PC-2017/18: DES Brute-force Attack

Alberto Ciolini
alberto.ciolini@stud.unifi.it

Alessandro Soci
alessandro.soci@stud.unifi.it

Abstract

We present the brute-force attack on Data Encryption Standard (DES) in three different implementation: a serial implementation in C++, a parallel implementation using OpenMP on the CPU and a parallel implementation using CUDA on the GPU. We compare the performances of the parallel approaches running on the CPU and on the GPU with the sequential implementation across a range of different passwords and number of threads.

Future Distribution Permission

1. Introduction

The **Data Encryption Standard** (**DES**) is a symmetric-key algorithm for the encryption of electronic data. Although insecure, it was highly influential in the advancement of modern cryptography.

Developed in the early 1970s at IBM and based on an earlier design by Horst Feistel, 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 eventually 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 works by encrypting groups of 64 message bits, which is the same as 16 hexadecimal numbers. To do the encryption, DES uses "keys" where are also apparently 16 hexadecimal numbers long, or apparently 64 bits long. However, every 8th key bit is ignored in the DES algorithm, so that the effective key size is 56 bits. But, in any case, 64 bits (16 hexadecimal digits) is the round number upon which DES is organized [3].

1.1. DES implementation

DES is a *block cipher*, meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. Thus DES results in a *permutation* among the 2^{64} possible arrangements of 64 bits, each of which may be either 0 or 1. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block L and a right half R.
DES operates on the 64-bit blocks using key sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used (i.e. bits numbered 8, 16, 24, 32, 40, 48, 56, and 64).

1.1.1 Key Generation

The 64-bit key is permuted according to *PC-1* table (See Fig. 1). Since the first entry in the table is "57", this means that the 57th bit of the original key K becomes the first bit of the permuted key K+. The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Note only 56 bits of the original key appear in the permuted key.
Next, split this key into left and right halves, C0 and D0, where each half has 28 bits.

PC-1

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

Figure 1. PC-1 table.

With C_0 and D_0 defined, we now create sixteen blocks C_n and D_n, $1 <= n <= 16$. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1}, respectively, for $n = 1, 2, ..., 16$, using the following schedule of ”left shifts” of the previous block (See Fig. 2). To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Figure 2. Shifts.

This means, for example, C_3 and D_3 are obtained from C_2 and D_2, respectively, by two left shifts, and C_{16} and D_{16} are obtained from C_{15} and D_{15}, respectively, by one left shift. In all cases, by a single left shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3,..., 28, 1.

We now form the keys K_n, for $1 <= n <= 16$, by applying the *PC-2* permutation table to each of the concatenated pairs C_nD_n (See Fig. 3). Each pair has 56 bits, but PC-2 only uses 48 of these.

PC-2

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

Figure 3. PC-2 table.

Therefore, the first bit of K_n is the 14th bit of C_nD_n, the second bit the 17th, and so on, ending with the 48th bit of K_n being the 32th bit of C_nD_n.

Algorithm 1 Key Generation

```
procedure KEY-GENERATION(key_str[64])
    p_key[56];
    for i=0 to 56 do
        p_key[i] ← key_str[PC1[i]-1];
    for i=0 to 28 do
        C0 ← p_key[i];
    for i=0 to 28 do
        D0 ← p_key[i+28];
    key[16][48];
    for i=0 to 16 do
        if i=0 or i=1 or i=8 or i=15 then
            $shift \leftarrow 1$;
        else
            $shift \leftarrow 2$;
        while shift=0 do
            temp1 ← C0[0];
            temp2 ← D0[0];
            for j=0 to 27 do
                C0[j] ← C0[j+1];
                D0[j] ← D0[j+1];
            C0[27] ← temp1;
            D0[27] ← temp2;
            $shift \leftarrow shift - 1$;
        for j=0 to 24 do
            key[i][j] ← C0[PC2[j]-1]
        for j=24 to 48 do
            key[i][j] ← D0[PC2[j]-1-28]
```

1.1.2 Encode Message

There is an *initial permutation IP* of the 64 bits of the message data *M*. This rearranges the bits according to the following table (See Fig. 4), where the entries in the table show the new arrangement of the bits from their initial order. The 58th bit of *M* becomes the first bit of *IP*. The 50th bit of *M* becomes the second bit of *IP*. The 7th bit of *M* is the last bit of *IP*. Next divide the permuted block *IP* into a left half L_0 of 32 bits, and a right half R_0 of 32 bits.

IP

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

Figure 4. IP table.

We now proceed through 16 iterations, for $1 <= n <= 16$, using a function *f* which operates on two blocks - a data block of 32 bits and a key K_n of 48 bits - to produce a block of 32 bits. Let + denote *XOR* addition, (bit-by-bit addition modulo 2). Then for *n* going from 1 to 16 we calculate

$$L_n = R_{n-1} \tag{1}$$

$$R_n = L_{n-1} + f(R_{n-1}, K_n) \tag{2}$$

This results in a final block, for $n = 16$, of $L_{16}R_{16}$. That is, in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For the right 32 bits in the current step, we *XOR* the left 32 bits of the previous step with the calculation *f*.

It remains to explain how the function *f* works. To calculate *f*, we first expand each block R_{n-1} from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in R_{n-1}. We'll call the use of this selection table the function *E* (See Fig. 5). Thus $E(R_{n-1})$ has a 32 bit input block, and a 48 bit output block.

Thus the first three bits of $E(R_{n-1})$ are the bits in positions 32, 1 and 2 of R_{n-1} while the last 2 bits of $E(R_{n-1})$ are the bits in positions 32 and 1.

E BIT-SELECTION TABLE

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

Figure 5. Selection table.

Next in the *f* calculation, we XOR the output $E(R_{n-1})$ with the key K_n:

$$K_n + E(R_{n-1})$$

We have not yet finished calculating the function *f*. To this point we have expanded R_{n-1} from 32 bits to 48 bits, using the selection table, and XORed the result with the key K_n . We now have 48 bits, or eight groups of six bits. We now use them as addresses in tables called "*S boxes*". Each group of six bits will give us an address in a different S box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the S boxes) for 32 bits total.

Write the previous result, which is 48 bits, in the form:

$$K_n + E(R_{n-1}) = B_1B_2B_3B_4B_5B_6B_7B_8,$$

where each B_i is a group of six bits. We now calculate

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$$

where $S_i(B_i)$ referres to the output of the *i-th* S box.

To repeat, each of the functions S_1, S_2,..., S_8, takes a 6-bit block as input and yields a 4-bit block as output. The table to determine S1 is shown and explained below (See Fig. 6):

If S_1 is the function defined in this table and B is a block of 6 bits, then $S_1(B)$ is determined as follows: The first and last bits of B represent in base 2 a number in the decimal range 0 to 3 (or

S1

Row No.	Column Number 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Figure 6. S1 table.

binary 00 to 11). Let that number be *i*. The middle 4 bits of B represent in base 2 a number in the decimal range 0 to 15 (binary 0000 to 1111). Let that number be *j*. Look up in the table the number in the *i-th* row and *j-th* column. It is a number in the range 0 to 15 and is uniquely represented by a 4 bit block. That block is the output $S_1(B)$ of S_1 for the input B. For example, for input block $B = 011011$ the first bit is ”0” and the last bit ”1” giving 01 as the row. This is row 1. The middle four bits are ”1101”. This is the binary equivalent of decimal 13, so the column is column number 13. In row 1, column 13 appears 5. This determines the output; 5 is binary 0101, so that the output is 0101. Hence $S_1(011011) = 0101$.

The final stage in the calculation of *f* is to do a permutation *P* of the *S-box* output to obtain the final value of *f*:

$$f = P(S_1(B_1)S_2(B_2)...S_8(B_8))$$

The permutation *P* is defined in the following table (See Fig. 7). P yields a 32-bit output from a 32-bit input by permuting the bits of the input block.

P

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

Figure 7. P table.

At the end of the sixteenth round we have the blocks L_{16} and R_{16}. We then reverse the order of the two blocks into the 64-bit block

$$R_{16}L_{16}$$

and apply a final permutation IP^{-1} (See Fig. 8).

IP^{-1}

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

Figure 8. IP^{-1} table.

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

Decryption is simply the inverse of encryption, follwing the same steps as above, but reversing the order in which the subkeys are applied.

Algorithm 2 Encrypt

```
procedure CRYPT(text, key)
    l, r, round=16;
    for i=0 to 64 do
        text ← text[IP[i]-1];
    l = text(0,32), r=(32,64);
    while round=0 do
        for i=0 to 64 do
            expansion ← r[E[i]-1];
        for i=0 to 64 do
            xor_out ← expansion[i] ⊕ key[round];
        for i=0 to 16 do
            row ← get_S_row;
            col ← get_S_col;
            s_out ← S[i][row][col]
        for i=0 to 32 do
            pc[i] ← s_out[P[i]-1];
        for i=0 to 32 do
            new_r ← pc[i] ⊕ l[i];
        l ← r_old;
        round ← round - 1;
    encrypted_text ← r + l;
    for i=0 to 64 do
        encrypted_text ← encrypted_text[FP[i]-1];
```

2. Brute-force Attack

In this section we talk about brute force strategy implementation details. Since the password space could have so many different values, it is computationally impraticable testing all passwords in a reasonable time. In this context we want to examine only the behaviour of our solution in order to study the speedup, so we assumed [0-9] as possible characters set and 8-characters passwords.

Once we obtained the plain password and encrypted it with a fixed key, we start iterating on the password space testing all the possible combinations with the key to find a match with the user encrypted password.

2.1. Details

Iterating all over the password space, we aim to discover a function to map an index of the iteration to an unique 8-characters key. Using a mathematic formalism we introduce our method.

Let:

- k the length of password
- S the set of possible symbols with $n = |S|$
- $I = \{0...n^k\}$ the index space
- K the space of all possible passwords combinations

so we want to find:

$$f : I \to K \quad (3)$$
$$i \in I \mapsto c \in K \quad (4)$$

such that f is bijective.

The next figure shows a possible solution to determinate f based on an algorithmic approach.

Algorithm 3 Generate Combination

```
1: procedure GENERATE-COMBINATION($i$,$n$,$k$)
2:     for j=0 to k-2 do
3:         $v_j \leftarrow \left\lfloor \frac{i}{n^{k-1-j}} \right\rfloor$
4:         $i = i \mod n^{k-1-j}$
5:     $v_{k-1} = i \mod n$
```

2.2. Serial Implentation

The serial implementation have to loop over all password-space necessarily. So, the algorithm is the following:

Algorithm 4 Brute Force

```
1: procedure BRUTE-FORCE(crypted_psw, key)
2:     for $i \in I$ do
3:         new_psw = f(i) // Generate Combination
4:         if crypted_psw=crypt(new_psw, key) then
5:             found = true
```

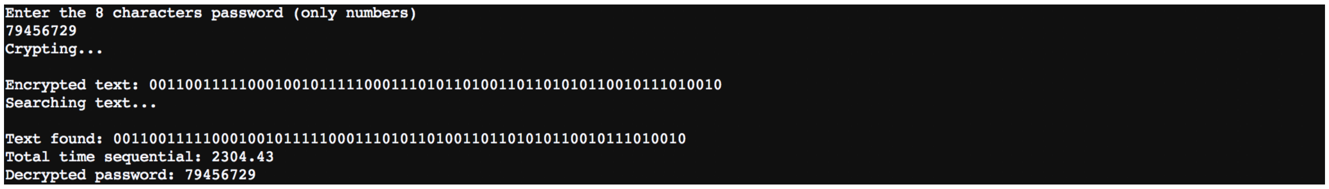
2.3. OpenMP Implemenation

The parallelism was achieved with **OpenMP** framework; more specifically we used the OpenMP *parallel-for* directive to split the password-searching process. This directive made possible to automatically distribute all the workload to a fixed number of threads. There are different loop scheduling types provided by OpenMP [1], the most relevant ones are reported below:

- **static**: Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size
- **dynamic**: Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue
- **guided**: Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations

Since we reasonably consider the workload for each iteration of the loop statement index-independent, we assumed that time-per-iteration is approximately constant. This led to choose **static** as scheduling strategy, without synchronizing any part of the code.

The parallel loop is summarised below in a general form.



```
Enter the 8 characters password (only numbers)
79456729
Crypting...

Encrypted text: 0011001111100010010111110001110101101001101101010110010111010010
Searching text...

Text found: 0011001111100010010111110001110101101001101101010110010111010010
Total time sequential: 2304.43
Decrypted password: 79456729
```

Figure 9. Example of brute-force attack.

Algorithm 5 OpenMP Brute Force

```
1: procedure OPENMP-BRUTE-FORCE(crypted_psw, key)
2:     Parallel for $i \in I$ do
3:         new_psw = gen_comb(i)
4:         if crypted_psw=crypt(new_psw, key) then
5:             found = true
```

2.4. CUDA Implementation

The second approach used to parallelize brute-force attack is *CUDA*, a parallel computing platform and application programming interface (API) model created by Nvidia. It allows to use a CUDA-enabled GPU for general purpose processing. The platform is designed to work with programming languages such as C, C++, and Fortran. This implementation has been written and developed using the C++ programming language.

The main idea is to divide all possible combination in a greater number of subset, depending on number of threads. So each thread compute a part of all possible itarations, reducing the computational cost considerably.
We have preallocated many variables necessary for the implementation of encrypting and conversion, only the values of key, preventively trasformed from 2D array to 1D array, and of encrypted text are transferred from CPU (Host) to GPU (Device). Based on number of thread, every array is preallocated with the normal size multiplied with the number of threads, because there can be conflict between different threads. Next, we call the CUDA kernel: *brute_force*.

3. Results

All the results, in terms of optimal number of threads and execution time, are obtained on a machine with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz (total 24 cores) and a Nvidia Titan X GPU.

We recall that, for computational time reasons, results were obtained using 8-characters password with only numbers so that limit size of search space.

3.1. Number of threads

First, for each parallel technology, the implementation has been tested with different numbers of threads (from 1, corresponding to serial implementation, to 4096 - and even 100000000 with CUDA), in order to evaluate the performances in each configuration and the number of threads that yields the best results.

With the OpenMP technology, the best results are obtained using 4 threads in a machine that has 24 cores in the CPU, meaning 2.6 threads per core (See Fig. 10).

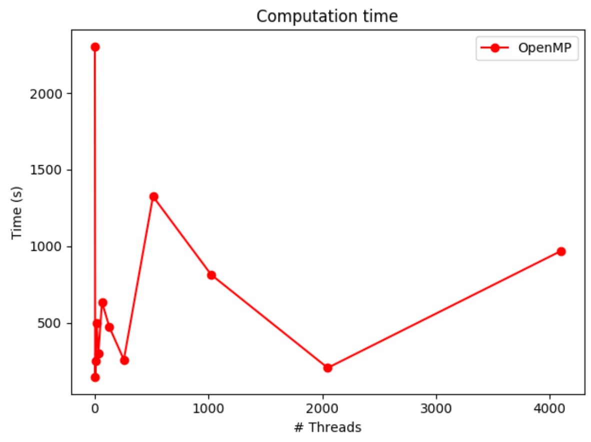


Figure 10. Computation time OpenMP.

With the CUDA technology, the best results are obtained using at least a thread per combination in each parallel task. Infact, using the 8-character password (10^8 combinations with only number), the best performances are achieved using 10^8 threads. Decent results, but far from the best, are obtained with 8 threads (See figure 11).

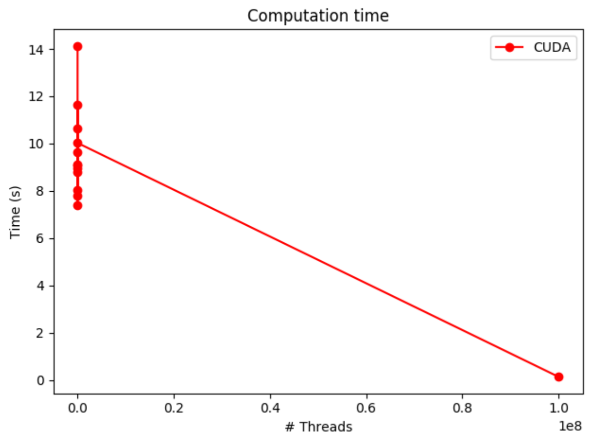


Figure 11. Computation time CUDA.

3.2. Serial vs Parallel

Once evaluated the optimal number of threads for each implementation, the different approaches have been compared across various combinations with increasing difficulty.

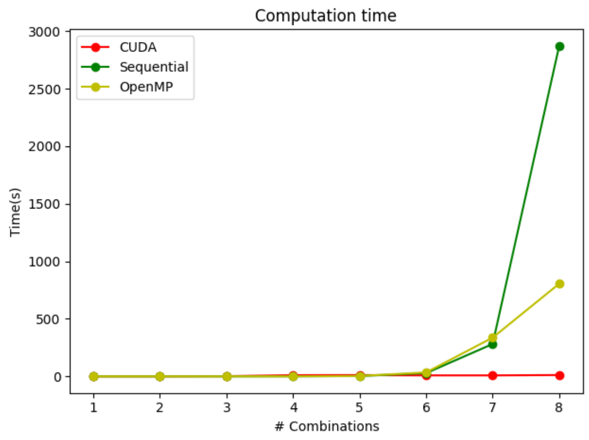


Figure 12. Serial vs OpenMP vs CUDA with various combinations with increasing difficulty.

As shown in Fig. 12 the serial approach is the slowest compared to all the other parallel approaches. In general with easy passwords the serial implementation has comparable performances with the other approaches, but from a certain difficulty it is outperformed.

The fastest implementation in every circumstance is CUDA with as many threads as combinations, confirming what the modern trend for scientific computation is highlighting: GPUs are suited for these tasks.

3.3. Speed Up

At this stage particular emphasis was given to the speedup concept to figure out if the taken multithreading approach brought benefits.

The speedup S_p is defined as $S_p = \frac{t_s}{t_p}$ where, P is the number of processors, t_s is the completion time of the sequential algorithm and tp is the completion time of the parallel algorithm [2].

In Fig. 13 and 15 is shown the speed up obtained with the different parallel approaches with relation to the serial algorithm.

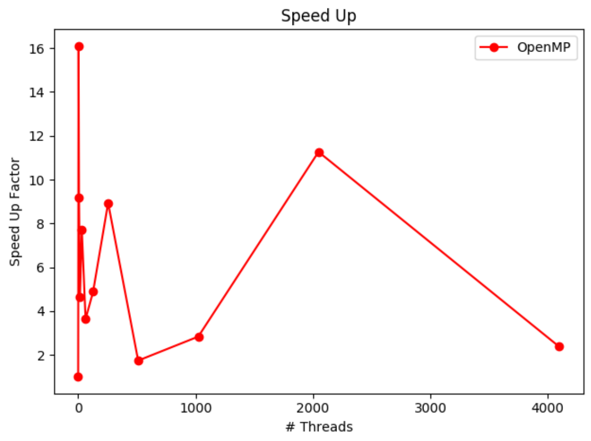


Figure 13. Speed Up OpenMP with fixed combination but varying number of threads.

4. Conclusion

We described a OpenMP and CUDA based method to exploit parallelism in the search process of a DES encrypted password.

The obtained results are consistent with what was expected, being aware of the limits of our machine and size of searching space. Since our procedure operates on indipendent chunks of the password-space, we didnt need to syncronize any part of the code. So, we can reasonably assume a sub-linear speedup trend increasing the number of processors.

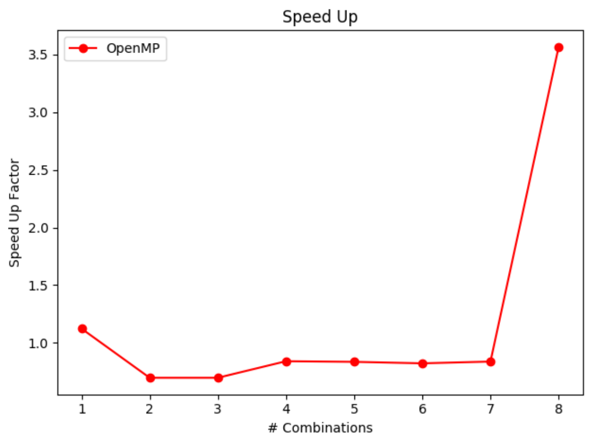


Figure 14. Speed Up OpenMP with fixed thread but trying more combinations with increasing difficulty.

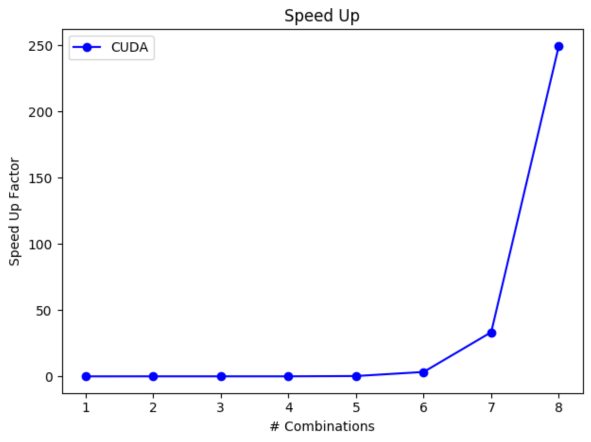


Figure 15. Speed Up CUDA with fixed combination but varying number of threads.

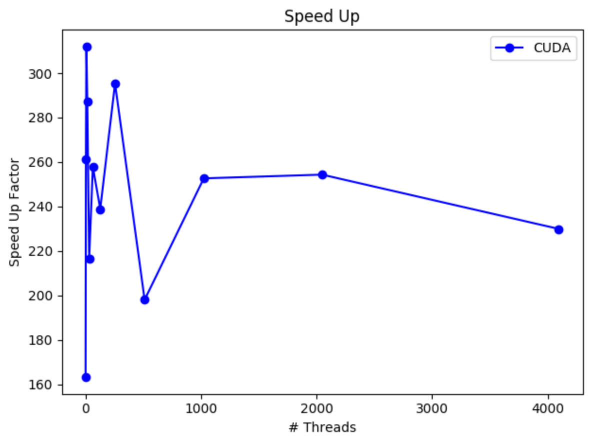


Figure 16. Speed Up CUDA with fixed thread but trying more combinations with increasing difficulty.

References

[1] Intel. Openmp loops cheduling. https://software.intel.com/en-us/articles/openmp-loop-scheduling.

[2] L. S. C. Lin. *Principles of parallel programming*. Pearson.

[3] Wikipedia. Data encryption standard. https://it.wikipedia.org/wiki/Data_Encryption_Standard.