DES Password Decryption

Edoardo Cagnes
edoardo.cagnes@stud.unifi.it

Andrea Giorgi
andrea.giorgi2@stud.unifi.it

Abstract

In this paper we present three different implementations of DES (Data Encryption Standard) Password Decryption with a brute force approach: a serial implementation in C++, a parallel implementation using OpenMP on the CPU and a parallel implementation using CUDA on the GPU.

1. Introduction

The Data Encryption Standard is a symmetric-key algorithm for the encryption of digital data based on an earlier design by Horst Feistel. Although today it's not secure anymore, it was highly influential in the evolution of modern cryptography.
After a brief introduction of the used techniques, we will focus on how we've implemented DES and the results obtained during various tests.

1.1. Techniques used

The sequential variant was made in C++.

To develop the parallel implementation on the CPU, we used **OpenMP**, an API for shared-memory parallel programming and designed for systems in which each thread or process can potentially have access to all available memory. When using OpenMP, we view our system as a collection of cores or CPUs, all of which have access to the main memory. It's a set of compiler directives (instructing the compiler on how to parallelize the code: this is the core of OpenMP), library routines (modifying and checking the number of threads and how many processors there are in the multiprocessor system), and environment variables (to alter the execution of OpenMP applications; e.g. the number of max. threads to use). The main advantage of OpenMP is that it doesn't require to restructure the sequential code.

To develop the parallel implementation on the GPU, we used **CUDA**, a parallel computing platform and application programming interface (API) model created by NVIDIA, that allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages, including C, C++, Fortran, and Python.

CUDA provides two API levels for managing the GPU device and organizing threads:

- **CUDA Driver API**: it's a low-level API and it's relatively hard to program, but it provides more control over how the GPU device is used.
- **CUDA Runtime API:** it's a higher-level API implemented on top of the driver API. Each function of the runtime API is broken down into more basic operations issued to the driver API.

1.2. Tools used

To develop our software, we used two different tools:

- **CLion**: a smart C and C++ editor developed by Jet Brains [1] that, thanks to native C and C++ support, including modern C++ standards, libc++ and Boost, knows your code through and through and takes care of the routine while you focus on the important things. It has an integrated debugger to investigate and solve problems with ease.
- **Nsight**: a full-featured IDE developed by NVIDIA and powered by the Eclipse platform [2] that provides an all-in-one integrated environment to edit, build, debug and profile CUDA-C applications.

2. Data Encryption Standard

Developed in the early 1970s at IBM, the algorithm was submitted to the National Bureau of Standards (NBS) following the agency's invitation to propose a candidate for the protection of sensitive, unclassified electronic government data. In 1976, after consultation with the National Security Agency (NSA), the NBS selected a slightly modified version (strengthened against differential cryptanalysis, but weakened against brute-force attacks), which was published as an official Federal Information Processing Standard (FIPS) for the United States in 1977.

DES is the archetypal block cipher, i.e. an algorithm that takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. In the case of DES, the block size is 64 bits. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt. The key ostensibly consists of 64 bits; however, only 56 of these are actually used by the algorithm. Eight bits are used solely for checking parity and are successively discarded, hence the effective key length is 56 bits.
The key is nominally stored or transmitted as 8 bytes, each one with odd parity. According to ANSI INCITS 92-1981: *One bit in each 8-bit byte of the KEY may be utilized for error detection in key generation, distribution, and storage. Bits 8, 16, ..., 64 are for use in ensuring that each byte is of odd parity*.

2.1. Feistel Cipher

As anticipated, DES is based on an earlier design by Horst Feistel. In cryptography, a Feistel cipher is a symmetric structure used in the construction of block ciphers. The Feistel structure has the advantage that encryption and decryption operations are very similar, even identical in some cases, requiring only a reversal of the key schedule. Therefore, the size of the code or circuitry required to implement such a cipher is nearly halved. It works in this way:
Let F be the round function and let $K_0, K_1, \dots, K_n$ be the sub-keys for the rounds $0, 1, \dots, n$ respectively. Then the basic operation is as follows:

- Split the plaintext block into two equal pieces, (L_0, R_0);
- For each round $i = 0, 1, \dots, n$, compute: $L_{i+1} = R_i$ and $R_{i+1} = L_i \oplus F(R_i, K_i)$; where $\oplus$ means XOR.

Then the ciphertext is (R_{n+1}, L_{n+1}).
Decryption of a ciphertext (R_{n+1}, L_{n+1}) is accomplished by computing for $i = n, n-1, \dots, 0$:
$R_i = L_{i+1}$ and $L_i = R_{i+1} \oplus F(L_{i+1}, K_i)$.
Then the plaintext is (L_0, R_0).
A detailed demonstration about how the decryption process works can be found in [7].
One advantage of the Feistel model compared to a substitution–permutation network is that the round function F does not have to be invertible.

2.2. DES Implementation

DES uses 16 round Feistel structure. The block size is 64-bit. Even though the key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only). General structure of DES is depicted in figure 1.

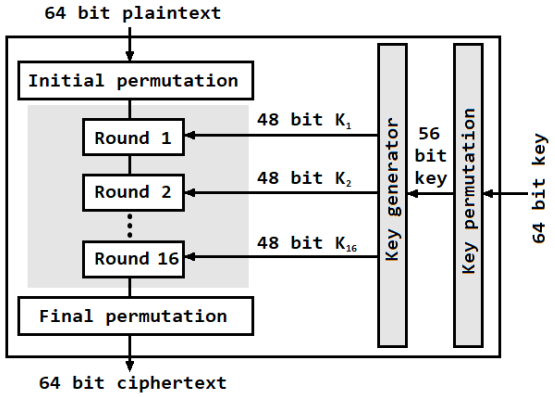

Fig. 1 - DES structure

Since DES is based on the Feistel Cipher, all that is required to specify DES is:

- Initial and final permutations;
- Round function F;
- Key generation process.

2.3. Initial and final permutations

The initial and final permutations are straight Permutation boxes (P-boxes) that are inverses of each other. They have no cryptography significance in DES. The initial permutation follows the table shown in Fig. 2

Initial permutation

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Fig. 2 - Initial permutation table

The first entry of this table is “58” and this means that the 58th bit of the initial message became the first of the permuted one. This is done for every bit of the message itself. The final premutation table follows the same rules described above.

Final permutation

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Fig. 3 - Final permutation table

2.4. The round function

The core of this cipher is the DES function, f. The DES function applies a 48-bit key to the rightmost 32 bits to produce a 32-bit output.

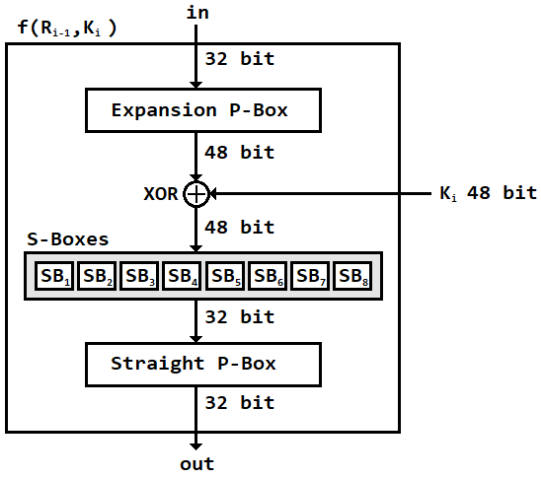

Fig. 4 - DES round function structure

- **Expansion Permutation Box**: since right input is 32-bit and the round key is a 48-bit, we first need to expand the right input to 48 bits. To achieve this, was used the permutation table shown in fig. 5. Obviously, to expand from 32 to 48, some bits are used more than once.

Expansion P-Box

32	1	2	3	4	5	4	5
6	7	8	9	8	9	10	11
12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21
22	23	24	25	24	25	26	27
28	29	28	29	30	31	32	1

Fig. 5 - Expansion permutation table

- **XOR**: after the expansion permutation, DES does XOR operation (bit by bit) on the expanded right section and the round key. The round key is used only in this operation.

- **Substitution Boxes**: the S-boxes carry out the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output each. Each S-box has a table with 4 rows and 16 columns that's used to generate the output. The result of the XOR operation is divided in 8 groups of six bit (grouped in sequence from the leftmost to the rightmost bit) and each one will be addressed in a different S-box.

$to\ Sbox_1$	$to\ Sbox_2$	$to\ Sbox_3$	...	$to\ Sbox_8$
100101	111011	111100	...	111110

The first and last bit of a group represents a number in the decimal range 0 to 3 in base 2. Let this be number *i*. The remaining 4 bits in the middle represent a number in the decimal range from 0 to 15 in base 2. Let this number be *j*. The output from the S-box assigned to the considered group of six bit will be the value (converted in binary) contained inside the i-th row and j-th column of the S-box table.

For example, consider the group "110011", the first and last bit are "11": converted to decimal, this number is 3, while the 4 middle bits are "1001", that converted to decimal is 9. So, we have to pick the value in row 2, column 9. This value is then converted to binary and that's the output of the considered S-box.

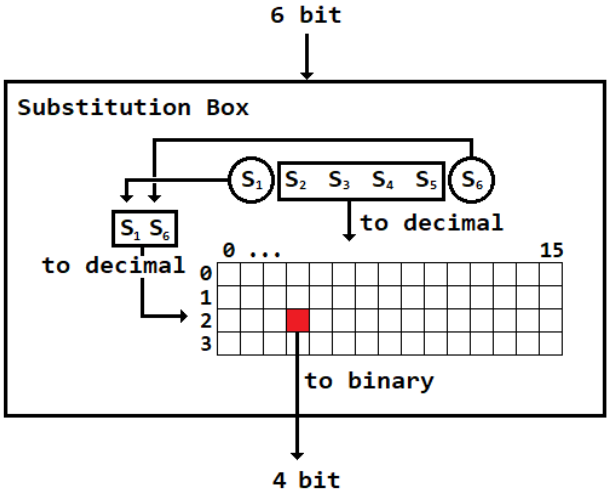

Fig. 6 - S-box structure

There is a total of eight S-box. The output of all eight s-boxes is then combined into a 32-bit section. Let $S_1 \ldots S_8$ the eight S-boxes and $G_1G_2G_3G_4G_5G_6G_7G_8$ the result of the XOR descripted previously (G is a group of 6 bits), the combined output of all S-boxes is:

$S_1(G_1)S_2(G_2)S_3(G_3)S_4(G_4)S_5(G_5)S_6(G_6)S_7(G_7)S_8(G_8)$

- **Straight Permutation**: the 32-bit output of S-boxes is then subjected to the straight permutation using the table in fig. 7.

Straight permutation

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Fig. 7 - Straight permutation table

2.5. Key Generation

The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key.
The initial key is a string of 64 bit, usually obtained converting an 8-char keyword. It's then permuted (and reduced to 56 bits) according to the table in fig 8 (the rules are the same of all the permutations seen so far).

Parity bit drop table

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Fig. 8 - Parity bit drop table; this reduces the key from 64 to 56 bit dropping every bit multiple of 8 (was once used to protect the key from transmission errors)

The 56-bit key is then split into left and right halves: KL_0 and KR_0 (each one has 28 bits). According to the fig. 9, sixteen pairs of blocks KL_i and KR_i ($1 \leq i \leq 16$) are created performing a certain number of left shifts: the i^{th} blocks are made by shifting KL_{i-1} and KR_{i-1} the number of times assigned to the iteration i. To do a left shift, each bit is moved to one place to the left (except for the first one that is cycled at the end of the block).

Shift table

1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Fig. 9 - Number of shifts to perform; the iteration i is the index. For example, the fourth iteration will require two shifts to be performed while the nineth just one.

Finally, to obtain the 48-bit round-key K_i with $1 \leq i \leq 16$, another permutation (the one in fig. 10) is applied to the concatenated pairs KL_iKR_i

Key compression table

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Fig. 10 - Key compression table

3. Implementation

Now we'll describe the problem that we want to solve, then we will present the sequential implementation first and the two parallel variants later.

3.1. Description of our problem

We have:

- a block named **DesEncryptor** that encrypts strings of 8 characters
- an **encrypted password**
- a **string generator**

We know:

- the set of all possible characters that can make the password up (we will later refer to this as the "allowed characters array")

We want:

- to find the combination that, once encrypted, matches the password

This is called Brute Force Attack.

3.2. Sequential Implementation

This was basically made by implementing the DES Algorithm in C++. We've also taken [8] as example. Firstly, we've created all the arrays required to perform permutation and substitutions and, later, we've made both the key generator and the `encrypt()` function. We've also made some utility functions to convert string to hexadecimal and binary (as explained before, DES works with bits).
After running some tests (in order to check that everything works correctly), we've wrapped everything inside a class named `DesEncryptor`. We've then made a new function to generate all possible passwords, using an iterative approach. Let ***n*** be the number of allowed characters, ***k*** the length of a password. If we consider the array of allowed characters as {a, b, c}, all possible password will be:

aaa	*aab*	*aac*
aba	*abb*	*abc*
aca	*acb*	*acc*
baa	*bab*	*bac*
...	...	...

These are dispositions with repetitions and the number of all possible passwords is defined as $D'_{n,k} = n^k$.
If we check these values, we can see that those can be represented by just their indices inside the allowed characters array:

000 001 002
010 011 012
020 021 022
100 101 102
... ...

These are also all **numbers of length *k* expressed in base *n***. Therefore, "generating all possible passwords" can be achieved by firstly generating every number of length k in base n, and then converting each digit using it as index of the allowed characters array.
We now have everything we need to build our password generator.

Algorithm 1 Pseudocode of `generatePassword()`

```
1: function generatePassword(n,k,iteration)
2:     for i=0 to k-2 do
3:         out[i] = charset[⌊iteration / n^(k-1-i)⌋]
4:         iteration = iteration mod n^(k-1-i)
5:     out[k-1] = charset[iteration mod n]
```

Now we simply use a for-loop to iterate over all possible password and find the one that, once encrypted, matches the ciphertext we want to decrypt. With a break statement, we will then terminate the execution of our software.

3.3. First parallel implementation - OpenMP

We used the parallel-for directive to distribute the iterations above multiple threads. The OpenMP framework provides different loop scheduling, these are the most relevant:

- **static**, the iterations are distributed almost equally among threads just when the loop starts;
- **dynamic**, specifies a dynamic distribution between threads. Whenever one of those ends its chunk of iterations, a new chunk is assigned;
- **guided**, very close to dynamic scheduling but the number of assigned iterations decreases during the execution in order to better handle any load imbalance.

Among those, we've chosen to use the static scheduling because the workload of each iteration is almost constant and index-independent.
The parallel-for directive, however, can only be used if the for-loop doesn't have a break statement, so we have to develop a strategy to stop the for-loop once we've found the password. To achieve that, OpenMP provides the for-cancellation directive: inside the loop are placed some so called "cancellation point" and the break statement is replaced with the directive: `#pragma omp cancel for`. Whenever a thread encounters the cancellation point, it checks if any of the other threads ran the for-cancellation directive. If the answer is affirmative, the loop stops itself. It's basically a parallel construct for the break statement.

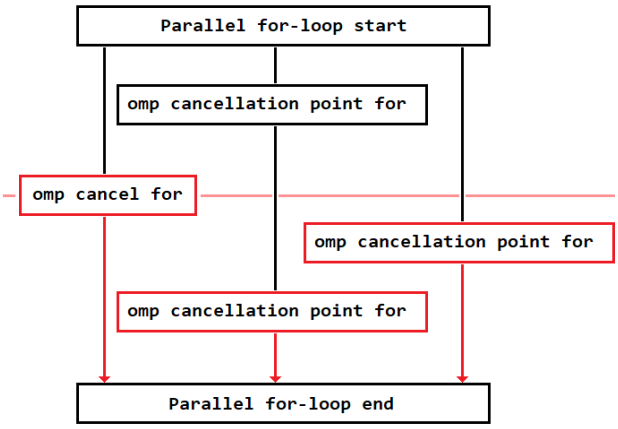

Fig. 11 - For-cancellation logic

The for-cancellation directive is an expensive operation and it's only executed if the environment variable `OMP_CANCELLATION` is set to true. Therefore, to run this software regardless of the system used, we've made a guard to force `OMP_CANCELLATION` to true and then restart the execution.

Algorithm 2 Pseudocode of OpenMP implementation

```
1: function main()
2:     if(OMP_CANCELLATION == false)
3:         restart with OMP_CANCELLATION = true
4:     parallel for i=0 to possiblePasswords do
5:         plaintext = generatePassword(n,k,i)
6:         ciphertext = encrypt(plaintext)
7:         if(password == ciphertext)
8:             cancel for
9:         cancellation point
```

3.4. Second parallel implementation - CUDA

As explained before, this kind of parallelization uses the GPU instead of the CPU. The code made for sequential implementation was rearranged to work correctly on CUDA. In particular the sixteen round-keys were converted to a 1D array and transferred from CPU (Host) to GPU (Device). To avoid any kind of conflict between threads, every array was preallocated with its normal size multiplied by the number of threads.
In CUDA, threads are organized within blocks: each block can hold up to 1024 threads. During the execution, threads of a block are divided into warps and each of those always contains 32 threads. To optimize computation, it's important to create thread blocks that result in mostly full

warps. In our software this is achieved by forcing the number of threads (if these are more than 32) to be multiples of 32.
Once the software starts, the user is asked to write down the 8 characters string password. The software later asks the number of threads and then, after having allocated all the resource with `cudaMalloc` and `cudaMemcpy`, it calls the CUDA kernel `findPassword`.

4. Results

To analyze the performance in a reasonable time, we decided to reduce the allowed characters solely to numbers from 0 to 9. This will drastically reduce the possible password space while we can still study the behavior and the speedup of our software.
All the following tests are made on a computer with Intel® Core(TM) i7-9700K CPU with a frequency of 3.60GHz and a Nvidia GeForce GTX 1070Ti GPU.

4.1. Sequential Implementation

We've tried to decrypt several passwords distributed among all the password space. The behavior of this variant was like we expected: the more iterations are required to generate a password with the algorithm 1, the more time will be required to complete the brute force attack.

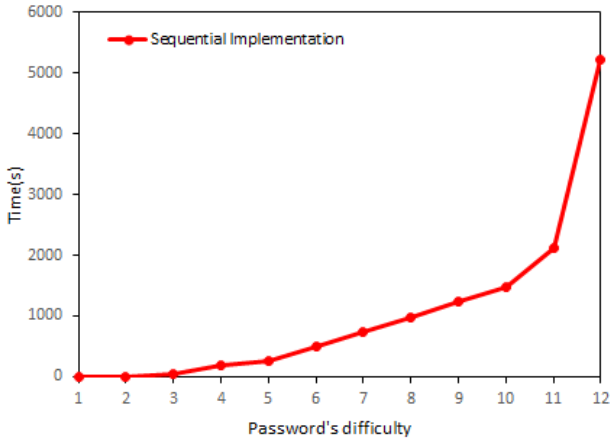

Fig. 12 - Sequential implementation computation time; on the horizontal axis there is the password's difficulty: the higher this value is, the more iterations will be required to discover the considered password

4.2. OpenMP Implementation

Before trying different passwords among all the password space, this implementation was tested with different numbers of threads starting from 1 (that gave us results almost similar to the sequential variant) up to 4096.

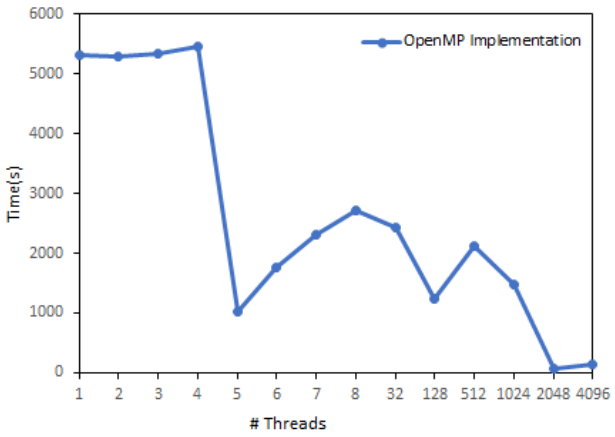

Fig. 13 - OpenMP computation time with different threads; note that the number of threads is not in scale, otherwise it would be very difficult to undertand the values between 1-32

As shown in the graph in fig. 13, there is a big improvement passing from 4 to 5 threads but this depends by where the password to decrypt is inside the chunk of iteration. Let *i* be the iteration that will disclose the password according to algorithm 1, in the worst-case scenario if this *i* is the last of the chunk of iteration assigned to an OpenMP thread, it will require the most time possible. Otherwise, the computation time will be minimal if *i* is at the beginning of a chunk. This last scenario is almost the same as when we used 5 threads.

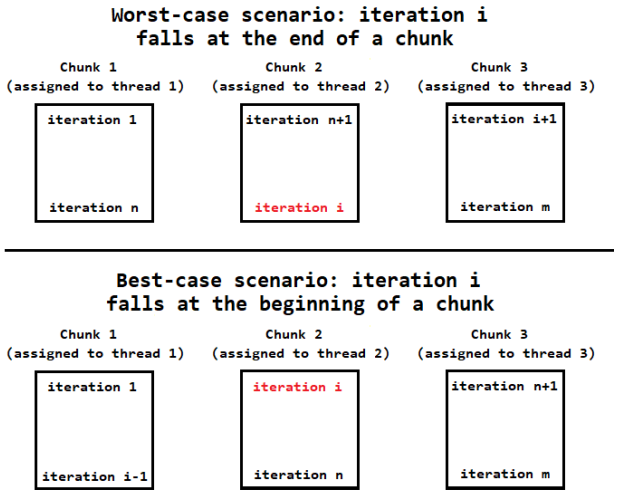

Fig. 14 - Graphical explanation of worst-case scenario vs best-case scenario

This behavior tends to become less relevant with greater number of threads: the smaller the chunk of iterations is, the lesser is the distance in term of time from the first to the last iterations of the chunk. To verify this assumption, we've tested the OpenMP variant again varying the number of threads, but this time within the 2048-4096

range and we've discovered that the computation time, apart from some fluctuations, is almost the same.
Once we've found the optimal number of threads, we've run the same tests as the sequential variant: after fixing the number of threads, we've tested the OpenMP implementation with several passwords with different difficulties. Results are shown in fig.15

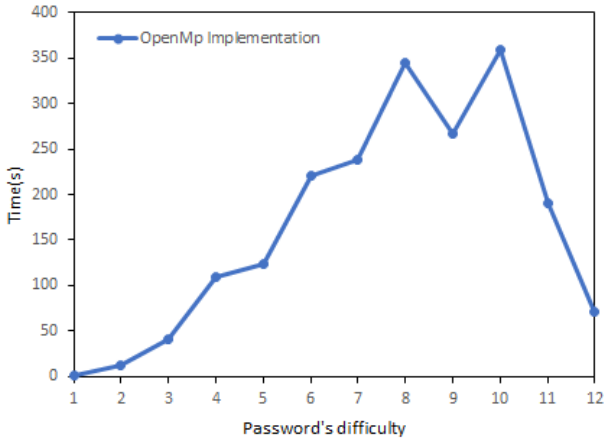

Fig. 15 – OpenMP computation time with different password's difficulty.

Now we consider the speed up factor defined as $S_p = \frac{t_s}{t_p}$ where t_s is the time required for sequential implementation and t_p the time required for the parallel one. In fig.16 and fig.17 there are two graphs showing the speed up factor of OpenMP implementation to the sequential variant.

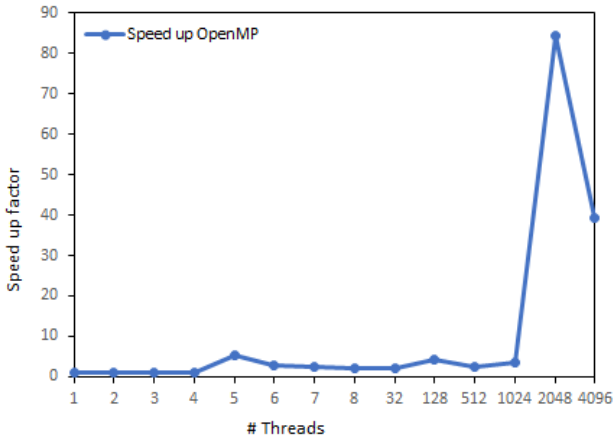

Fig. 16 - Speed up factor of OpenMP calculated fixing the password's difficulty but changing the number of threads

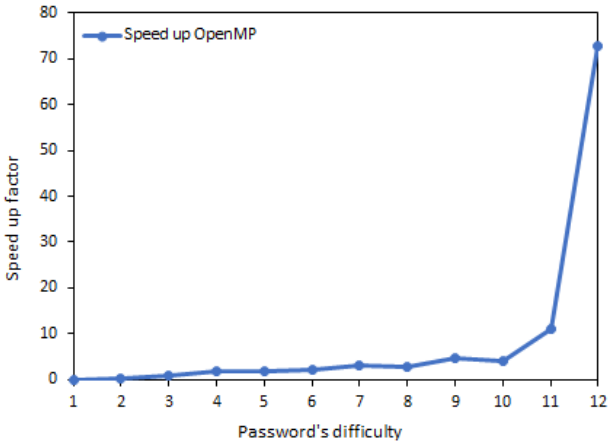

Fig. 17 - Speed up factor of OpenMP calculated fixing the number of threads but varying the password's difficulty

4.3. CUDA Implementation

Using the GPU parallelism, we've also made some tests with a great number of threads. Starting from a 32 to 10^8 the results are shown in fig.18.

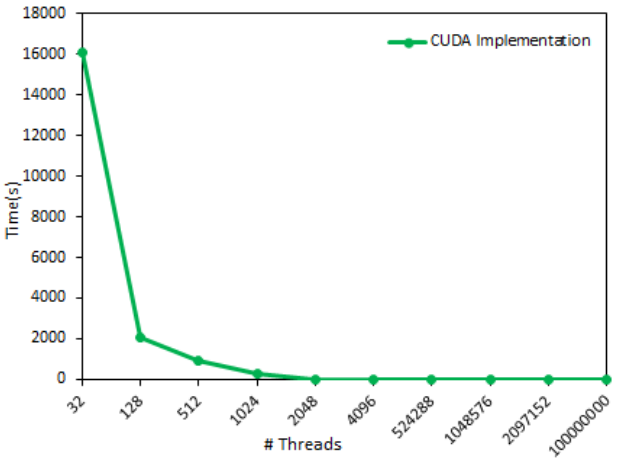

Fig. 18 - CUDA implementation computation time with different threads; with a small number of threads CUDA is a lot slower than the sequential variant

The best results are obtained with 10^8 threads which means a parallel thread per password. With this number of threads, the time to decrypt the password is almost identical regardless of the password's difficulty.
However, with a small number of threads, CUDA implementation slows down and it's outperformed by both sequential and OpenMP variant. In fig.19 we show the speed up factor of the CUDA implementation to the sequential variant varying the number of threads.

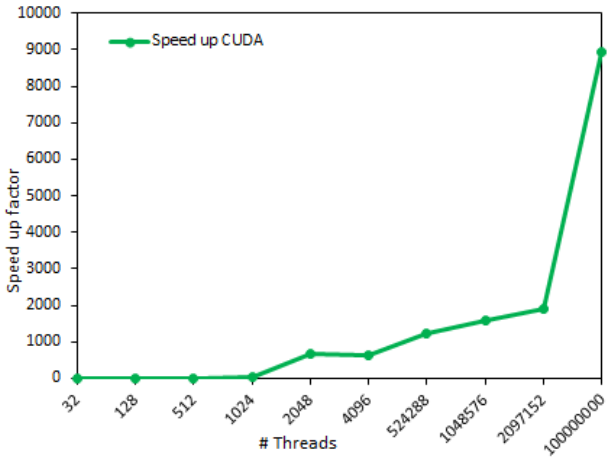

Fig. 19 - Speed up factor of CUDA varying the number of threads; with 10^8 threads CUDA is almost 9000 times more performant than the sequential variant

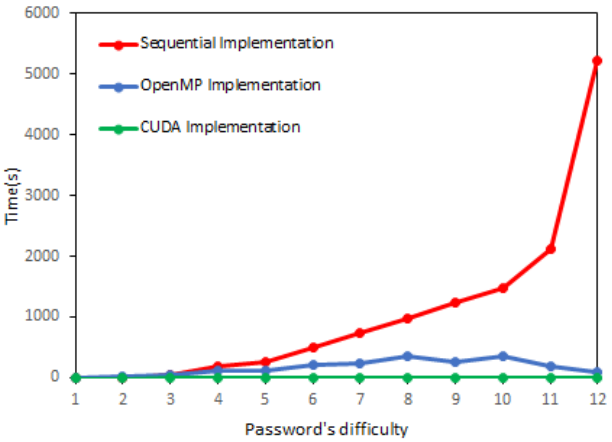

Fig. 20 - Comparison between the three different implementations; both parallel versions run with the optimal number of threads

5. Conclusion

We've presented three different implementations to decrypt a DES encrypted password with a brute force approach. The obtained results show that CUDA, with a great number of threads, outspeeds the other two methods, proving that nowadays GPU are suited for this kind of computations.
Regarding parallel implementations, since our software works on independent chunks of the password space, there was no need to synchronize any part of the code.

References

[1] Marco Bertini, *Parallel Computing 2020 course slide* *https://www.micc.unifi.it/bertini/teaching-parallel-computing.html*

[2] JetBrains. *Clion* *https://www.jetbrains.com/clion/*

[3] NVIDIA *Nsight Eclipse Edition* *https://developer.nvidia.com/nsight-eclipse-edition*

[4] Wikipedia, *Data Encryption Standard* *https://en.wikipedia.org/wiki/Data_Encryption_Standard*

[5] Wikipedia, *Feistel cipher* *https://en.wikipedia.org/wiki/Feistel_cipher*

[6] Tutorialspoint, *Data Encryption Standard* *https://www.tutorialspoint.com/cryptography/data_encryption_standard.htm*

[7] Michele Boreale, *Note per il corso di Codici e Sicurezza*, Edizione 2012/2013

[8] GeeksforGeeks, *Data Encryption Standard (DES) Set 1* *https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/*

[9] Jaka's Corner, *OpenMP: Cancel and cancellation points* *http://jakascorner.com/blog/2016/08/omp-cancel.html*