COMPUTER ARCHITECTURE PROJECT

PARALLELIZATION OF DES ALGORITHM

RICCARDO FIORINI, MARIELLA MELECHI, ERICA RAFFA

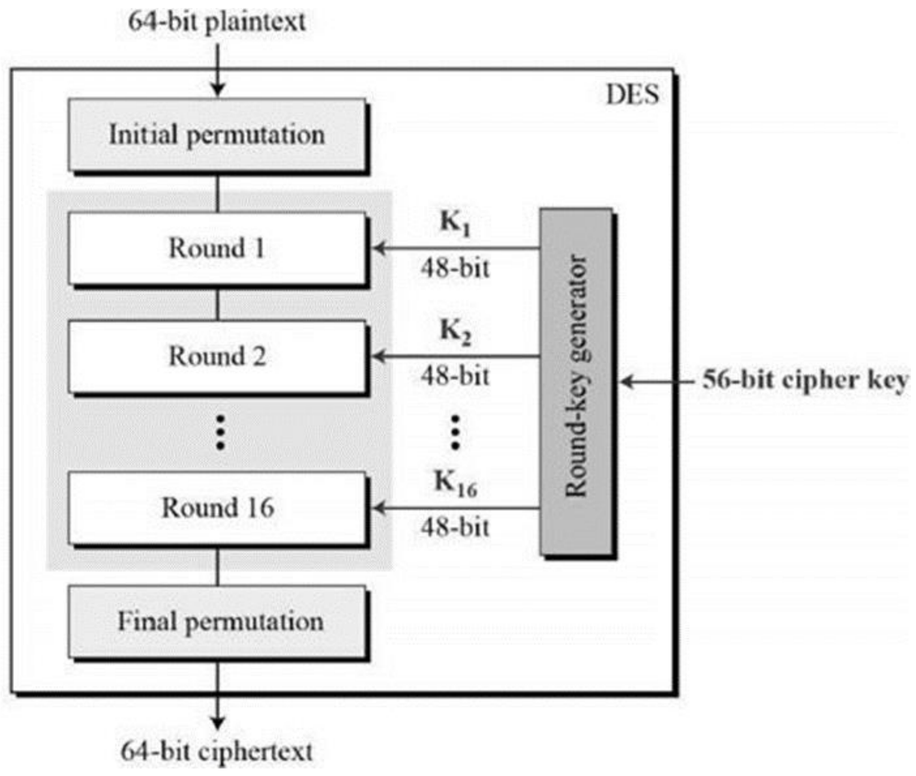
2022/2023

Table of Contents

1 Algorithm Introduction........3
2 CPU Implementation........4
2.1 Results........5
2.1.1 Elapsed time........5
2.1.2 Cumulative speedup........6
2.1.3 Throughput........7
2.2 Profiling........8
2.3 Optimization........9
2.4 CPU Conclusions........9
3 GPU Implementation........11
3.1 GPU System Information........11
3.2 Goal and results........12
3.2.1 Configurations........13
3.3 Bottleneck........14
3.4 Implementation........14
3.5 GPU Conclusions........14

1. ALGORITHM INTRODUCTION

The Data Encryption Standard (DES) is a symmetric-key block cipher where 64-bit plaintext blocks are encrypted into 64-bit cyphertext blocks using a key of 56 bits. The algorithm consists of several steps where the plaintext undergoes a series of elaborations:



- Each round performs the steps of substitution and transposition.
- In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
- Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
- Now each LPT and RPT go through 16 rounds of the encryption process where different functions are performed (xor, s-box, expansion)
- In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block to obtain the ciphertext.

We also performed the decrypt function using the same steps but reversing the keys.

2. CPU IMPLEMENTATION

For CPU parallel implementation of DES algorithm, the **plaintext** is read from a text file and stored in a string which is then divided into blocks of length **8 byte**, and they are parallelly **encrypted** and **decrypted**. In our solution the initial text is divided into strings depending on the number of threads to balance the load.
We analysed the algorithm's behaviour using a CPU with the following characteristics:

Processor:

Processor	AMD Ryzen 7 Mobile 4800H
#Cores	8
#Hardware Threads	16

Cache:

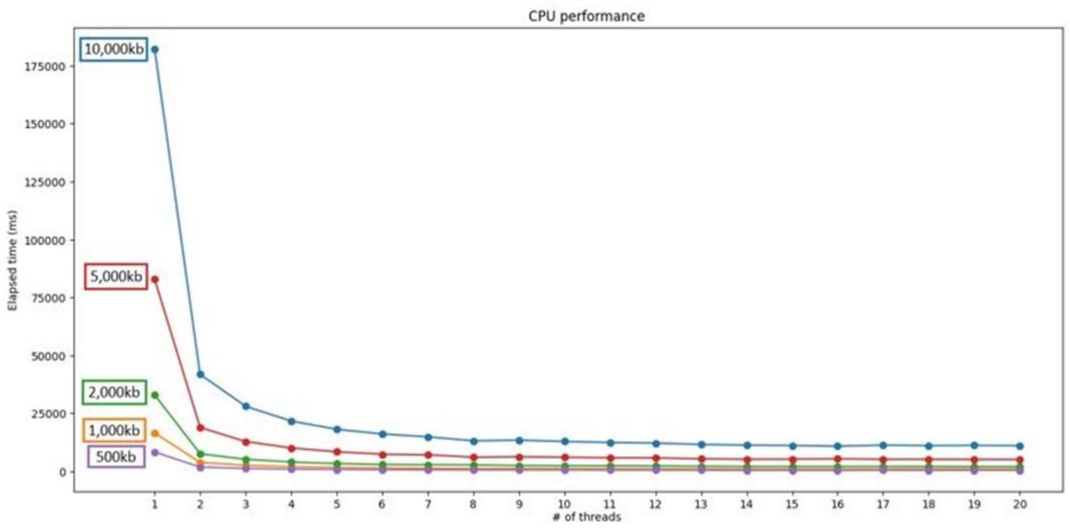
L1 Data Cache	8 x 32 KBytes	8-way
L1 Inst. Cache	8 x 32 KBytes	8-way
Level 2 Cache	8 x 512 KBytes	8-way
Level 3 Cache	2 x 4 MBytes	16-way

The **Goal** of this project is to analyse the performance of the algorithm and improve the elapsed time as the complexity of the input data increases.

2.1 RESULTS

We analysed the time taken by the algorithm to **encrypt** and **decrypt** a text (with 30 executions per configuration) with different plaintext complexities from 500Kb to 10000Kb, considering both the sequential execution with a single thread and then **parallelizing**, progressively increasing the number of threads to study the variation of the algorithm's **performance**.

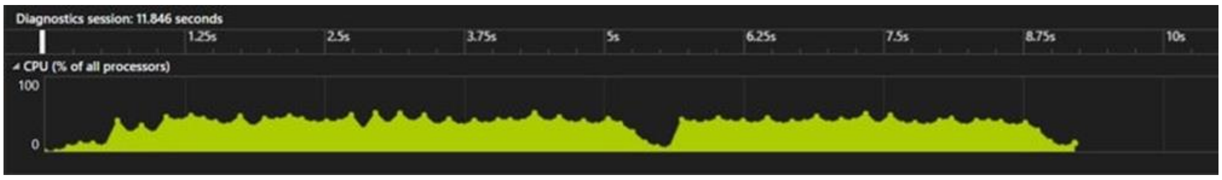
2.1.1 Elapsed time



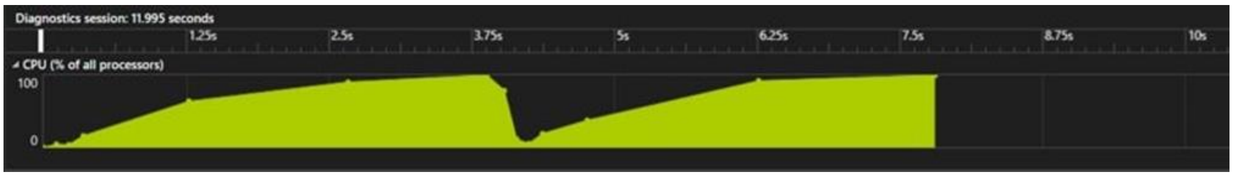
Elapsed time - CPU

For each text complexities, the elapsed time decreases as the number of threads increases because each thread encrypts and decrypts, in parallel, a portion of the text and then saturates when the quantity of **logical threads** reaches the number of **hardware threads** of the machine, in our case **16**.

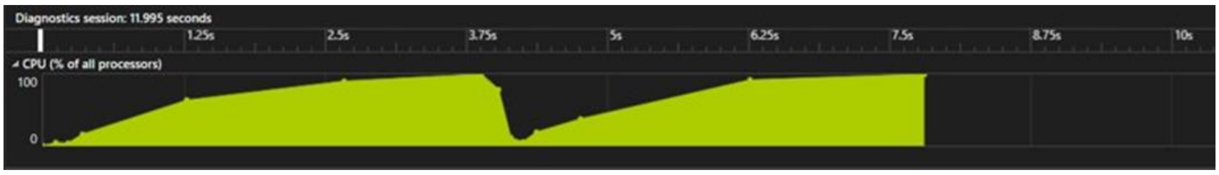
This happens because the CPU has reached its **maximum utilization** and, even if we have 100 threads, the max number of threads in parallel does not exceed 16, the elapsed time starts to increase and the CPU utilization decreases. At this point the portion of text of each thread becomes minimal and negligible compared to the **cost** of creating and maintaining the threads so that we do not reach any gain in performance by increasing the number of threads.
Moreover, we can observe that the algorithm is **highly scalable**, since the elapsed time decreases faster for bigger files.


Diagnostics session: 11.846 seconds
1.25s
2.5s
3.75s
5s
6.25s
7.5s
8.75s
10s
CPU (% of all processors)
100
0

CPU utilization for 8 threads


Diagnostics session: 11.995 seconds
1.25s
2.5s
3.75s
5s
6.25s
7.5s
8.75s
10s
CPU (% of all processors)
100
0

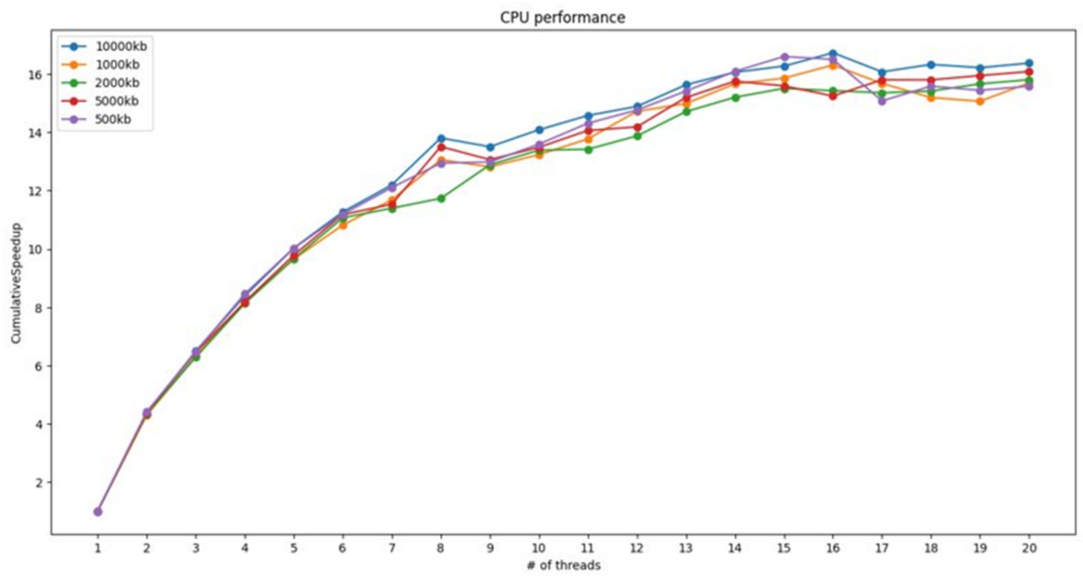
CPU utilization for 16 threads


Diagnostics session: 11.995 seconds
1.25s
2.5s
3.75s
5s
6.25s
7.5s
8.75s
10s
CPU (% of all processors)
100
0

CPU utilization for 100 threads

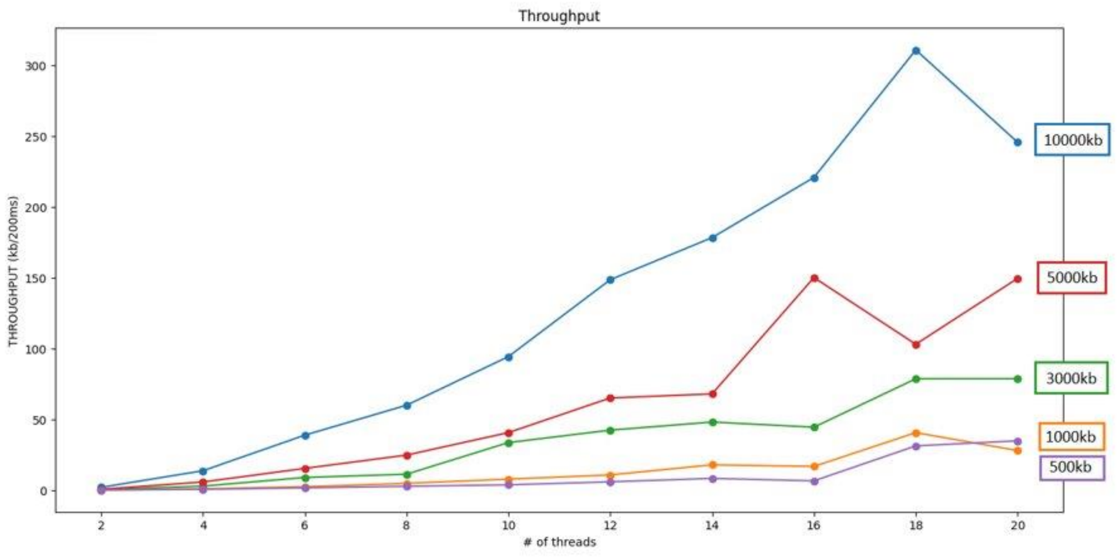
2.1.2 Cumulative speedup

As we expected, the **cumulative speedup** reaches the maximum value (16) with 16 threads and then **stabilizes**.



2.1.3 Throughput

Furthermore, analysing the **throughput** of the algorithm, considering the amount of encrypted Kb in **200 ms**, we notice how it increases with the number of threads, except for very small files where it stabilizes very soon because the piece of text to be encrypted for each thread becomes very small and they manage to finish before 200 ms.



2.2 PROFILING

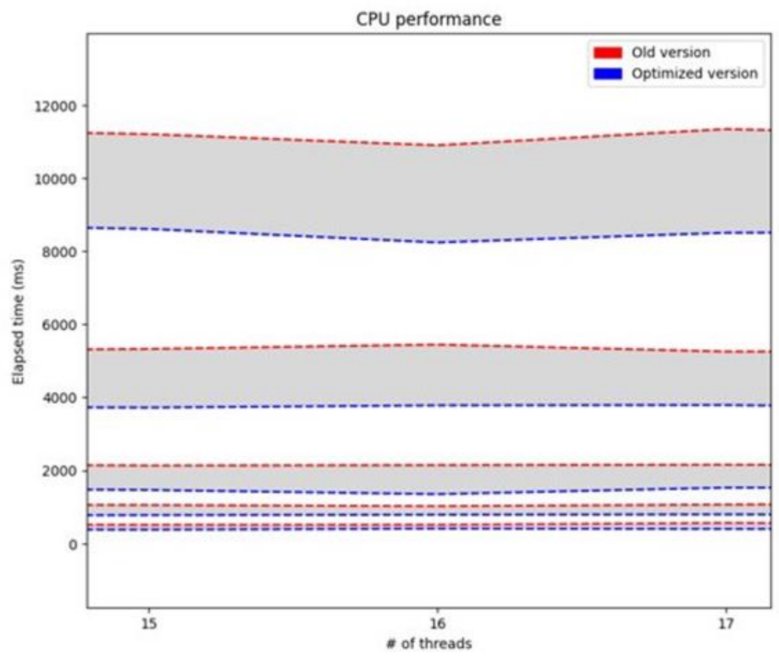
To achieve our goal, we have carried out the profiling of our algorithm to understand if it could be improved, analysing the bottlenecks and the functions on which the CPU takes more time.
Moreover, we decided to measure the time intervals of the **parallel computations** with *chrono::steady_clock*, since is specifically designed for this kind of measurements and the time between ticks of this clock is constant.

Front-End Bound	20.7%
Memory Bound	78.7 %
LLC Miss Count	11,700,637

As we expect the elapsed time depends on the amount of memory required to hold the working data, reaching 78.7% of **Memory Bound**. In fact, we have a high number of misses and a front-end bound percentage of 20,7% (*unutilized issue-slots when there is no Back-End stall, bubbles where Front-End delivered no uOps while Back-End could have accepted them*).
We also understood that the functions on which the CPU takes more time are *XOR* and *Permutation* that are performed during the encryption rounds. We tried to solve this overhead by replacing some functions with more efficient ones:

- **push_back() =>** emplace_back().
- **reserve()** to pre-allocate memory for strings.
- **logical xor =>** delete the if branch

2.3 OPTIMIZATION

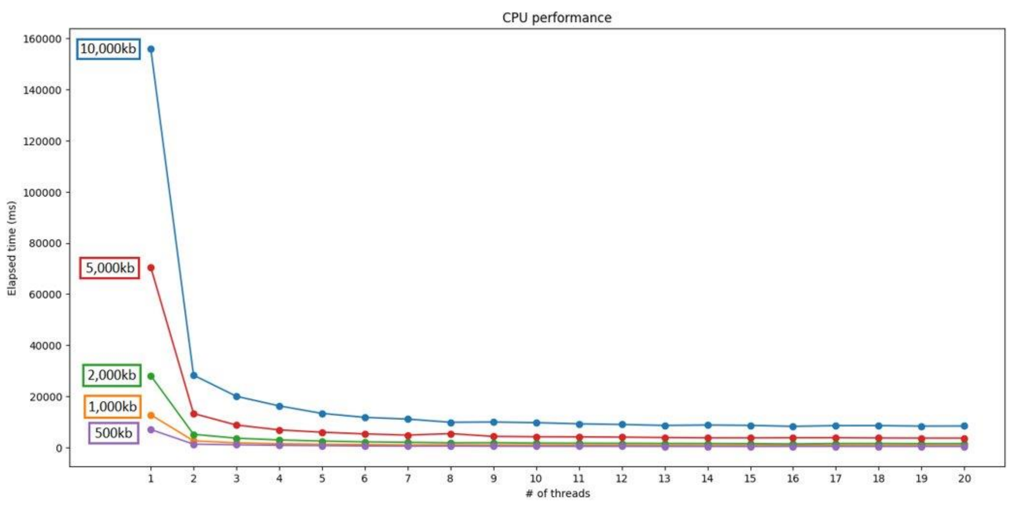


With these considerations we managed to optimize and accelerate the algorithm by about **35%** for both small and large files.
In fact, it can be seen how the number of **misses** and therefore the **memory bound** have decreased, as well as the elapsed time, while the **speed-up** has increased.

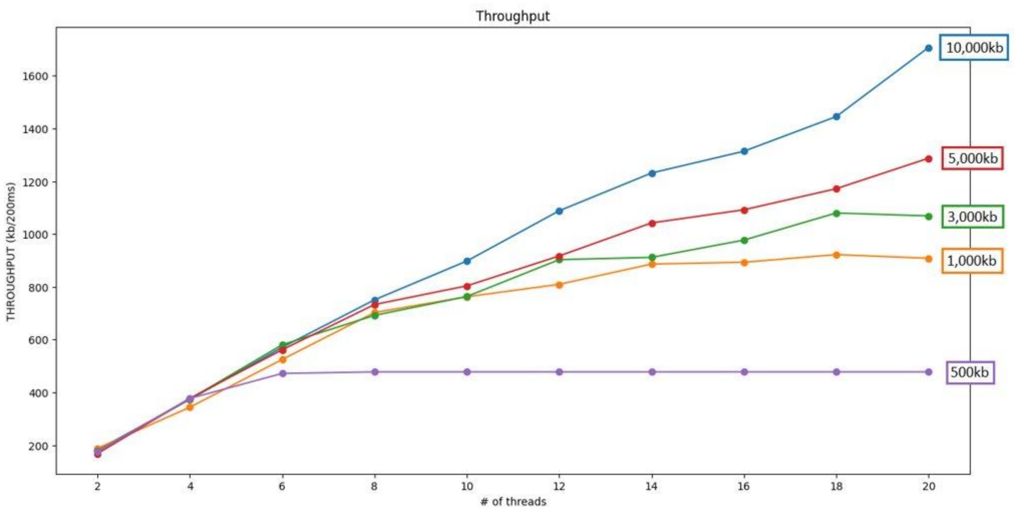
Front-End Bound	24.0%
Memory Bound	73.6 %
LLC Miss Count	6,500,364

2.4 CPU CONCLUSION

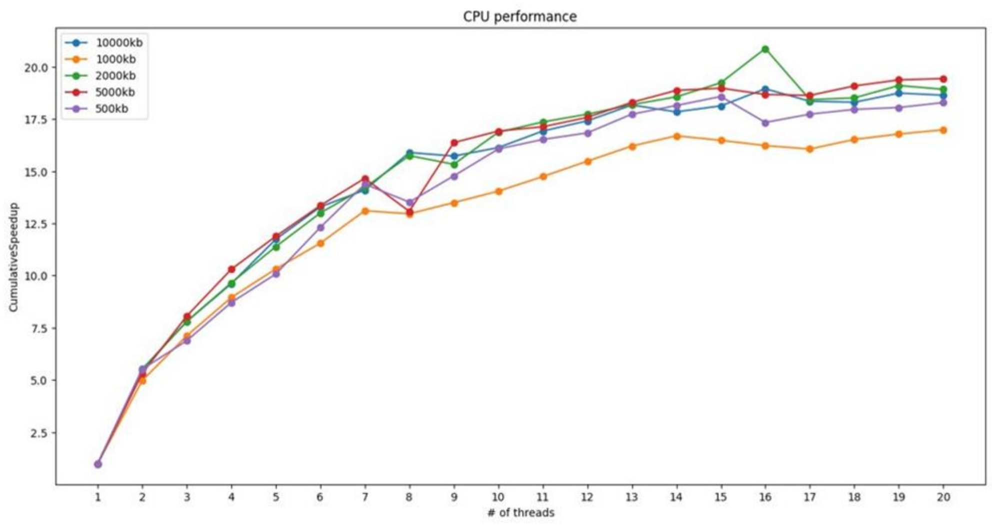
We achieved our goal of improving the **execution times** of the algorithm by using a parallel implementation, taking advantage of the **maximum CPU utilization**, speeding the algorithm up by **35%**.



Elapsed time cpu - optimized



Throughput cpu - optimized



Speedup cpu – optimized

3. GPU IMPLEMENTATION

We tried to achieve a major optimization of our algorithm with the GPU implementation in CUDA C. We adopted the same strategy of the CPU implementation: the plaintext is read from a text file and stored in a string which is then divided into blocks of length 8 byte, and they are parallelly encrypted and decrypted.

3.1 GPU System Information

We analysed the algorithm's behaviour using a GPU with these characteristics:

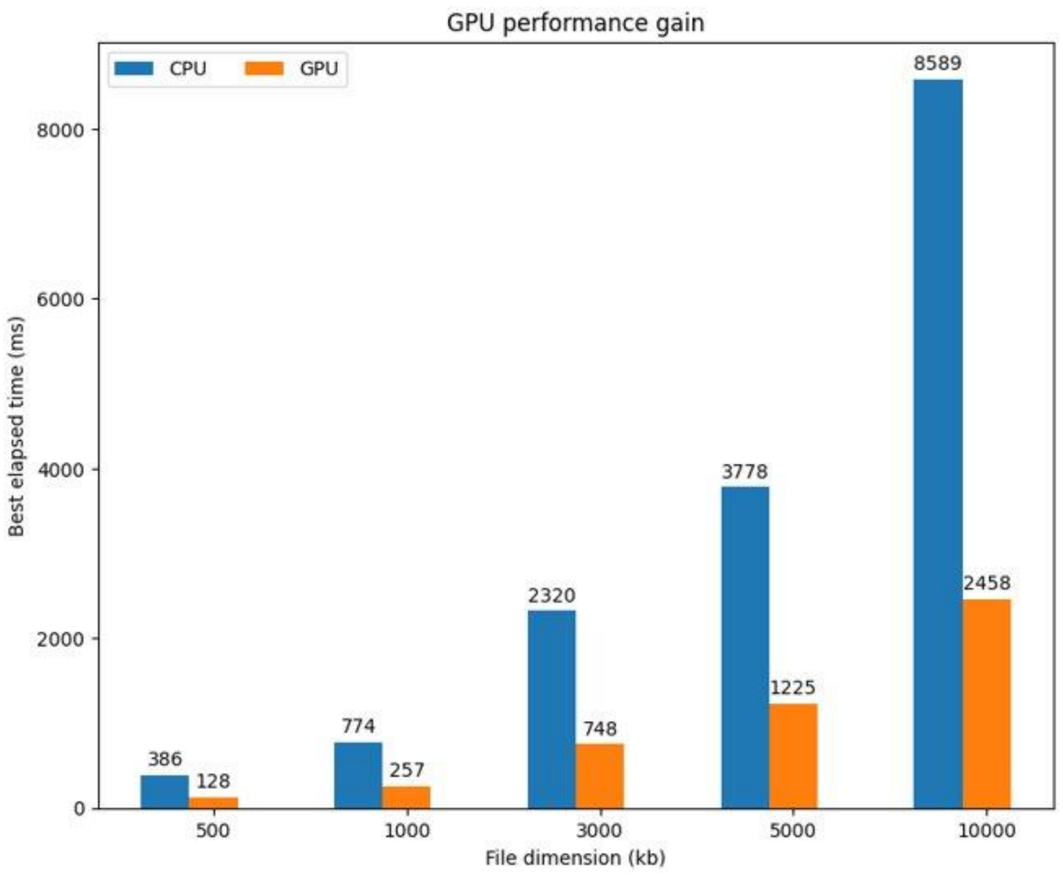
Global Memory:	4096 Mbytes
# Cuda Cores:	1024 CUDA Cores
GPU Max Clock Rate:	1485 MHz
Memory Clock Rate:	6001 MHz
Memory Bus Width:	128-bit
L2 Cache Size:	1048576 bytes
Warp Size:	32
# Threads / Multiprocessor:	1024
# Threads / Block:	1024
Max size of Thread Block (x, y, z):	(1024, 1024, 64)
Max size of Grid Size (x, y, z):	(2147483647, 65535, 65535)

Data retrieved from CUDA Device Query (Runtime API)

3.2 Goal and results

Our **goal** was to reduce the elapsed time obtained with the CPU implementation, so to improve the overall performance of the algorithm.

The **results** that we obtained are show in the following graph, where we compare the elapsed time achieved with both CPU and GPU implementations:

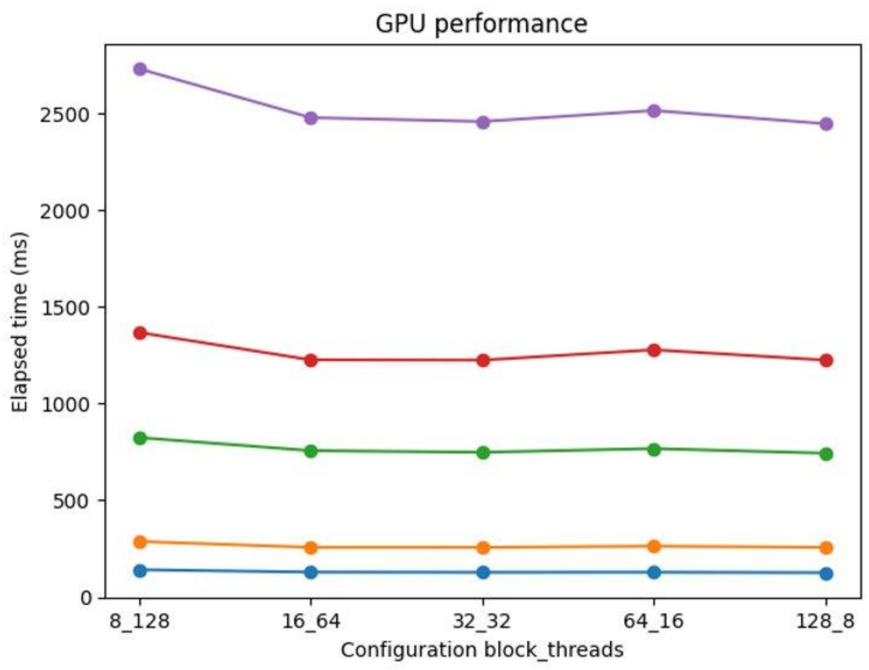


Comparison of elapsed time – CPU (left) GPU (right)

As we can see, we have achieved a further performance improvement thanks to the GPU, reducing the **elapsed time** of about 70%.

3.2.1 Configurations

We analyzed the mean elapsed time taken by the algorithm of encryption and decryption of a text (with 30 executions per configuration) with different plaintext size (from 500Kb to 10000Kb), progressively increasing and decreasing the number of **blocks** in the grid and the number of **threads per block**, to study the variation of the algorithm's performance.



Elapsed time(ms) - GPU

As we can notice, there is an overall **performance improvement**: the best elapsed time is achieved when the number of **blocks** increases, instead of increasing the number of **threads** of a single block.

3.3 Bottleneck

This behaviour is due to the **memory-bound** nature of the kernel, as we can see in the following table:

File Size	Compute Throughput (%)	Memory Throughput (%)
500 Kb	5.43	75.82

1000 Kb	5.48	77.90
3000 Kb	5.50	78.45
5000 Kb	5.57	79.99
10000 Kb	5.57	80.49

Data retrieved from NVIDIA Nsight Compute

A further analysis with the NVIDIA Nsight profiler revealed that the **high memory throughput**, together with the **high memory access time**, reduces the **warp** utilization, limiting the performance improvement with a growing number of threads per block.

3.4 Implementation

Two main point in our implementation:

- Encryption keys stored in shared memory, to speed up the access time
- Distributed plaintext towards the threads of the grid, to ensure a balanced workload

Moreover, we decided to measure the time intervals of the **parallel computations** using a **GPU-specific timer**. A problem with using host-device synchronization points, such as *cudaDeviceSynchronize()*, is that they stall the GPU pipeline. For this reason, CUDA offers a relatively light-weight alternative to CPU timers via the *CUDA event API*.

3.5 GPU Conclusions

We achieved our goal of improving the **elapsed time** of the algorithm by using a parallel implementation with CUDA C, taking advantage of different configurations of grid and number of threads, speeding the algorithm up by **70%**, compared to our CPU implementation.