UNIVERSITÀ DI PISA

MSc in Computer Engineering

Computer Architecture Project

Parallelized Brute Force Attack on AES key using CUDA-C

TEAM MEMBERS:

Tommaso Bertini
Fabrizio Lanzillo
Federico Montini

`https://github.com/FabrizioLanzillo/Parallelized-AES-Brute-Force-Attack-with-Cuda`

Anno Accademico 2022-2023

Summary

1 Introduction

The Advanced Encryption Standard (AES) is a symmetric cryptographic algorithm, born in 2001, when a couple of Belgian cryptographers (Joan Daemen and Vincent Rijmen) submitted a proposal to NIST during the AES selection process. This algorithm is a variant of the Rijndael block cipher family using different key length and block size.
Actually AES is still considered as one of the most secure algorithm and is widely used thanks to its advantageous trade-off between performances and security. There are three main versions of this standard, each with a different number of rounds and key lengths: 128, 192, 256 bits for 10, 12, or 14 rounds respectively and the most efficient attack for key-recovery is only 4 times more efficient than exhaustive search.
Another aspect that has to be considered to understand its security are the Encryption modes: AES has different encryption modes, starting from the simplest and the least secure one, which is *ECB*, to more complex ones such as *CTR* or *CFB*.
In our project we choose the **CBC** version since it is the most used one and allows us to parallelize the decryption. In the following picture we can see, from an external point of view, its structure:

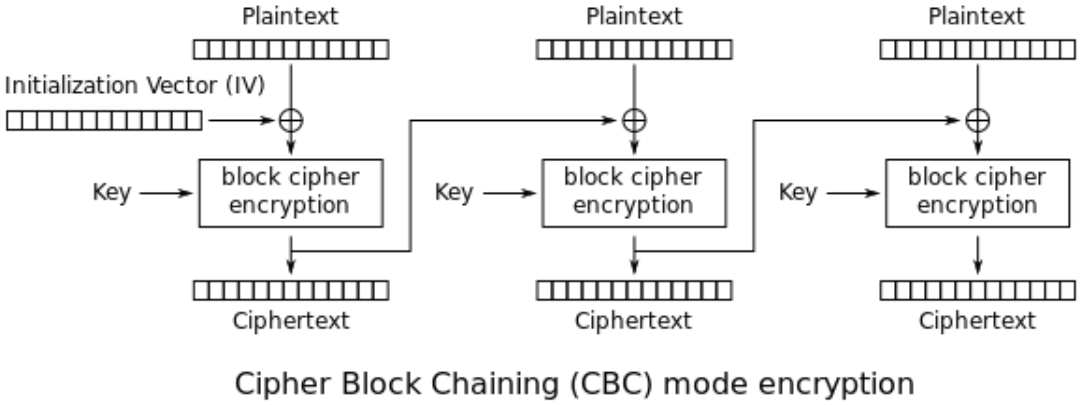

Figure 1: AES CBC Encryption Block scheme

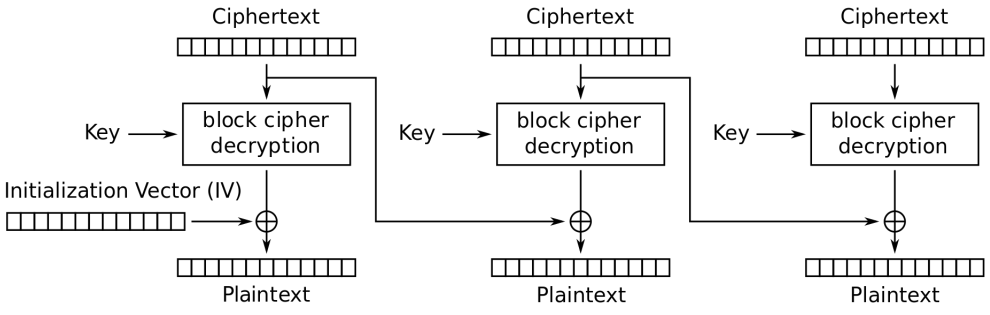

Figure 2: AES CBC Decryption Block scheme

1.1 The algorithm

Let's proceed with analyzing what AES needs in order to perform encryption of a given plaintext and what steps are required to perform it.
It needs:

- **Key** of 128, 192 or 256 bits
- **Plaintext** of any length
- A **nonce**, called IV, fresh for every new plaintext

AES as we stated in the previous paragraph is a block cipher, at the current state it encrypts (and decrypt) the plaintext dividing it in block of 16 Bytes, to understand what happens if the plaintext isn't multiple of the block size look for PKCS 7.
In Figure 3 we can observe the internal composition for the decryption operation:

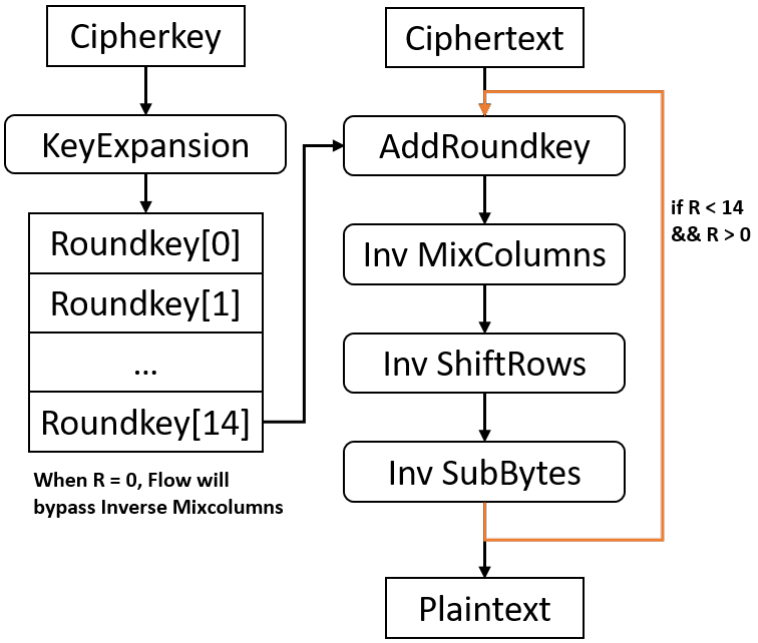

Figure 3: Round scheme of AES-256

- The AES **key expansion** algorithm takes as input a eight-word (32-byte) key and produces a linear array of 60 words (240 bytes). This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the 14 rounds of the cipher
- In the **AddRoundKey** step, the subkey that corresponds to the current round is combined with the state.
- In the **MixColumns** step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes
- The **ShiftRows** step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset

- In the **SubBytes** step, each byte $a_{i,j}$ in the state array is replaced with a SubByte $S(a_{i,j})$ using an 8-bit substitution box.

1.2 Reasons behind AES parallelization

Nowadays, as the computational capabilities available through common laptops and desktops grow, it is necessary to question the security properties of various cryptographic algorithms. As we have seen in the previous section the most efficient algorithm for key-recovery, at the moment, is brute-forcing itself, to understand the feasibility of this attack we are going to implement various versions for this attack, whose performances will be measured during their execution on various architectures, we will start with a simple example of execution on the CPU without taking advantage of the multi-threading mechanism, then to a version with multi-threading to end with the version parallelized and executed on GPUs. Our goal will be to optimize the code, which will be written in Cuda-C, in order to retrieve a key portion of about 30 bits (1,073,741,824 possible combinations) in **less than 2,5 s**.

2 CPU implementation

In order to identify which architecture is the best to perform bruteforcing, we started with the simplest: a C++ version that does not exploit the multithreading mechanism. This version is fully sequential, a single thread cycles over each key attempting, at each iteration, to perform decryption and checks if the plaintext obtained match with the expected one, if they correspond the program terminates.
The known key portion is given to the bruteforce function that will handle it by concatenating to the known portion an index that varies, via a for loop, between 0 and the maximum number of iterations (ex. 2^{30} for 30-bits) via an increment. The concatenation operation, however, consists of two subphases (Figure 4): a first in which the missing bit portion is added to the MSB and a second in which the missing whole bytes are concatenated via a memcpy.

```
// First copy the bytes that are whole
for(int j=0;j <  numcycles; j++){
    //This part must be executed only if there is a part of a byte remaining to be inserted (like last 4 bits in case of 20 bits)
    if(num_bits_to_hack % 8 != 0 && j == num_bits_to_hack/8){
        //The addition of unsigned number perform the append correctly until the value inside pointer[j]
        // overcome the capacity of the bit to be copied, but this will never happen since we stop the cycle before it happen
        bytes_to_hack[j] = tmp + pointer[j];
        continue;
    }
    ascii_character = char(i >> (8*j));
    sprintf((char*)&bytes_to_hack[j],"%c",ascii_character);
}

// we assemble the key with the new character, cycle needed to order the bytes in the correct way,
// otherwise it will result in a swap of the cycled bytes
for (int j = 0; j < (num_bits_to_hack/8) + 1; j++){
    if(num_bits_to_hack % 8 != 0){
        memcpy(&hacked_key[AES_KEY_BYTES_LENGTH - j - 1], &bytes_to_hack[j], 1);
    }
    else if(j < (num_bits_to_hack/8)){
        memcpy(&hacked_key[AES_KEY_BYTES_LENGTH - j -1], &bytes_to_hack[j],  1);
    }
}
```

Figure 4: Creation of the key for this iteration

The AES implementation that has been used is the one derived from the block-scheme. Is also important to notice that from now on, all the time samples are obtained measuring the execution of the whole decryption function, including the portion needed to attempt the decryption since it is necessary to discriminate the correct key from the other. All the timestamps are obtained using the **chrono library** and we made 10 repetitions for each configuration and extracted a 99% confidence interval for each measurement.

2.1 CPU configuration

From now on all the obtained results that regards the CPU versions are executed on a machine with the following architecture

- Processor: AMD Ryzen™ 5 5000 G-Series Desktop with 6 cores and 12 concurrent threads and a base clock of 3.9GHz.
- RAM: 16GB
- L2 Cache: 3MB
- L3 Cache: 16MB

2.2 CPU Version - Fully sequential version

In the following picture can be seen the results obtained from this kind of implementation. Like we expected the result are really far away from what is our objective.

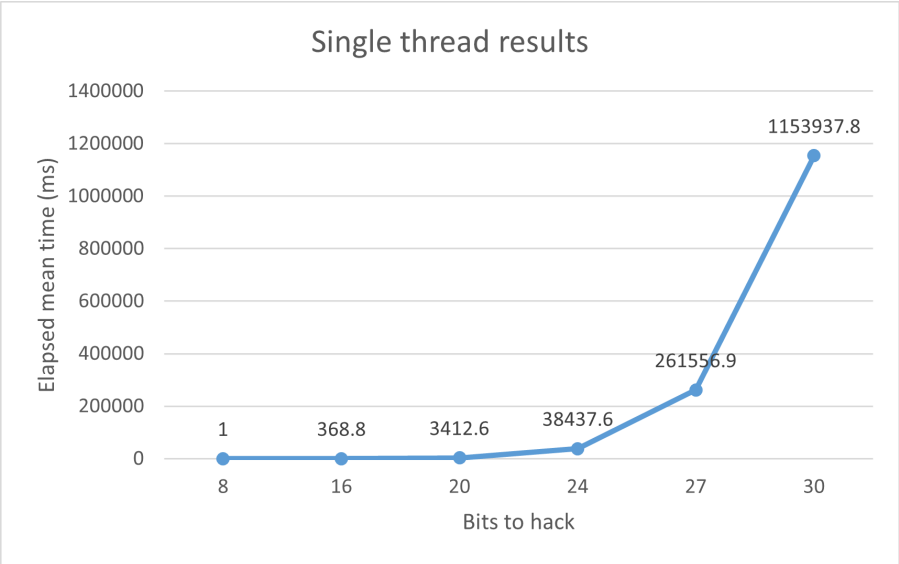

Figure 5: Results on CPU - Fully sequential version

If our objective was to remain under 2,5 s, the results shows that, in the worst scenario (i.e. 30-bit) we need an amount of time that's near **20 minutes of execution**. Looking at the curve can be seen the main problem that makes the bruteforce unfeasible on a CPU: the time required grows exponentially with the length of the key to discover. We expect this trend also on our optimized GPU version but for way bigger key bit-lengths.

2.3 CPU Version - Multithreaded version

This version is a slightly modified sequential version. The various threads, which will be allocated by the main thread, will take a portion of the key space analyzing every possible combination in that space, the main thread will suspend waiting for one of the children's signal allowing the child threads to take advantage of the maximum power from the processor. The reduction in the space in which each thread can execute, coupled with the parallelization they can exploit leads to expect a significant improvement in the timing for key-retrieval. We implemented the multi-thread version using **thread library** and tested various configurations for threads number.

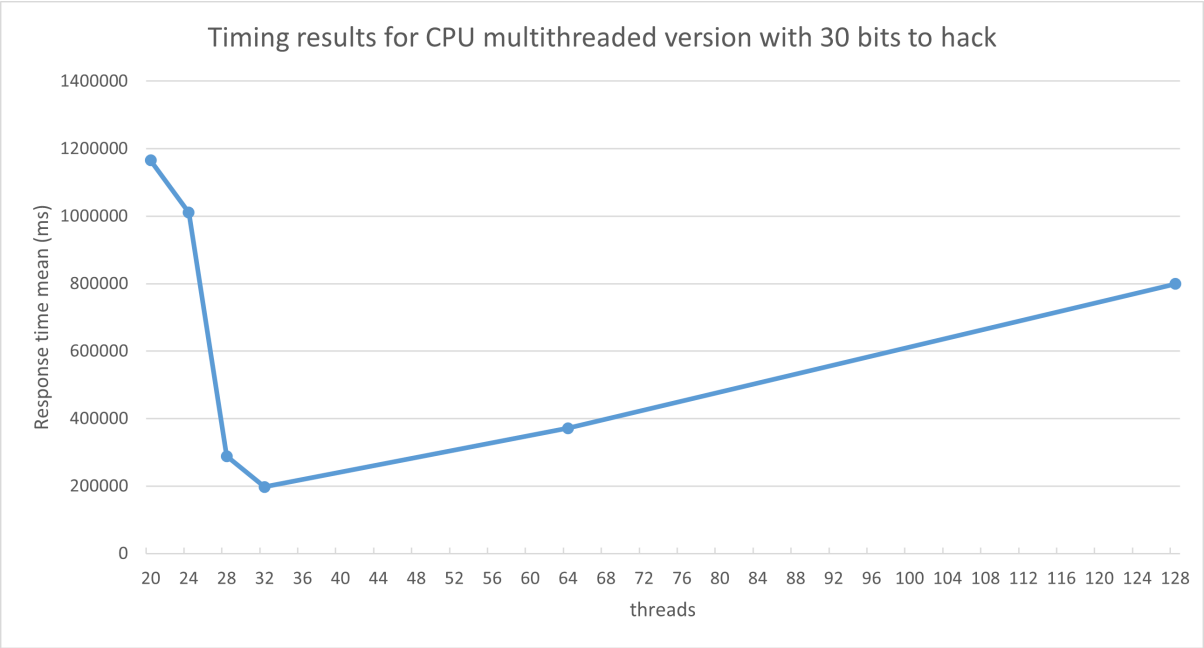

Figure 6: Results on CPU - Timing results variating the amount of threads with 30 bit to hack

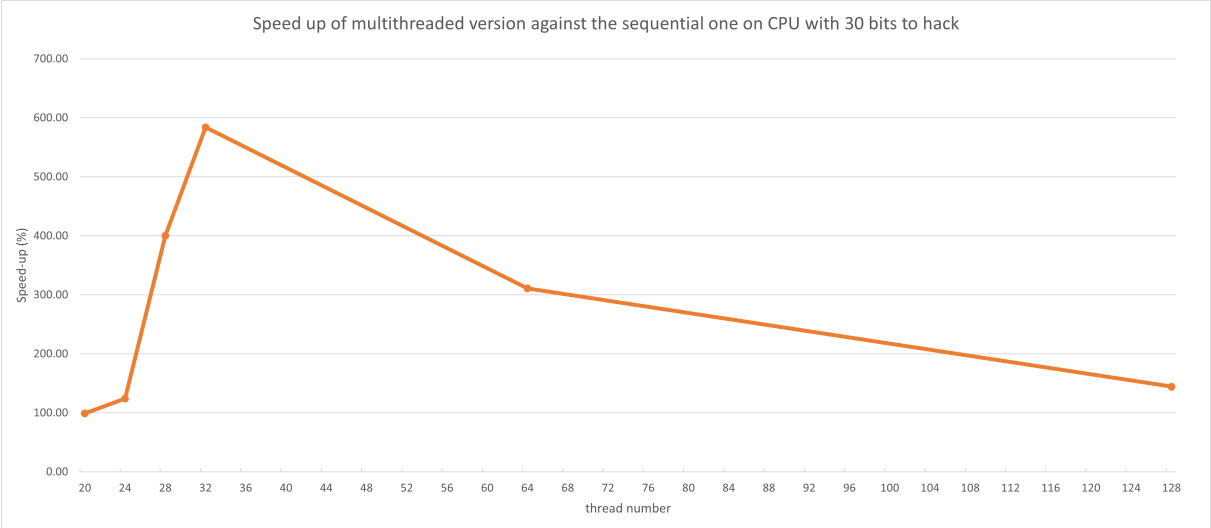

Figure 7: Results on CPU - Speed-up results variating the amount of threads with 30 bit to hack

In 6 and 7 can be observed the the timing results of the multithreaded version and its speed-up w.r.t the single-threaded version seen earlier. Looking at the graphs, it is clear that the system benefits from the key-space division. After a slow start the curve gain speed-up after that at least 20 threads are allocated, generating a major speed-up to **saturation**, which occurs around 32 threads. After 32 threads the advantage we gain by further dividing the key space is masked by the overhead required to schedule the other threads that will proceed in time sharing.
In any case, comparing the best result obtained with 32 threads with what we aim to achieve, we immediately observe that we are far from what we hope for. In fact, the

version with 32 threads returned timings on the order of 3 to 4 minutes, being at least two orders of magnitude away from the expected result. In order to get as close as possible, we decided to change architecture, thus **switching to the vectorial architecture of a GPU**.

3 GPU Implementation

In the implementation using the GPU, the primary goal was to figure out what was the **best trade-off** between **the number of blocks** and **the number of threads per block**, in order to **optimize the device utilization**.
Indeed, by changing these parameters, **the computational load for each individual thread is reduced**.
Thus, we can figure out how much is it possibile to increase the maximum number of threads (number of blocks multiplied by the number of threads per block) in order to achieve the best configuration.

Specifically, the computational load, mentioned above, is computed by dividing the maximum number of iterations (ex. 2^{30} for 30 bits) by the maximum number of threads.
This value will obviously change depending on the parameters chosen during the different tests.

3.1 GPU Configuration

The results obtained from the GPU version are based on a Device with the following characteristics:

- **Device**: "**NVIDIA GeForce GTX 1050**"
- **Streaming Multiprocessors (SM)**: **5**
  - **CUDA Cores per SM**: **128**
  - **Maximum number of threads per SM**: **2048**
  - **Concurrent Warp per SM**: **64**
- **Warp size**: **32**
- **Maximum number of threads per block**: **1024**
- Total Number of CUDA Cores: 640
- GPU Max Clock rate: 1493 MHz (1.49 GHz)
- L2 Cache Size: 524288 bytes
- Memory Clock rate: 3504 Mhz
- Memory Bus Width: 128-bit
- Total amount of shared memory per block: 49152 bytes
- Total shared memory per multiprocessor: 98304 bytes
- Total amount of global memory: 2001 MBytes (2098331648 bytes)
- Total number of registers available per block: 65536
- CUDA Driver Version / Runtime Version: 12.0 / 12.0

3.2 GPU Version - Initial Results

As mentioned earlier in the analysis of the CPU version, **our goal is to reach a maximum execution time of 2.5 seconds**, in the worst case scenario (i.e. 30-bit).
In order to get as close as possible to the target threshold, we measured the **execution times of different tests**, where each of them **performs a different combination of the parameters** i.e., the number of blocks and the number of threads per block, in order to find the best trade-off of these two parameters.
In particular, there are a total of **28 tests** where we gradually increased the level of parallelization of our architecture. For each of these tests 10 repetitions were performed, extracting a 99% confidence interval for each measurement.
The tests are grouped in **4 steps**:

1. In the first step we performed **5 different tests**, where we **used only one of the SM available on the device**, and to do this, we declared **only one block**, since a block is scheduled on only one SM and it remains there until the execution is completed. The only difference among the tests was **the number of threads per block**, that were **progressively increased**.
   The 5 tests performed are (blocks, threads):
   - 1, 32 (base case)
   - 1, 64
   - 1, 128
   - 1, 256
   - 1, 512 (maximum number of threads per block in our case)
2. In the second step, **4 tests were performed** and in these we increased the number of blocks in order to **use all the SM**(the total number of SMs on the device is 5) on the device.
   The 4 tests performed are (blocks, thread):
   - 2, 512
   - 3, 512
   - 4, 512
   - 5, 512
3. In the third step, **3 tests were performed** and in these **the number of blocks were increased proportionally to the number of SMs**, with the purpose of reaching the maximum number of concurrent warps.
   As we can see from our device specification, the maximum number of concurrent warps per SM is 64 and the warp size is 32. Therefore, the maximum number of concurrent threads per SM is 64*32=2048, which is also a data given in the device specification.
   Hence, given that our device has 5 SMs, the maximum number of concurrent threads in the device is 2048*5 = 10240.
   By setting the number of threads per block to 512, which is the maximum in our case, we get 10240/512=20 blocks.

To summarize, **the maximum number of concurrent warps in our device is obtained using 20 blocks and 512 threads**.
The 3 tests performed are (blocks, thread):

- 10, 512
- 15, 512
- 20, 512

4. In the fourth step, **16 tests were performed** and we **exponentially increased the number of blocks**, to get the best combination of number of blocks and number of threads per block.
   The best combination of these parameters, also takes into account the computational load of each thread.
   This is because **the more we increase the level of parallelization, the more the load of each thread decreases**, so the best solution is not necessarily the one with more threads and blocks because maybe we could lose more time in overhead by context change.
   The 16 tests performed are (blocks, thread):

   - 32, 512
   - 64, 512
   - 128, 512
   - 256, 512
   - 512, 512
   - 1024, 512
   - 2048, 512
   - 8192, 512
   - 16384, 512
   - 32768, 512
   - 65536, 512
   - 131072, 512
   - 262144, 512
   - 524288, 512
   - 1048576, 512

We can see, in the plot in the figure 8, how by means of all these tests we managed to **approach the 2.5-second threshold**.

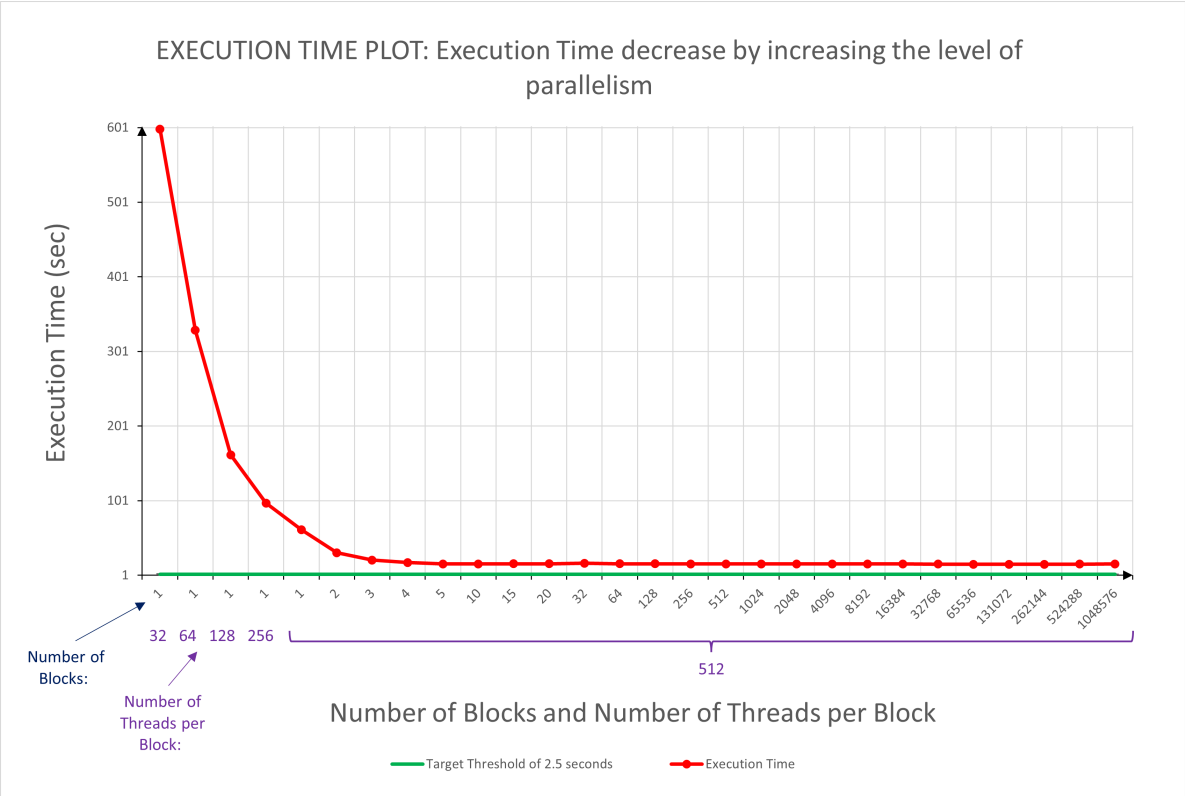

Figure 8: Execution Time Plot: Execution Time decrease by increasing the level of parallelism

If we analyze the execution time in terms of speed-up, as we can see in the figure 9, we can better analyze the behavior of our architecture as the steps change.
The speed-up of each test is computed against the base case represented by the first test in the first step.
The formula by which it is calculated is the following:

$$Speed - Up\ Test\ \#n = \frac{Mean\ Execution\ Time\ Test\ \#n}{Mean\ Execution\ Time\ Test\ \#1}$$

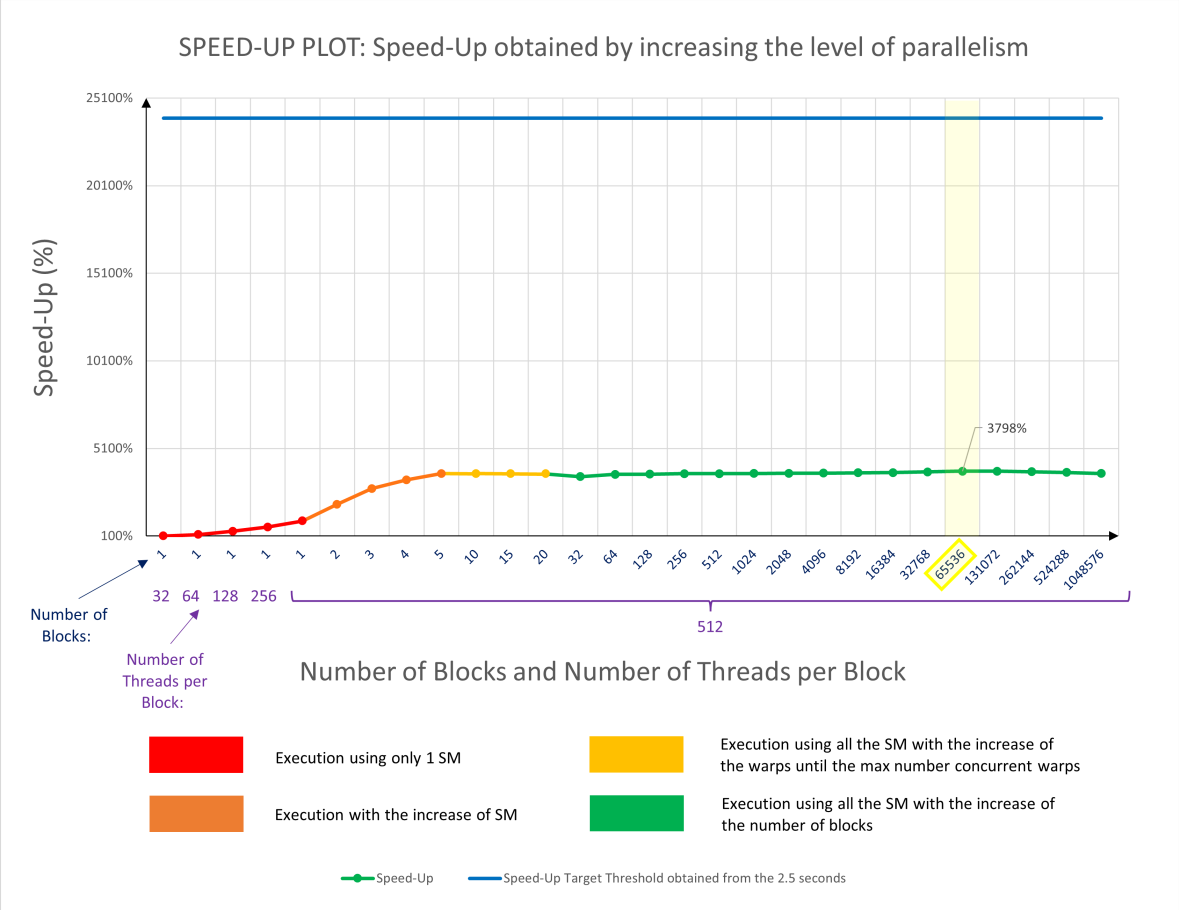

Figure 9: Speed-Up Plot: Speed-Up obtained by increasing the level of parallelism

The problem, however, is that despite the fact that **we have greatly improved the CPU performance from about 20 min execution time to about 15.77 seconds**, which is the best result, obtained in the test where we used 65536 blocks and 512 threads per block.
However the execution time trend does not improve beyond this value, even though the number of total threads increases.
To better understand this behavior, we took advantage of some profiling tools.

3.3 GPU Version - NVIDIA Nsight System Optimizations

To achieve a better performance, **we analyzed our configuration by using some profiling software**. The first software we used is **NVIDIA Nsight system** that is a system-wide performance analysis tool designed to visualize an application's algorithms, and identify the largest opportunities to optimize.
Through this software we saw that with the adopted configuration **the interaction with memory was well handled** as shown in the figure 10, because the **transfers occurred before and after the kernel call**.

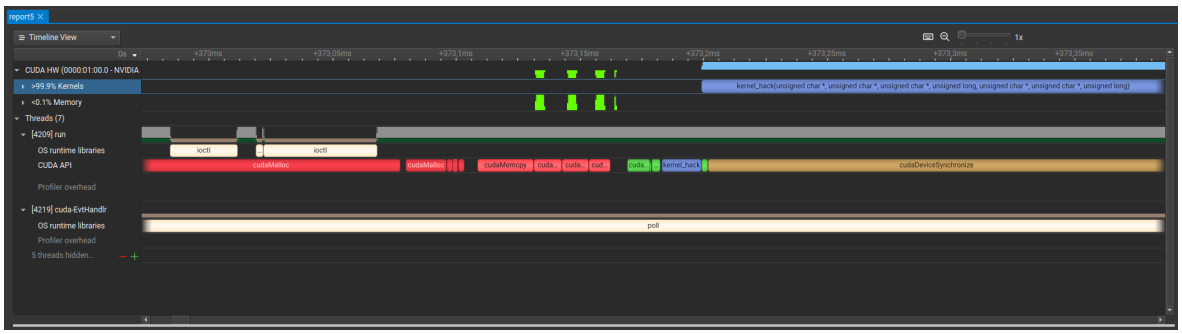

Figure 10: NVIDIA Nsight System: Memory Transfer Analysis

Moreover, we also **found a problem that we thought was causing the slow down of our configuration**. In fact, as we can see in the figure 11 right after the kernel call, the "**cudaDeviceSynchronize**" function was invoked, taking up **almost all of the execution time of the kernel itself**.

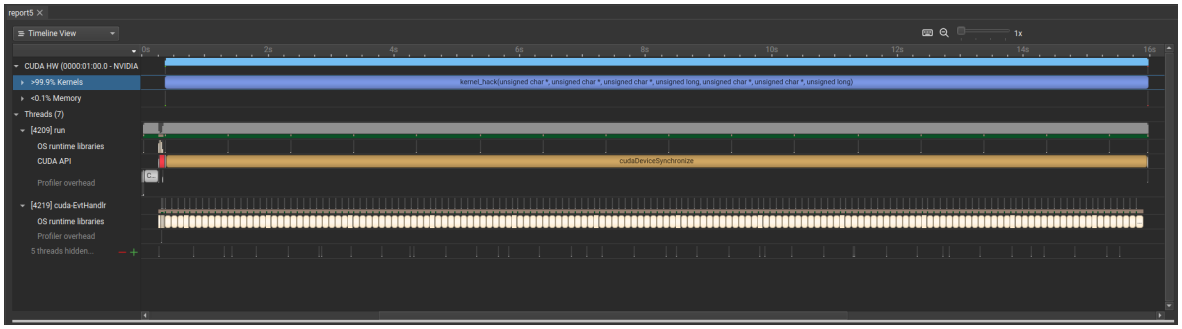

Figure 11: NVIDIA Nsight System: CudaDeviceSyncronize Analysis

The reason for this behavior relates to the fact that **when the key**, suitable for decrypting the chipertext, **is found by a thread, the thread, since it cannot communicate with the other threads in the other blocks, cannot notify them to stop their execution**.
Therefore, it is necessary to wait for all other threads to finish their execution, and this **is highly inefficient**.
To solve this issue, we introduced a boolean variable "hack over" in the **global memory space of the GPU**.
Since it is in the global memory space, **all threads in all blocks can access it**. Whenever a thread needs to try a new key first, the variable "hack over" must be checked.
If this is equal to "True", it means that the key has already been found and we need to stop execution, otherwise we need to continue searching for the key.
With this method **we don't even have to worry about issues related to simultaneous write accesses to the variable**, because since there is only one key that can decrypt the chipertext, **only one thread can change the value of that variable** and in addition, this operation will only happen once.

After this optimization we were able to **get a significant improvement in the performances**, which is clearly evident from the following Speed-Up plot in the figure 12, **which emphasizes the difference with the previous non-optimized version**.

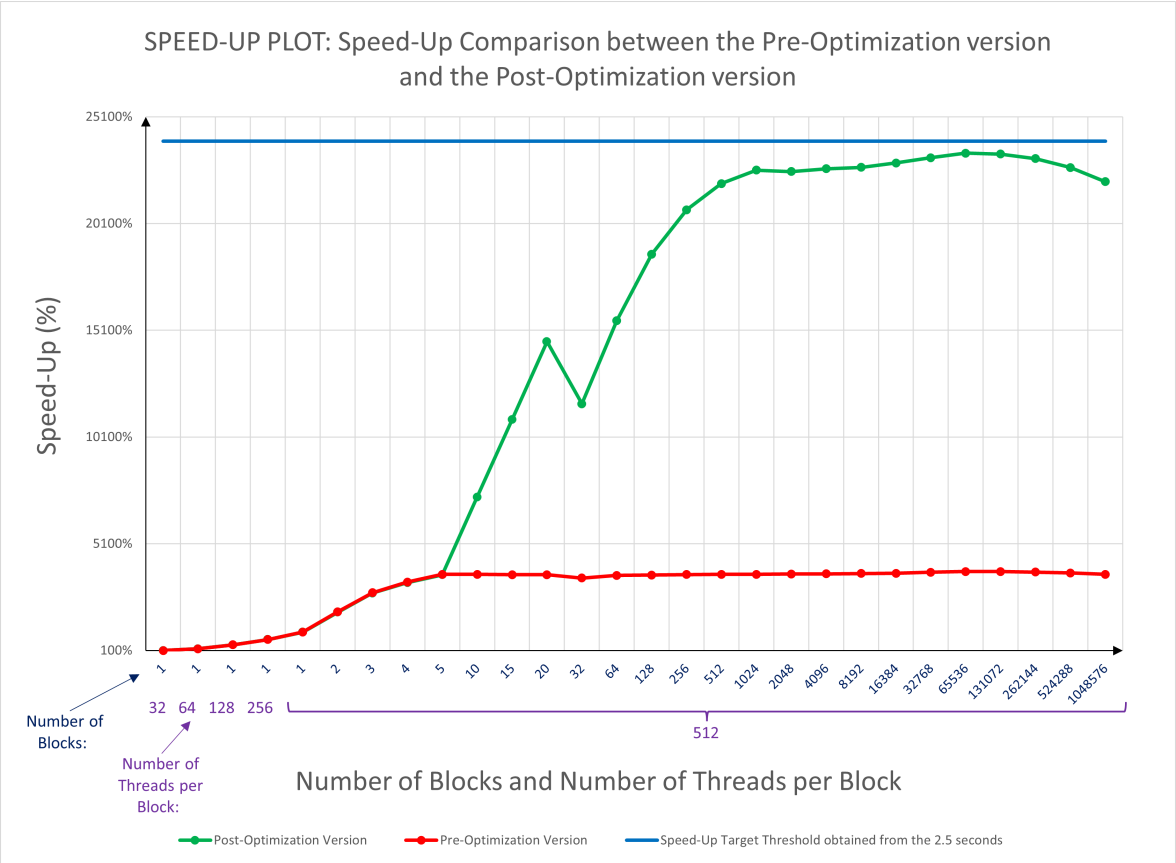

Figure 12: Speed-Up Plot: Speed-Up Comparison between the Pre-Optimization version and the Post-Optimization version

As we can see from the plot in the figure 12, **the speed-ups of the two versions deviate from each other after we increase the number of blocks for each MS**. In fact, if we **do not adopt a policy to stop the threads after the key is found, will be very hard to decrease the timing enough to reach the goal**. This, because even if we decrease the computational load for each thread, through the parallelization, all threads will still be executed, so the overall time remains the same.

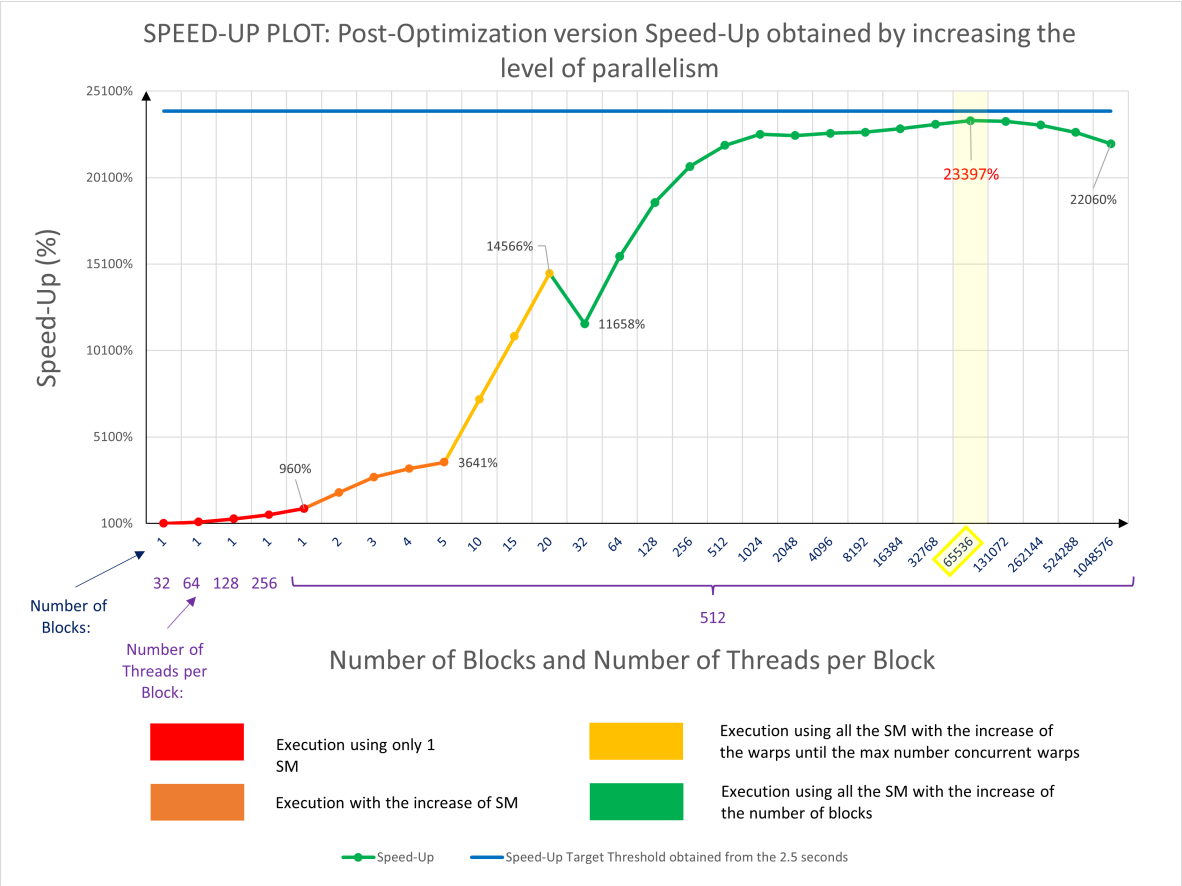

Figure 13: Speed-Up Plot: Post-Optimization version Speed-Up obtained by increasing the level of parallelism

From this plot in the figure 13 we can see that we achieve a better speed-up in **phase 2** (orange) **than in phase 1** (red), this is because by **increasing the number of SMs** the **level of parallelization also increases**.
In **phase 3** (yellow) we have a **real performance boost**, because we **add blocks proportionally to the number of SMs** until we reach the maximum number of concurrent threads.
In **phase 4** (green) we first have a performance decrease, because we increase the number of total threads, but the **load per thread is still high**, and since **the total threads are not all concurrent** we have a performance decrease.
However, as we can see in subsequent tests that the **Speed-Up continues to increase**, with the goal of finding the **best trade-off** between number of blocks, number of threds per block, and computational load per single thread.

Again from this graph we can see that the **best performance** is achieved when we use a **configuration** where the **number of blocks is 65536 and the number of threads per block is 512**.
In fact, with this configuration we were able to **reduce the execution time from a mean value of 15.77 seconds to one of 2.55 seconds**.
So we are **very close to the goal of reaching the threshold of the 2.5 seconds** but we have not accomplished it yet.

3.4 GPU Version - NVIDIA Nsight Compute Optimizations

Since the achieved timelines still do not meet our initial goal, we continued the analysis also using the tool provided by Nvidia: "**Nsight Compute**". In order to understand what problems plagued the code, we performed profiling of the optimized version through Nsight System and obtained the following results:

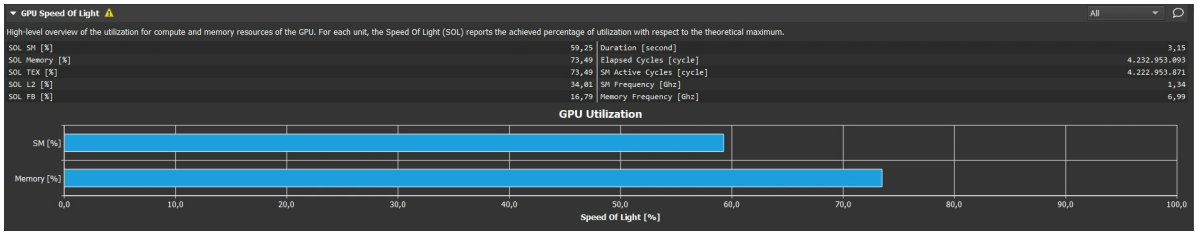

Figure 14: Nsight Compute: SOL Section before optimizations

Taking a first look at the results, we immediately observe how the use of resources, although slightly higher in the memory compartment, are quite high and balanced. This indicates to us how the GPU resources are really being exploited by the algorithm. This section, in any case indicates to us that the component that is likely to generate a bottleneck is the memory.

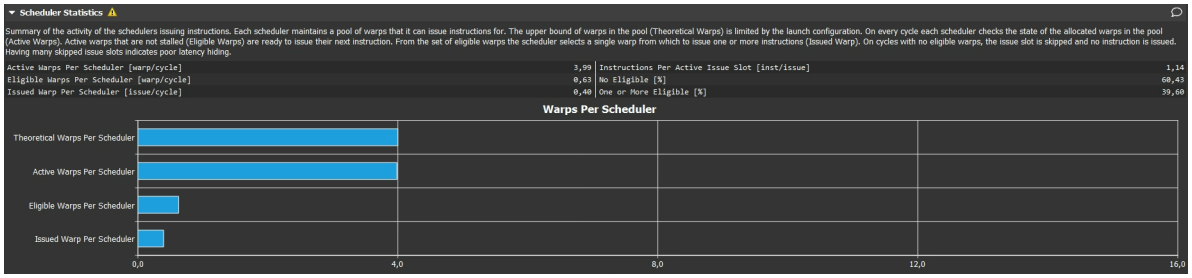

Figure 15: Nsight Compute: Scheduler statistics sections before optimizations

Another section that shows criticality is the **scheduler statistics** section. We can immediately observe that for more than 60 percent of the samples obtained during profiling, there is not a single warp available to run. This deadlock, probably due to the bottleneck on memory seen earlier, is very relevant to execution times since, despite the large level of parallelization, the execution slots that are actually exploited are about 40% of the total.

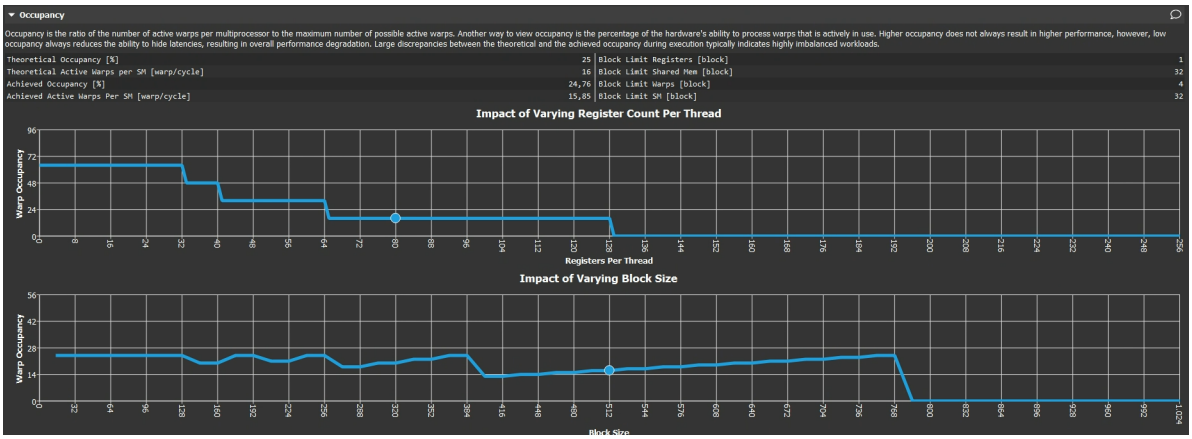

Figure 16: Nsight Compute: Warp occupancy section before optimizations

As a final relevant aspect, there is **warp occupancy**. This section lets us know how many warps are actually active on an SM versus how many it can actually support and is very important as it clearly defines the current level of parallelization. As can be seen in this version the occupancy reaches just 25% of the GPU's theoretical potential, from the graphs below we can also observe the cause: **the number of registers assigned to each thread**. In the optimized version we will have to go and redefine the amount of registers which, at the cost of a few more data transfers, will guarantee a level of parallelization much closer to the theoretical maximum.

3.4.1 GPU Version - Optimizations

In order to improve performance, we modified the code in several places trying to optimize and improve parallelization and memory organization. To optimize the number of registers, we used the **__launch_bounds__()** function, which through two parameters **MAX_THREADS_PER_BLOCK** and **MIN_BLOCK_PER_SM** estimates the amount of registers per block that maximize the warp occupancy.
As for the memory bottleneck, on the other hand, we encountered more difficulty. According to what the Source counter section states, the large memory usage is due to Uncoalesced Accesses in the global memory.

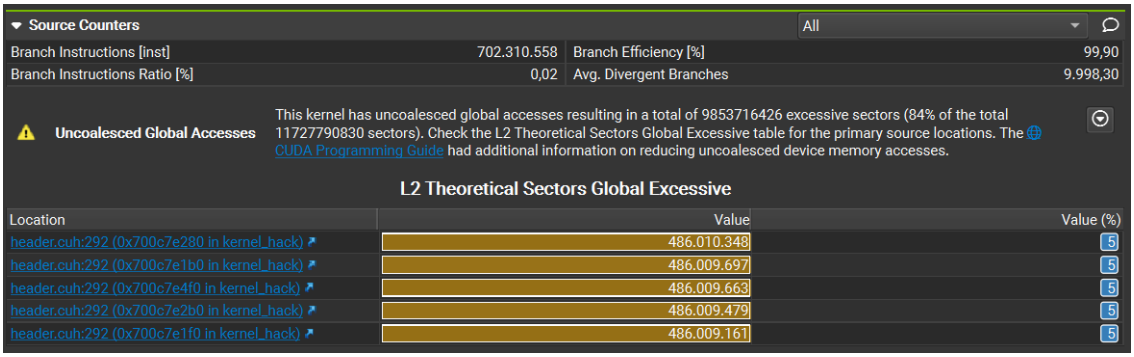

Figure 17: Nsight Compute: Source Counter section before optimizations

The instruction that causes this type of access was identified through Compute by linking the source code and its disassembled (SAAS) and found to be the one in figure 18.

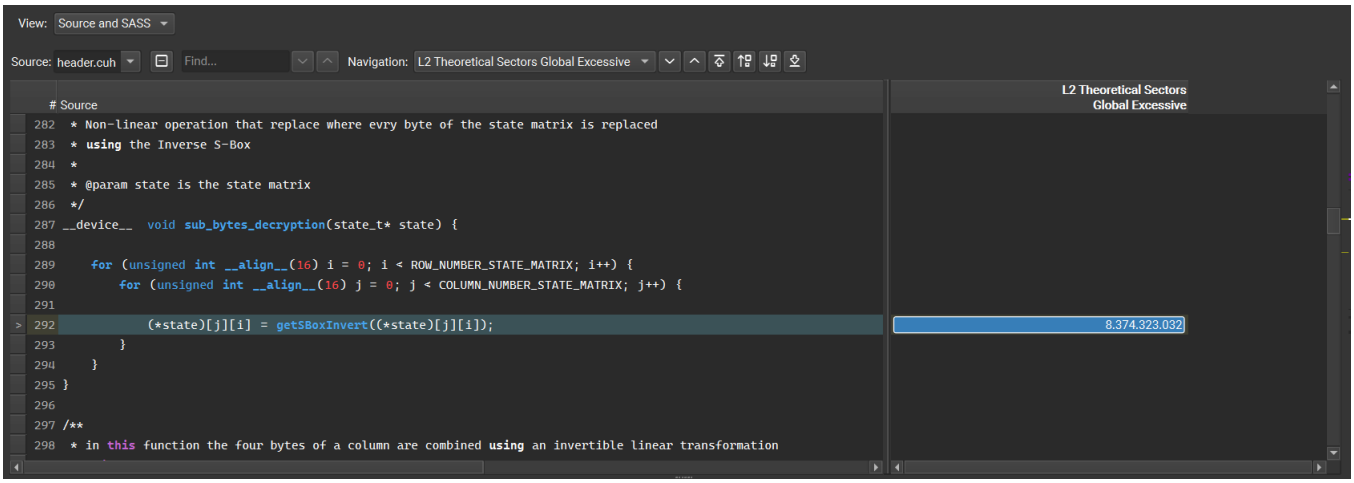

Figure 18: Nsight Compute: line that generates the uncoalesced accesses

After a thorough study of the algorithm, we learned that access to the S-Boxes (which is what this line of code is concerned with) is done in a sparse manner based on the value contained in the state matrix, which varies round by round based also on the ciphertext to be decrypted. Turnend out that there is no easy way to reduce this kind of uncoalesced accesses without changing the algorithm itself, we ended up trying a different approach to optimize the memory bottleneck.

In order to improve memory accesses, we inserted keywords within the code that would allow the compiler to make, at compile time precisely, optimizations:

- **__restricted__** keyword: This keyword allow to specify if a specific parameter of a device function will be a read-only value
- **__align__** keyword: This keyword, added during a variable declaration, allows to align them and make the access time needed to read/write those variables way smaller

In the end we reorganized the variables declarations (especially the big ones) in order to reduce the number of useless copies and organize the memory in order to fit it in the smallest space possible.

3.5 GPU Version - NVIDIA Nsight Compute Results

After the optimization we ended up obtaining the following results.

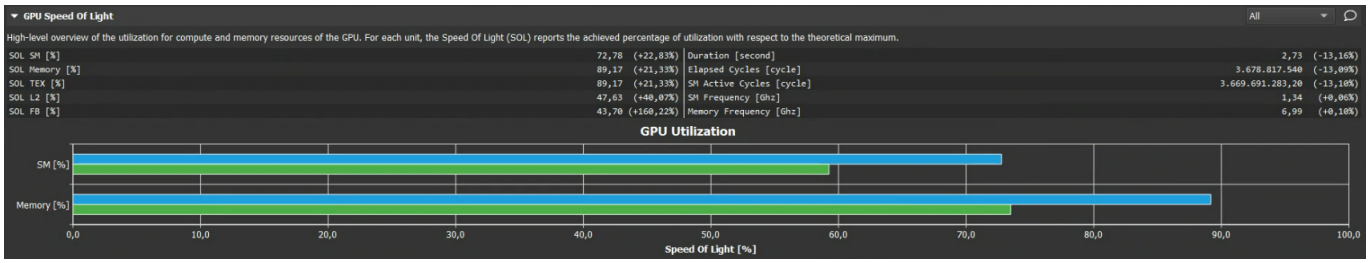

Figure 19: Nsight Compute: SOL section before optimizations

As we can see from the previous image extracted on the optimized version of the algorithm, we have achieved several improvements. The resource utilization has been increased by more than 20% for both SM and Memory section leading to a -13% for the execution time.

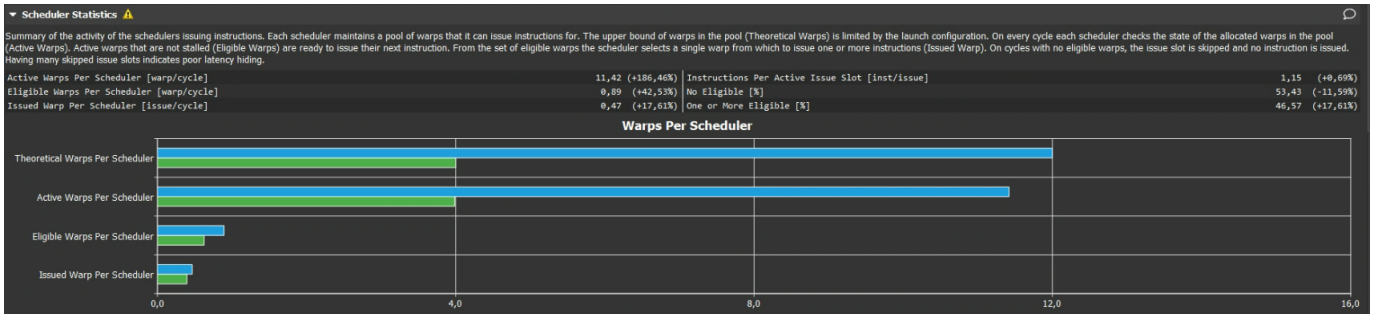

Figure 20: Nsight Compute: Scheduler statistics section before optimizations

Regarding scheduler statistics, on the other hand, we can observe an increase of more than 180% in active warps per scheduler and a better, though not optimal balance of issue slots exploited (+17%).

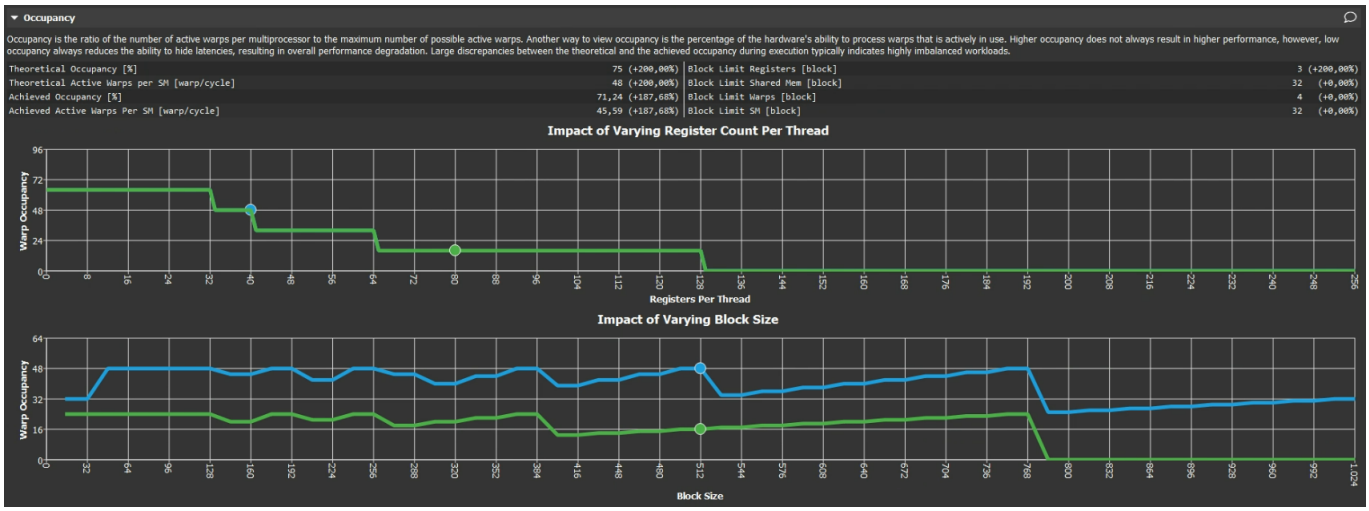

Figure 21: Nsight Compute: Occupancy section before optimizations

Finally going to look at the warp occupancy the best results obtained are what we can observe in figure 21. We achieved a 200% increase on theoretical and actual warp occupancy thus increasing the level of parallelization. The only observation is, why did we stop at 40 registers and not go below that? For example, 32 registers seems to be the amount that optimizes warp occupancy. This has been done by setting the values in the launch_bounds equal to 512 for MAX_THREADS_PER_BLOCK and 4 for MIN_BLOCKS_PER_SM. But largely higher timings were observed even compared to the unoptimized version, so we took a step back by stopping at **40 registers**.

4 Conclusion

After the Nsight Compute analysis we reached out goal, the time required to locate the key by having to hack 30 bits turns out to be **2.328 ± 0.009 seconds** with a confidence interval at 99%. Via Compute we obtained a speed-up of 110% compared to the version not optimized using the help of this tool and 25647% compared to the sequential version with 1 block and 32 threads on the GPU.
Although the result has been achieved there still remain many optimizations that, through careful study of the algorithm, can be made. First of all, as we saw earlier, one can conduct a study that aims to analyze some possible solutions for the problem of Uncoalesced global accesses or optimize the execution time by distributing the work on the CPU as well. Anyway, due to the complexity of the algorithm itself, is not easy to develop and optimize such code, it also does not approach in the most efficient way the memory management that the GPU uses thus reducing the room for improvement. In each case the GPU version scale way better w.r.t the CPU version as the length of the key to be hacked grows, making our goal attainable despite great efforts.