MSc Computer Engineering
Computer Architectures

Drastic Event Simulator

Mirco Concu
Edoardo Pantè
Antonino Nigro

Anno accademico 2022/23

1. Introduction

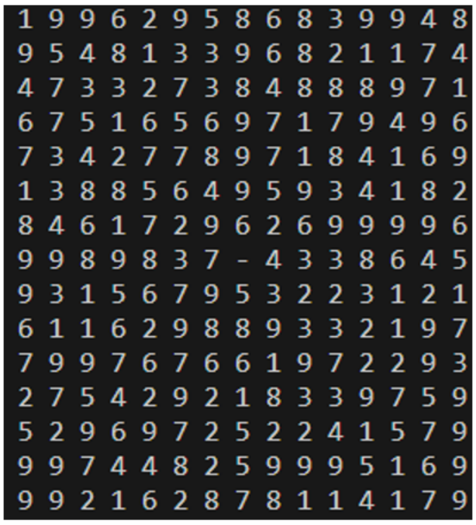
A Drastic Event Simulator (DES) is a program that computes the damage performed on an area after a catastrophic event has happened. The area was schematized as a matrix where each element represents a single block of the area and that has assigned a value going from zero to nine. This value indicates the grade of resistivity of the single piece of the area. To function properly the program need some input by the users such as:

- Matrix size
- Event power
- Coordinates of the event
- Number of threads
- Other debugging settings

The formula used to compute the devastation was as follows:

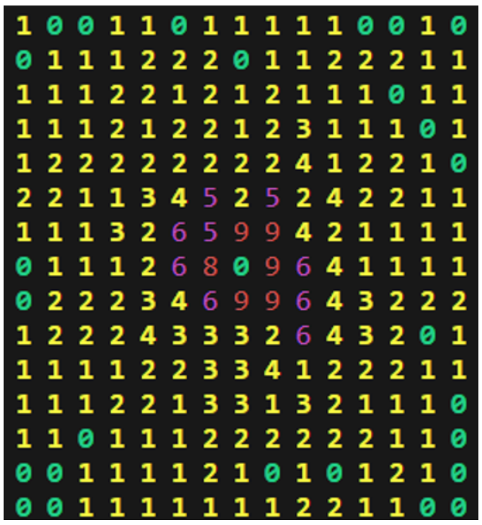
$$\frac{power - resistivity}{distance}$$

Where power is the event power, resistivity is the single element of the matrix value and distance is the distance from the epicenter of the event. The program returns as output a matrix that shows the damage on the area. We can observe in the picture below an example of an input and output matrix.



```
1 9 9 6 2 9 5 8 6 8 3 9 9 4 8
9 5 4 8 1 3 3 9 6 8 2 1 1 7 4
4 7 3 3 2 7 3 8 4 8 8 8 9 7 1
6 7 5 1 6 5 6 9 7 1 7 9 4 9 6
7 3 4 2 7 7 8 9 7 1 8 4 1 6 9
1 3 8 8 5 6 4 9 5 9 3 4 1 8 2
8 4 6 1 7 2 9 6 2 6 9 9 9 9 6
9 9 8 9 8 3 7 - 4 3 3 8 6 4 5
9 3 1 5 6 7 9 5 3 2 2 3 1 2 1
6 1 1 6 2 9 8 8 9 3 3 2 1 9 7
7 9 9 7 6 7 6 6 1 9 7 2 2 9 3
2 7 5 4 2 9 2 1 8 3 3 9 7 5 9
5 2 9 6 9 7 2 5 2 2 4 1 5 7 9
9 9 7 4 4 8 2 5 9 9 9 5 1 6 9
9 9 2 1 6 2 8 7 8 1 1 4 1 7 9
```

Before the event



```
1 0 0 1 1 0 1 1 1 1 1 0 0 1 0
0 1 1 1 2 2 2 0 1 1 2 2 2 1 1
1 1 1 2 2 1 2 1 2 1 1 1 0 1 1
1 1 1 2 1 2 2 1 2 3 1 1 1 0 1
1 2 2 2 2 2 2 2 2 4 1 2 2 1 0
2 2 1 1 3 4 5 2 5 2 4 2 2 1 1
1 1 1 3 2 6 5 9 9 4 2 1 1 1 1
0 1 1 1 2 6 8 0 9 6 4 1 1 1 1
0 2 2 2 3 4 6 9 9 6 4 3 2 2 2
1 2 2 2 4 3 3 3 2 6 4 3 2 0 1
1 1 1 1 2 2 3 3 4 1 2 2 2 1 1
1 1 1 2 2 1 3 3 1 3 2 1 1 1 0
1 1 0 1 1 1 2 2 2 2 2 2 1 1 0
0 0 1 1 1 1 2 1 0 1 0 1 2 1 0
0 0 1 1 1 1 1 1 1 2 2 1 1 0 0
```

Damage found

2. CPU implementation

We have four different types of implementations regarding the CPU, in particular we have:

- des_mono
- des_column
- des
- des_sequentialized

In order to have consistent values all the parameters described in the previous paragraph are kept constant between the different implementations.

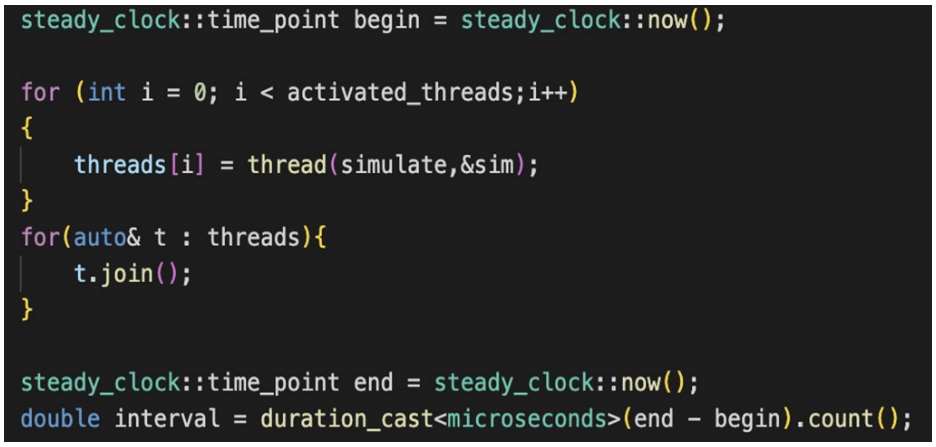
2.1 Hardware specifications

For this project we have adopted the M1 Pro processor which is Apple proprietary hardware that presents an ARM architecture. In particular the processor has the following specifics:

- 8 Core - 8 Thread
  - 6 performance @ 3.22 GHz
  - 2 efficiency @ 2.06 GHz
- Performance Core Cache
  - L1i - 192 KB
  - L1d - 128 KB
  - L2 Shared - 28 MB
  - L3 - 16 MB
- Efficiency Core Cache
  - L1i - 128 KB
  - L1d - 64 KB
  - L2 Shared - 4 MB
  - L3 - 14 MB

2.2 Code specifications

To develop the code it was used the C++ programming language. In the picture below is shown how the execution time was measured.



```
steady_clock::time_point begin = steady_clock::now();

for (int i = 0; i < activated_threads;i++)
{
    threads[i] = thread(simulate,&sim);
}
for(auto& t : threads){
    t.join();
}

steady_clock::time_point end = steady_clock::now();
double interval = duration_cast<microseconds>(end - begin).count();
```

2.3 Analysis

The first implementation that was performed is ***des_mono***. In this implementation every thread gets one single element of the matrix and performs the computation.

The performance was basically bad and increasing threads we had no speed up but a consistent speed-down. This is due to the fact that every thread is working on a single element that is 2 bytes long and when each of them has finished and wants to store the results, they are basically writing the same cache line, usually 64 bytes long, and so they must wait for previous threads. This behavior leads to an increase in the execution time because of all cache misses that are provoked by the conflict on the same cache line by multiple threads. So not only are we not using the cache, but it is slowing down the performance.

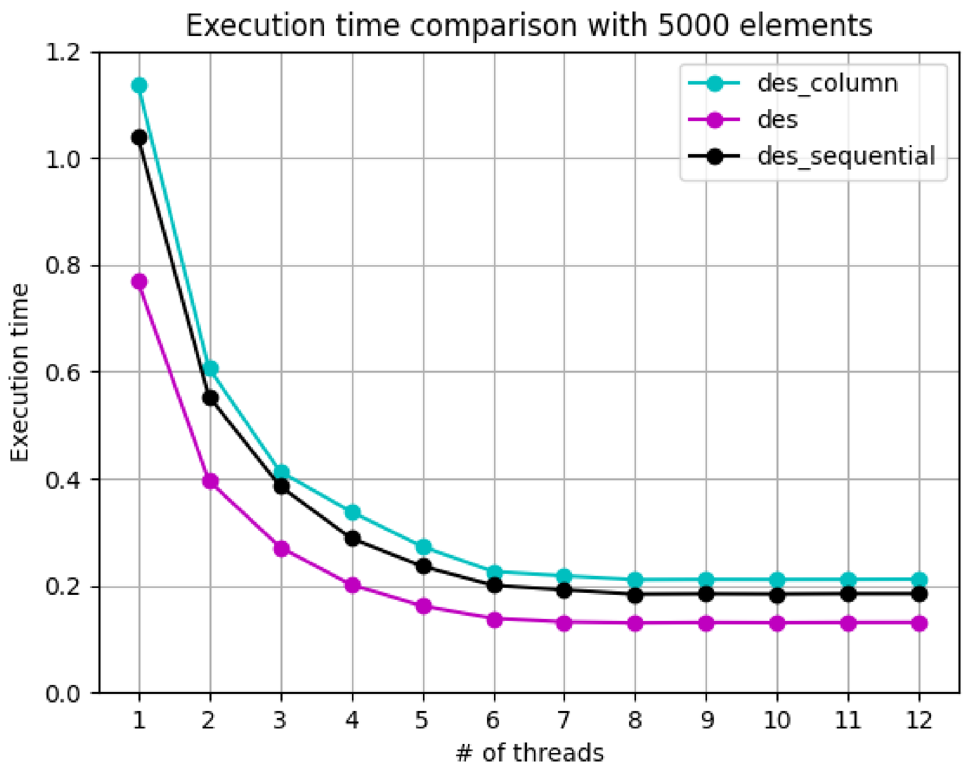
For this reason we decided to test two different alternatives: ***des_column*** and ***des***. Talking about ***des_column***, in this implementation every thread gets a set of elements of the matrix, that can be a row or a column. In particular every thread gets a column of the matrix. Here we found out that, even in this case, where we get a set of elements of the matrix these elements are far from each other in memory and so, each element must be loaded in cache singularly instead of being loaded all together as expected in a normal program. This shows us that even the order of accessing data can change the performance of the application.

With this in mind we moved to our third implementation (***des***) where every thread gets an entire single row of the matrix. Here we have encountered the best performance possible because we have seen that the speed up scaled with the number of physical threads, reaching his peak using 8 threads. This is caused by the optimal use of the Spatial Locality and so the full use of the cache that has improved the performance to the best.

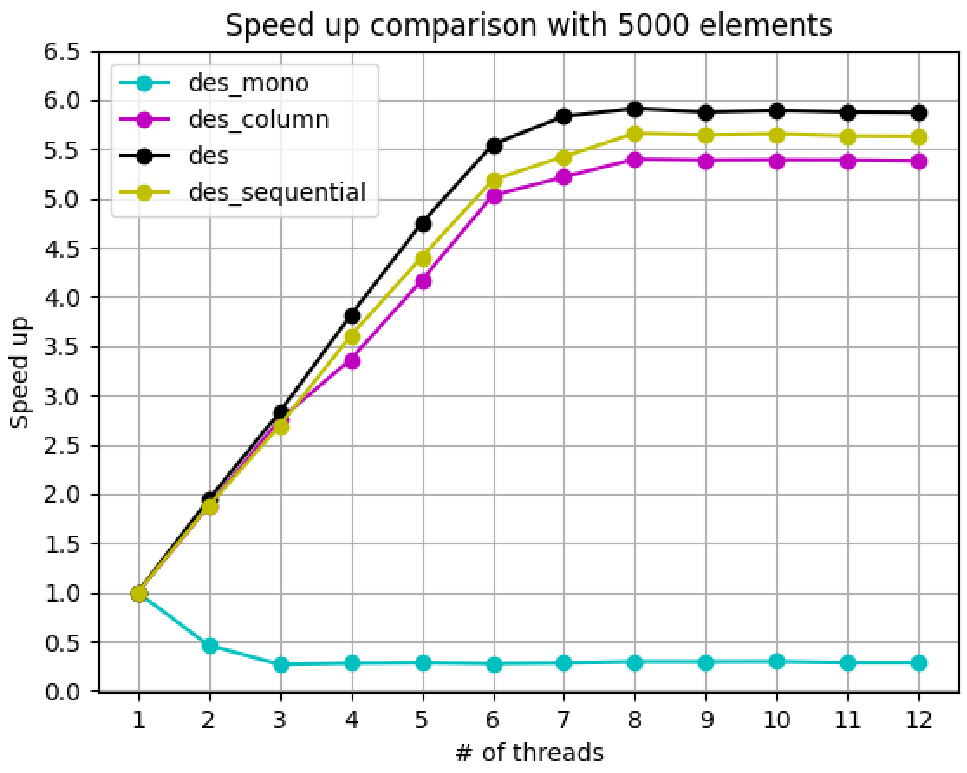
Here is where the idea of ***des_sequentialized*** was born. This implementation takes the elements of the columns of the matrix and moves them into a new matrix where these columns become lines (in a certain way we are performing a transpons). At this point the behavior is the same as the des implementation but as the last step is performed the reverse of the transpons so in this way we have returned to the original matrix. The improvement consists in the fact that we are accessing the data in a columnar way but then we work in a copy in which is used the ***des*** implementation, and so even if we have a copy, we have a speed up and a general improvement of performance. This is very useful to show that this algorithm can be exported to be used in different ways, even not the one we propose.

Another important thing to note is that, taking for example the ***des*** speed-up curve, we can observe that we have a constant increase in the speed-up till the sixth thread (we have six performance cores) and then a slightly less steep increase till the eighth thread (this due to the two efficiency cores). This goes to demonstrate that all cores in the processor are employed (taking into account that this processor has one thread per core).

In the graphs below we can see all the different speedups and execution times of the single implementations. Is clear that des is the best case scenario but comparing that to the des_sequentialized we have slightly worse performance than the best scenario but still acceptable.



Execution time comparison with 5000 elements
Execution time
# of threads
des_column
des
des_sequential
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
5
6
7
8
9
10
11
12



Speed up comparison with 5000 elements
Speed up
# of threads
des_mono
des_column
des
des_sequential
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6
7
8
9
10
11
12

2.4 Power consumption

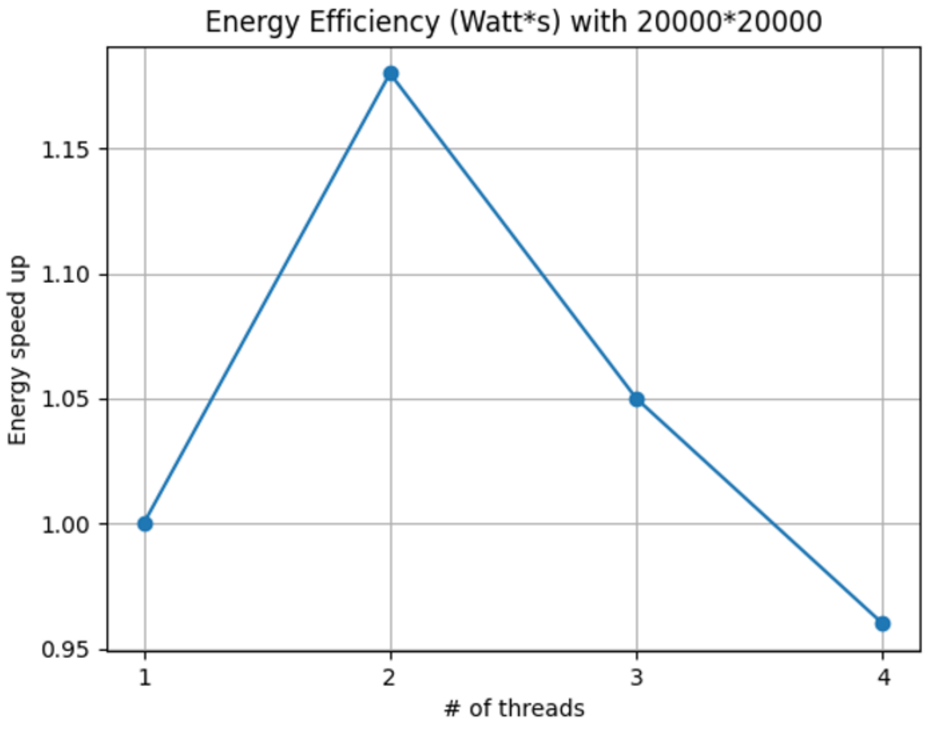
Unfortunately we weren't able to analyze the power consumption on the M1 Pro processor because of Apple proprietary hardware. For this reason we analyzed the power consumption on a different processor, in particular on an Intel i5-4790K with 4 Core. The programs used to perform the analysis were mainly two:

- CoreTemp 1.18
- HWMonitor

We used two of them at different times so we could be sure that the results were correct. These programs read an internal register that provides information for the entire packet. We also used WSL2 instead of a pure Linux machine. In the following graph we can see that we have the peak of energy efficiency using only two threads of the four available. We could not verify exactly with a fine instrument like Perf, due to the implementation of WSL that hides the real behavior of the CPU, but we are pretty sure that this is caused by the Hit/Miss rate cache. We know that the element that requires energy is the L2 cache memory, in particular during the recovery of a Miss action. In fact, we evaluate the energy efficiency by running the application in background and sampling the temperature every second, and then compute the average for each configuration. Then we divided the average for the running time, so we could compare the results.

$$Energy\ consumption\ =\ Average\ Energy\ /\ Running\ time$$

The difference is little, but we could see that with more threads we had less time, proportional with the number of physical cores used, but the energy consumption has increased not linearly, that’s why we have the worst energy efficiency using all four physical cores.



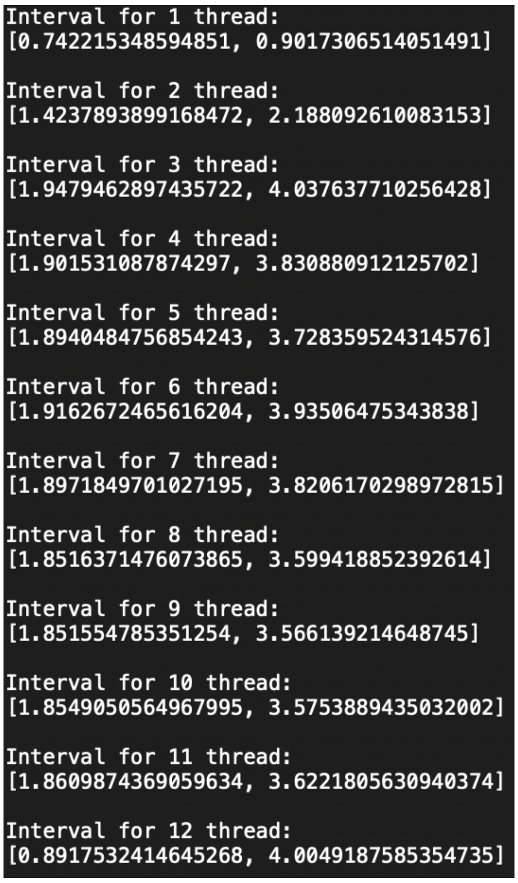
2.5 Confidence interval

By implementing a python script and using the below formulas we were able to determine the confidence intervals of all the implementations.

$$\overline{X} = \frac{1}{n} \cdot \sum_{i=1}^{n} X_i \qquad S^2 = \frac{\sum_{i=1}^{n} \left(X_i - \overline{X}\right)^2}{n-1}$$

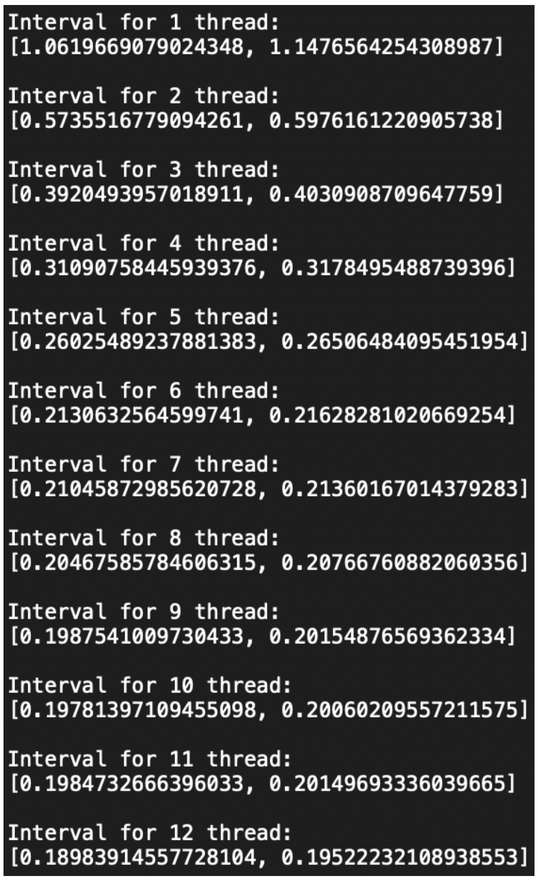
$$\left[\overline{X} - \frac{S}{\sqrt{n}} \cdot z_{\alpha/2}, \overline{X} + \frac{S}{\sqrt{n}} \cdot z_{\alpha/2}\right]$$

The pictures below show the different implementations and the resulting intervals.



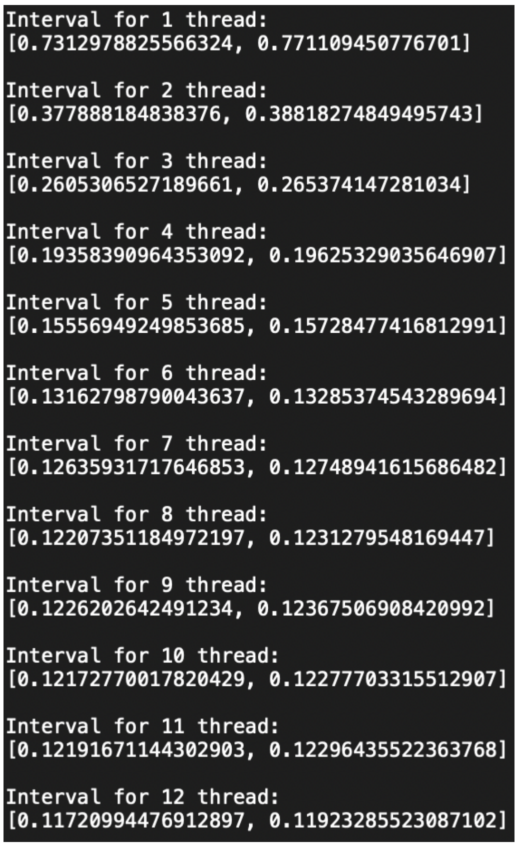
```
Interval for 1 thread:
[0.742215348594851, 0.9017306514051491]

Interval for 2 thread:
[1.4237893899168472, 2.188092610083153]

Interval for 3 thread:
[1.9479462897435722, 4.037637710256428]

Interval for 4 thread:
[1.901531087874297, 3.830880912125702]

Interval for 5 thread:
[1.8940484756854243, 3.728359524314576]

Interval for 6 thread:
[1.9162672465616204, 3.93506475343838]

Interval for 7 thread:
[1.8971849701027195, 3.8206170298972815]

Interval for 8 thread:
[1.8516371476073865, 3.599418852392614]

Interval for 9 thread:
[1.851554785351254, 3.566139214648745]

Interval for 10 thread:
[1.8549050564967995, 3.5753889435032002]

Interval for 11 thread:
[1.8609874369059634, 3.6221805630940374]

Interval for 12 thread:
[0.8917532414645268, 4.0049187585354735]
```

des_mono *intervals*



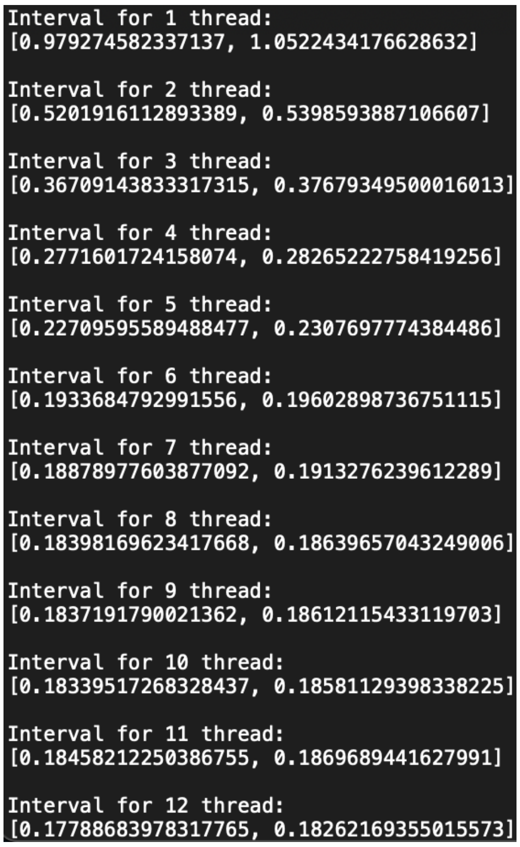
```
Interval for 1 thread:
[1.0619669079024348, 1.1476564254308987]

Interval for 2 thread:
[0.5735516779094261, 0.5976161220905738]

Interval for 3 thread:
[0.3920493957018911, 0.4030908709647759]

Interval for 4 thread:
[0.31090758445939376, 0.3178495488739396]

Interval for 5 thread:
[0.26025489237881383, 0.26506484095451954]

Interval for 6 thread:
[0.2130632564599741, 0.2162828102066925]

Interval for 7 thread:
[0.21045872985620728, 0.21360167014379283]

Interval for 8 thread:
[0.20467585784606315, 0.20766760882060356]

Interval for 9 thread:
[0.1987541009730433, 0.20154876569362334]

Interval for 10 thread:
[0.19781397109455098, 0.20060209557211575]

Interval for 11 thread:
[0.1984732666396033, 0.20149693336039665]

Interval for 12 thread:
[0.18983914557728104, 0.19522232108938553]
```

des_column *intervals*



```
Interval for 1 thread:
[0.7312978825566324, 0.771109450776701]

Interval for 2 thread:
[0.377888184838376, 0.38818274849495743]

Interval for 3 thread:
[0.2605306527189661, 0.265374147281034]

Interval for 4 thread:
[0.19358390964353092, 0.19625329035646907]

Interval for 5 thread:
[0.15556949249853685, 0.15728477416812991]

Interval for 6 thread:
[0.13162798790043637, 0.13285374543289694]

Interval for 7 thread:
[0.12635931717646853, 0.12748941615686482]

Interval for 8 thread:
[0.12207351184972197, 0.1231279548169447]

Interval for 9 thread:
[0.1226202642491234, 0.12367506908420992]

Interval for 10 thread:
[0.12172770017820429, 0.12277703315512907]

Interval for 11 thread:
[0.12191671144302903, 0.12296435522363768]

Interval for 12 thread:
[0.11720994476912897, 0.11923285523087102]
```

***des** intervals*



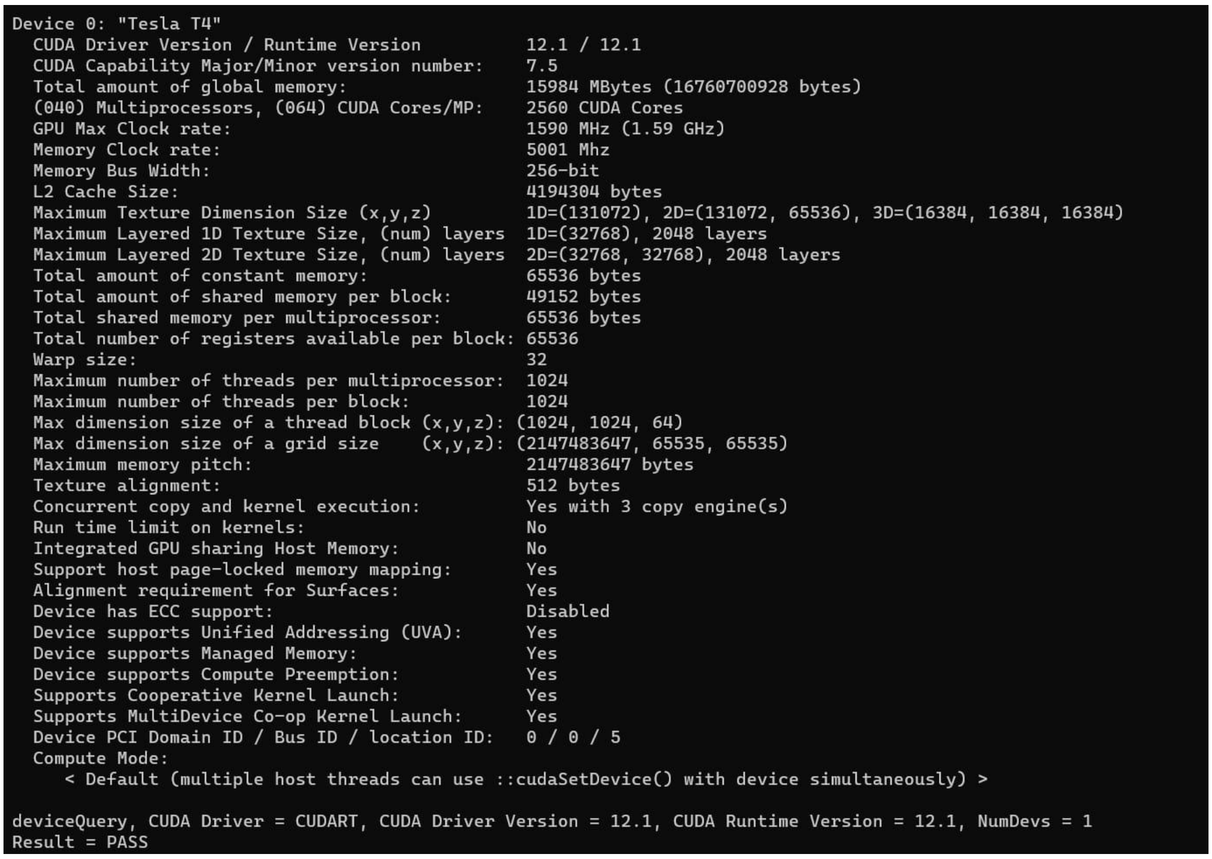
```
Interval for 1 thread:
[0.979274582337137, 1.0522434176628632]

Interval for 2 thread:
[0.5201916112893389, 0.5398593887106607]

Interval for 3 thread:
[0.36709143833317315, 0.37679349500016013]

Interval for 4 thread:
[0.2771601724158074, 0.28265222758419256]

Interval for 5 thread:
[0.22709595589488477, 0.2307697774384486]

Interval for 6 thread:
[0.1933684792991556, 0.19602898736751115]

Interval for 7 thread:
[0.18878977603877092, 0.1913276239612289]

Interval for 8 thread:
[0.18398169623417668, 0.18639657043249006]

Interval for 9 thread:
[0.1837191790021362, 0.18612115433119703]

Interval for 10 thread:
[0.18339517268328437, 0.18581129398338225]

Interval for 11 thread:
[0.18458212250386755, 0.1869689441627991]

Interval for 12 thread:
[0.17788683978317765, 0.18262169355015573]
```

***des_sequential** intervals*

3. GPU implementation

3.1 Hardware specifications

For this part we have adopted the Nvidia Tesla T4 GPU that was kindly granted to us by the University of Pisa. In particular the GPU has the following specifics:

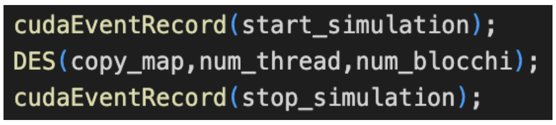


```
Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          12.1 / 12.1
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15984 MBytes (16760700928 bytes)
  (040) Multiprocessors, (064) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 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:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  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
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 5
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.1, CUDA Runtime Version = 12.1, NumDevs = 1
Result = PASS
```

3.2 Code specifications

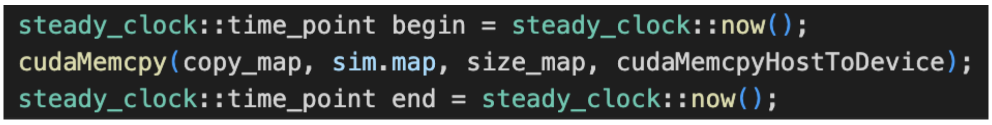
We used CUDA C++ language to develop the code. To calculate the execution time to carry out our analyses we used the CudaEvents, which allowed us to evaluate only the actual execution time of the GPU not taking in consideration the work on the CPU. We took in consideration three different times that are:

- ***Total execution time:*** take in consideration the whole elapsed time
- ***Copy time:*** time elapsed to copy the data from the CPU to the GPU
- ***Simulation time:*** consider only the time to perform the simulation



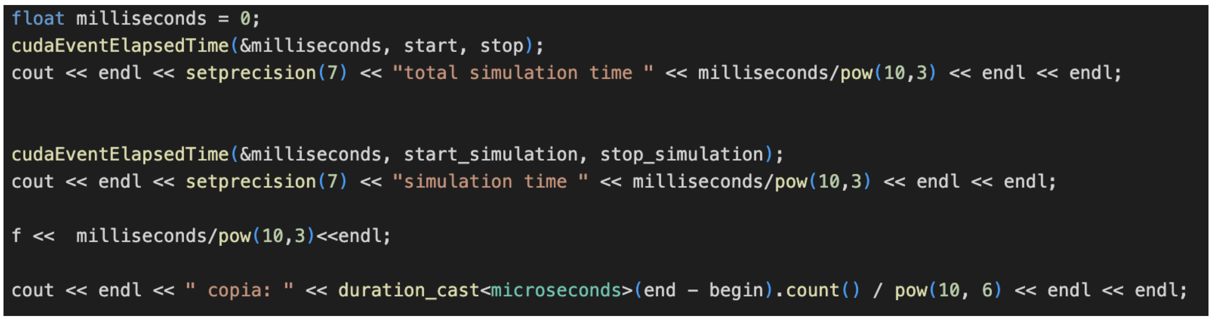
```
cudaEventRecord(start_simulation);
DES(copy_map,num_thread,num_blocchi);
cudaEventRecord(stop_simulation);
```

*Screenshot of the **simulation time**.*



```
steady_clock::time_point begin = steady_clock::now();
cudaMemcpy(copy_map, sim.map, size_map, cudaMemcpyHostToDevice);
steady_clock::time_point end = steady_clock::now();
```

*Screenshot of the **copy time**.*

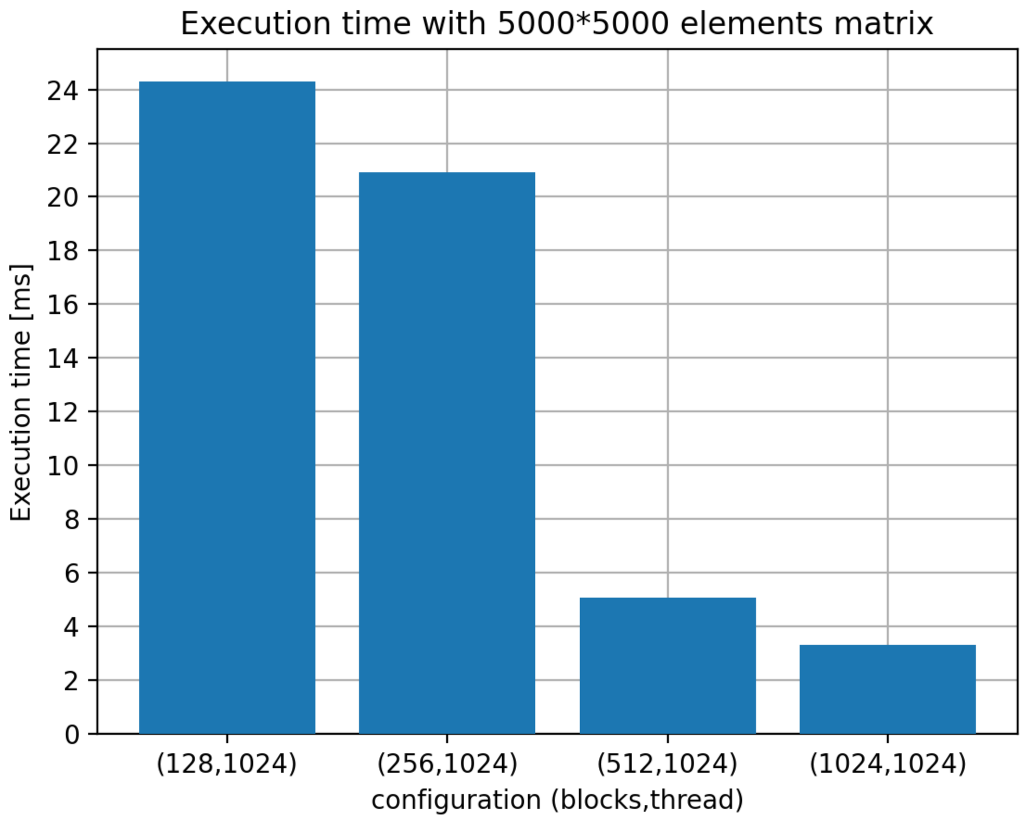


```
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
cout << endl << setprecision(7) << "total simulation time " << milliseconds/pow(10,3) << endl << endl;


cudaEventElapsedTime(&milliseconds, start_simulation, stop_simulation);
cout << endl << setprecision(7) << "simulation time " << milliseconds/pow(10,3) << endl << endl;

f <<  milliseconds/pow(10,3)<<endl;

cout << endl << " copia: " << duration_cast<microseconds>(end - begin).count() / pow(10, 6) << endl << endl;
```

*Screenshot of the evaluation of the **different execution times**.*

3.3 GPU analysis

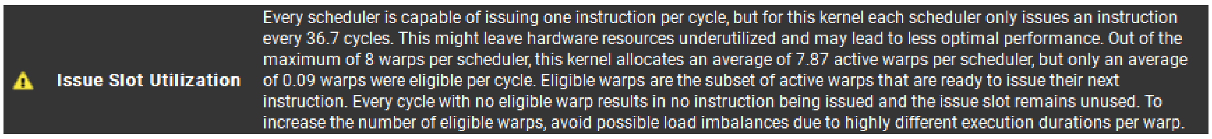
The algorithm implemented for the GPU is implemented differently from the CPU's one to maximize the performance. In this one since, we have a larger number of threads that can be run and a completely different architecture.
The choice we made was to assign to each thread an equal number of elements.
The analysis we did was to launch the algorithm with a different number of threads and blocks. By setting the the value of threads and varying the blocks in a set of {128, 256 … 1024} we gain the result given in the following graph:



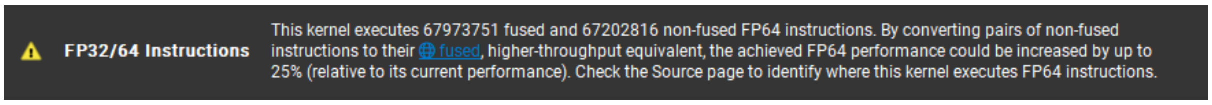
As we can use more blocks we improve the performance but from (512,1024) the performance saturates. This is due to the saturation of the pipelines that are fully used and the architecture cannot provide more performance increasing the number of threads. To verify this behavior, we used the Nsight System. NVIDIA Nsight Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, help you identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs. The first thing that was reported was the high compute throughput: 85%. This means that we are using pretty much almost all that the device can give:

Compute (SM) Throughput [%]	85
Memory Throughput [%]	0,28

On the contrary, memory throughput is very low but it's in the algorithm: each thread accesses an element at time or a group of elements but they don't need synchronization, there is no interference between them and so there is no latency for memory conflict. Using shared memory gives no advantages with this algorithm but it can be modeled to be more complex and maybe with some implementation it can be more memory bound. Another message given by the profiler was that every 8 schedulers could issue an average of 7.8 active warps but only 0.9 were eligible per cycle. This means that every scheduler could issue only 1 instruction per 36 cycles, leading to bad performance and likely to be a bottleneck. Looking further in the reports from Nsight profiler, we saw that even if we had a very high Throughput, we was executing only about 40 instruction per cycles, and in the meantime no instruction is issued compromising the performance, and so the profiler inadvises us to try to use the equivalent fused instruction to improve performance and so we did.


⚠ **Issue Slot Utilization**

Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 36.7 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 8 warps per scheduler, this kernel allocates an average of 7.87 active warps per scheduler, but only an average of 0.09 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, avoid possible load imbalances due to highly different execution durations per warp.


⚠ **FP32/64 Instructions**

This kernel executes 67973751 fused and 67202816 non-fused FP64 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP64 performance could be increased by up to 25% (relative to its current performance). Check the Source page to identify where this kernel executes FP64 instructions.

Basically, the bottleneck was using non-fused operations that are very slow and more precise than we need. The profiler advises us to try to reduce the data size for a better cache L1 optimization but we couldn't because the algorithm could not work correctly.
So, we translate all the main operations on the kernel with the fused equivalent and obtain a huge improvement. It will be covered in the following paragraph.

3.5 CPU-GPU comparison

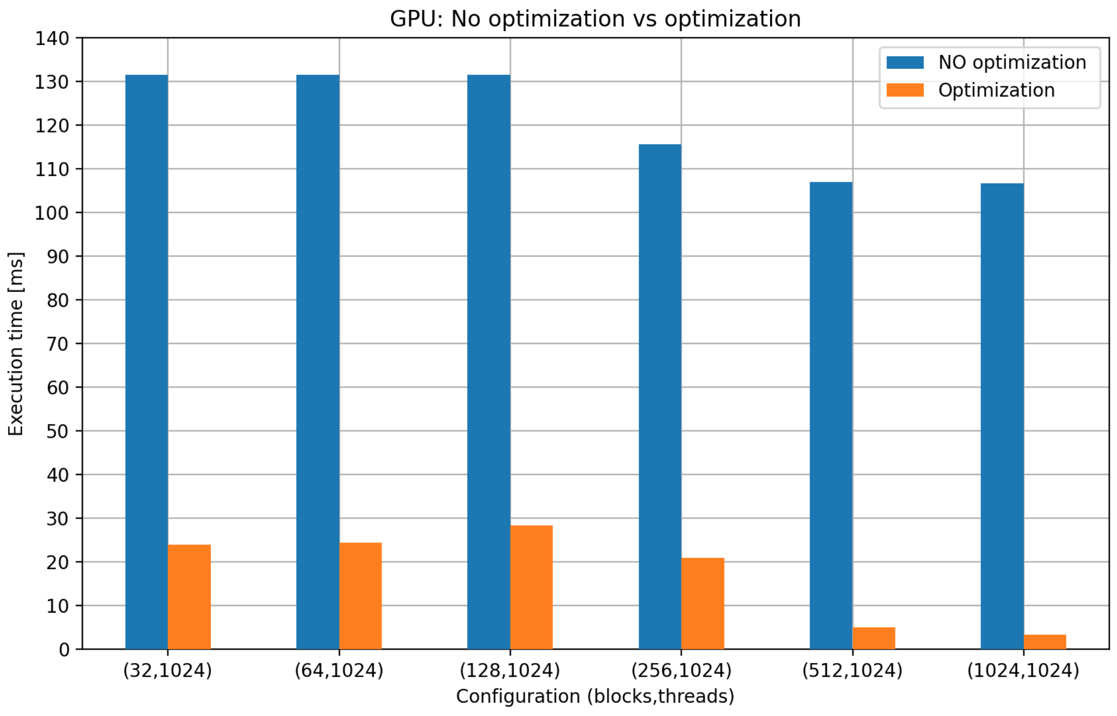
As shown before, the GPU version is about only 20% faster than the CPU version using the smallest matrix as input (5000x5000). The GPU version can be 1.2 up to 2 times faster than the CPU version. This result is interesting, in fact we know that a GPU is a device that can use a very complex and powerful architecture to speed-up the execution and can use its own global memory that is way faster than the main memory composed by the RAM. The GPU has more memory and way more available threads than the CPU but it is not so much faster. As shown in the previous paragraph, the profiler indicates that we reach about 85% of the maximum theoretical throughput, so we basically reached the limit and the only way to go further was to try to use different data sizes, but that was not possible on this algorithm. A reason for this result is that the CPU architecture is optimized for running sequential code and this architecture is pushed to the limit to achieve the maximum number of instructions per seconds. The GPU is optimized for running multiple threads at once, but it cannot keep up with the speed of the CPU cores. The CPU core has a frequency that is double than the GPU frequency, about 3 Ghz for CPU against 1.5 Ghz for the GPU. This difference in speed can explain why the CPU can be so fast, every single core of the CPU can execute a very high number of instructions while the GPU is slower, but can compensate with the number of threads that can execute. This compensation increases using a very big matrix. Another aspect to be considered is that the algorithm is not memory bound, so having a bigger and faster memory is not a big deal and is not a big advantage for the GPU. So, as last, the GPU is always faster than the CPU, but not so much.

3.5 CPU-GPU comparison (GPU optimized)

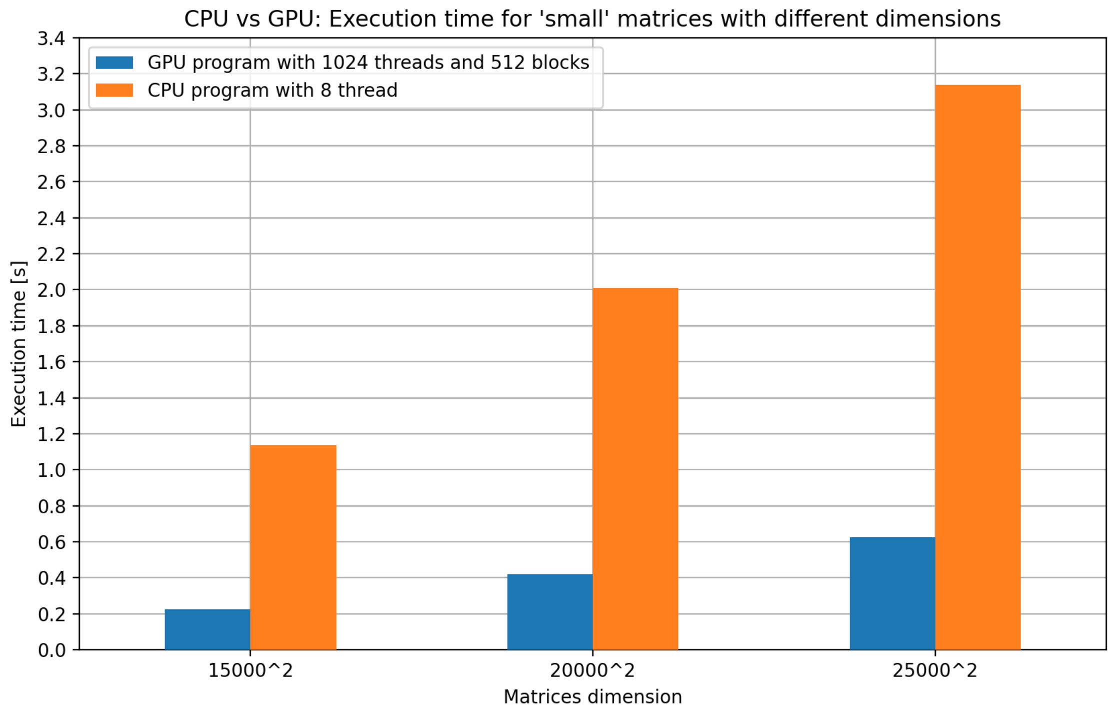
We have seen previously that the CPU has better performance on executing instruction per cycle, and so we used the fused equivalent instruction, as suggested by the profiler, to increase the instruction per cycle. What we have done is presented in the following table:

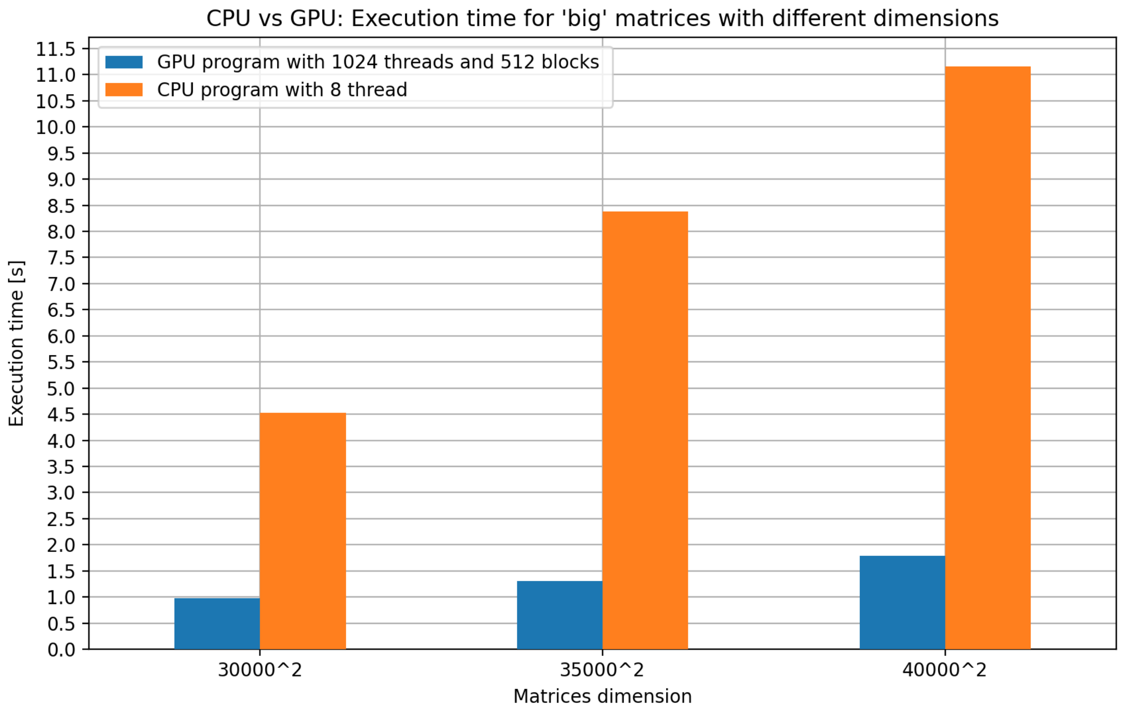
Non fused instruction	**Fused instruction**
*(x*x)+y*	*__fmaf_rn(x,x,y)*
x-y	*__fsub_rn(x,y)*
*x*x*	*__powf(x,2)*

With these modifications, we had an average speed-up of 15 from the base case. This result is obtained because the bottleneck of the GPU's ICP was very high as seen in the profiler. Using less precise and more fast instructions permits us to remove this bottleneck and to have more instructions executed per cycle and the same Throughput.
In the above graph we can see a comparison between the version non optimized and the one optimized.



In the following the graph we can see a comparison in execution time between the program ran by the CPU and the one ran by the GPU for smaller matrix (dimension that goes from 30000*30000 to 40000*40000) and for big matrices (dimension that goes from 15000*15000 to 25000*25000).





3.6 Confidence interval

Using the same formulas shown for the CPU and implementing a new python script we were able to calculate the confidence intervals for all the different configurations (number of blocks / number of threads). For the sake of simplicity we decided to not include the confidence intervals here in the documentation but are available in the following files:

- **Standard GPU**: GPUIntervals.txt
- **Optimized GPU**: GPUOptimizedIntervals.txt

4. Conclusions

The Drastic Event Simulator is an efficient algorithm that scales well when executed in CPUs that have as many cores as possible. Thanks to the fact that it can be handled with very simple synchronization between threads, it has practically no threads overhead. Spatial and Temporal Locality are very well implemented and, as seen in the CPU analysis, the algorithm can be modified to use these localities for different implementations of the simulation. It's not memory bound, so it can be fitted in any device just slicing the data and re-running the algorithm with each slice. Another aspect that must be considered is that there is no interference between threads, so there are a very low number of cache misses, so the bottleneck is the number of physical threads that the machine can run simultaneously. When it is executed in a GPU the performance is way better thanks to the parallel architecture that is made of. Every aspect is similar to the CPU part but the bottleneck is present, and is the fact that it cannot execute the same instruction at the same time. In fact as we saw, it is pretty low in executing complex instructions while the CPU is way better optimized for this kind of operation. This brings us to try to resolve this bottleneck thanks to the profiler and try to simplify the operation. As suggested by the profiler, we use the “fused instruction” that approximates the result but is very fast. These instructions permit us to have the real speed up and to fully use the architecture of the GPU. So, in conclusion, if we don’t need to evaluate a precise and complex formula for calculating the destruction we can optimize the algorithm using fused instruction and have a huge boost in terms of execution times. Otherwise, the improvement is not significant and can be used CPU instead.