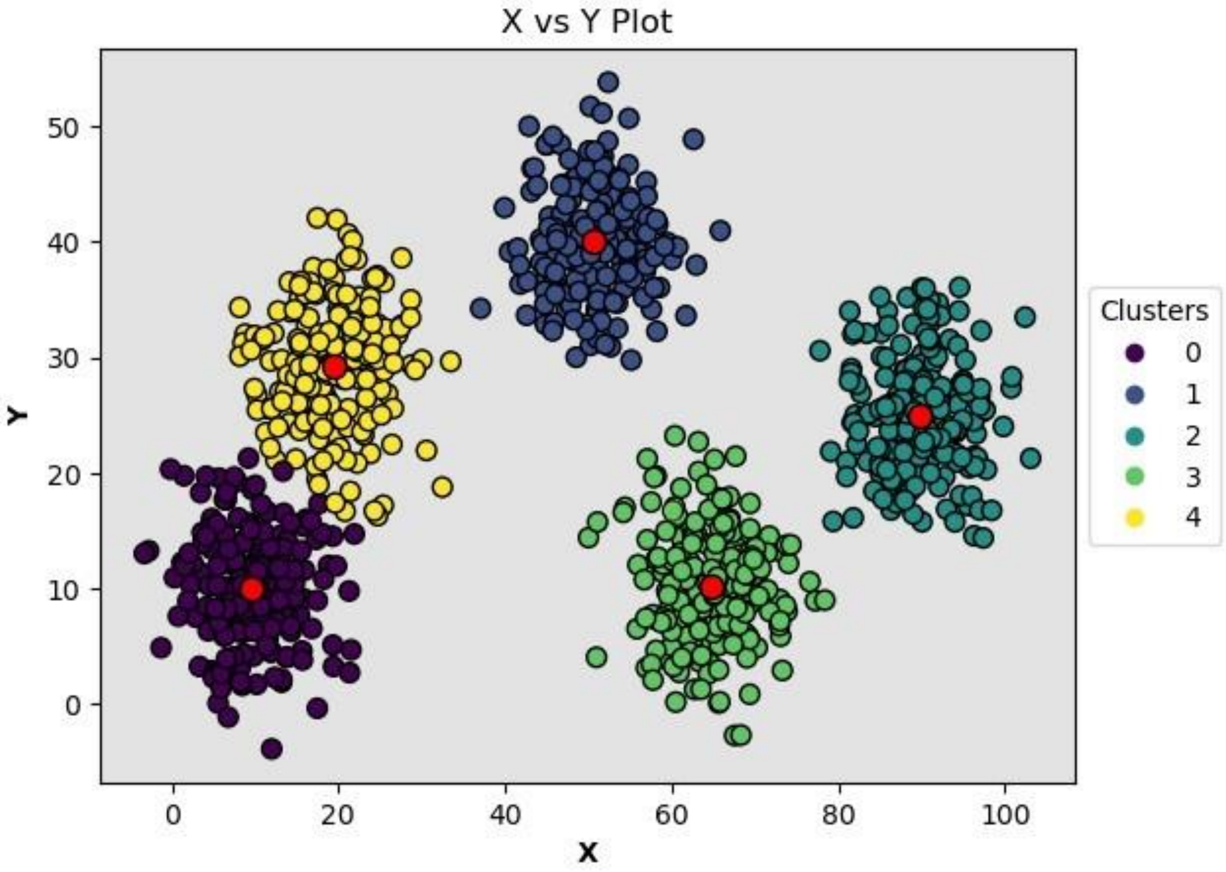
Output:

- A set of K points called Centroids
- The membership of each point in the dataset to one of the K clusters

Parameters:

- K=5
- Max # of iterations = 10

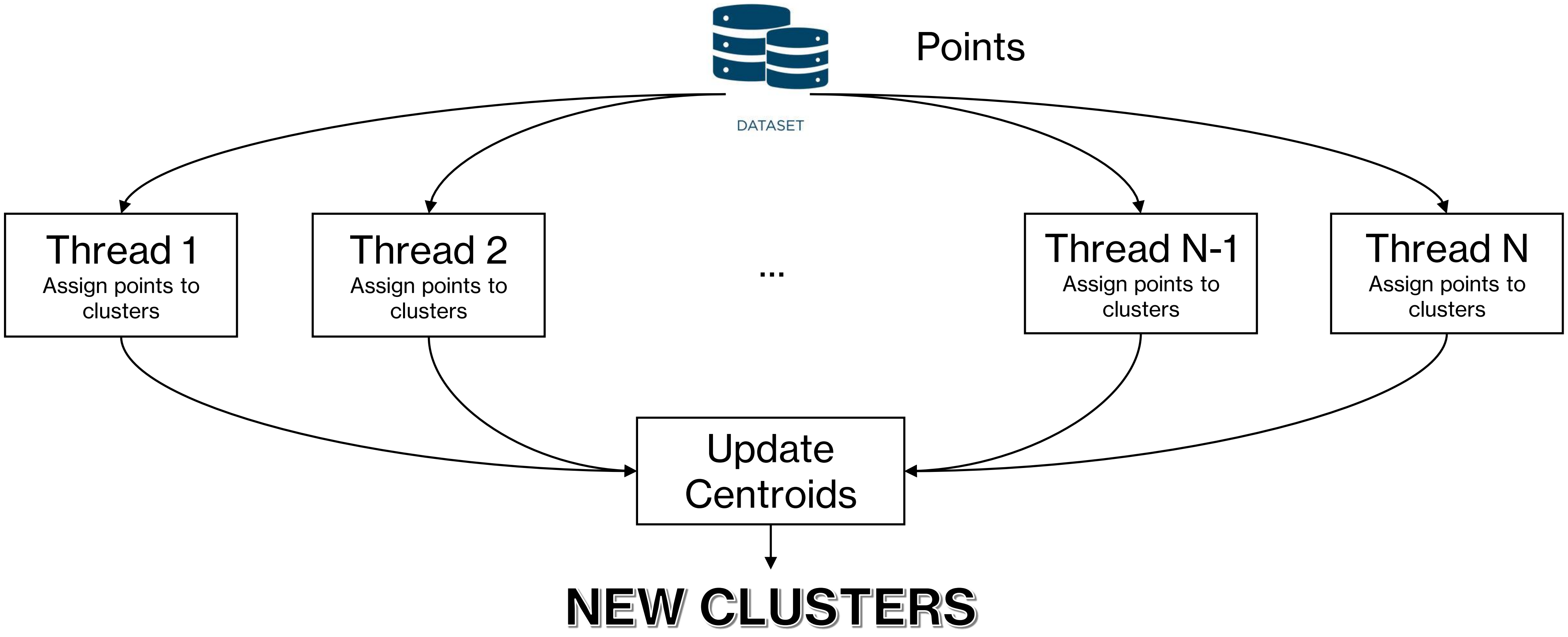
5) Repeat from 3 until Max # of iterations



The Goal

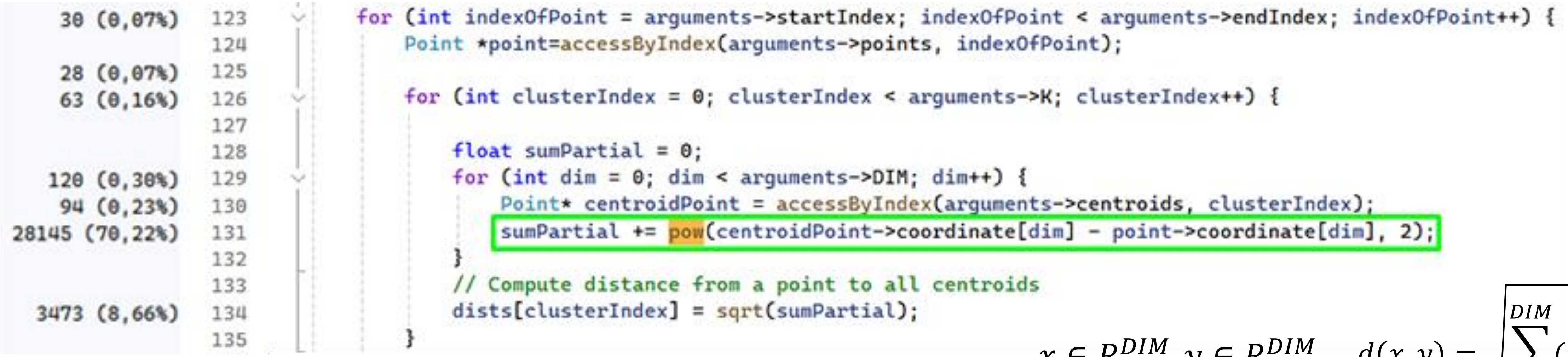
To obtain a solution that has an execution time of maximum 5 seconds on 10M dataset points.

How to make it work in parallel


Points
DATASET
Thread 1
Assign points to clusters
Thread 2
Assign points to clusters
...
Thread N-1
Assign points to clusters
Thread N
Assign points to clusters
Update Centroids
NEW CLUSTERS

How to make it work in parallel

Visual Studio



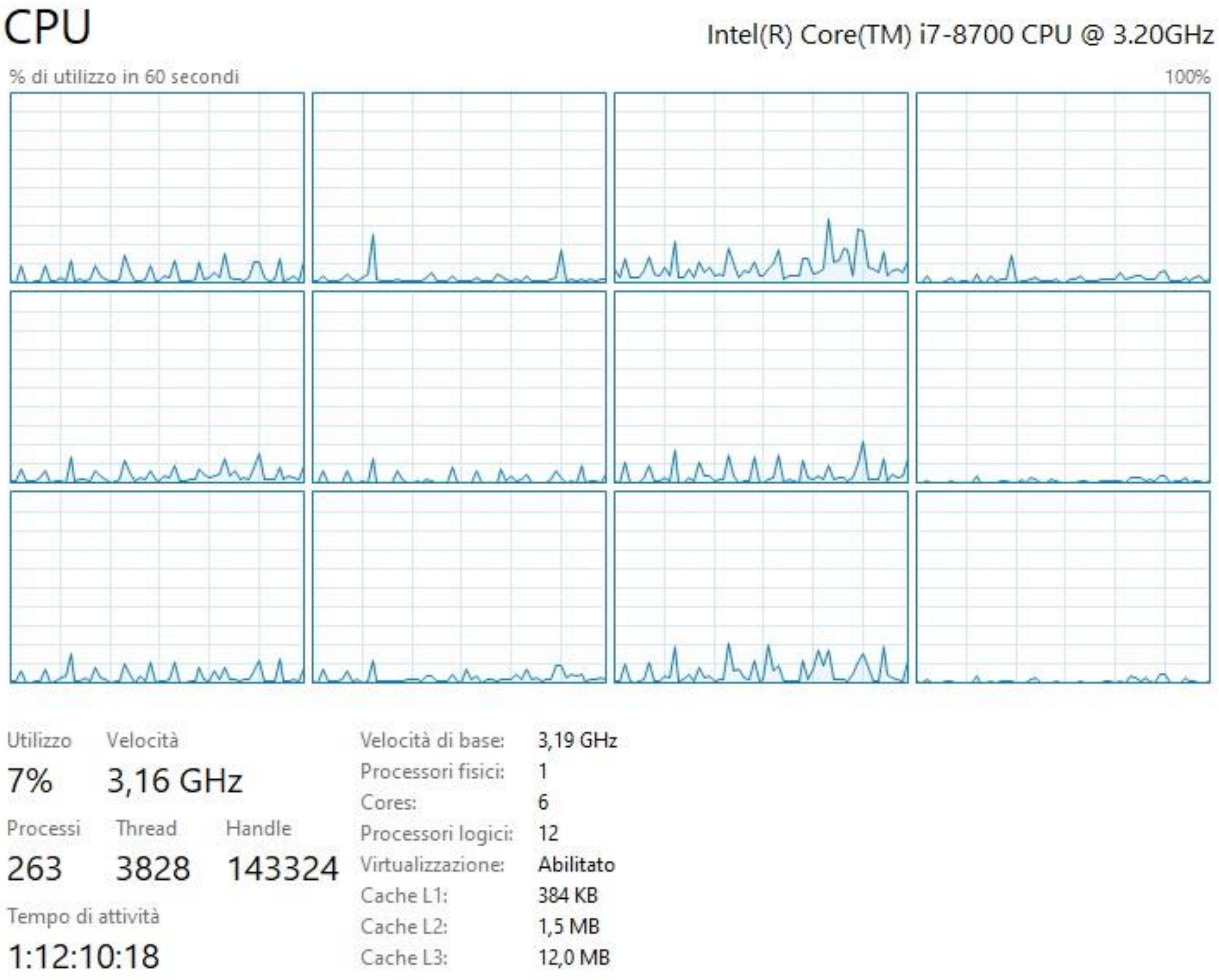
```
30 (0,07%)      for (int indexOfPoint = arguments->startIndex; indexOfPoint < arguments->endIndex; indexOfPoint++) {
                    Point *point=accessByIndex(arguments->points, indexOfPoint);

28 (0,07%)
63 (0,16%)          for (int clusterIndex = 0; clusterIndex < arguments->K; clusterIndex++) {


                        float sumPartial = 0;
120 (0,30%)             for (int dim = 0; dim < arguments->DIM; dim++) {
94 (0,23%)                  Point* centroidPoint = accessByIndex(arguments->centroids, clusterIndex);
28145 (70,22%)              sumPartial += pow(centroidPoint->coordinate[dim] - point->coordinate[dim], 2);
                        }
                        // Compute distance from a point to all centroids
3473 (8,66%)            dists[clusterIndex] = sqrt(sumPartial);
                    }
```

$$x \in R^{DIM}, y \in R^{DIM} \qquad d(x,y) = \sqrt{\sum_{i=1}^{DIM} (x_i - y_i)^2}$$

Idea: exploit thread parallelism to work independently on different chunks of equal size of the dataset and then merge the results to update the centroids as in the sequential version.

Architecture used

- CPU: Intel® Core™ i7-8700 Processor
  - 6 Cores, 12 Threads
  - Clock: Up to 3.20 Ghz
  - Max Turbo Frequency: 4.60 GHz
  - Cache:
    - ➢ Level 1: 384 KB
    - ➢ Level 2: 1.5 MB
    - ➢ Level 3: 12 MB
- RAM: 16 GB DDR4 2133 MHz



CPU

Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz

% di utilizzo in 60 secondi

100%

Utilizzo 7%

Velocità 3,16 GHz

Processi 263

Thread 3828

Handle 143324

Tempo di attività 1:12:10:18

Velocità di base: 3,19 GHz

Processori fisici: 1

Cores: 6

Processori logici: 12

Virtualizzazione: Abilitato

Cache L1: 384 KB

Cache L2: 1,5 MB

Cache L3: 12,0 MB

Architecture used

- CPU: Intel® Core™ i7-8700 Processor
  - 6 Cores, 12 Threads
  - Clock: Up to 3.20 Ghz
  - Max Turbo Frequency: 4.60 GHz
  - Cache:
    - ➢ Level 1: 384 KB
    - ➢ Level 2: 1.5 MB
    - ➢ Level 3: 12 MB
- RAM: 16 GB DDR4 2133 MHz

Cache details

- Level 1 :
  - Data : 32 Kbytes 8-way set associative;
  - Instructions : 32 Kbytes 8-way set associative;
- Level 2 : 256 Kbytes 4-way set associative;
- Level 3: 12 Mbytes 16-way set associative;

Measuring method

Function clock() in time.h C library

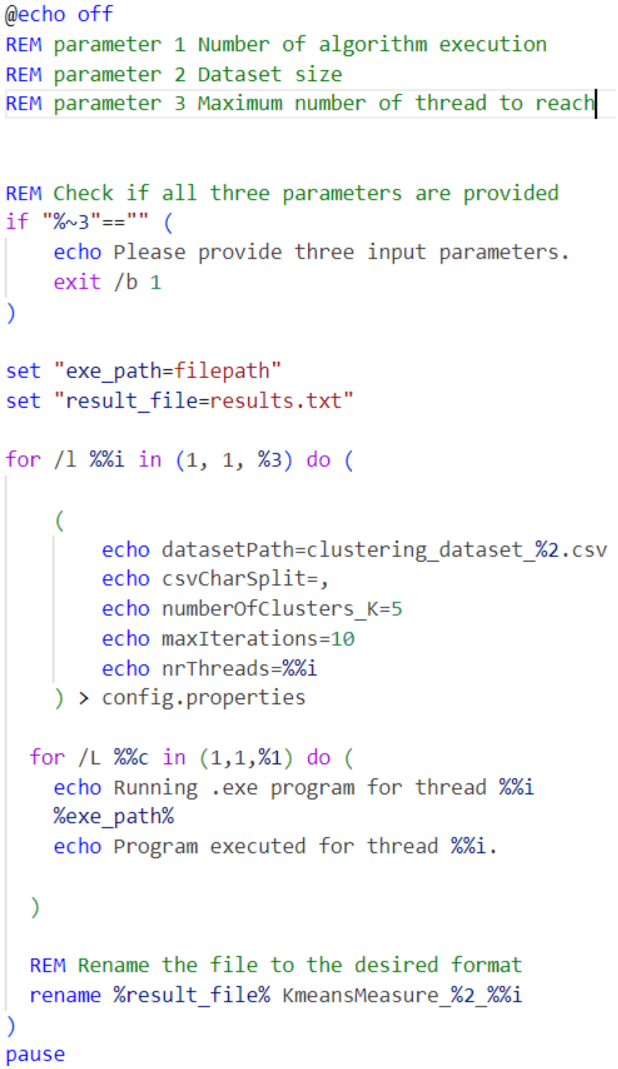
Why:

- It calculates CPU time and not Wall-clock time (with small errors)
- On modern machines, precision of ms
- C is a standard
- C is the language in which we developed the algorithm

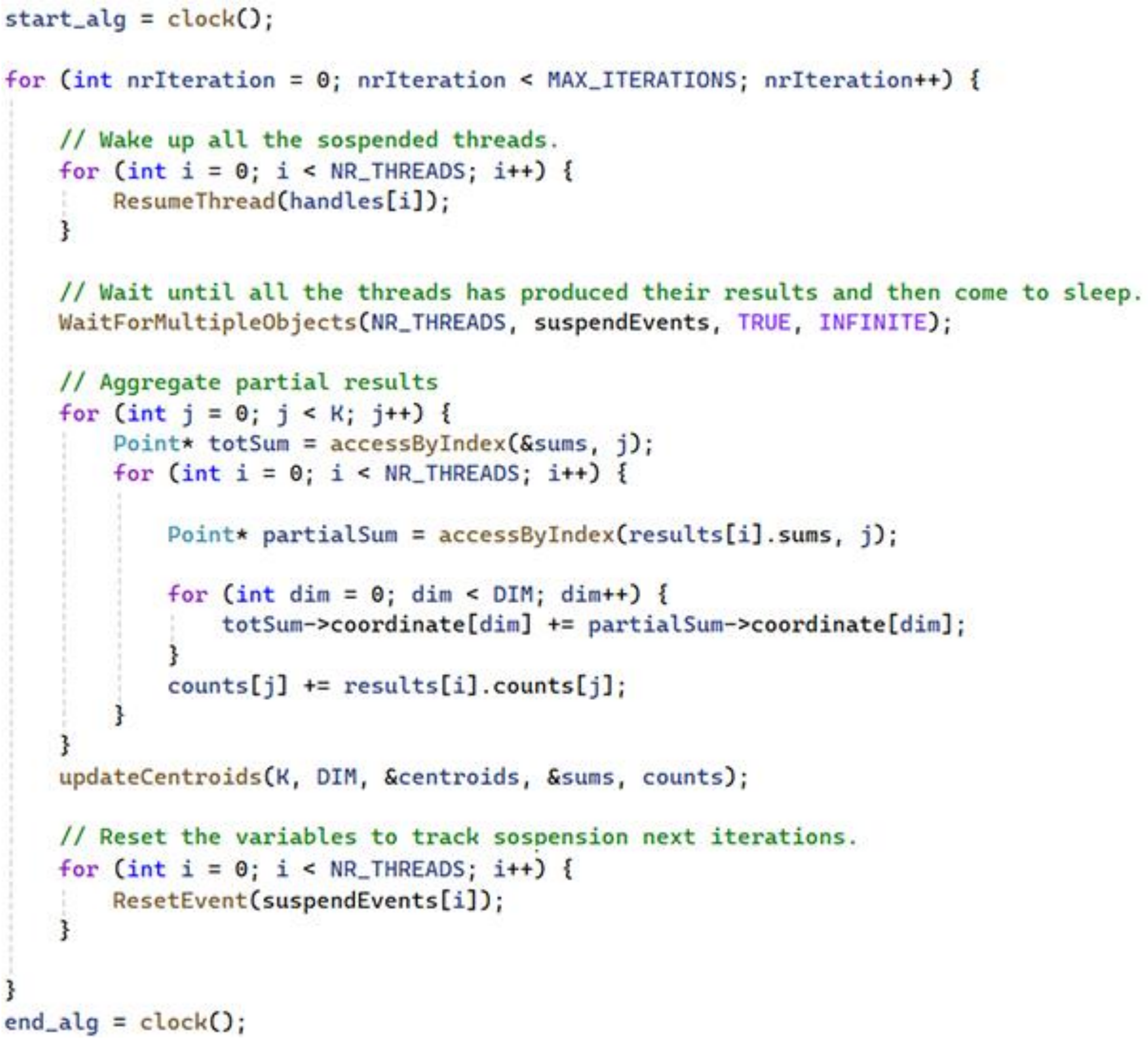
How it works:

The function clock() defined in this library returns the corresponding cpu time at the instant when it’s called.

Measuring method

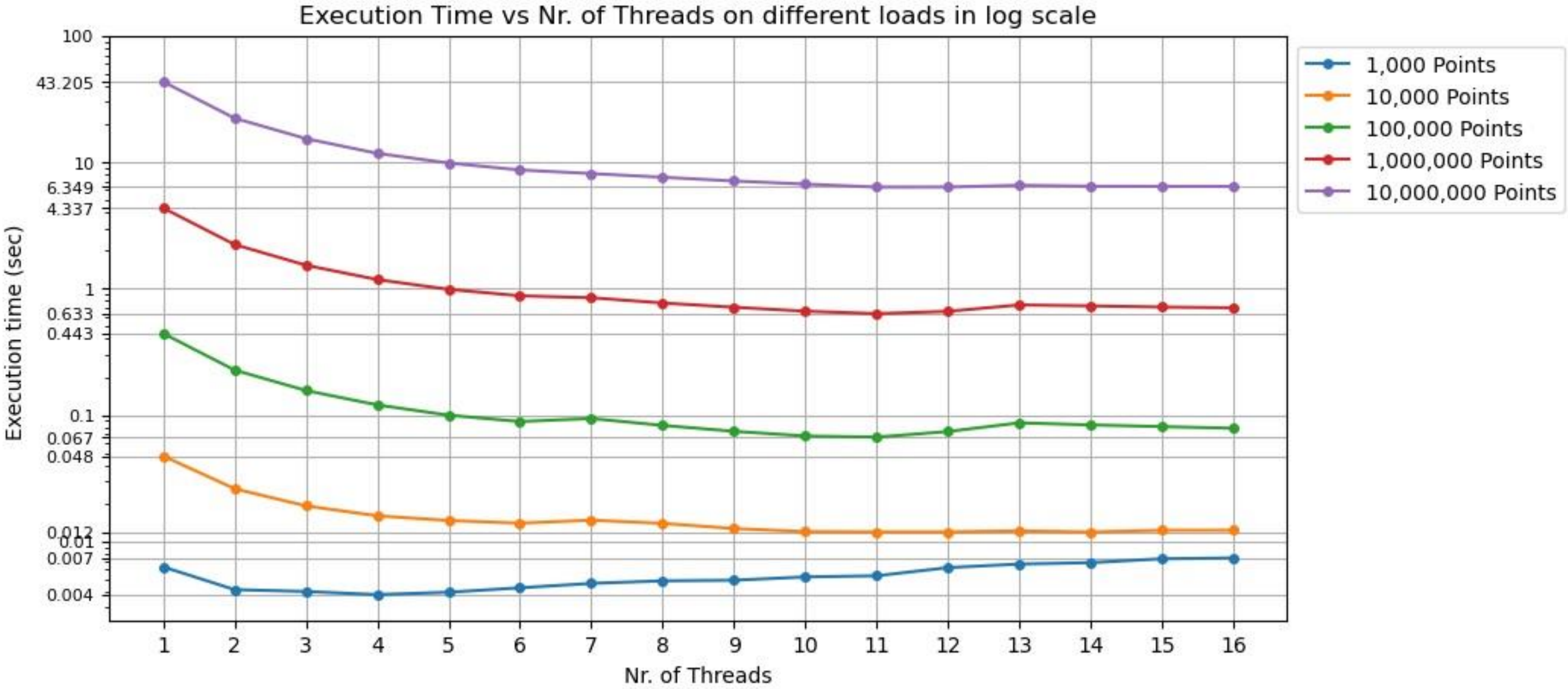


```
@echo off
REM parameter 1 Number of algorithm execution
REM parameter 2 Dataset size
REM parameter 3 Maximum number of thread to reach


REM Check if all three parameters are provided
if "%~3"=="" (
    echo Please provide three input parameters.
    exit /b 1
)

set "exe_path=filepath"
set "result_file=results.txt"

for /l %%i in (1, 1, %3) do (

    (
        echo datasetPath=clustering_dataset_%2.csv
        echo csvCharSplit=,
        echo numberOfClusters_K=5
        echo maxIterations=10
        echo nrThreads=%%i
    ) > config.properties

  for /L %%c in (1,1,%1) do (
    echo Running .exe program for thread %%i
    %exe_path%
    echo Program executed for thread %%i.

  )

  REM Rename the file to the desired format
  rename %result_file% KmeansMeasure_%2_%%i
)
pause
```

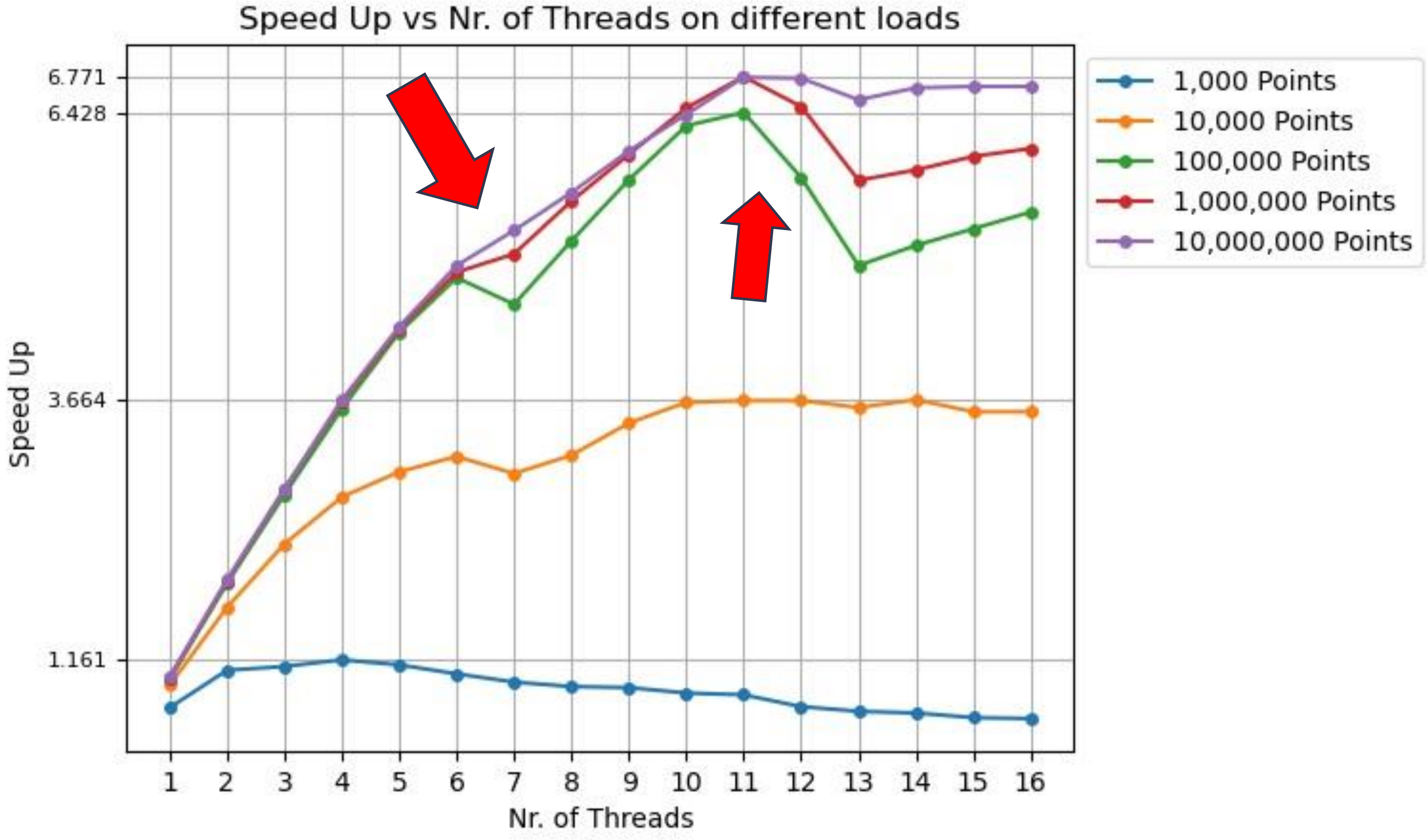


```
start_alg = clock();

for (int nrIteration = 0; nrIteration < MAX_ITERATIONS; nrIteration++) {

    // Wake up all the sospended threads.
    for (int i = 0; i < NR_THREADS; i++) {
        ResumeThread(handles[i]);
    }

    // Wait until all the threads has produced their results and then come to sleep.
    WaitForMultipleObjects(NR_THREADS, suspendEvents, TRUE, INFINITE);

    // Aggregate partial results
    for (int j = 0; j < K; j++) {
        Point* totSum = accessByIndex(&sums, j);
        for (int i = 0; i < NR_THREADS; i++) {

            Point* partialSum = accessByIndex(results[i].sums, j);

            for (int dim = 0; dim < DIM; dim++) {
                totSum->coordinate[dim] += partialSum->coordinate[dim];
            }
            counts[j] += results[i].counts[j];
        }
    }
    updateCentroids(K, DIM, &centroids, &sums, counts);

    // Reset the variables to track sospension next iterations.
    for (int i = 0; i < NR_THREADS; i++) {
        ResetEvent(suspendEvents[i]);
    }

}
end_alg = clock();
```

CPU time comparison



Speed Up


Speed Up vs Nr. of Threads on different loads
Speed Up
6.771
6.428
3.664
1.161
Nr. of Threads
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
1,000 Points
10,000 Points
100,000 Points
1,000,000 Points
10,000,000 Points

Speed Up – What’s Up?

Intel VTune

CORE 0	CPU 0	CPU 1
CORE 1	CPU 2	CPU 3
CORE 2	CPU 4	CPU 5
CORE 3	CPU 6	CPU 7
CORE 4	CPU 8	CPU 9
CORE 5	CPU 10	CPU 11

How the software threads are assigned to our logical core units?

The O.S. schedule them according his own policy

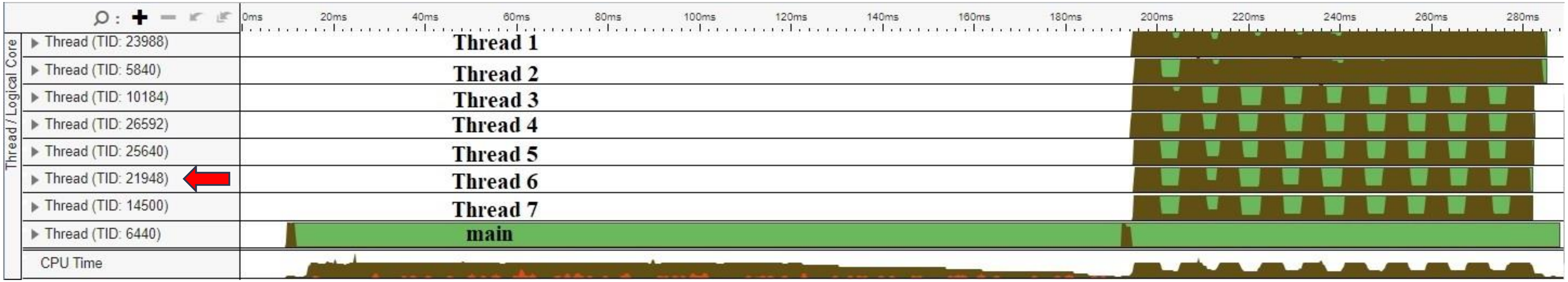
Speed Up – What’s Up?

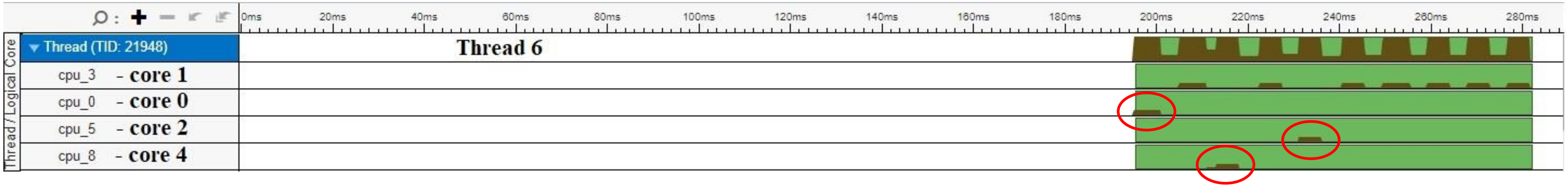
Intel VTune

Intel VTune

- 100 000 Points dataset - 7 Threads - Clusters=5 & Max # of iterations = 10

🟢 Running 🟤 CPU in use



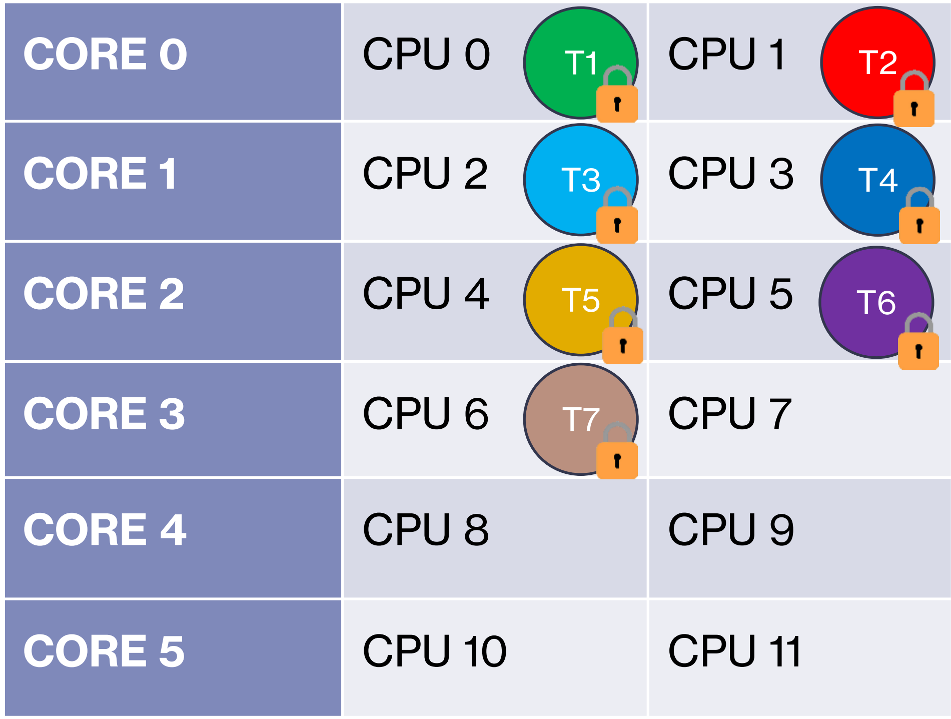


Speed Up – What’s Up?

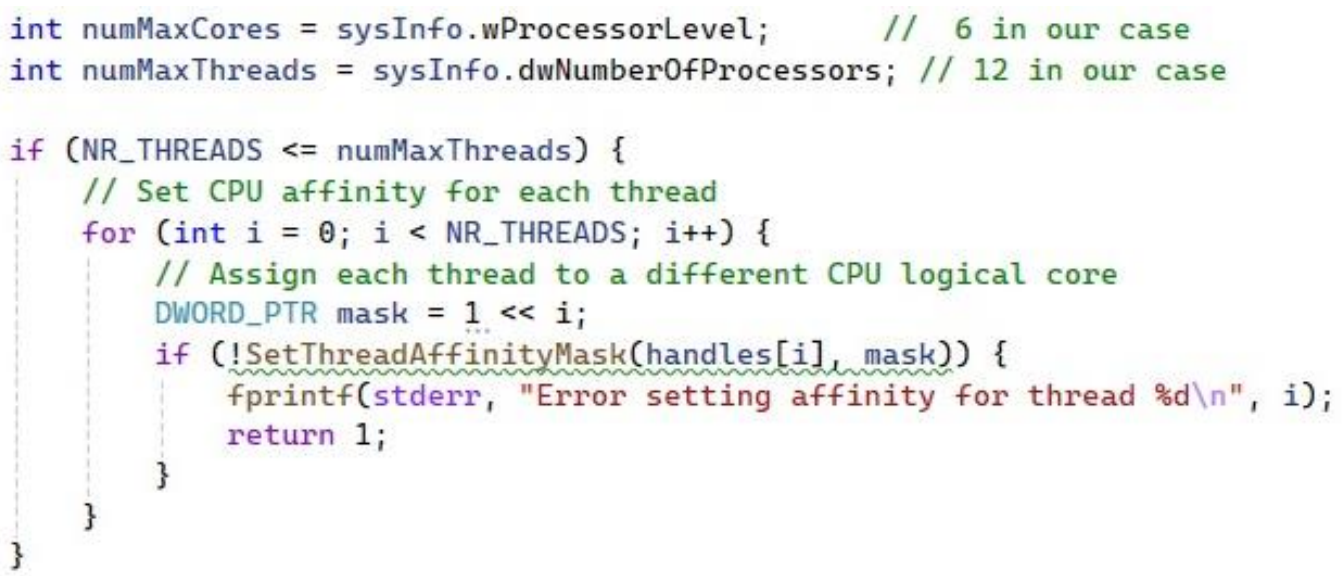
Affinity:

The explicit request to O.S. to assign a process or thread to a specific logical core inside a specific physical core.

Using affinity → Following logical cores order



CORE		
CORE 0	CPU 0 T1	CPU 1 T2
CORE 1	CPU 2 T3	CPU 3 T4
CORE 2	CPU 4 T5	CPU 5 T6
CORE 3	CPU 6 T7	CPU 7
CORE 4	CPU 8	CPU 9
CORE 5	CPU 10	CPU 11



```
int numMaxCores = sysInfo.wProcessorLevel;       //  6 in our case
int numMaxThreads = sysInfo.dwNumberOfProcessors; // 12 in our case

if (NR_THREADS <= numMaxThreads) {
    // Set CPU affinity for each thread
    for (int i = 0; i < NR_THREADS; i++) {
        // Assign each thread to a different CPU logical core
        DWORD_PTR mask = 1 << i;
        if (!SetThreadAffinityMask(handles[i], mask)) {
            fprintf(stderr, "Error setting affinity for thread %d\n", i);
            return 1;
        }
    }
}
```

Following logical cores order

Intel VTune

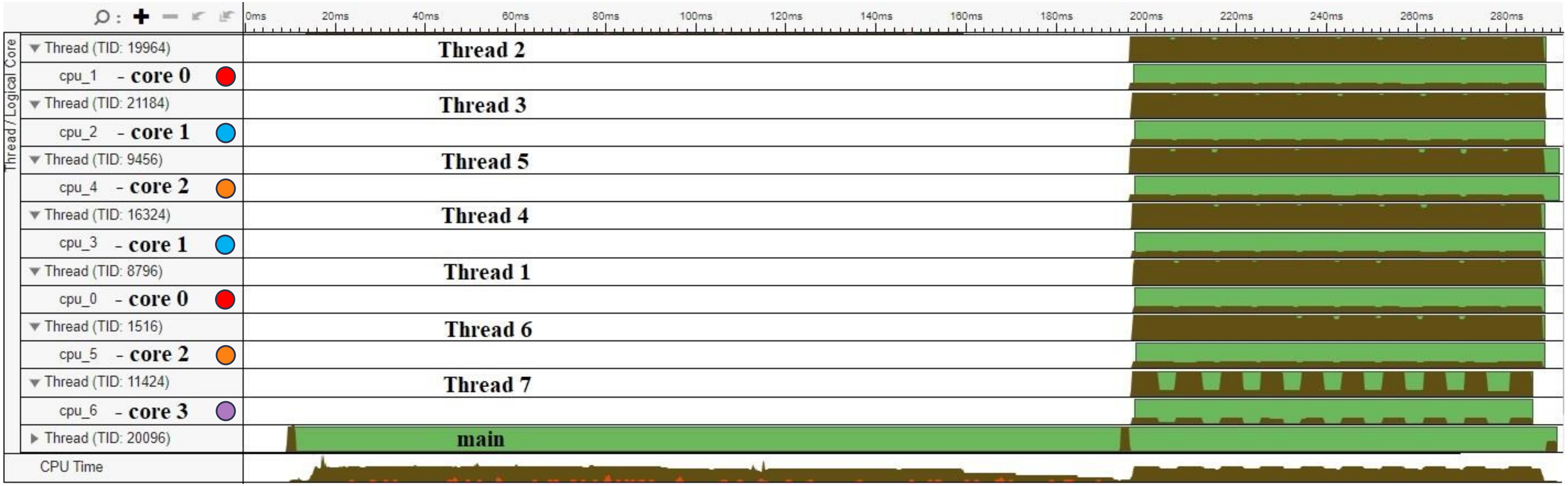
Intel VTune

- 100 000 Points dataset

- 7 Threads

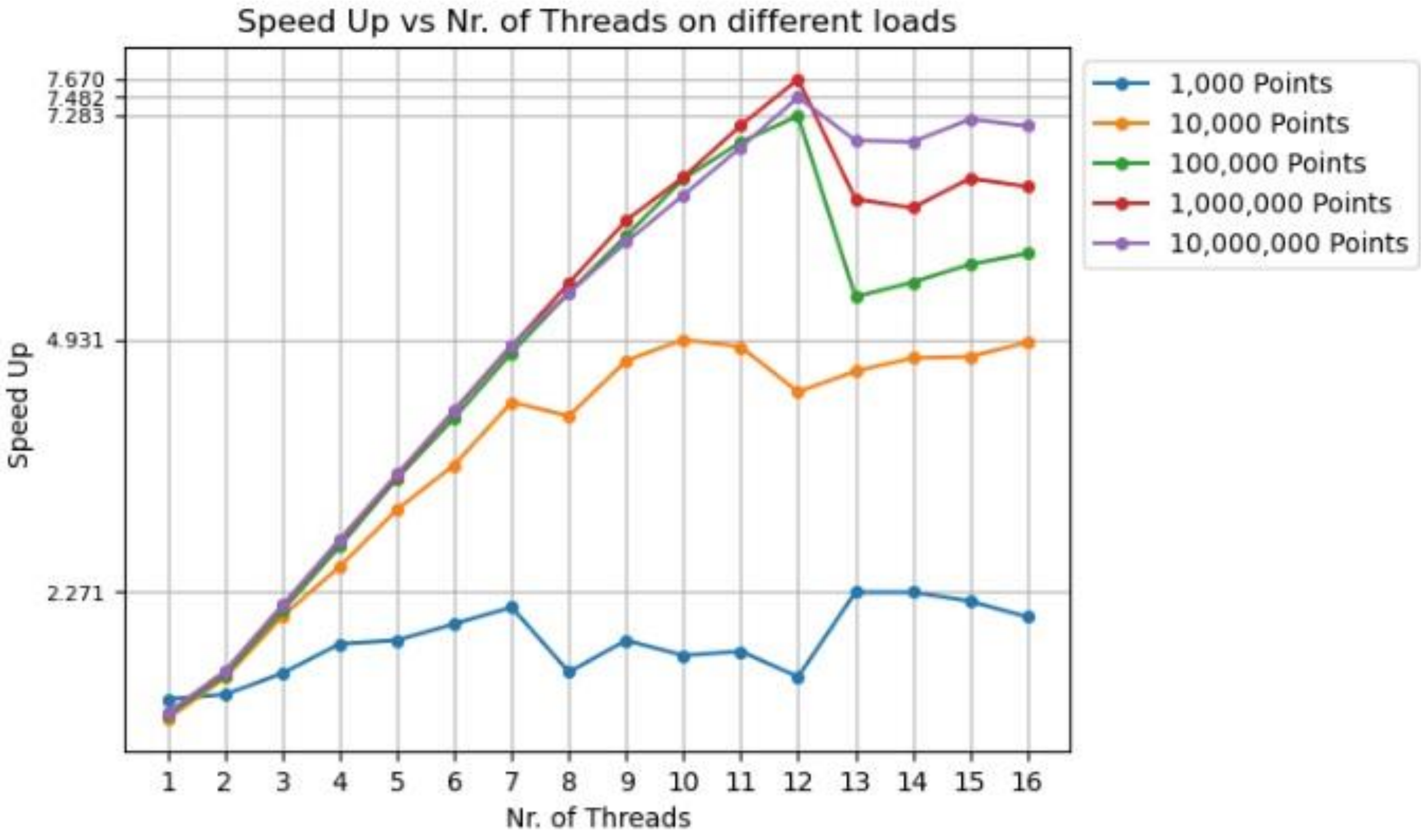
- Clusters=5 & Max # of iterations = 10

🟢 Running 🟤 CPU in use

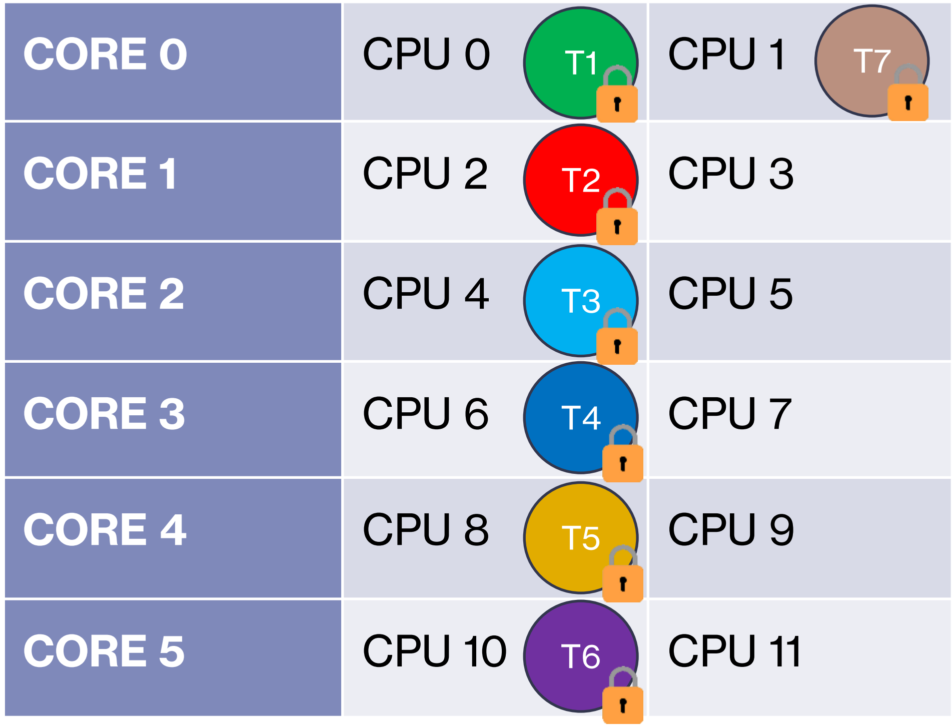


Speed Up

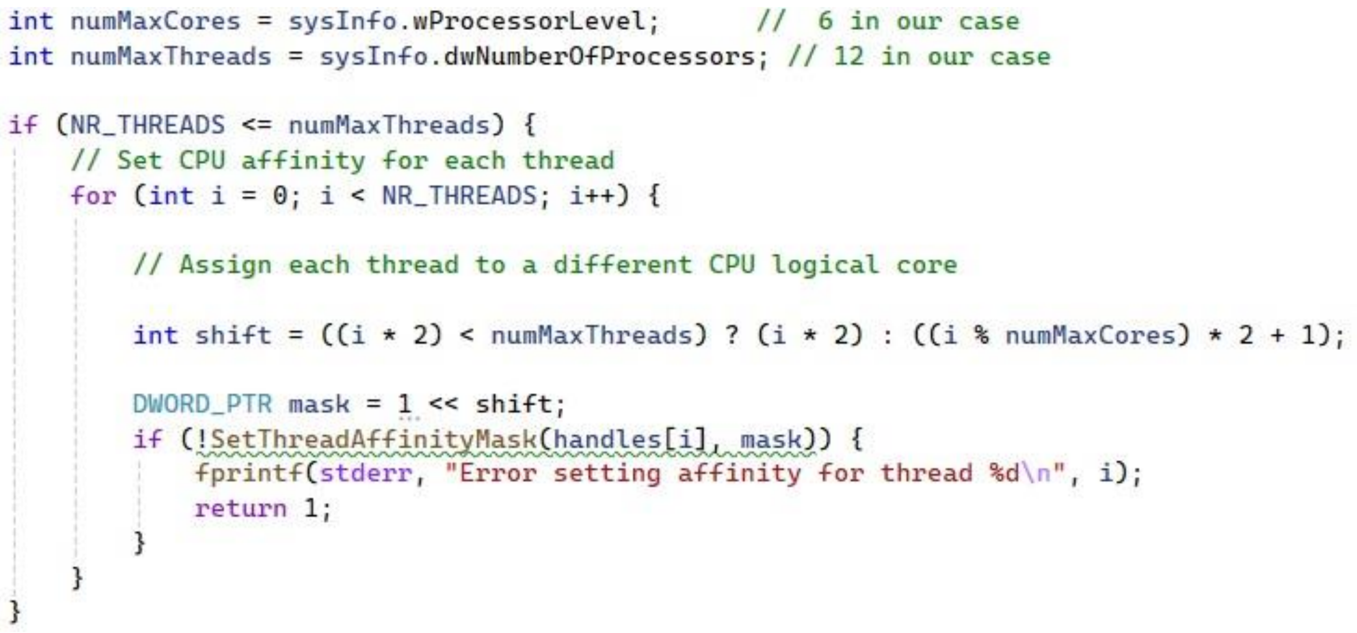
affinity set to follow logical cores order


Speed Up vs Nr. of Threads on different loads

Using affinity → Following physical cores order



CORE 0	CPU 0	T1	CPU 1	T7
CORE 1	CPU 2	T2	CPU 3	
CORE 2	CPU 4	T3	CPU 5	
CORE 3	CPU 6	T4	CPU 7	
CORE 4	CPU 8	T5	CPU 9	
CORE 5	CPU 10	T6	CPU 11	



```
int numMaxCores = sysInfo.wProcessorLevel;       //  6 in our case
int numMaxThreads = sysInfo.dwNumberOfProcessors; // 12 in our case

if (NR_THREADS <= numMaxThreads) {
    // Set CPU affinity for each thread
    for (int i = 0; i < NR_THREADS; i++) {

        // Assign each thread to a different CPU logical core

        int shift = ((i * 2) < numMaxThreads) ? (i * 2) : ((i % numMaxCores) * 2 + 1);

        DWORD_PTR mask = 1 << shift;
        if (!SetThreadAffinityMask(handles[i], mask)) {
            fprintf(stderr, "Error setting affinity for thread %d\n", i);
            return 1;
        }
    }
}
```

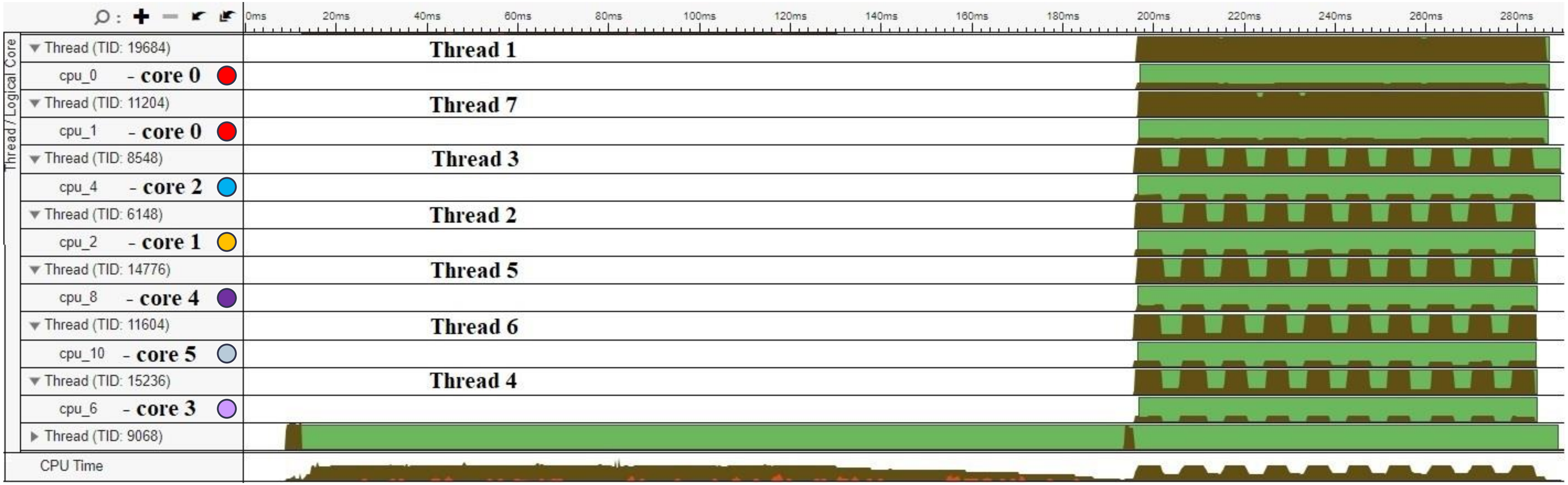
Following physical cores order

- 100 000 Points dataset

- 7 Threads

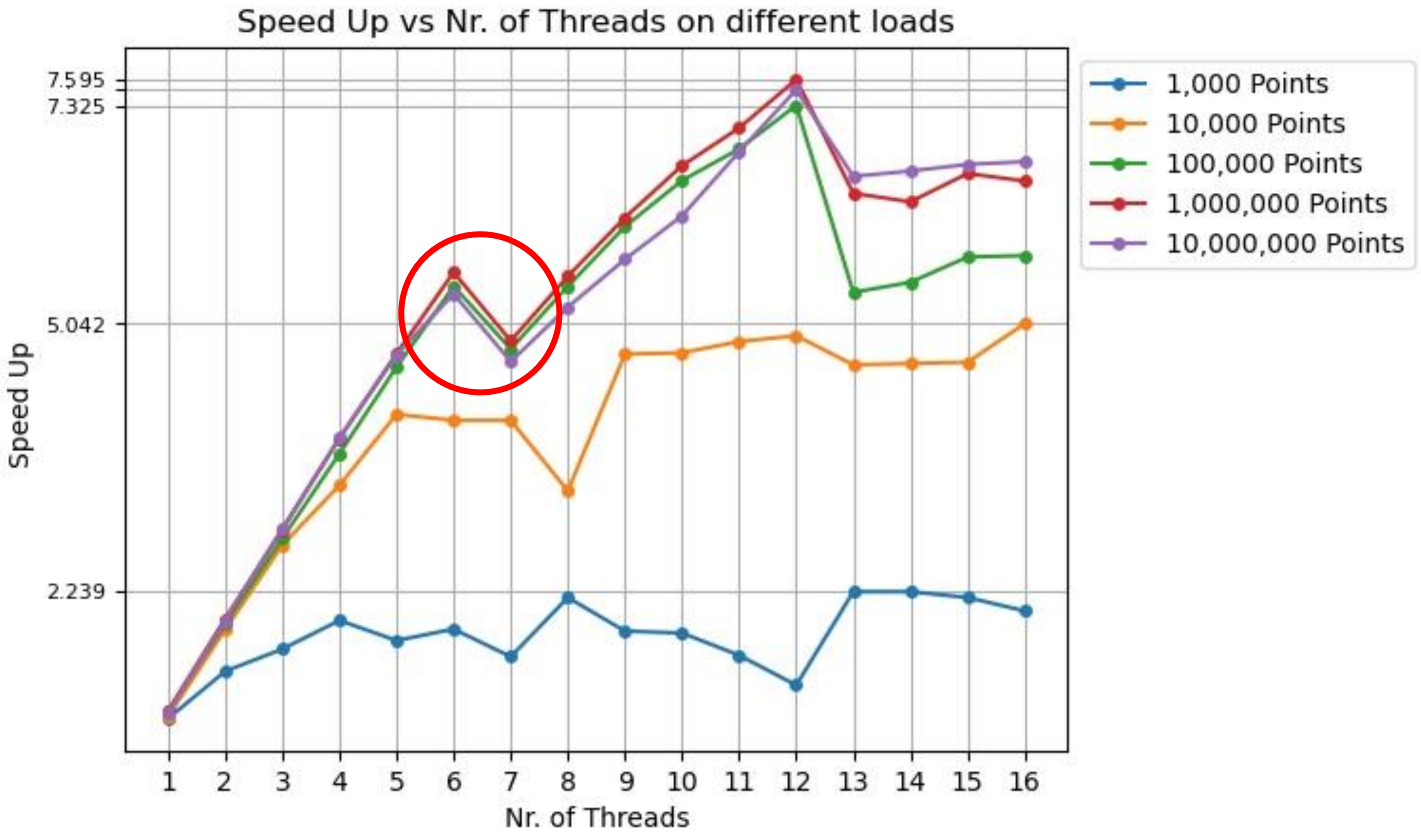
- Clusters=5 & Max # of iterations = 10

● Running ● CPU in use



Speed Up

affinity set to follow physical cores order

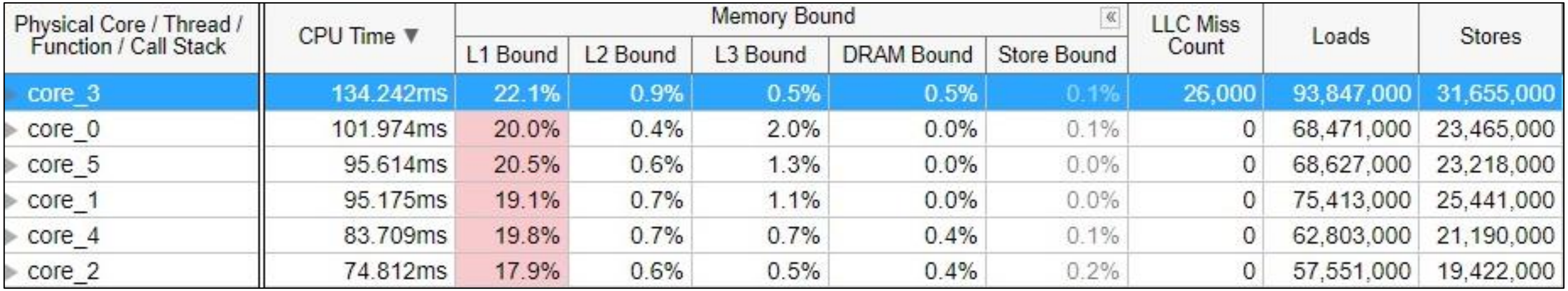


What’s the cause of this effect?

Memory bound : issues on the memory subsystem (cache) that causes the introduction of stalls inside pipeline slots due miss.

How to interpret the values

Intel VTune



Physical Core / Thread / Function / Call Stack	CPU Time ▼	Memory Bound					LLC Miss Count	Loads	Stores
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound			
core_3	134.242ms	22.1%	0.9%	0.5%	0.5%	0.1%	26,000	93,847,000	31,655,000
core_0	101.974ms	20.0%	0.4%	2.0%	0.0%	0.1%	0	68,471,000	23,465,000
core_5	95.614ms	20.5%	0.6%	1.3%	0.0%	0.0%	0	68,627,000	23,218,000
core_1	95.175ms	19.1%	0.7%	1.1%	0.0%	0.0%	0	75,413,000	25,441,000
core_4	83.709ms	19.8%	0.7%	0.7%	0.4%	0.1%	0	62,803,000	21,190,000
core_2	74.812ms	17.9%	0.6%	0.5%	0.4%	0.2%	0	57,551,000	19,422,000

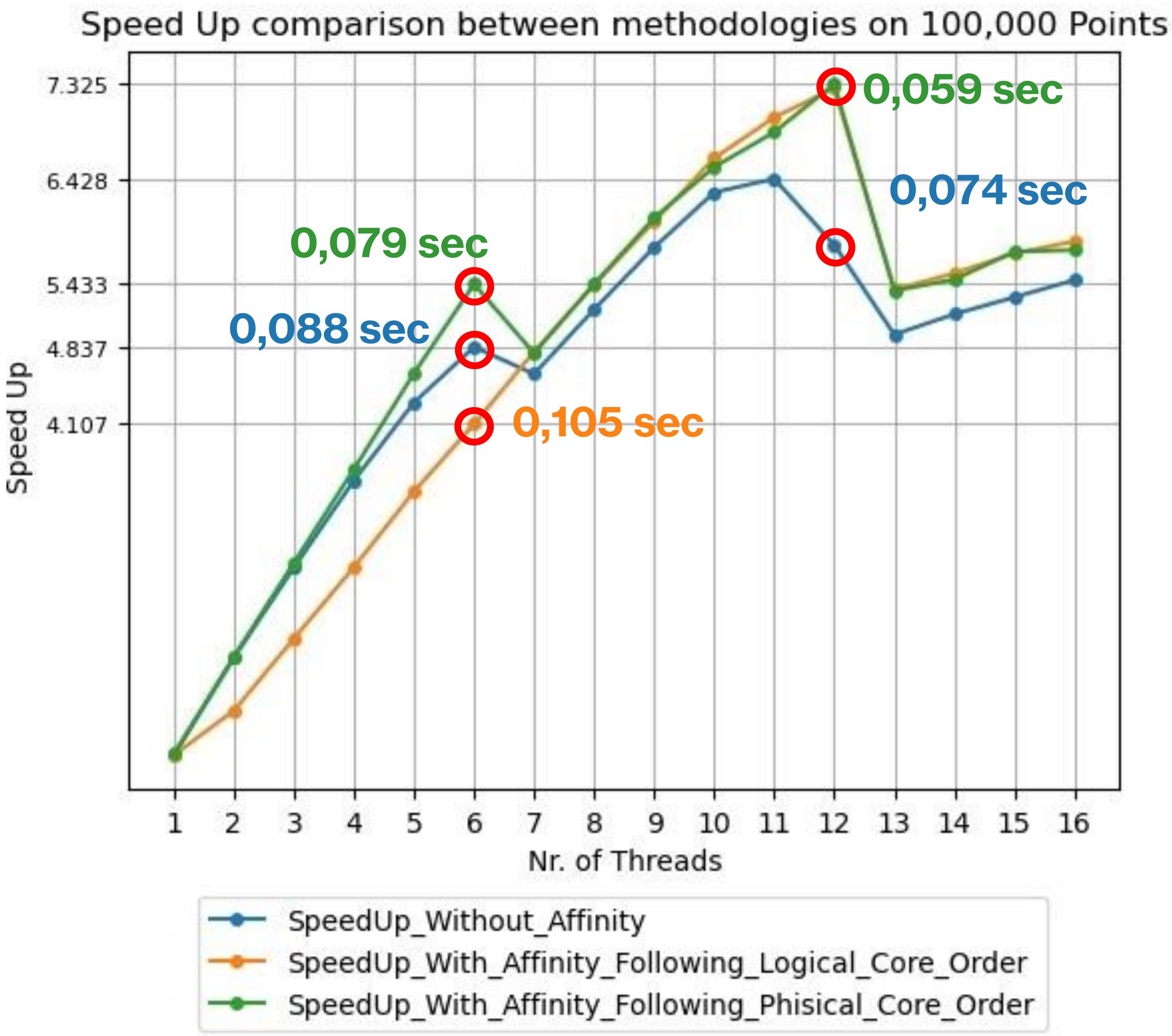
- 100 000 Points dataset

- 6 Threads

- Without Affinity

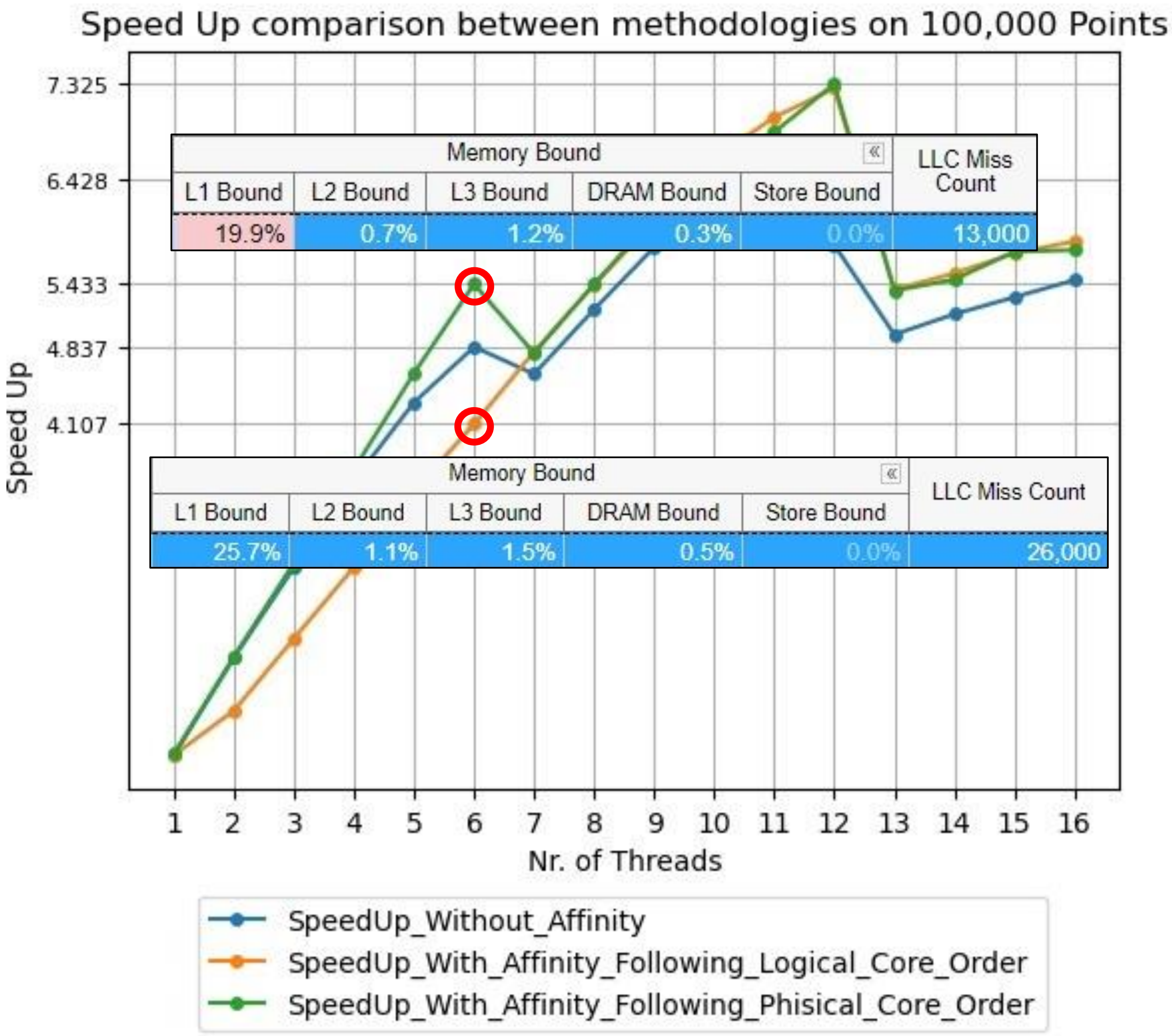
Speed Up

Comparison on 100.000 points with different affinity methodologies



Speed Up

Comparison on 100.000 points with different affinity methodologies

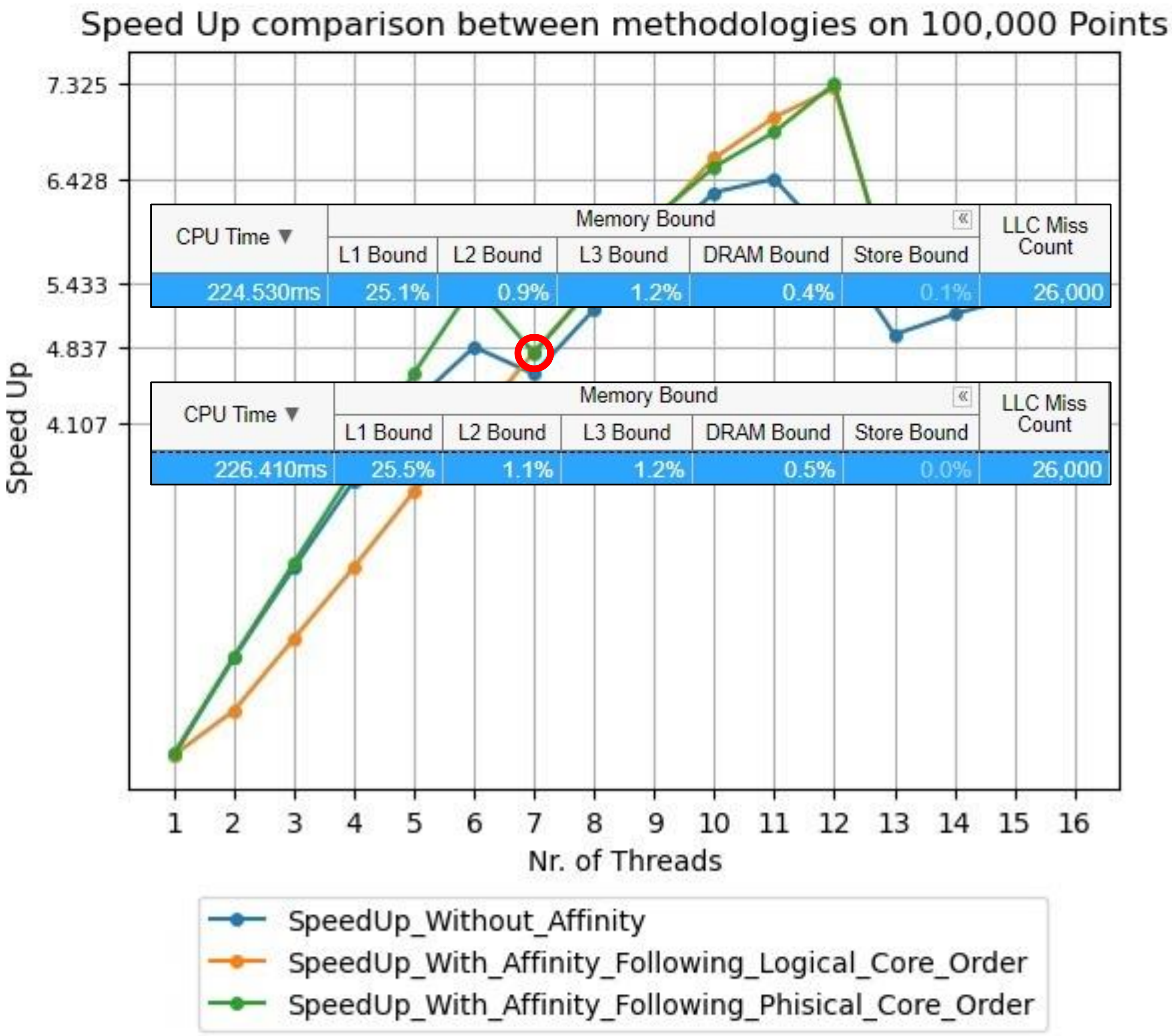


Memory Bound					LLC Miss Count
L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
19.9%	0.7%	1.2%	0.3%	0.0%	13,000

Memory Bound					LLC Miss Count
L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
25.7%	1.1%	1.5%	0.5%	0.0%	26,000

Speed Up

Comparison on 100.000 points with different affinity methodologies

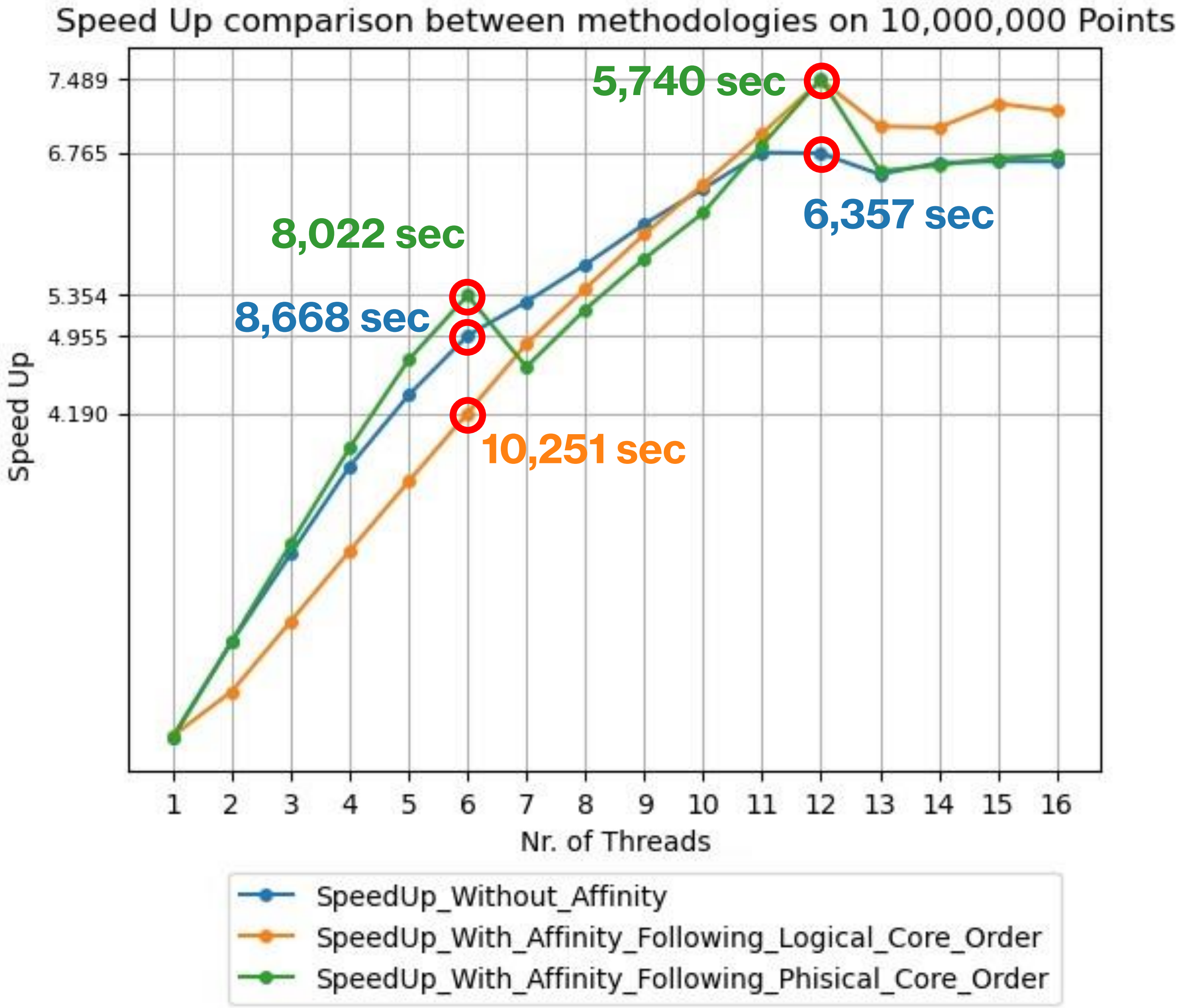


CPU Time ▼	Memory Bound					LLC Miss Count
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
224.530ms	25.1%	0.9%	1.2%	0.4%	0.1%	26,000

CPU Time ▼	Memory Bound					LLC Miss Count
	L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	
226.410ms	25.5%	1.1%	1.2%	0.5%	0.0%	26,000

Speed Up

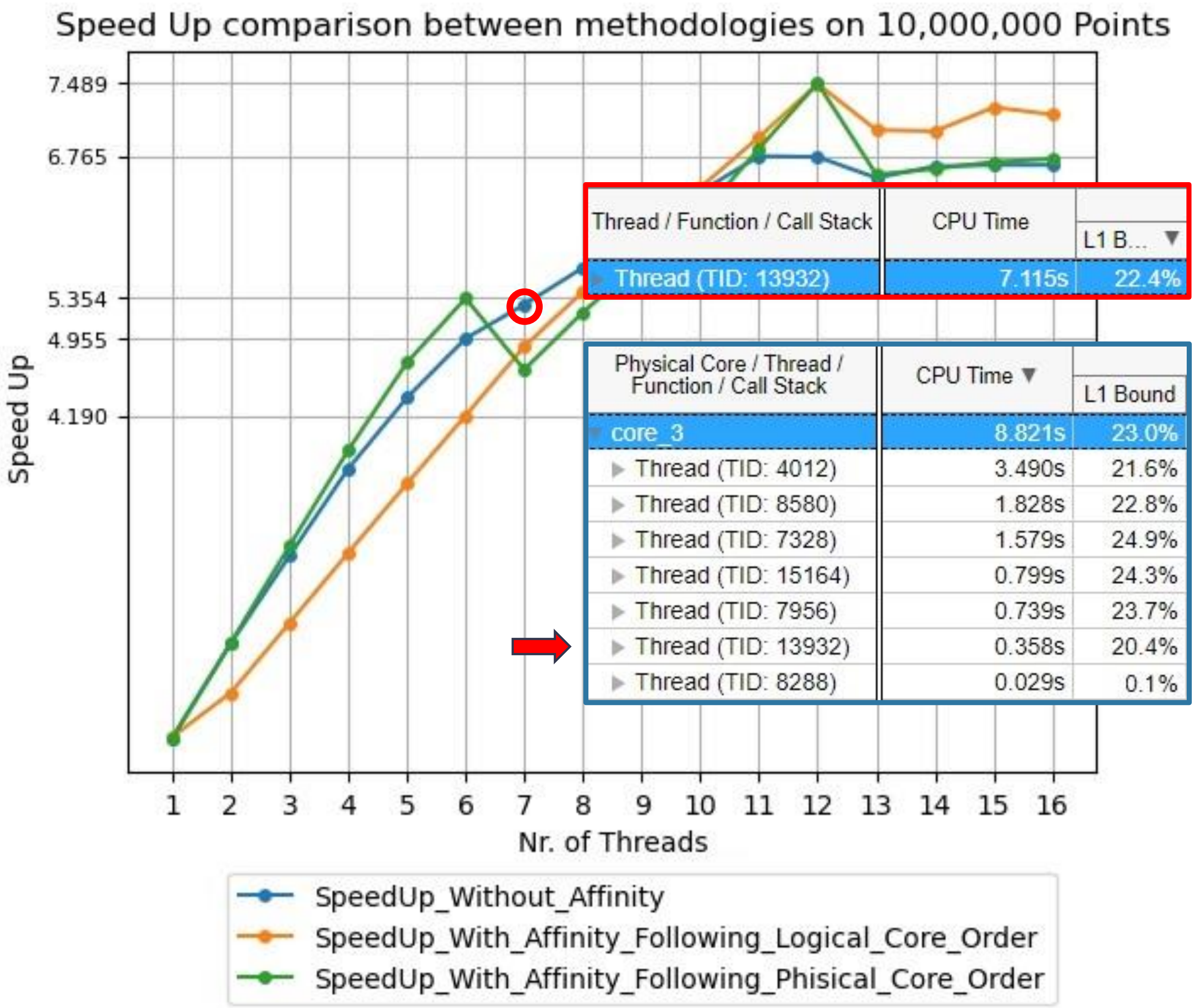
Comparison on 10.000.000 points with different affinity methodologies



Speed Up

Comparison on 10.000.000 points with different affinity methodologies

The O.S obtains a smoother curve increasing the load and in some cases better than with fixed affinity management.



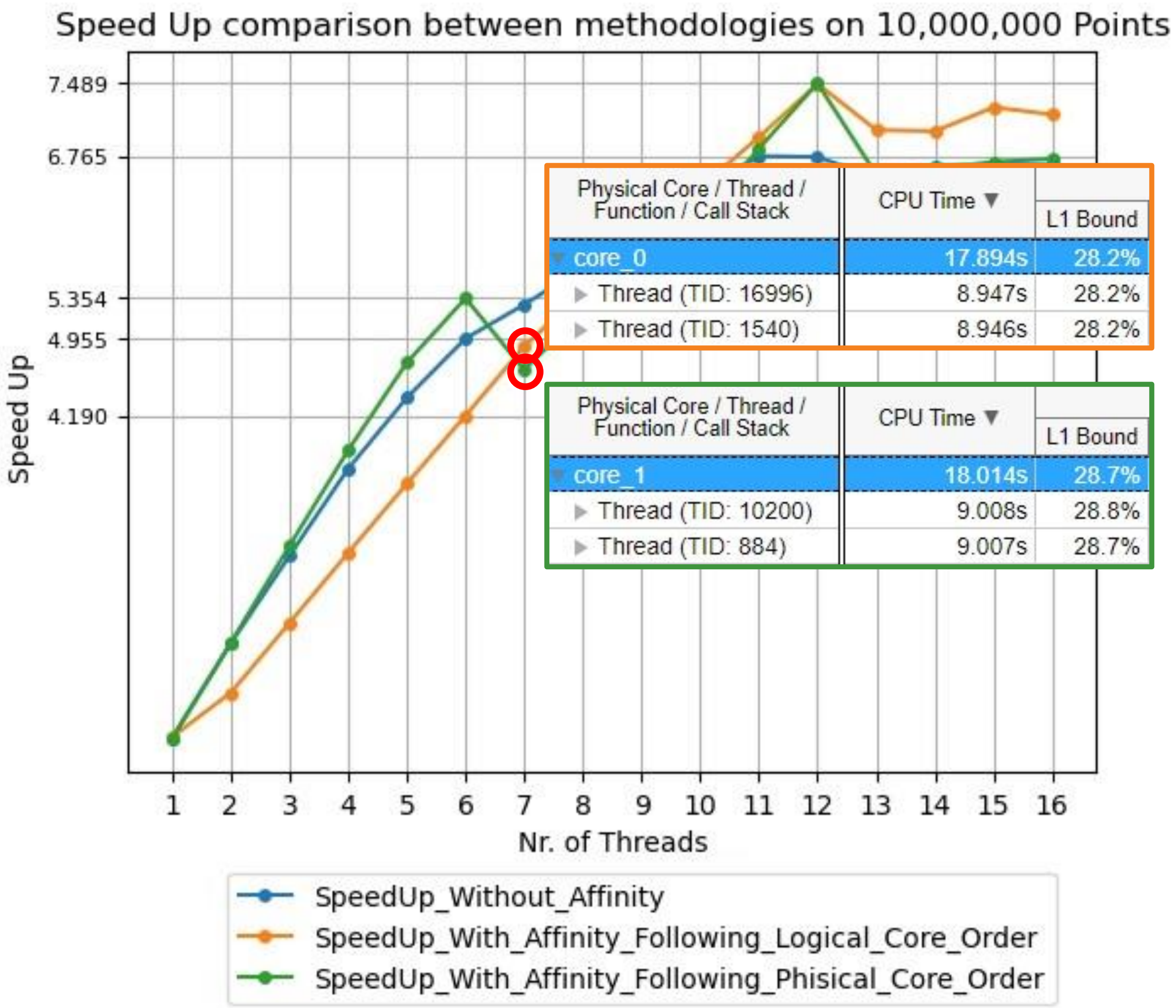
Thread / Function / Call Stack	CPU Time	L1 B... ▼
Thread (TID: 13932)	7.115s	22.4%

Physical Core / Thread / Function / Call Stack	CPU Time ▼	L1 Bound
core_3	8.821s	23.0%
▶ Thread (TID: 4012)	3.490s	21.6%
▶ Thread (TID: 8580)	1.828s	22.8%
▶ Thread (TID: 7328)	1.579s	24.9%
▶ Thread (TID: 15164)	0.799s	24.3%
▶ Thread (TID: 7956)	0.739s	23.7%
▶ Thread (TID: 13932)	0.358s	20.4%
▶ Thread (TID: 8288)	0.029s	0.1%

Speed Up

Comparison on 10.000.000 points with different affinity methodologies

The O.S obtains a smoother curve increasing the load and in some cases better than with fixed affinity management.

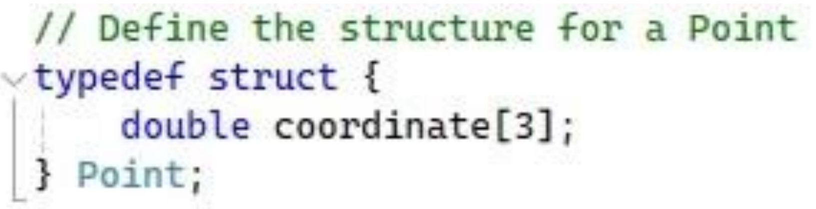


Physical Core / Thread / Function / Call Stack	CPU Time ▼	L1 Bound
core_0	17.894s	28.2%
▶ Thread (TID: 16996)	8.947s	28.2%
▶ Thread (TID: 1540)	8.946s	28.2%

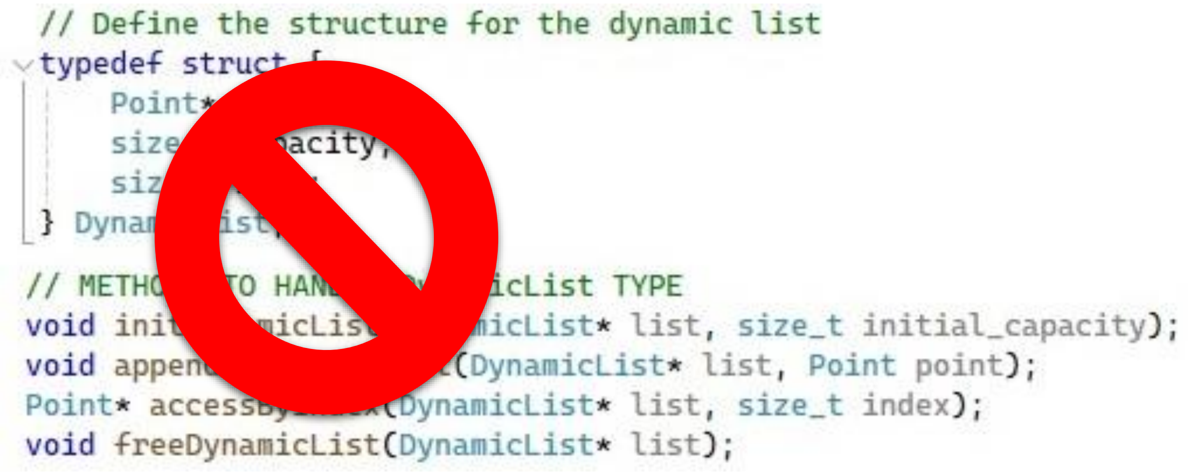
Physical Core / Thread / Function / Call Stack	CPU Time ▼	L1 Bound
core_1	18.014s	28.7%
▶ Thread (TID: 10200)	9.008s	28.8%
▶ Thread (TID: 884)	9.007s	28.7%

Optimizations

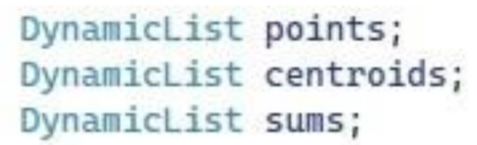
OPT.1: Data structures



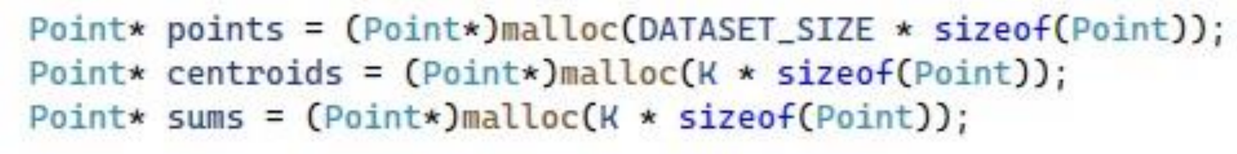
```
// Define the structure for a Point
typedef struct {
    double coordinate[3];
} Point;
```



```
// Define the structure for the dynamic list
typedef struct {
    Point*
    size     acity;
    siz
} Dynam     ist;

// METHO     TO HAN       icList TYPE
void init     micLis     micList* list, size_t initial_capacity);
void appen          (DynamicList* list, Point point);
Point* accessB      (DynamicList* list, size_t index);
void freeDynamicList(DynamicList* list);
```



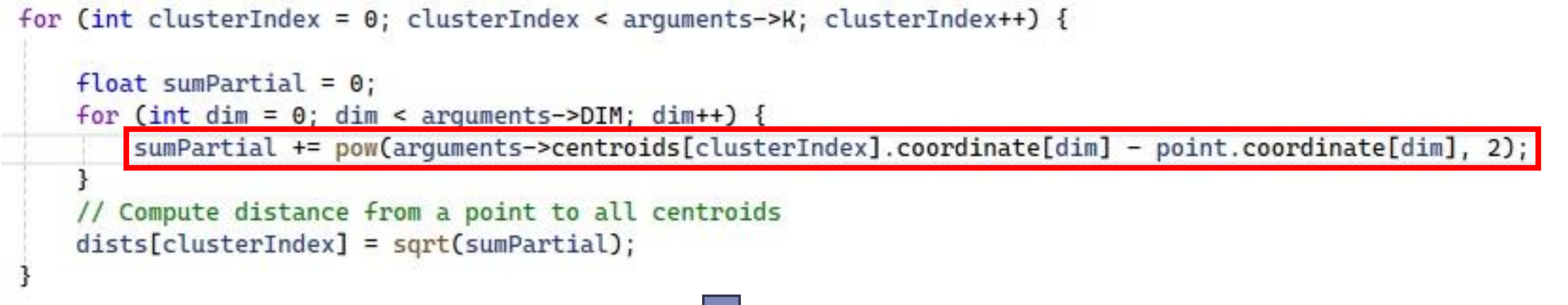
```
DynamicList points;
DynamicList centroids;
DynamicList sums;
```



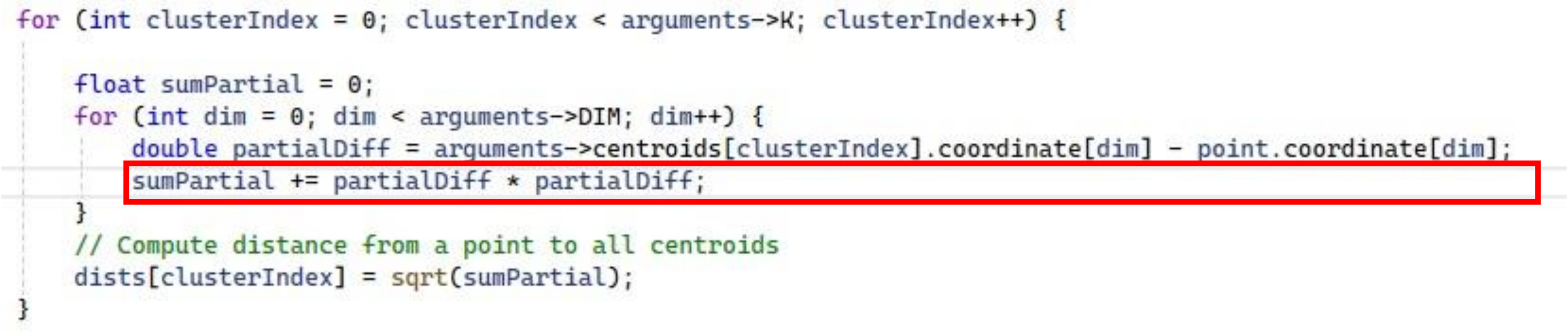
```
Point* points = (Point*)malloc(DATASET_SIZE * sizeof(Point));
Point* centroids = (Point*)malloc(K * sizeof(Point));
Point* sums = (Point*)malloc(K * sizeof(Point));
```

Optimizations

OPT.2: Math.h



```
for (int clusterIndex = 0; clusterIndex < arguments->K; clusterIndex++) {

    float sumPartial = 0;
    for (int dim = 0; dim < arguments->DIM; dim++) {
        sumPartial += pow(arguments->centroids[clusterIndex].coordinate[dim] - point.coordinate[dim], 2);
    }
    // Compute distance from a point to all centroids
    dists[clusterIndex] = sqrt(sumPartial);
}
```



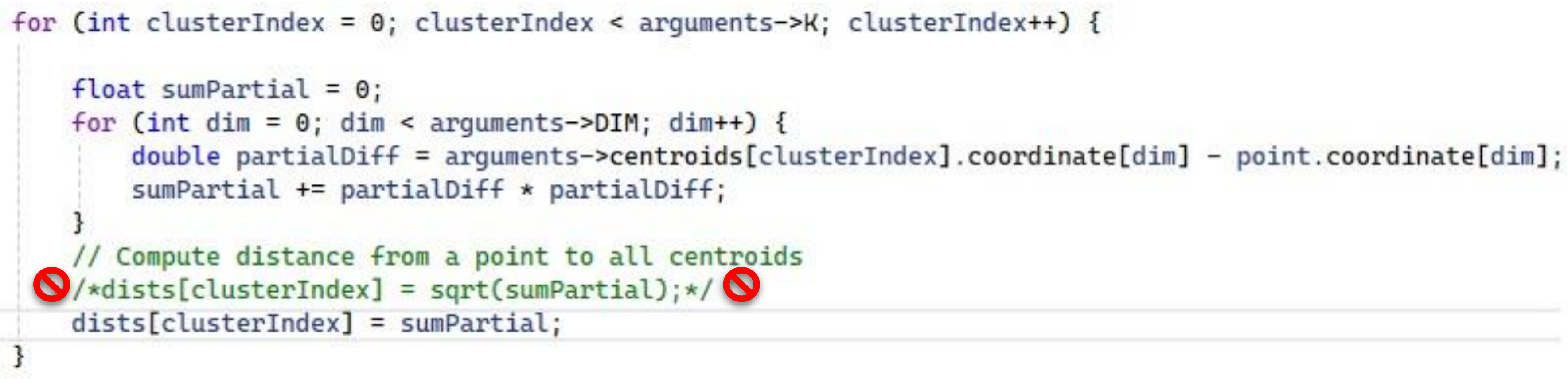
```
for (int clusterIndex = 0; clusterIndex < arguments->K; clusterIndex++) {

    float sumPartial = 0;
    for (int dim = 0; dim < arguments->DIM; dim++) {
        double partialDiff = arguments->centroids[clusterIndex].coordinate[dim] - point.coordinate[dim];
        sumPartial += partialDiff * partialDiff;
    }
    // Compute distance from a point to all centroids
    dists[clusterIndex] = sqrt(sumPartial);
}
```

Optimizations

OPT.3: To act on the algorithm ... a little

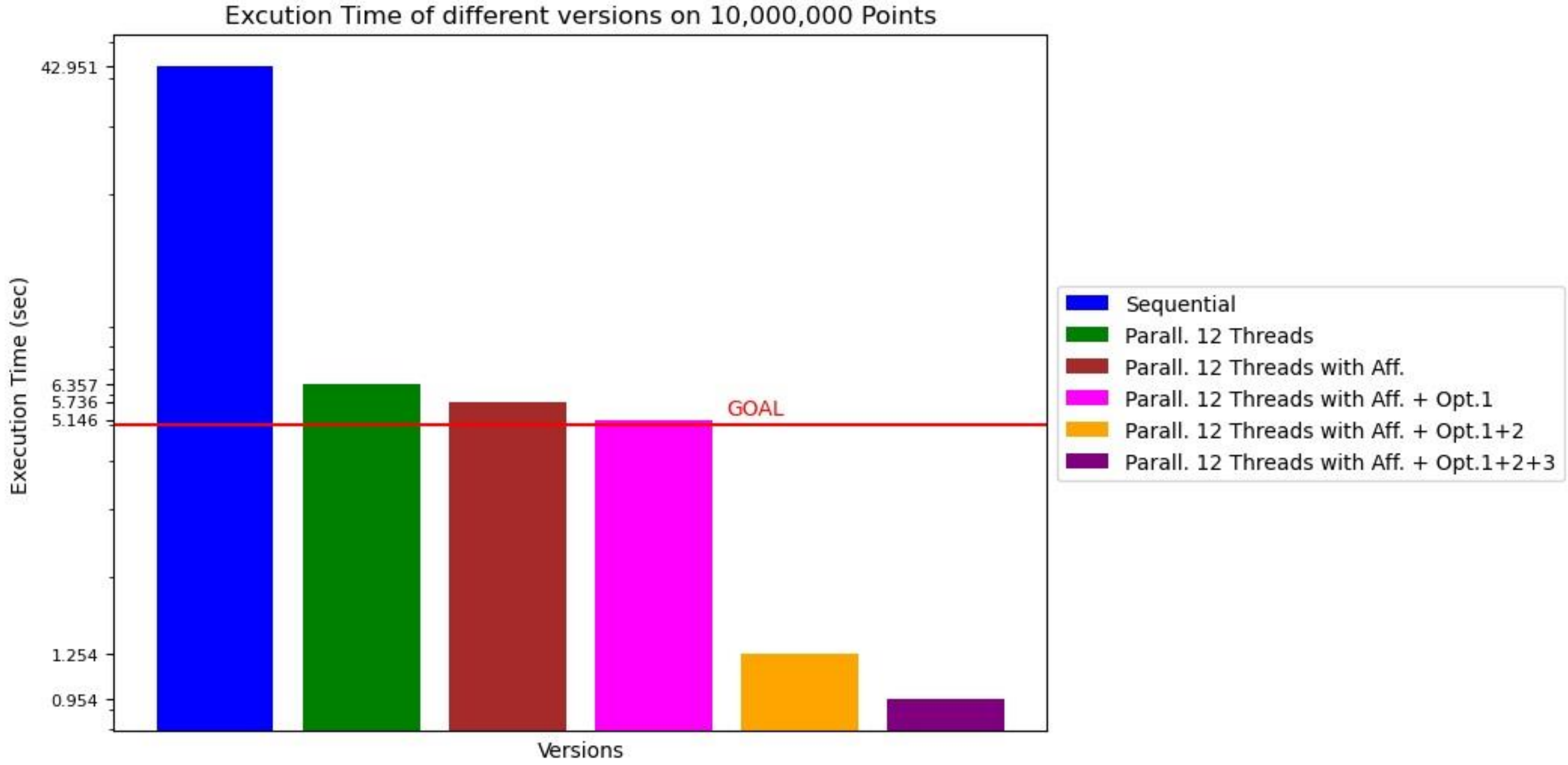
We are not actually interested in obtaining the exact distance, but only to discover the minimum among all the centroids.

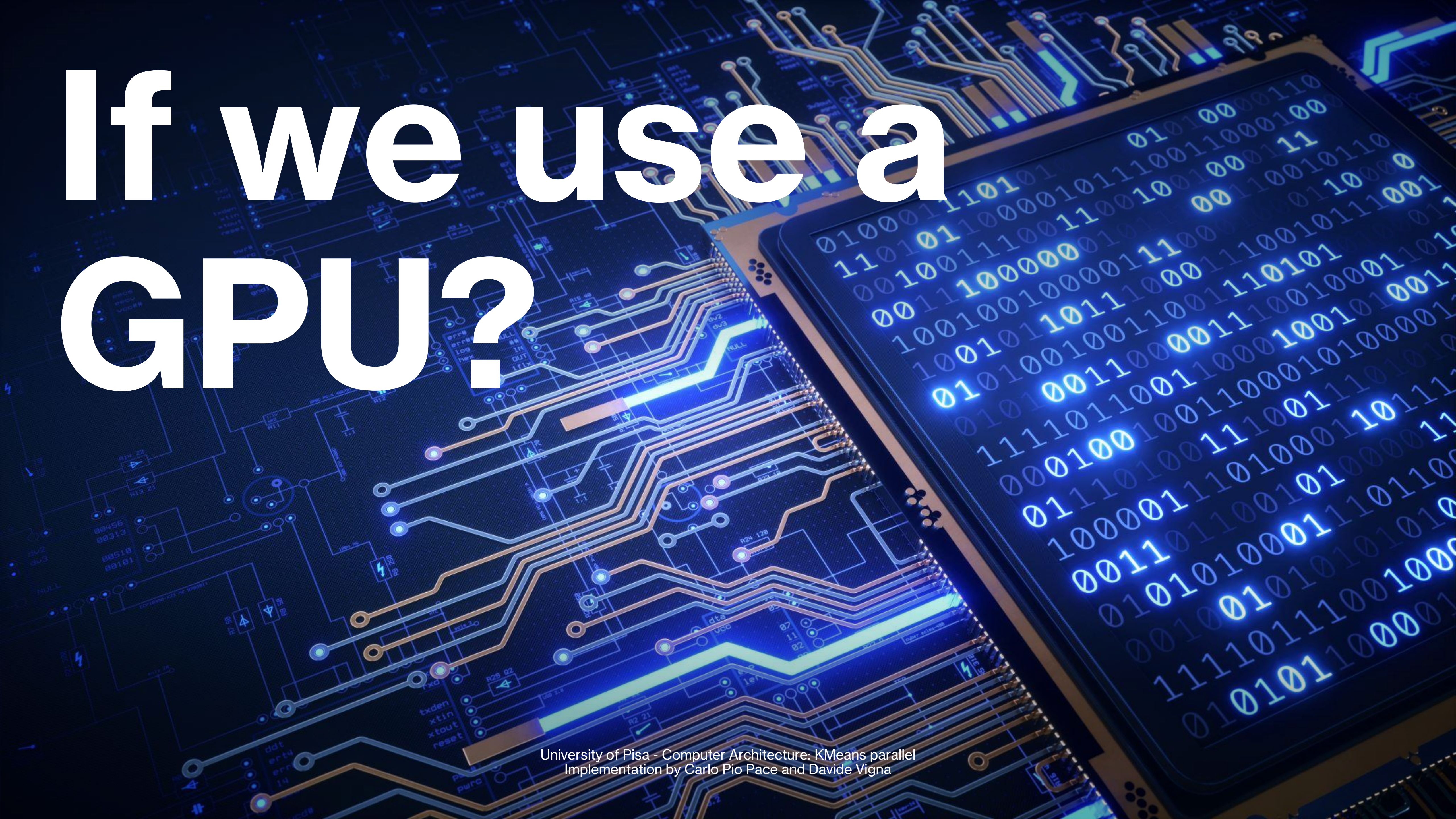
We can avoid to compute the square root.



```
for (int clusterIndex = 0; clusterIndex < arguments->K; clusterIndex++) {

    float sumPartial = 0;
    for (int dim = 0; dim < arguments->DIM; dim++) {
        double partialDiff = arguments->centroids[clusterIndex].coordinate[dim] - point.coordinate[dim];
        sumPartial += partialDiff * partialDiff;
    }
    // Compute distance from a point to all centroids
    /*dists[clusterIndex] = sqrt(sumPartial);*/
    dists[clusterIndex] = sumPartial;
}
```

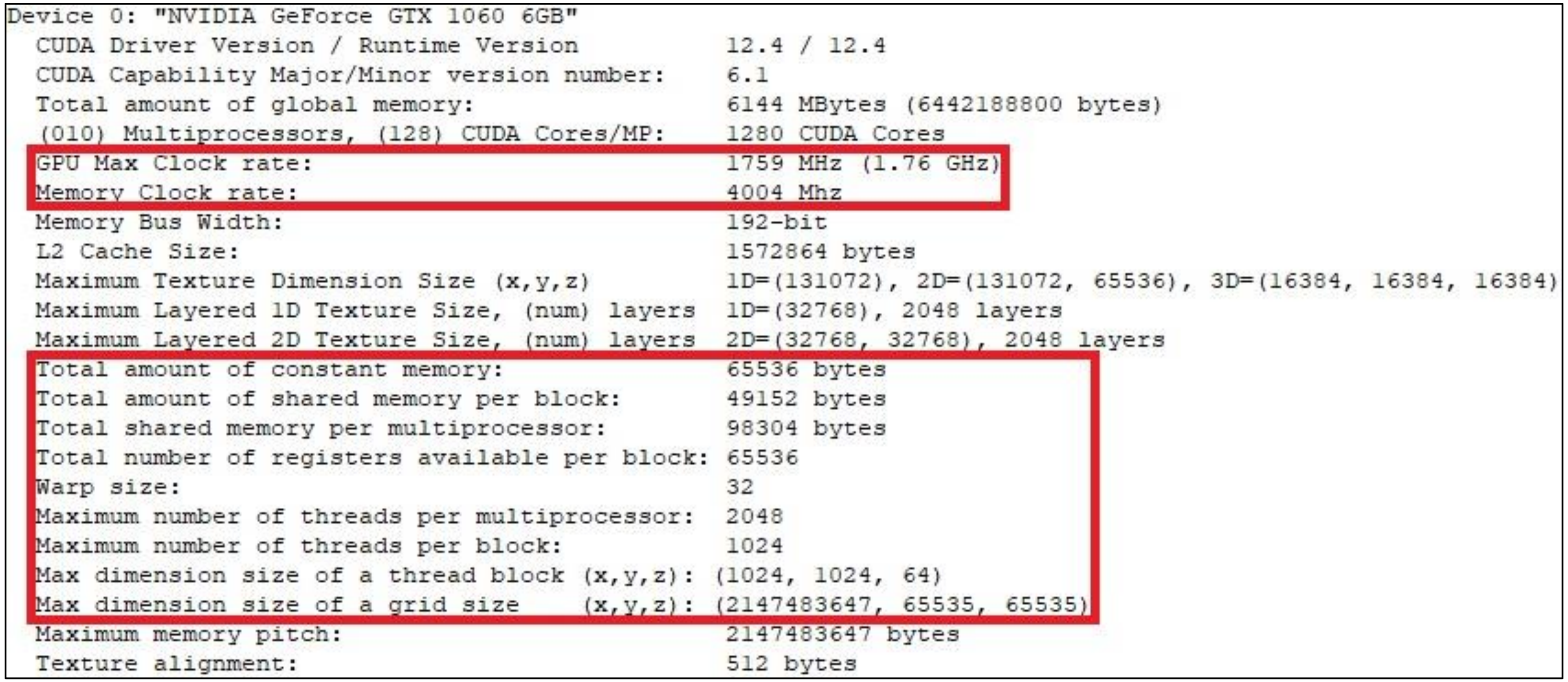
Optimizations results





If we use a GPU?

GPU Nvidia GeForce GTX 1060



```
Device 0: "NVIDIA GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version          12.4 / 12.4
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6144 MBytes (6442188800 bytes)
  (010) Multiprocessors, (128) CUDA Cores/MP:    1280 CUDA Cores
  GPU Max Clock rate:                            1759 MHz (1.76 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
```

GPU – Initial Version

We mantain the same exact structure of the parallel CPU version

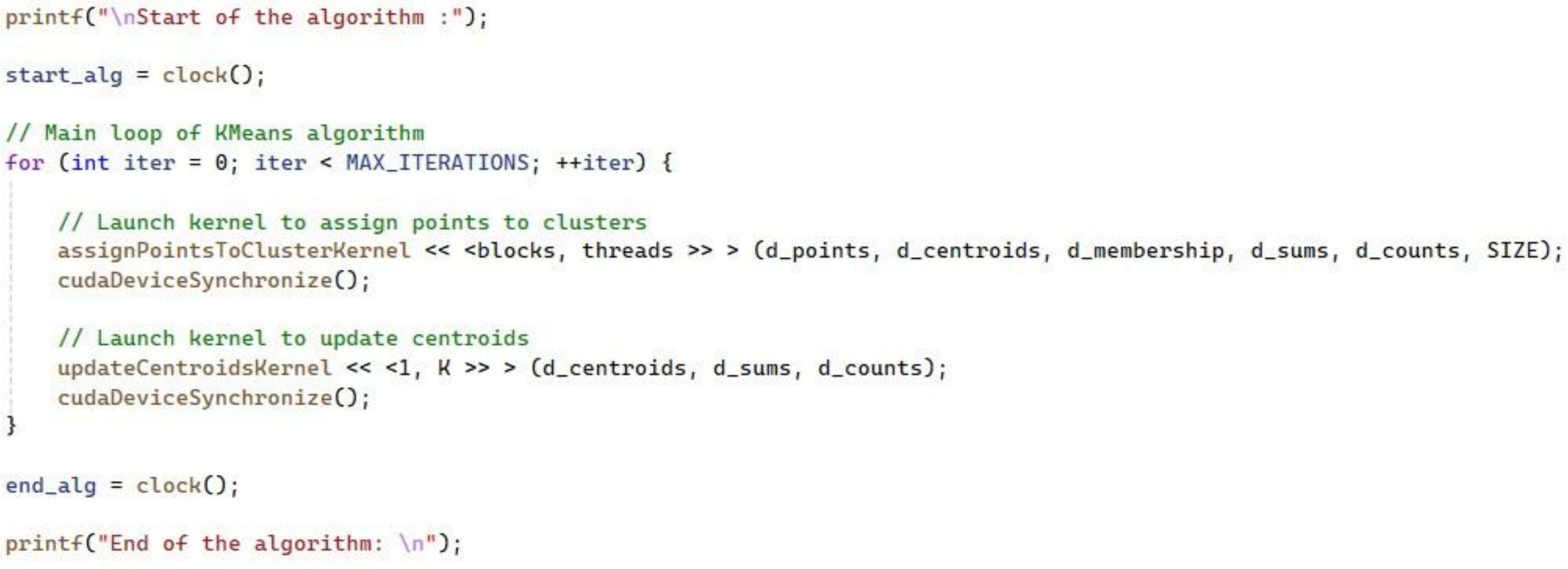
- 2 kernel functions

```
1) assignPointsToClusterKernel<< <blocks, threads >> > (d_points, d_centroids,
d_membership, d_sums, d_counts, SIZE);

2) updateCentroidsKernel << <1, K >> > (d_centroids, d_sums, d_counts);
```

- Same measurement methods: function clock() in time.h C library
  - Focus on pure execution time

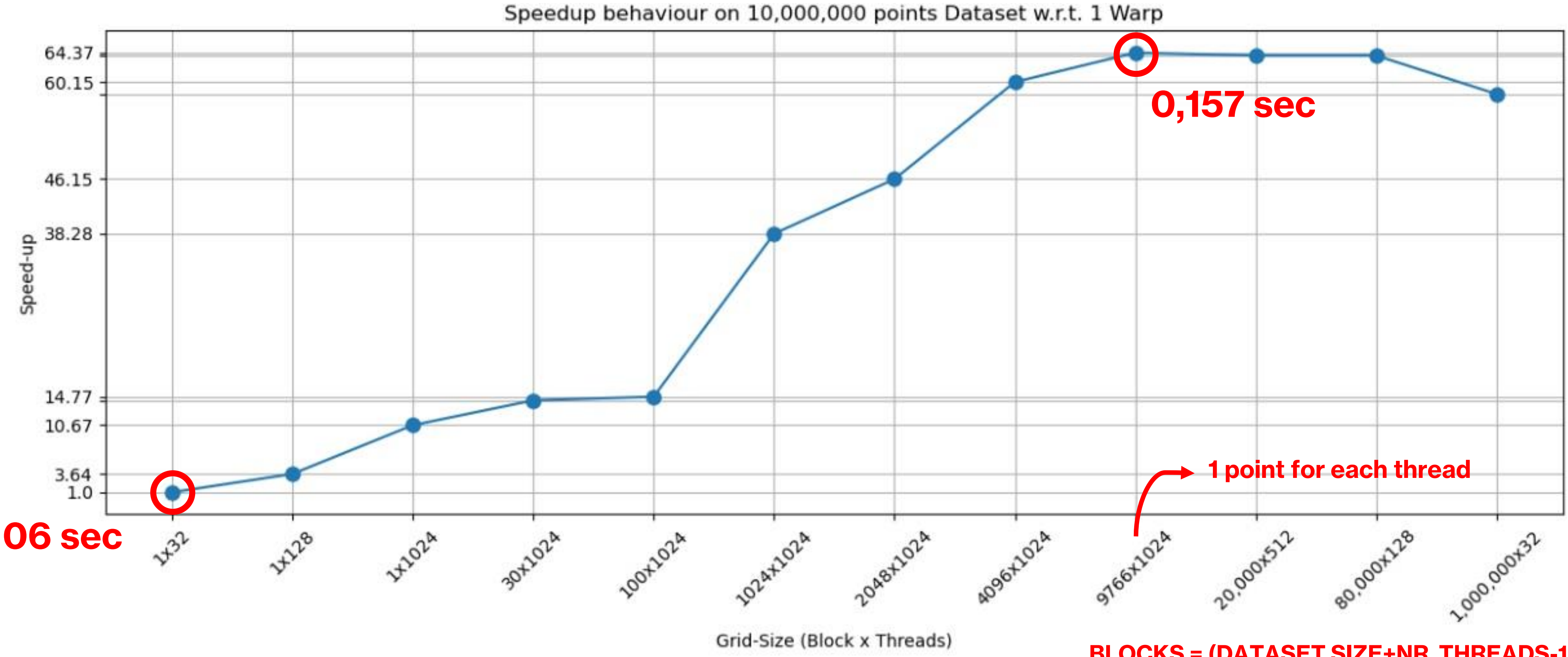
- What is its behavior? How do we select the # of blocks and threads?

GPU – Initial Version



```
printf("\nStart of the algorithm :");

start_alg = clock();

// Main loop of KMeans algorithm
for (int iter = 0; iter < MAX_ITERATIONS; ++iter) {

    // Launch kernel to assign points to clusters
    assignPointsToClusterKernel << <blocks, threads >> > (d_points, d_centroids, d_membership, d_sums, d_counts, SIZE);
    cudaDeviceSynchronize();

    // Launch kernel to update centroids
    updateCentroidsKernel << <1, K >> > (d_centroids, d_sums, d_counts);
    cudaDeviceSynchronize();
}

end_alg = clock();

printf("End of the algorithm: \n");
```

GPU – Initial Version



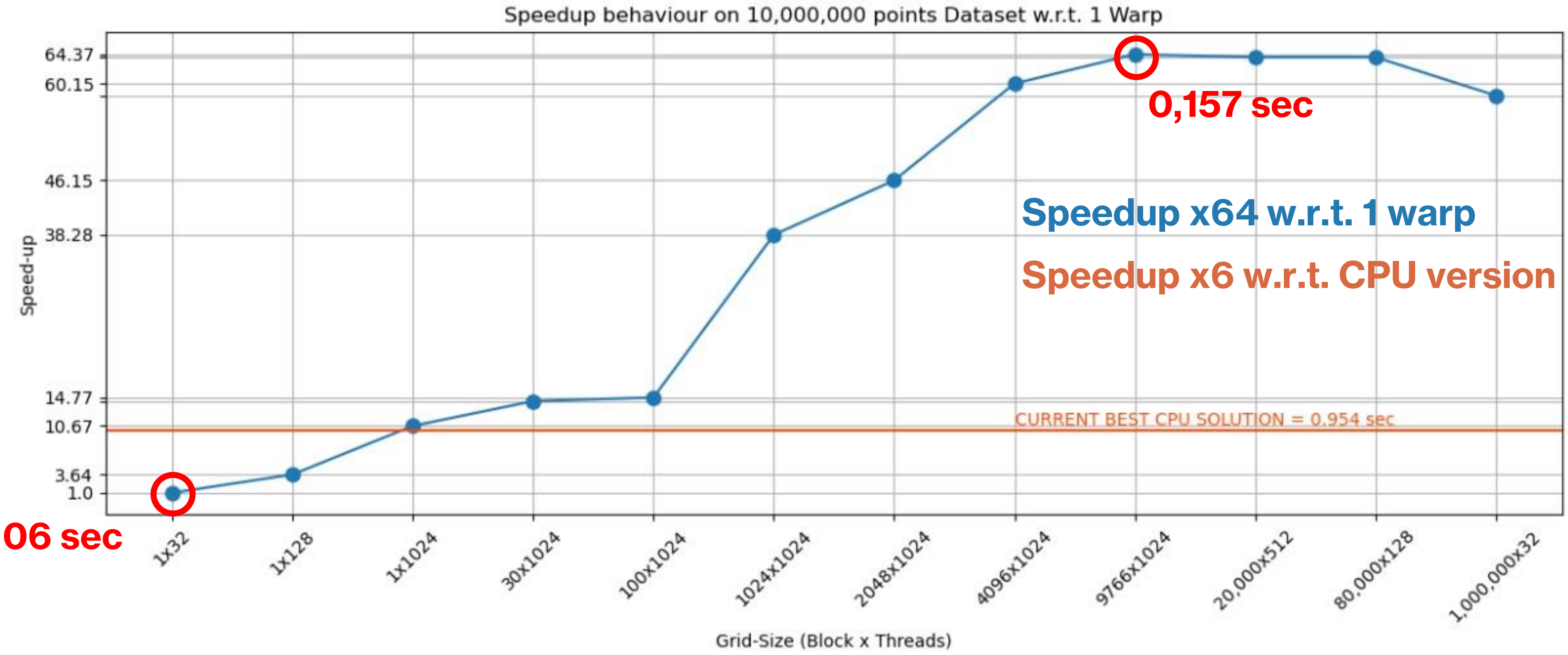
0,157 sec

10,106 sec

1 point for each thread

BLOCKS = (DATASET SIZE+NR_THREADS-1)/ NR_THREADS

GPU – Initial Version



10,106 sec

0,157 sec

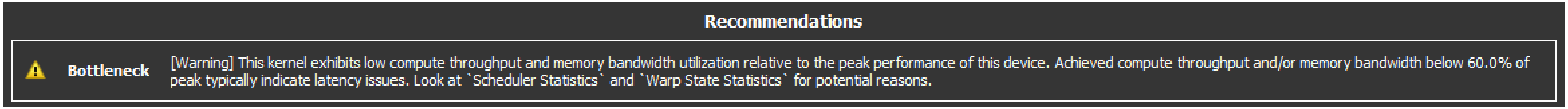
Speedup x64 w.r.t. 1 warp

Speedup x6 w.r.t. CPU version

GPU – Profiler

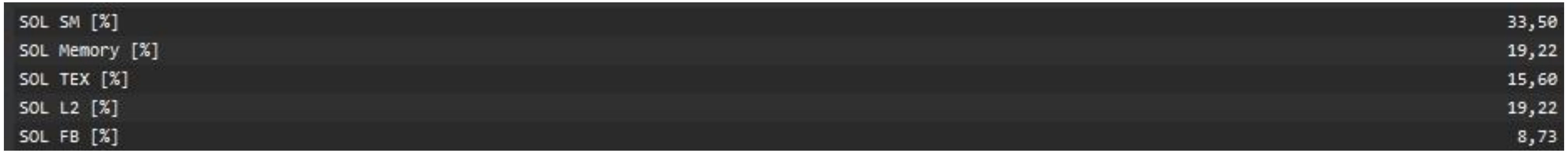
NVIDIA
Nsight™
Compute

v.2019

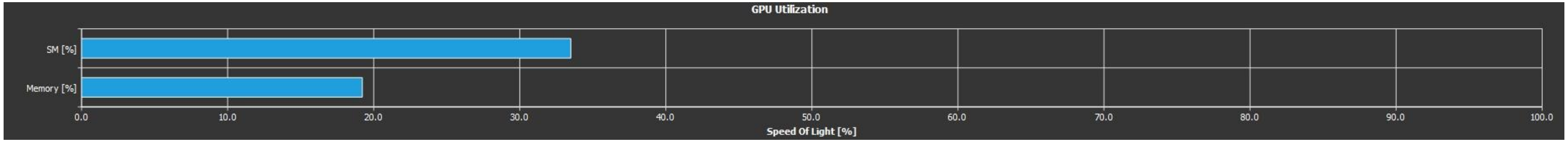


Recommendations

⚠ **Bottleneck** [Warning] This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at `Scheduler Statistics` and `Warp State Statistics` for potential reasons.



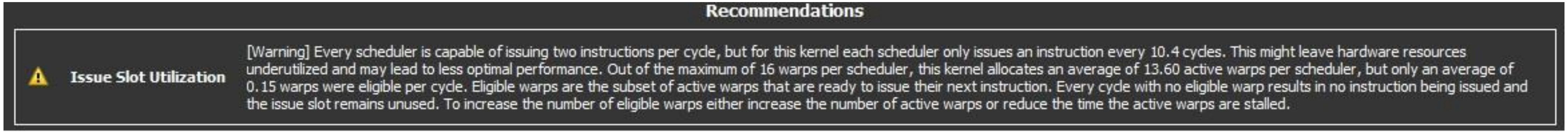
SOL SM [%]	33,50
SOL Memory [%]	19,22
SOL TEX [%]	15,60
SOL L2 [%]	19,22
SOL FB [%]	8,73



GPU – Profiler

NVIDIA Nsight™ Compute

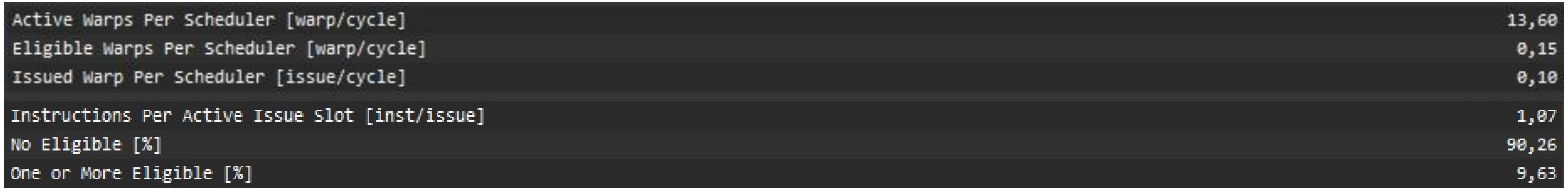
v.2019



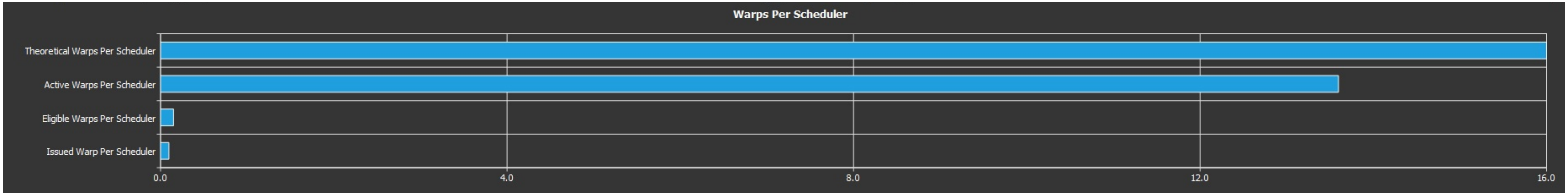
Recommendations

⚠ **Issue Slot Utilization**

[Warning] Every scheduler is capable of issuing two instructions per cycle, but for this kernel each scheduler only issues an instruction every 10.4 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 13.60 active warps per scheduler, but only an average of 0.15 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

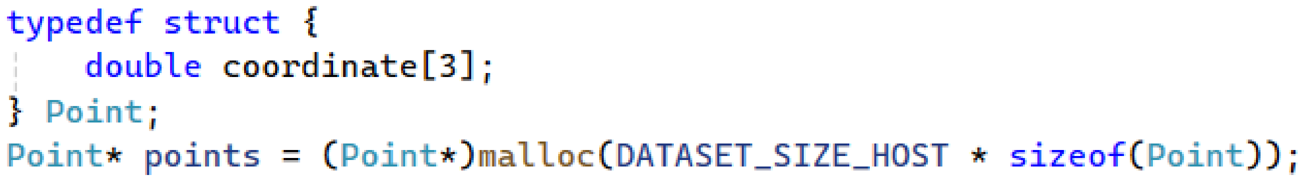


Active Warps Per Scheduler [warp/cycle]	13,60
Eligible Warps Per Scheduler [warp/cycle]	0,15
Issued Warp Per Scheduler [issue/cycle]	0,10
Instructions Per Active Issue Slot [inst/issue]	1,07
No Eligible [%]	90,26
One or More Eligible [%]	9,63



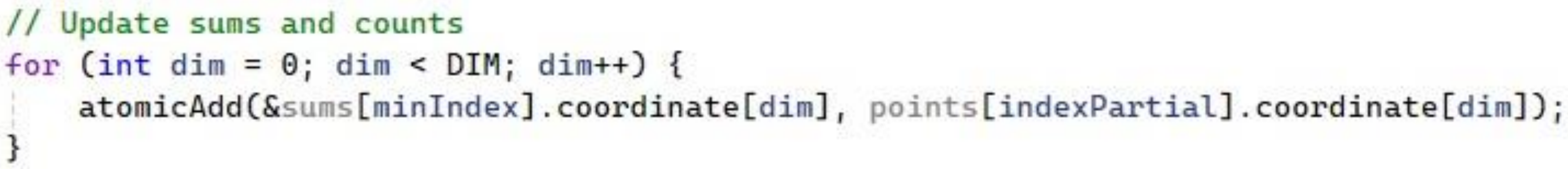
GPU – Considerations

Data structure used:



```
typedef struct {
    double coordinate[3];
} Point;
Point* points = (Point*)malloc(DATASET_SIZE_HOST * sizeof(Point));
```

Operation performed:



```
// Update sums and counts
for (int dim = 0; dim < DIM; dim++) {
    atomicAdd(&sums[minIndex].coordinate[dim], points[indexPartial].coordinate[dim]);
}
```

Atomic operations on double generate errors at compilation time!!

We used this one!

How to solve?

Two options:

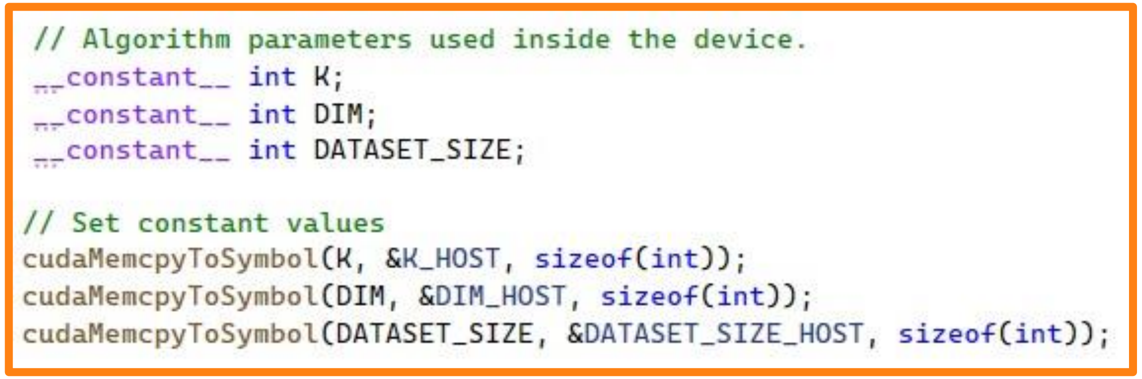
1. Harware option: nvcc kernel.cu -o KMeansCuda -arch=sm_61 (Faster)
2. Software (Slower)

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#gpu-feature-list

GPU – Considerations

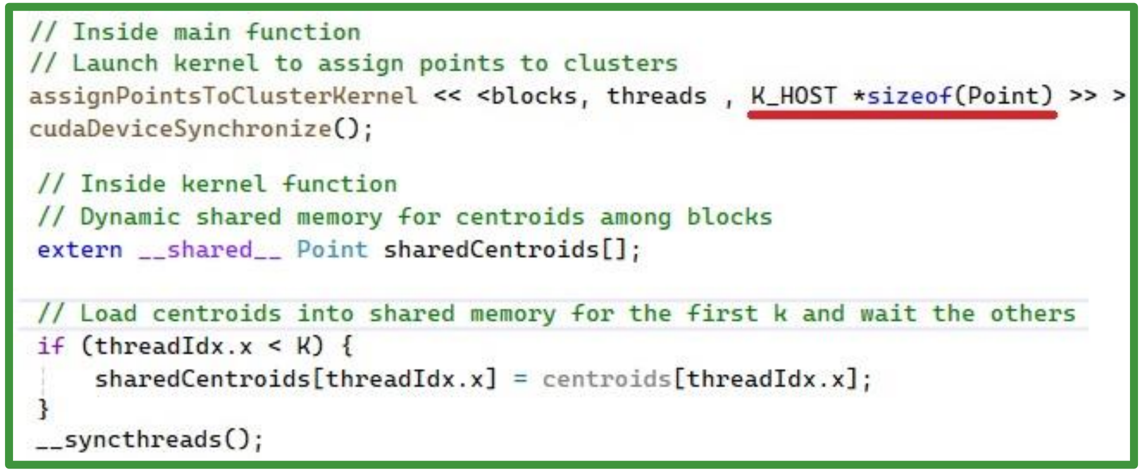
Memory usage

- Constant memory



```
// Algorithm parameters used inside the device.
__constant__ int K;
__constant__ int DIM;
__constant__ int DATASET_SIZE;

// Set constant values
cudaMemcpyToSymbol(K, &K_HOST, sizeof(int));
cudaMemcpyToSymbol(DIM, &DIM_HOST, sizeof(int));
cudaMemcpyToSymbol(DATASET_SIZE, &DATASET_SIZE_HOST, sizeof(int));
```

- Shared Memory

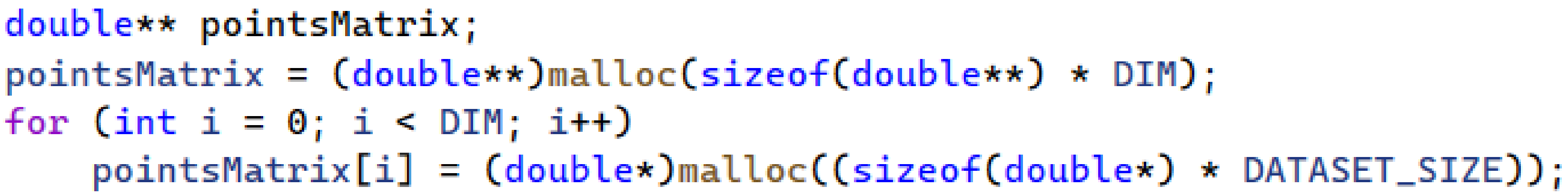


```
// Inside main function
// Launch kernel to assign points to clusters
assignPointsToClusterKernel << <blocks, threads , K_HOST *sizeof(Point) >> >
cudaDeviceSynchronize();

 // Inside kernel function
 // Dynamic shared memory for centroids among blocks
 extern __shared__ Point sharedCentroids[];

 // Load centroids into shared memory for the first k and wait the others
 if (threadIdx.x < K) {
     sharedCentroids[threadIdx.x] = centroids[threadIdx.x];
 }
 __syncthreads();
```

GPU – Considerations

To maximaze the usage of the architecure, considering that we are operating with bi-dimensional points, let’s exploit gpu features, and use a 2D grid, in order to do so, we need to change the data structure previuosly used, to a 2D matrix.

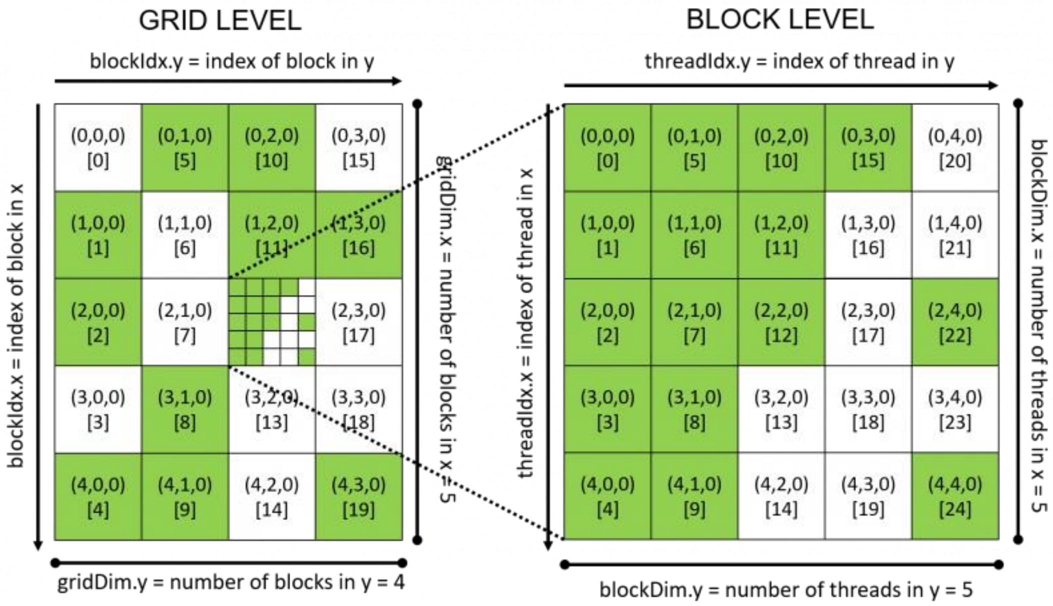
2D Matrix of doubles



```
double** pointsMatrix;
pointsMatrix = (double**)malloc(sizeof(double**) * DIM);
for (int i = 0; i < DIM; i++)
    pointsMatrix[i] = (double*)malloc((sizeof(double*) * DATASET_SIZE));
```

GPU – Considerations

2D-GRID


GRID LEVEL
blockIdx.y = index of block in y
blockIdx.x = index of block in x
gridDim.x = number of blocks in x = 5
gridDim.y = number of blocks in y = 4
BLOCK LEVEL
threadIdx.y = index of thread in y
threadIdx.x = index of thread in x
blockDim.x = number of threads in x = 5
blockDim.y = number of threads in y = 5

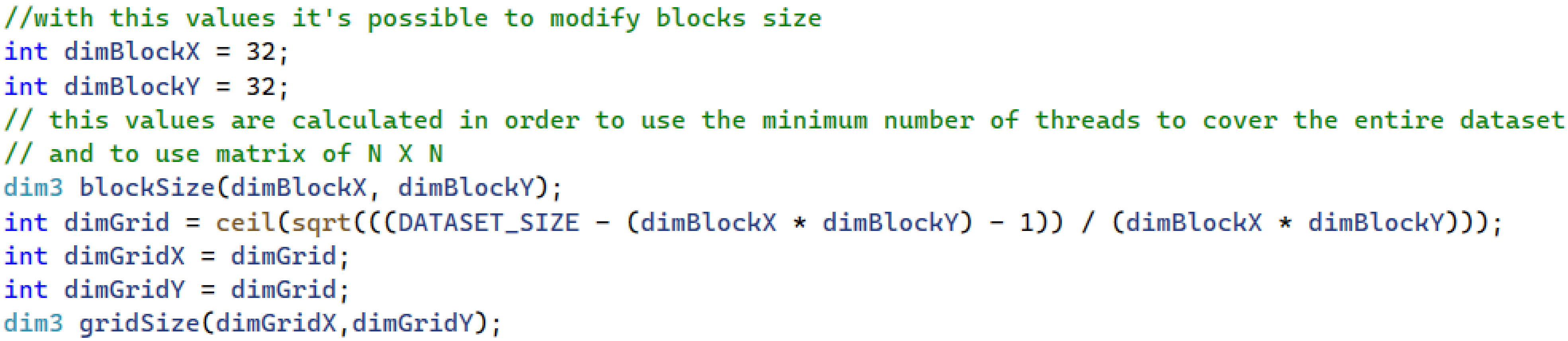
(0,0,0) [0]	(0,1,0) [5]	(0,2,0) [10]	(0,3,0) [15]
(1,0,0) [1]	(1,1,0) [6]	(1,2,0) [11]	(1,3,0) [16]
(2,0,0) [2]	(2,1,0) [7]		(2,3,0) [17]
(3,0,0) [3]	(3,1,0) [8]	(3,2,0) [13]	(3,3,0) [18]
(4,0,0) [4]	(4,1,0) [9]	(4,2,0) [14]	(4,3,0) [19]

(0,0,0) [0]	(0,1,0) [5]	(0,2,0) [10]	(0,3,0) [15]	(0,4,0) [20]
(1,0,0) [1]	(1,1,0) [6]	(1,2,0) [11]	(1,3,0) [16]	(1,4,0) [21]
(2,0,0) [2]	(2,1,0) [7]	(2,2,0) [12]	(2,3,0) [17]	(2,4,0) [22]
(3,0,0) [3]	(3,1,0) [8]	(3,2,0) [13]	(3,3,0) [18]	(3,4,0) [23]
(4,0,0) [4]	(4,1,0) [9]	(4,2,0) [14]	(4,3,0) [19]	(4,4,0) [24]

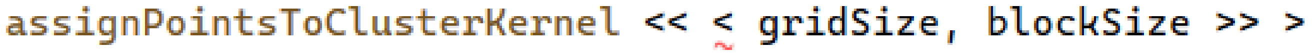
```
double** d_pointsMatrix;
double** d_pointsR;
d_pointsR = (double**)malloc(DIM * sizeof(double*));

cudaMalloc((void**)&d_pointsMatrix, DIM * sizeof(double*));

for (int i = 0; i < DIM; i++)
    cudaMalloc((void**)&d_pointsR[i], DATASET_SIZE * sizeof(double));
```

GPU – Considerations

We used block size large enough to cover the entire dataset, and in order to provide enough warps to schedule, using the architecture at it’s maximum.

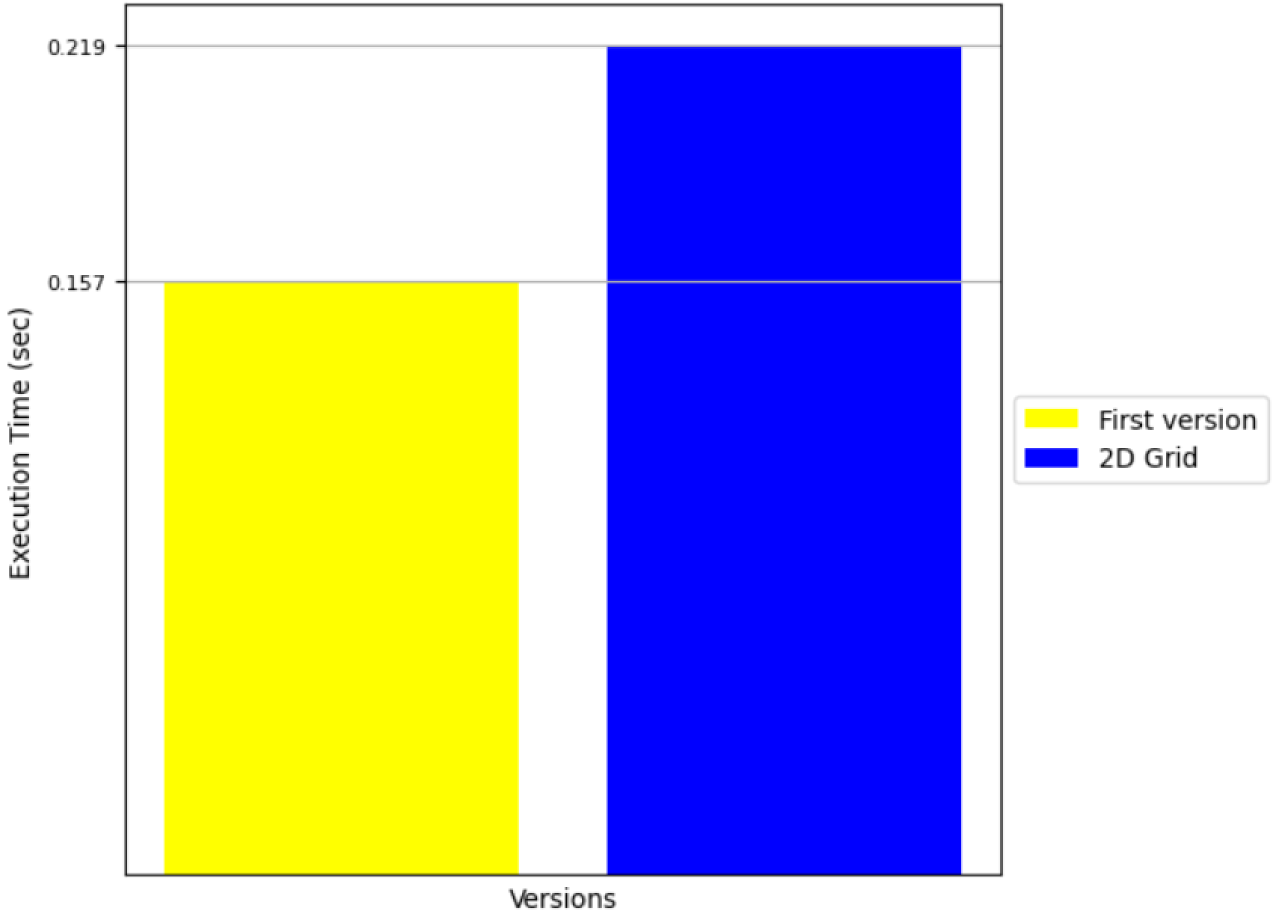


```
//with this values it's possible to modify blocks size
int dimBlockX = 32;
int dimBlockY = 32;
// this values are calculated in order to use the minimum number of threads to cover the entire dataset
// and to use matrix of N X N
dim3 blockSize(dimBlockX, dimBlockY);
int dimGrid = ceil(sqrt(((DATASET_SIZE - (dimBlockX * dimBlockY) - 1)) / (dimBlockX * dimBlockY)));
int dimGridX = dimGrid;
int dimGridY = dimGrid;
dim3 gridSize(dimGridX,dimGridY);
```



```
assignPointsToClusterKernel << < gridSize, blockSize >> >
```

Execution time comparison between Initial version and using a 2D Grid

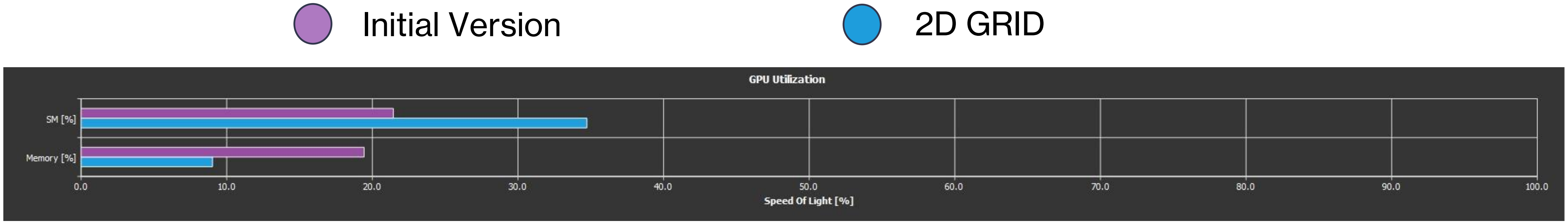


GPU – Profiler

NVIDIA
Nsight™
Compute

v.2019

We reached the goal of increasing the usage of the Multiprocessors (+60% SM utilization) but the exploiting of the memory subsystem of the cache is worse with the new version (-53%)

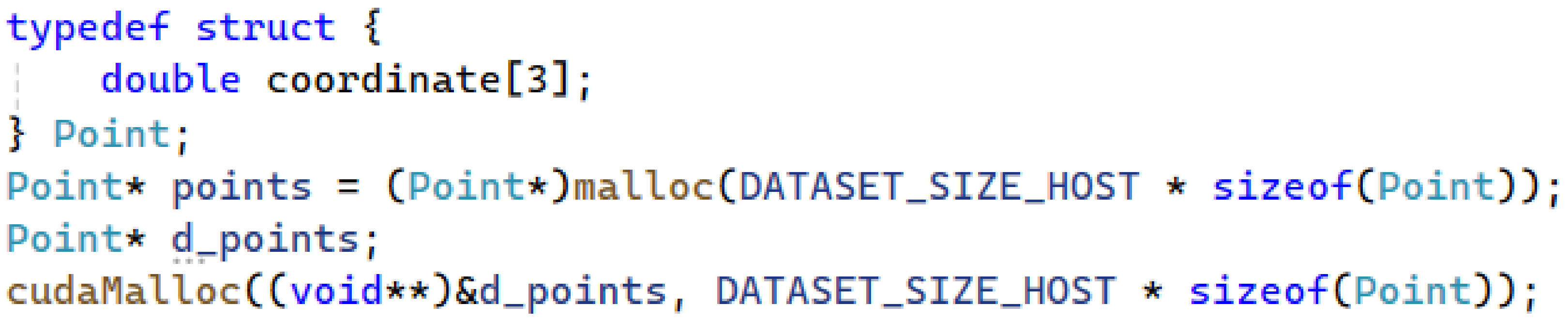


Why?

Flattening !

On gpu, if we are operating on bi-dimensional data, if is possible, is highly reccomended to perform a flattening and operate on one dimension data structure.

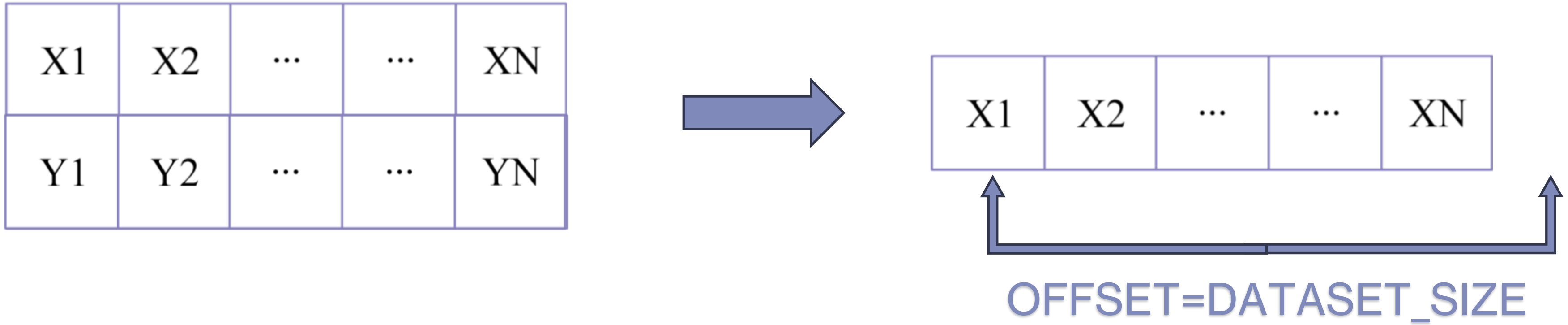
Our first implementation alredy used a data structure that exploited a sort of flattening.



```
typedef struct {
    double coordinate[3];
} Point;
Point* points = (Point*)malloc(DATASET_SIZE_HOST * sizeof(Point));
Point* d_points;
cudaMalloc((void**)&d_points, DATASET_SIZE_HOST * sizeof(Point));
```

Flattening !

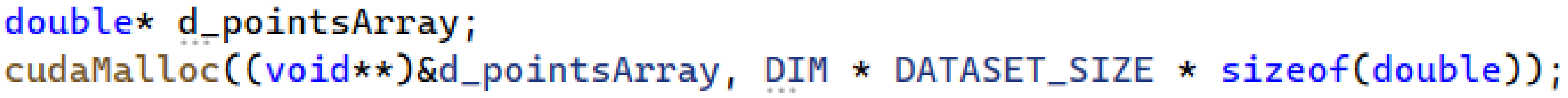
To prove this supposition we used another data structure, a 2D matrix flattened to an array.



X1	X2	···	···	XN
Y1	Y2	···	···	YN

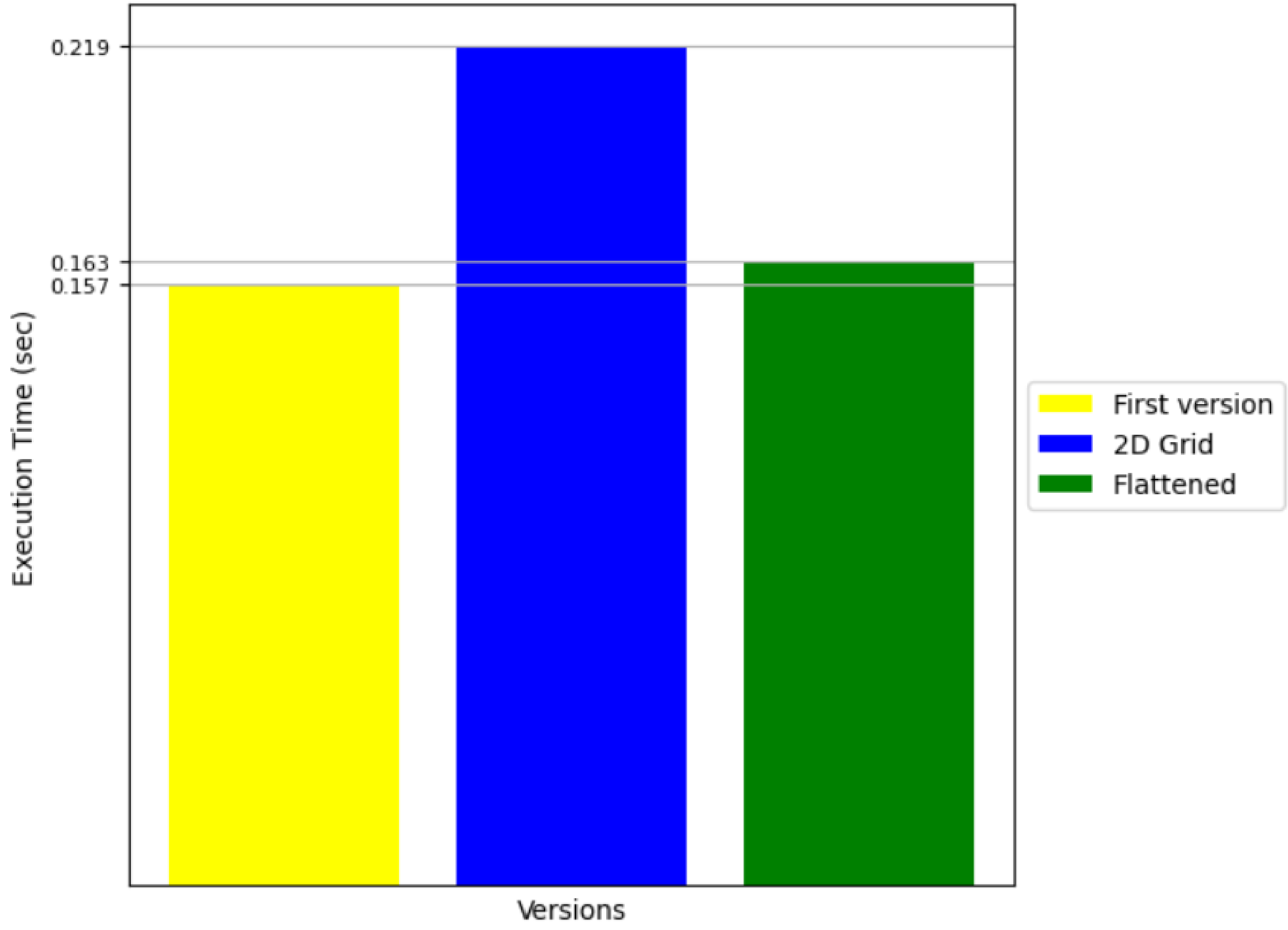
X1	X2	···	···	XN

OFFSET=DATASET_SIZE



```
double* d_pointsArray;
cudaMalloc((void**)&d_pointsArray, DIM * DATASET_SIZE * sizeof(double));
```

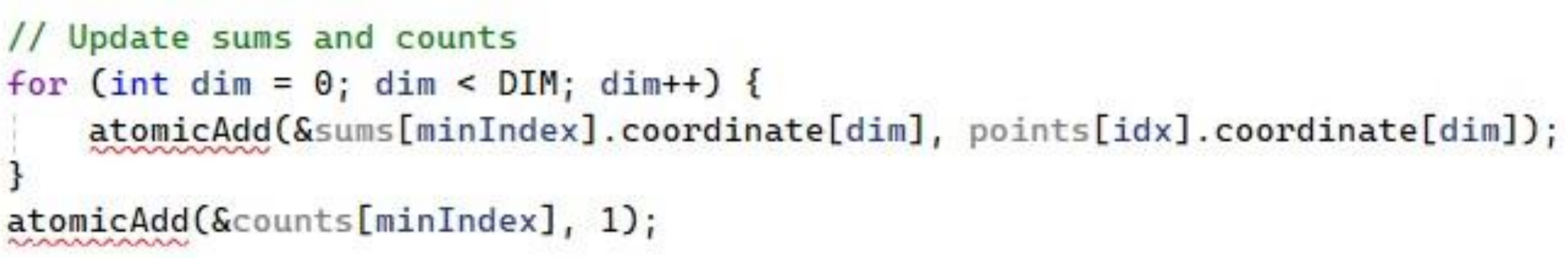
Comparison of execution time on 10 millions points between initial version,
2D Grid and Flattened version



GPU – Optimization

Reducing the number of calls on atomic operations: is there any way?

`assignPointsToClusterKernel`



```
// Update sums and counts
for (int dim = 0; dim < DIM; dim++) {
    atomicAdd(&sums[minIndex].coordinate[dim], points[idx].coordinate[dim]);
}
atomicAdd(&counts[minIndex], 1);
```

How many times? DATASET_SIZE x (DIM +1) →

In our case: DIM=2 and DATASET_SIZE = 10 000 000

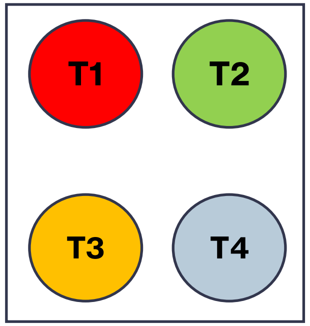
30 000 000 calls per iteration

Can we do better exploiting our GPU architecture?

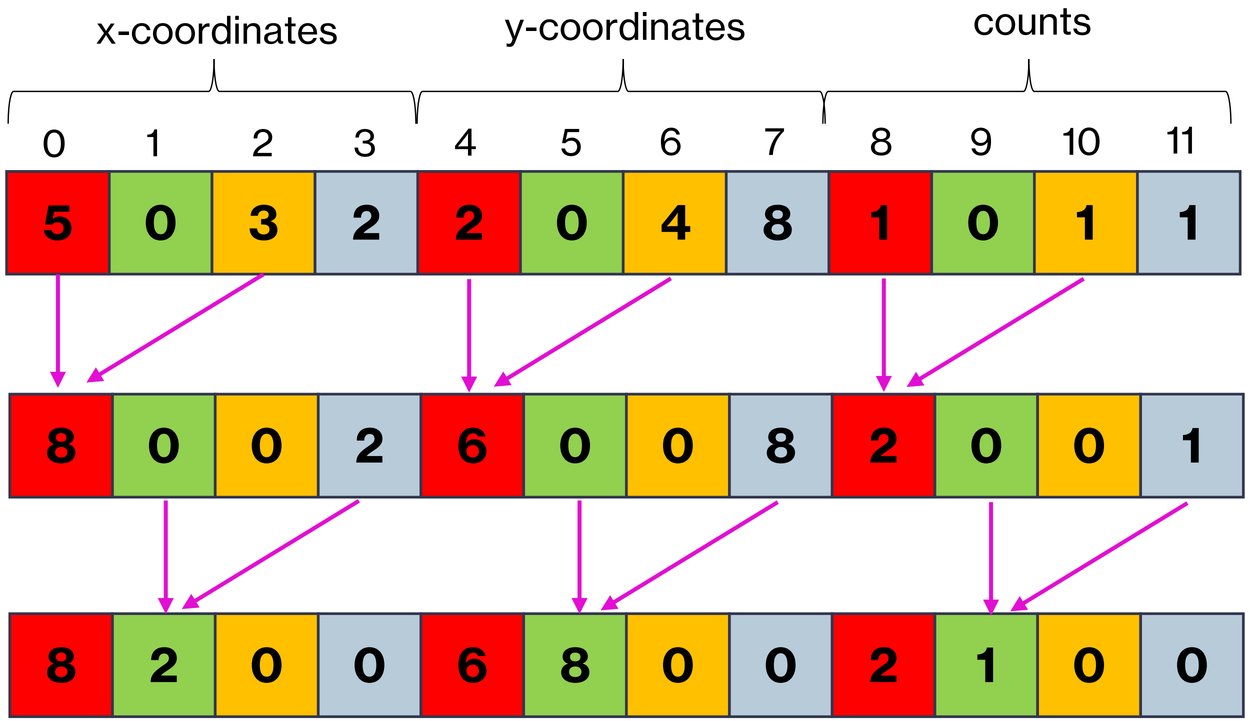
GPU – Optimization

Rewrite `assignPointsToClusterKernel` in order to perform for each cluster a PARALLEL - TREE – REDUCTION exploiting the shared memory in each block

Status of shared memory inside a block for i-cluster



Generic block



0	1	2	3	4	5	6	7	8	9	10	11
5	0	3	2	2	0	4	8	1	0	1	1
8	0	0	2	6	0	0	8	2	0	0	1
8	2	0	0	6	8	0	0	2	1	0	0

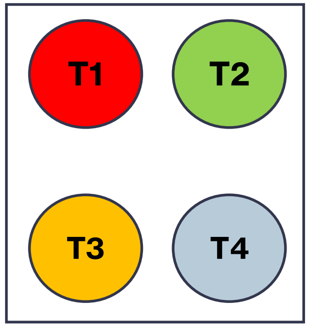
stride = blockDim.x/2

stride = stride /2

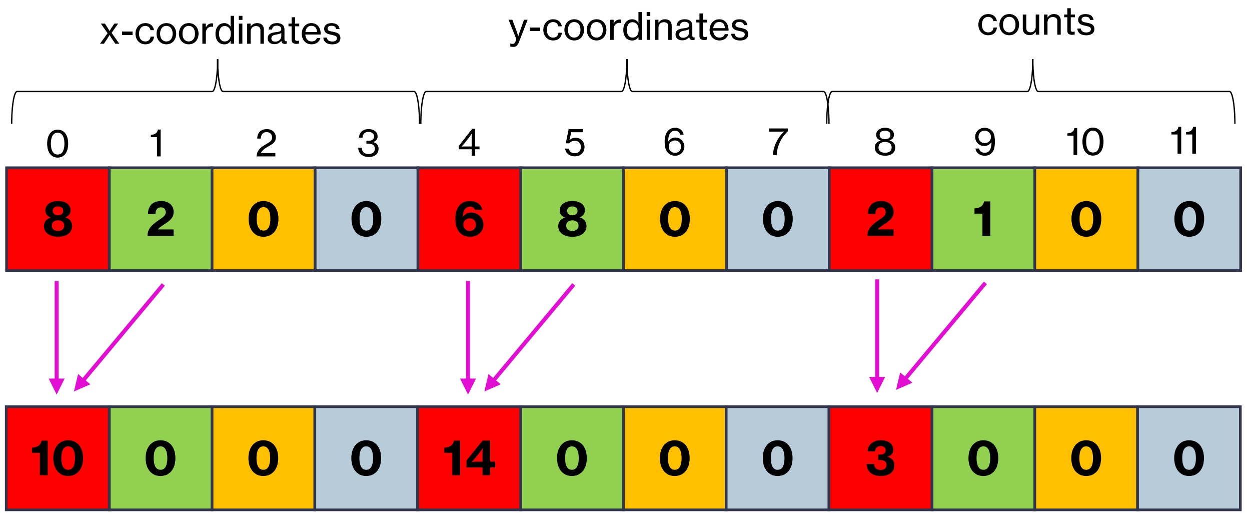
GPU – Optimization

Rewrite `assignPointsToClusterKernel` in order to perform for each cluster a PARALLEL - TREE – REDUCTION exploiting the shared memory of each block

Status of shared memory inside a block for i-cluster



Generic block



0	1	2	3	4	5	6	7	8	9	10	11
8	2	0	0	6	8	0	0	2	1	0	0
10	0	0	0	14	0	0	0	3	0	0	0

Now only T1 performs atomic opertations !!

A BLOCK NEVER HANDLES ONLY 4 THREADS!!

This is just an example: let see the impact on real data

GPU – Optimization

Reducing the number of calls on atomic operations

- Initial version: DATASET_SIZE x (DIM +1) → 30 000 000 calls per iteration
- Opt. version: NR_BLOCKS x (DIM +1) x K → ~ 150 000 calls per iteration

DIM=2
DATASET_SIZE = 10 000 000

NR_BLOCKS=9766
THREADS=1024
K(# of clusters)=5

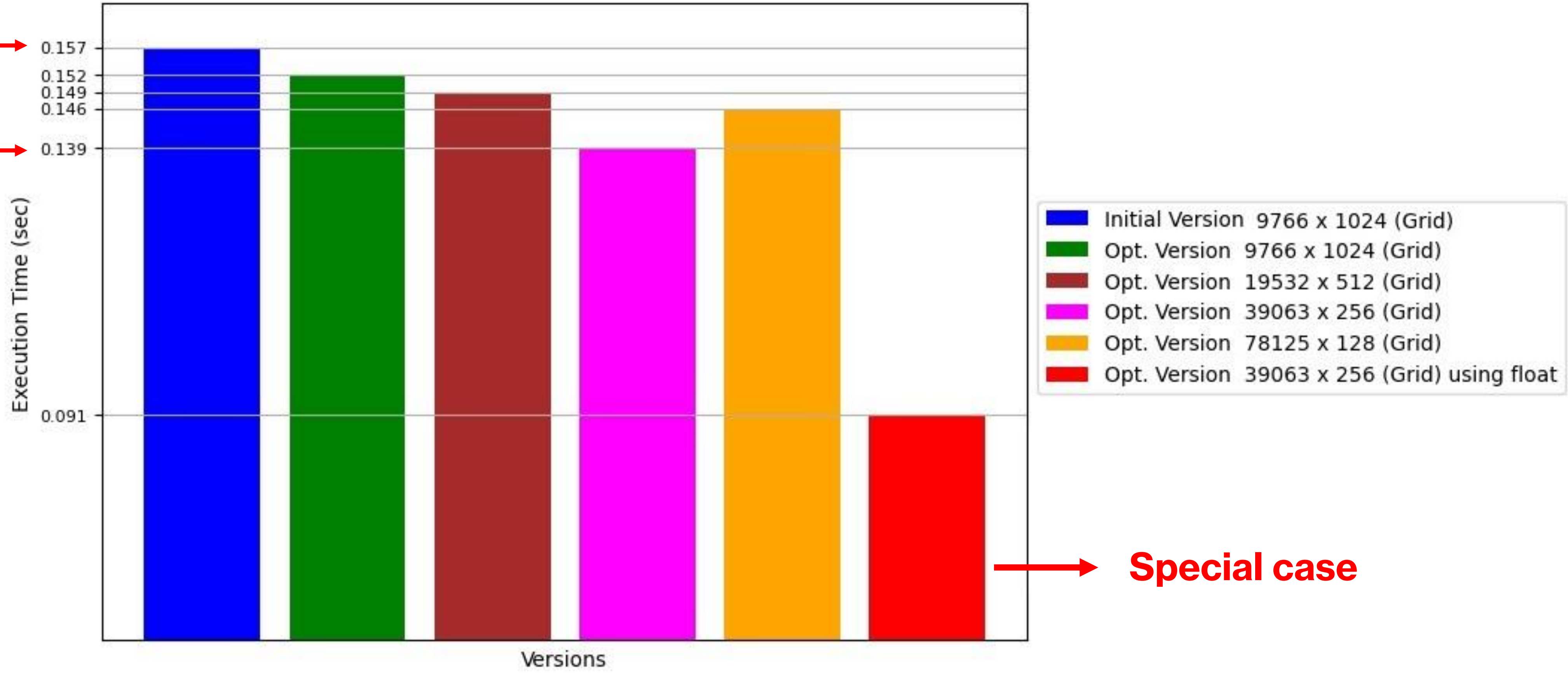
PAY ATTENTION: the opt. version introduces additional synchronizations among threads in the same block

```
__syncthreads();
```

We have to find a trade-off between # of blocks and # of threads

GPU – Optimization

-12%



Special case

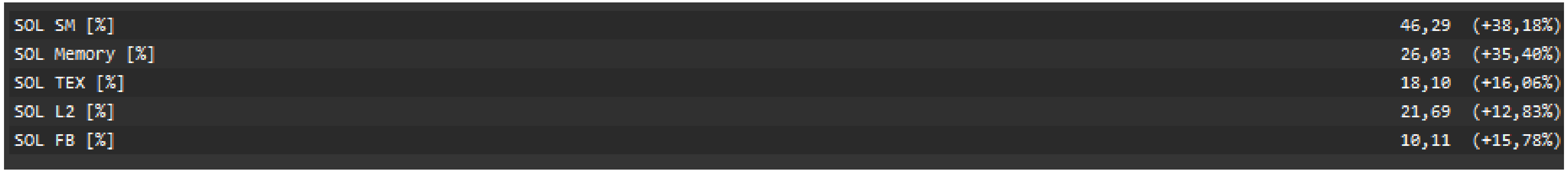
# Atomic Op.	30 M	150 K	300 K	600 K	1.2 M	300 K

GPU – Optimization

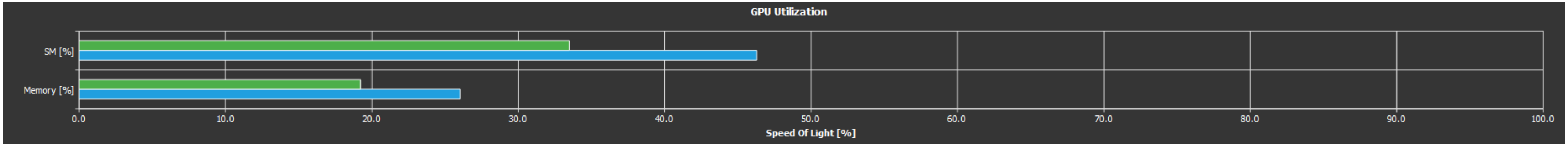
NVIDIA Nsight™ Compute

v.2019

🟢 Without parallel tree 🔵 With parallel tree



SOL SM [%]	46,29 (+38,18%)
SOL Memory [%]	26,03 (+35,40%)
SOL TEX [%]	18,10 (+16,06%)
SOL L2 [%]	21,69 (+12,83%)
SOL FB [%]	10,11 (+15,78%)



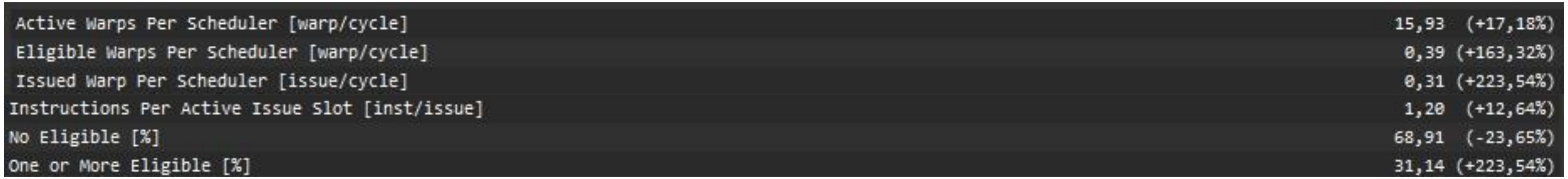
GPU – Optimization

NVIDIA Nsight™ Compute

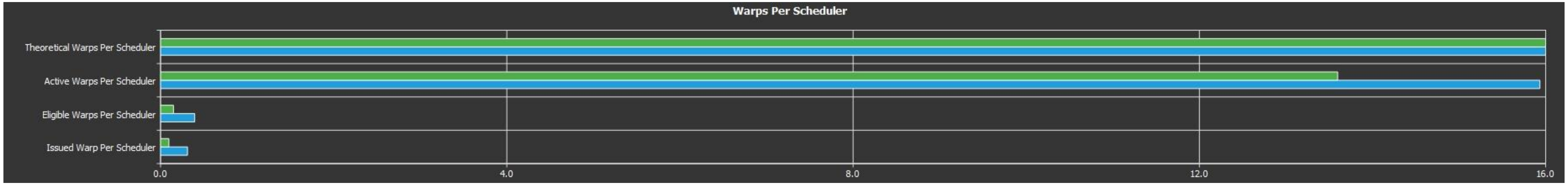
v.2019

🟢 Without parallel tree

🔵 With parallel tree



Active Warps Per Scheduler [warp/cycle]	15,93 (+17,18%)
Eligible Warps Per Scheduler [warp/cycle]	0,39 (+163,32%)
Issued Warp Per Scheduler [issue/cycle]	0,31 (+223,54%)
Instructions Per Active Issue Slot [inst/issue]	1,20 (+12,64%)
No Eligible [%]	68,91 (-23,65%)
One or More Eligible [%]	31,14 (+223,54%)



GPU – Optimization

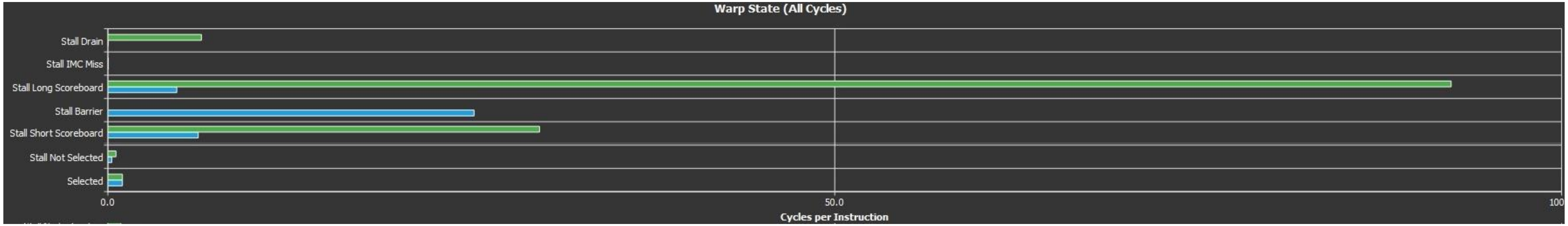
NVIDIA Nsight™ Compute

v.2019

```
__syncthreads();
```

● Without parallel tree

● With parallel tree



Optimizations summary

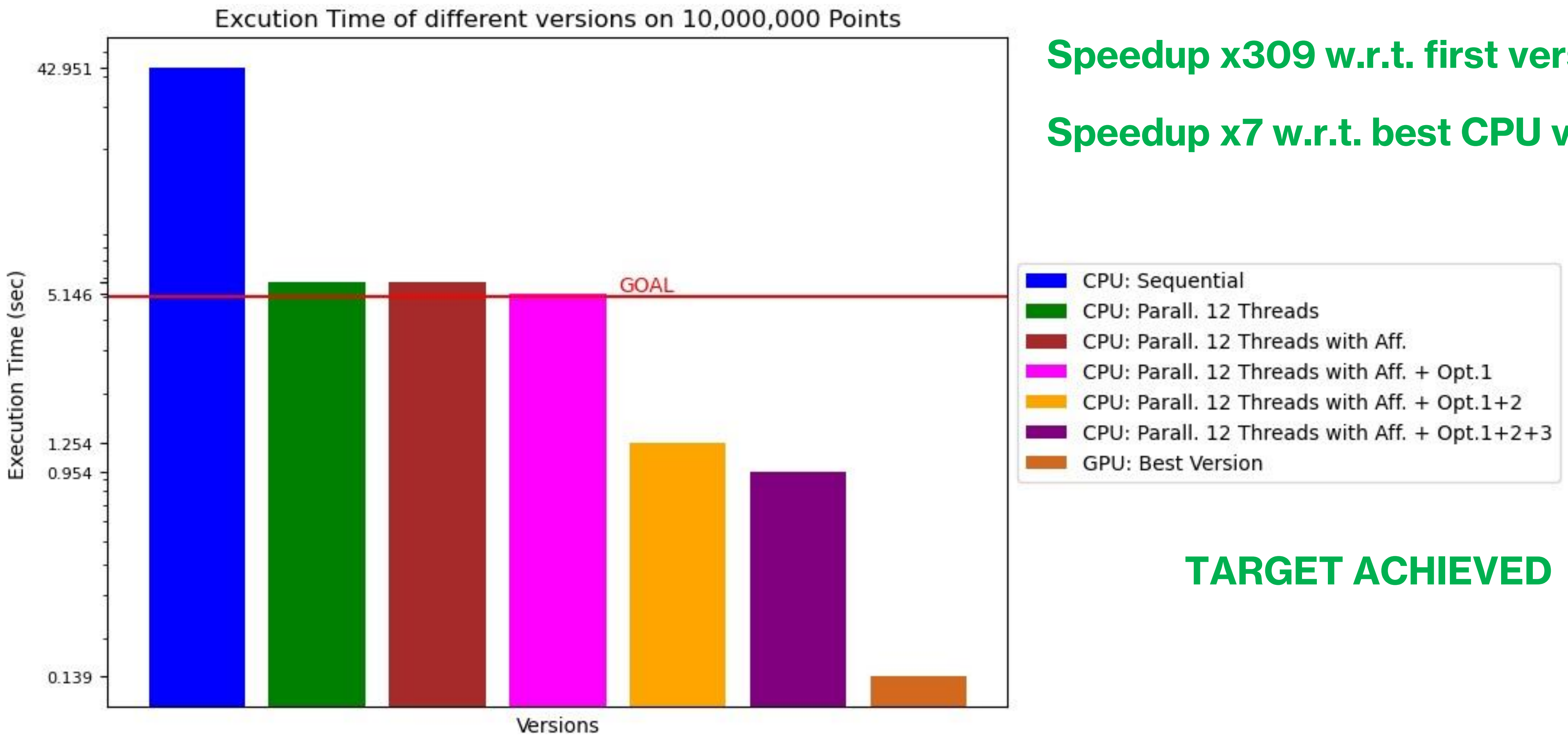
For CPU Parallel version:

1. From a dynamic list of points, to an array of points [Opt.1];
2. Switching from usage of pow() function, to executing directly the square of the number (N^2=N X N) [Opt.2];
3. Find the minimum distance between to points, without using the square root [Opt.3];

For GPU version:

1. Usage of parallel tree reduction technique;

Conclusions



Speedup x309 w.r.t. first version

Speedup x7 w.r.t. best CPU version

TARGET ACHIEVED

Thank you