Final Project - Discrete Logarithm Problem

Andrés Ricardo Pérez Rojas
Facultad de Ingeniería, Universidad Nacional de Colombia
Bogotá, Colombia
riperezro@unal.edu.co

I. Introduction

Discrete logarithms are defined for cyclic multiplicative groups. In groups formed by selecting the integers modulo some prime number p (Z_p^*), it is defined as the solution to the equation:

$$a^x \equiv b \pmod{m} \tag{1}$$

given a, b and m that are positive integers. Such x that satisfies the equation may not exist, and there isn't a simple method for checking its existence.

II. Conceptual Framework and Prior Work

A. Algorithm

The brute force solution for this problem just iterates over all possible values for x. That is all integers in the range $[0, m)$, having therefore a complexity of $O(m{*}\log m)$ if binary exponentiation is used.

A better algorithm for solving this problem, which was chosen for the implementations for this project, is called "Baby-step Giant-step" [1]. It was proposed by Daniel Shanks in 1971 [2] [3]. More precisely, it finds the discrete logarithm of an element in a finite abelian group.

Back on equation 1, we can express x as:

$$x = n * p - q \tag{2}$$

with n being a constant we can choose. p is considered the giant-step, and q the baby-step. Any possible x can be expressed in this way with a value of p in the range $[1, \lceil \frac{m}{n} \rceil]$ and q in the range $[0, n]$.

The equation now becomes:

$$a^{n*p-q} \equiv b \pmod{m} \tag{3}$$

Given that a and m are relatively prime (which is always the case if p is prime and a is not 0), there is an existing operation to obtain:

$$a^{n*p} \equiv b * a^q \pmod{m} \tag{4}$$

From equation 4 we define two functions:

$$f_1(p) = a^{n*p} \pmod{m} \tag{5}$$

$$f_2(q) = b * a^q \pmod{m} \tag{6}$$

By using this two functions, we can use a "meet-in-the-middle" approach:

1) Calculate f_1 for all possible values of p.
2) Calculate f_2 for all possible values of q and check if any of the values obtained for $f_1(p)$ had the same result.

If we choose the constant n to be the square root of the modulo m, the range of possible values for both p y q is of size $\sqrt{n}$:

- $p \in [1, \lceil \frac{m}{n} \rceil] = [1, \lceil \sqrt{m} \rceil]$
- $q \in [0, n] = [0, \sqrt{m}]$

We can evaluate the functions f_1 and f_2 in $O(\log m$ time with binary exponentiation. We can store the results obtained for f_1 using a hash table, or in a sorted list to be able to use binary search on it.

The time complexity of both steps is $O(n \log m)$ if we choose n as the square root of the modulo m, therefore having a total time complexity for the algorithm of: $O(\sqrt{m} \log m)$.

B. Why parallelize this algorithm?

Several cryptography algorithms are based in the much higher difficulty of calculating the discrete logarithm in comparison to its inverse: discrete exponentiation. One of this algorithms is the Diffie-Hellman key exchange [4]. This algorithm enables two users in a public network to generate a shared secret that can be used as a simetric key in private key cryptography algorithms in the future.

It is usually assumed that the best complexity for the discrete logarithm is trying all possible exponents by brute force, which would be $O(m)$ with m the chosen modulo. But, as previously shown, the "Baby-step Giant-step" algorithm reduces the time complexity significantly. It is still much more costly to calculate than a discrete exponentiation, but it is much lower than what is thought initially. By understanding the limits of its parallelization, we will have a better measure of the aparent computational cost of calculating the discrete logarithm, and be able to make more informed decisions about the required size of the chosen abelian group when creating a encryption system based on discrete exponentiation.

III. Design Considerations

The source code for the project can be found in the following GitHub repository: https://github.com/AndresRPerez12/DistributedSystems

A. Sequential

For the sequential version of the algorithm, a unordered_map was used, from the standard C++ library, in order to store the calculated values for f_1 for each p. The binary exponentiation was implemented recursively.

B. OpenMP

On the parallel version with OpenMP, it wasn't possible to use unordered_map from the standard C++ library since it doesn't support concurrent read and write operations. This was solved by using concurrent_unordered_map from the oneAPI Threading Building Blocks library from Intel. The binary exponentiation was implemented recursively.

C. CUDA

On the parallel version with CUDA it wasn't possible to use a hash table as in the previous cases, since we can't use CPU libraries on the GPU. There are some hash table implementations for CUDA, but they require additional installs. The selected approach was to create an array on the device on which threads write their results for the f_1 function, and another one for the corresponding p values. after filling this arrays, the $sort_by_key$ command was called from teh CPU, to sort both arrays using the values obtained as reference. This method sorts in a parallel fashion, and is significantly faster than sorting on the CPU.

Once we have the arrays sorted, when a thread has calculated a result for f_2 with a current q value, a binary search is run over the array with the stored values, and if the target results is found, the corresponding p value is used to calculate x.

An additional consideration for this case was the implementation of the binary exponentiation, given that the threshold for a stack overflow error when running recursive methods on the device is much lower than on the CPU. The method was implemented in an iterative fashion without increase its time complexity.

D. CUDA in Google Colab

When trying to run the same CUDA code that was used in the previous section on Google Colab machines, an unexpected problem arose. The existing implementation required the use of 128-bit integers in order to compute the multiplications required without overflow, even if immediately after the modulo was taken.

The root of the issue was that this feature was added fairly recently in CUDA version 11.5 [6], and the Google Colab machines support only up to version 11.2, as can be observed with the nvidia−smi command:

NVIDIA-SMI 460.32.03 Driver Version: 460.32.03 CUDA Version: 11.2

A solution was found, but it came at a performance cost. In a similar way as a discrete modular exponentiation can be calculated more efficiently than the brute force approach by using a divide-and-conquer strategy to reduce the number of multiplications required, a multiplication can be thought of as a series of additions and can also benefit from a divide-and-conquer approach. This of course is more costly in time than just using the standard $*$ operator, but it allows for the use of only 64-bit variables as the maximum value to compute before taking the modulo is around double the modulo value itself, instead of the square as is the case with a direct multiplication.

Similarly as in the fast exponentiation algorithm used up to this point, this "fast multiplication" algorithm has a complexity of $O(\log m)$. Since the fast exponentiation algorithm requires multiplications, its complexity now rises to $O(\log^2 m)$. Therefore, the total time complexity of the whole program increases to $O(\sqrt{m}\log^2 m)$. This basically negates any performance gains with respect to the CUDA program run on a local GPU, since the extra cores available in the Google Colab machines don't make up for the new increased time complexity, but still, the experiments were run with this approach.

E. MPI

On the parallel version with MPI it wasn't possible to use a hash table either, in this case since MPI uses a distributed memory architecture. The chosen approach was to again use a sorted list and applying binary search as a lookup strategy when calculating the results of the second function.The binary exponentiation was implemented recursively, since it is running on CPUs and doesn't run into stack overflow issues.

To be able to use this approach, it was required to gather all the results calculated by each process for f_1, as well as their corresponding values for p. Then, these lists must be sorted on the main process, since it is necessary to have all the values on memory to perform the sort. Then, each of the processes needs a complete copy of the sorted lists in order to check their calculated values for f_2 against the values already computed. Therefore, a broadcast was used to make sure each process had a full copy of both arrays. The final value for the answer x is brought to the main process via a reduce operation that chooses the maximum value among all processes, since the default value for this variable (that isn't overwritten if that specific process doesn't find an answer) is -1, so if any process finds a correct answer, its result will be greater than in all processes that didn't find it.

IV. Experiments

The performance of the algorithm will be measured by evaluating the time taken to solve the equation:

$$56439^x \equiv 27341644544150 \pmod{29996224275833} \quad (7)$$

With this response time, we will also calculate the speed up using as baseline The response time of the sequential implementation. The test will be executed using 1, 2, 4, and 8 threads. For CUDA, they will be executed with 5 blocks y 32, 64, 128m, and 256 threads per block. The response time reported is the average of 5 runs.

A bash script was used to run each program 5 times for each measurement, and to calculate the average of all runs. Each program receives as a console parameter the threads desired (for the parallel implementations) and a print flag. The program prints the response time and the found solution according to the flag. The response time is determined using the <sys/time.h> library.

The computer on which the tests were run has the following characteristics:

- CPU: Intel Core i7-7700HQ @ 2.80 GHz
- Cores: 8
- RAM: 16 GB
- OS: Ubuntu 22.04.1
- GPU: NVIDIA GeForce GTX 1050 with 4 GB. It has 5 multiprocessors and 128 cores per multiprocessor (640 in total)

For the CUDA on Google Colab experiments, the virtual machine available has the following characteristics:

- CPU: Intel(R) Xeon(R) CPU @ 2.30GHz
- Cores: 1
- RAM: 13 GB
- OS: Ubuntu 18.04.6 LTS
- GPU: NVIDIA Tesla T4 with 15 GB. It has 40 multiprocessors and 64 cores per multiprocessor (2560 in total)

For the MPI experiments, four virtual machines were set up on Google Cloud Platform (GCP) to configure the distributed memory architecture. The chosen type was an e2-medium, which has the following characteristics:

- CPU: Intel Broadwell
- Cores: 2 vCPUs
- RAM: 4 GB
- OS: Debian 11

V. Results

A. Sequential

Threads	Time (s)
1	20.2551942

TABLE I: Sequential response time

B. OpenMP

Threads	Time (s)	SpeedUp
1	22.4747398	0.9012426564
2	13.005094	1.557481568
4	7.5905904	2.668460967
8	4.9812848	4.06625901
16	5.0068542	4.045493116

TABLE II: OpenMP response time and speedup

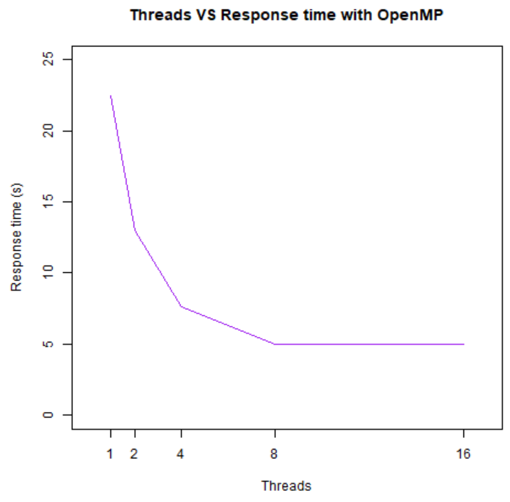

Fig. 1: OpenMP response time.

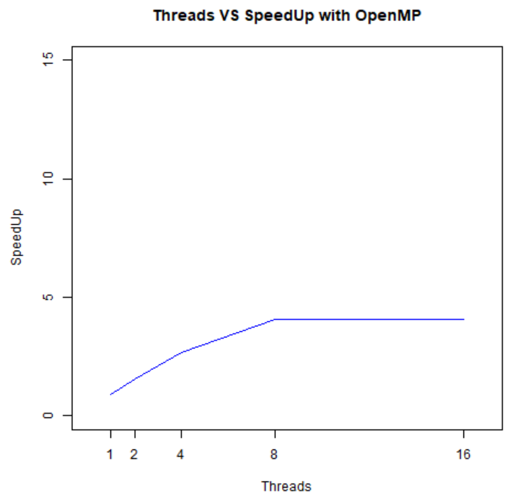

Fig. 2: OpenMP speedup.

C. CUDA

Blocks	Threads per block	Time (s)	SpeedUp
1	16	96.68897925	0.2094881377
1	32	47.594024	0.4255827202
1	64	22.77132825	0.8895042914
1	128	11.02069075	1.837924197
1	256	5.67413275	3.569742742
3	16	31.55769175	0.6418465064
3	32	15.99623175	1.266247859
3	64	7.6200815	2.658133538
3	128	3.7268105	5.434994401
3	256	1.9637275	10.31466647
5	16	19.9024236	1.017725007
5	32	9.948904	2.035922168
5	64	4.7314394	4.280979315
5	128	2.3556006	8.598738768
5	256	1.4052096	14.41435797

TABLE III: CUDA response time and speedup

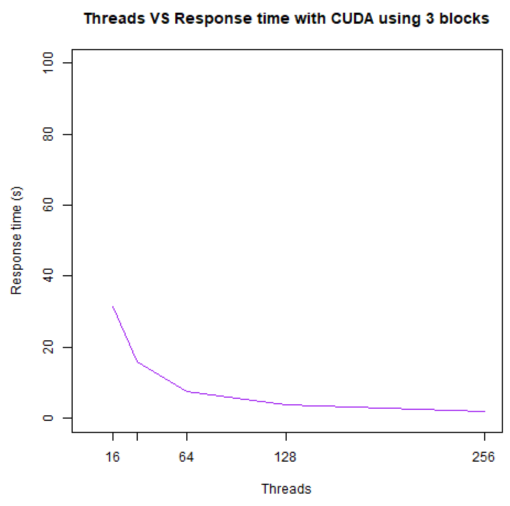

Fig. 5: CUDA response time with 3 blocks.

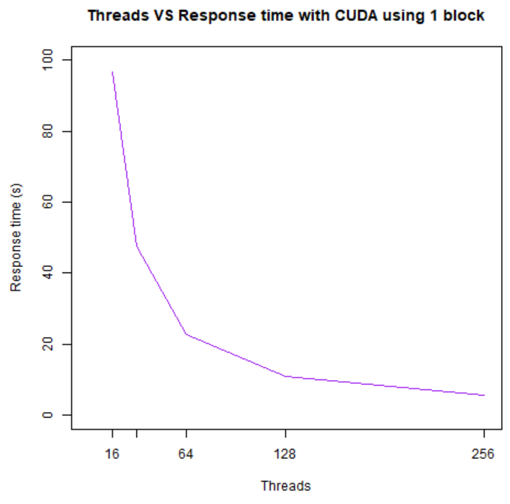

Fig. 3: CUDA response time with 1 block.

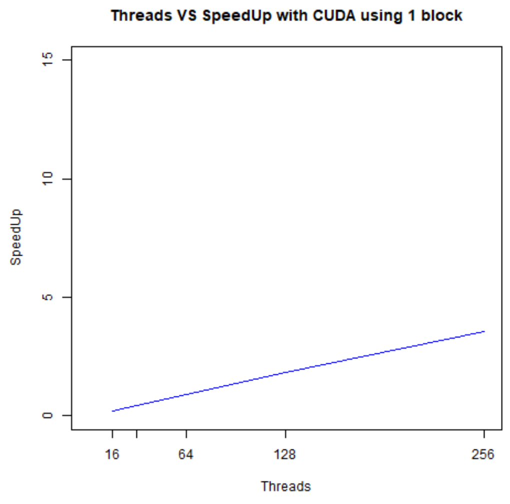

Fig. 4: CUDA speedup with 1 block.

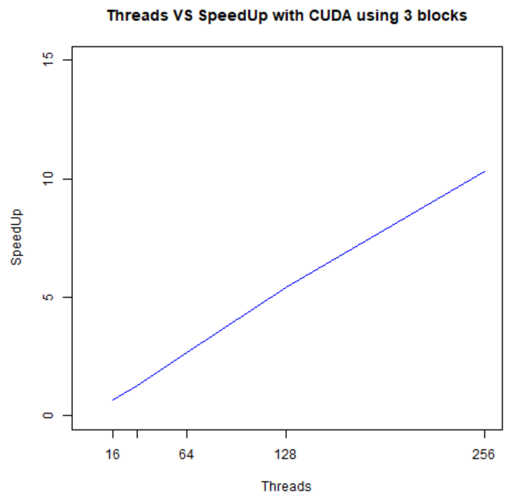

Fig. 6: CUDA speedup with 3 blocks.

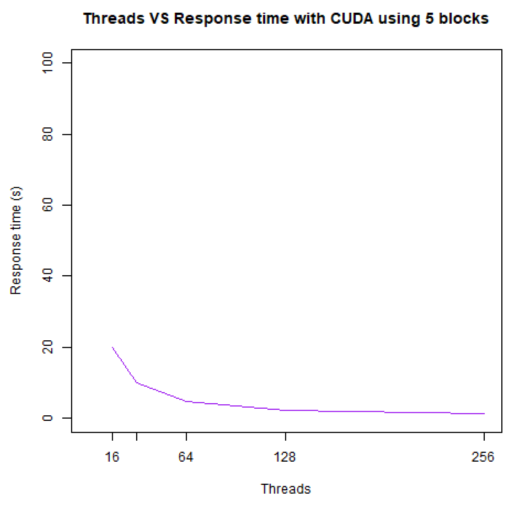

Fig. 7: CUDA response time with 5 blocks.

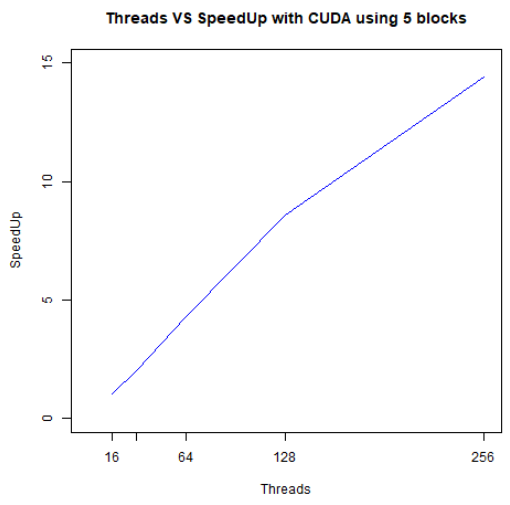

Fig. 8: CUDA speedup with 5 blocks.

D. CUDA in Google Colab

Blocks	Threads per block	Time (s)	SpeedUp
10	16	90.8100	0.22305
10	32	45.6077	0.44411
10	64	23.2912	0.86965
10	128	13.2577	1.52780
20	16	45.7019	0.44320
20	32	23.2826	0.86997
20	64	13.2381	1.5300
20	128	7.84282	2.5826
30	16	30.0092	0.6749
30	32	15.2847	1.3251
30	64	9.5318	2.1250
40	16	23.1367	0.8754
40	32	13.2482	1.5289
40	64	7.7763	2.6047
40	128	10.1740	1.9908

TABLE IV: CUDA response time and speedup in Google Colab

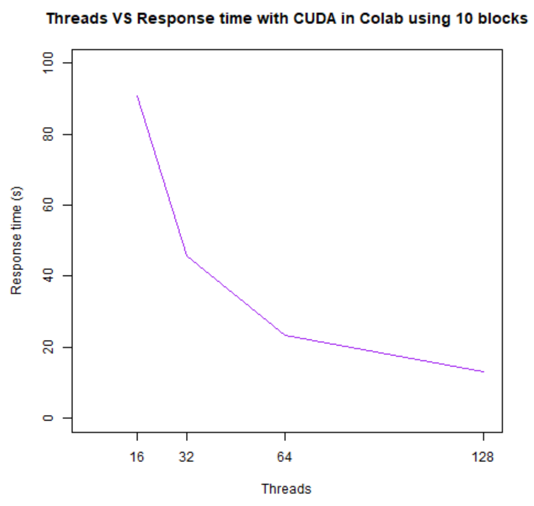

Fig. 9: CUDA response time with 10 blocks in Google Colab.

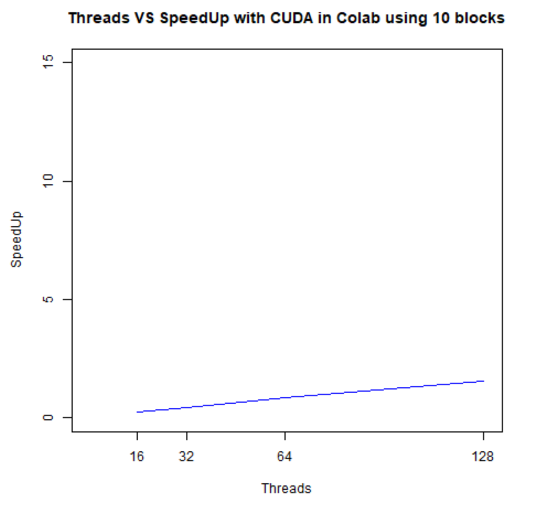

Fig. 10: CUDA speedup with 10 blocks in Google Colab.

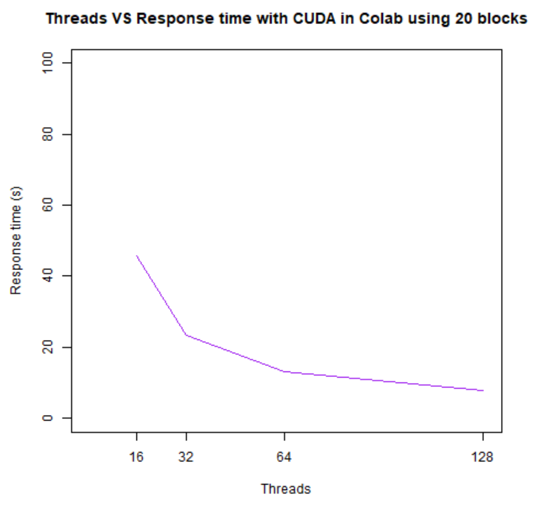

Fig. 11: CUDA response time with 20 blocks in Google Colab.

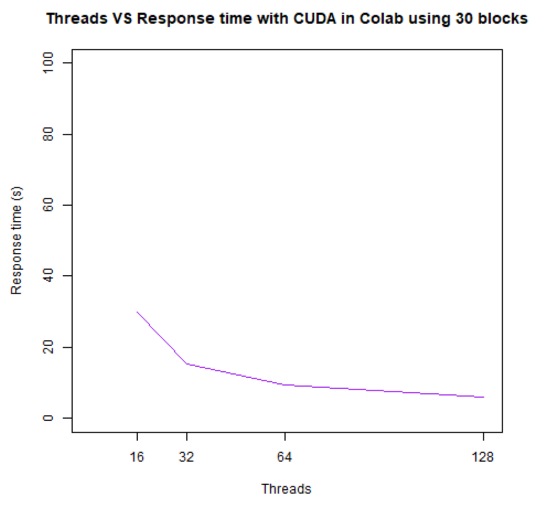

Fig. 13: CUDA response time with 30 blocks in Google Colab.

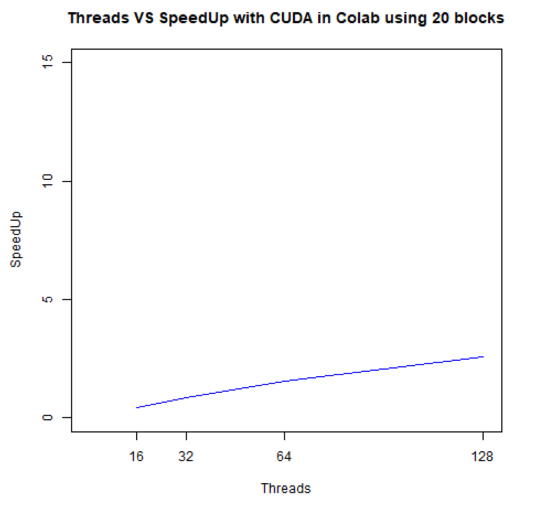

Fig. 12: CUDA speedup with 20 blocks in Google Colab.

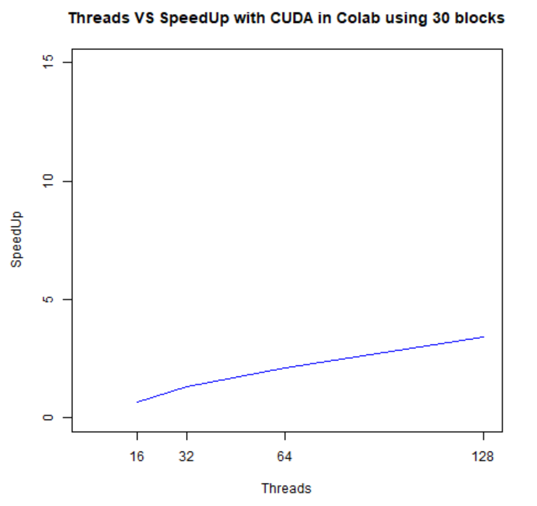

Fig. 14: CUDA speedup with 30 blocks in Google Colab.

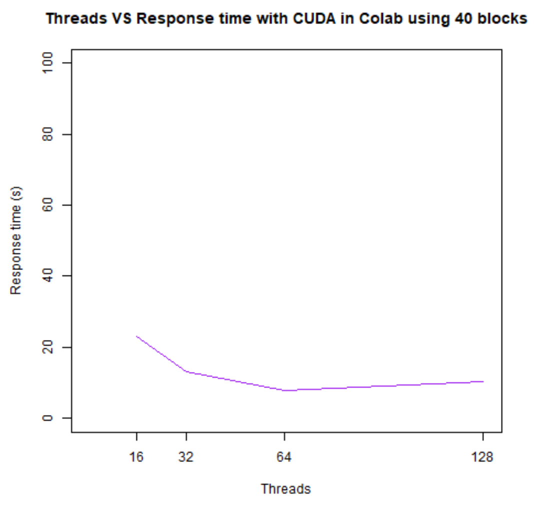

Fig. 15: CUDA response time with 40 blocks in Google Colab.

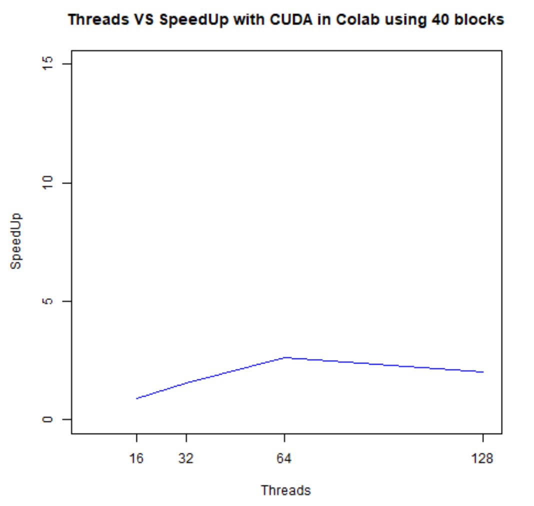

Fig. 16: CUDA speedup with 40 blocks in Google Colab.

E. MPI

Processes	Time (s)	SpeedUp
1	19.9499458	1.015300713
2	20.8253756	0.9726208347
3	16.63273375	1.217791044
4	14.512248	1.395730985
5	13.41639325	1.509734682
6	13.0172708	1.556024647
7	11.9046084	1.701458252
8	13.321141	1.520529976

TABLE V: MPI response time and speedup

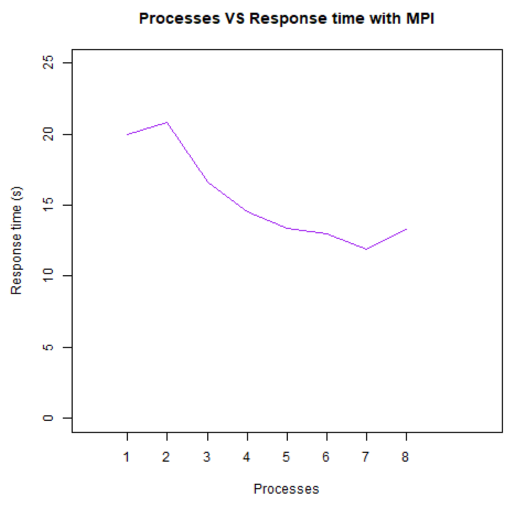

Fig. 17: MPI response time.

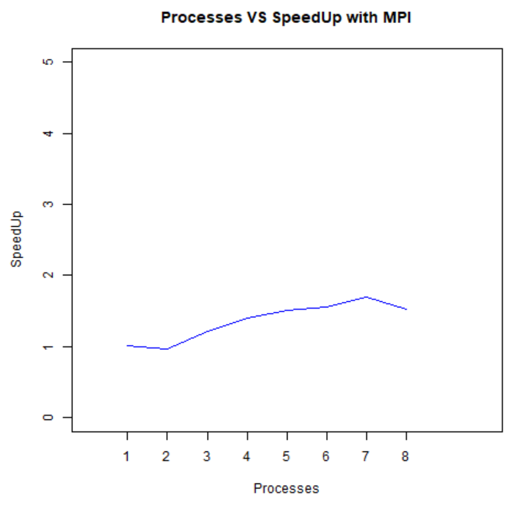

Fig. 18: MPI speedup.

VI. Conclusions

We obtained satisfactory results by parallelizing the discrete logarithm problem. For OpenMP, the speedup found increases as expected, as we are able to use CPU calls normally and therefore can use a very efficient hash table to store the calculated values. Given the nature of the approach, it is required to synchronize all threads before starting to calculate values for f_2. Therefore, the results we obtained for the speedup in which the response time isn't exactly reduced by a factor of the number of threads makes sense. The results are promising but this parallel approach depends directly on the available CPU cores. This has the implication that the maximum theoretical speedup is heavily limited by our current technology and CPU limitations. Given the size of the prime numbers usually used in cryptography (even reaching 128 bits), the time reduction wouldn't really be meaningful.

On the other hand, with CUDA we obtained interesting results. This implementation had the special situation of not being able to use a hash table directly. This was solved by using sort_by_key from the thrust library, that sorts the required arrays on the device in a parallel fashion. This makes the algorithm in general slower, having now a complexity of $O(\log \sqrt{M})$ to verify if a found value for f_2 was also obtained from f_1. In the results, we can notice that even with 16 threads in each of the 5 blocks, the speedup is still not significantly higher than 1 when compared against the sequential version that uses hash tables. In general, the speedup and time response show promising results with 1, 3 and 5 blocks. In all three cases, the speedup almost doubles as we double the amount of threads per block. And of course, the increase in speedup when increasing the number of threads is steeper the more block we are using. The implementation used for this project isn't dependent on using the shared memory that each block has, so the performance gains are directly connected to the complete number of threads we are using on the CPU. We can still get some performance gains when going from the maximum number of concurrent threads the GPU can handle to double that number, as suggested by NVIDIA.

For the other CUDA case, when running on Google Colab machines with the required changes to the algorithm as described on section III-D, the linear speedup trend observed on the local CUDA experiments can still be seen. Even if the slower algorithm causes the best response time to still be slower than on the local GPU that can execute the faster version of the algorithm for this input, the results show that when the Google Colab machines update their drivers and/or GPUs to support a CUDA version of at least 11.5, they can be used to solve this problem much faster than the sequential version and many consumer GPUs given their high core count. And even with the slower multiplication, a significant reduction from the sequential CPU based implementation was obtained.

From the results, we can see that the speedup does approximately duplicate as the threads are duplicated. This is very significant, given that there are GPUs on the market with much more CUDA cores tan the one used for these tests (a NVIDIA GeForce 4090 GPU has 25 times more cores than the 1050 for laptops used in the experiments [5], and cloud services like the Google Colab examples presented can go even higher). Since we can parallelize the algorithm in a GPU, the minimum time for a given abelian group will keep getting lower and lower. This is relevant for the selection of the size of the group such that it is sustainable in time. Even if the algorithm used in this architecture is less efficient than the sequential version running on a CPU, the hardware possibilities for its parallelization make it a very relevant measurement when choosing the group for a cryptography system.

Finally, the distributed memory architecture used by the MPI implementation seems to not really be suited for this problem and algorithm specifically. Each process can calculate the values of the f_1 function for its assigned p values. But these values must be sent back to the main process for sorting. This is a key difference with the CUDA implementation. They both rely on sorted lists for finding a match between the two functions, but in CUDA we don't have to move the values we calculated back and forth, and we can sort in a parallel fashion on the GPU itself since all the data is already there. With MPI, we have to bring all the data back to the main process to be able to sort it, and then broadcast a complete copy of the lists to each process, since it is required for them to check if there is a match. The costs of sending and receiving all this data increases as the number of processes does to, thus limiting the maximum performance gains we can get by parallelizing the algorithm this way.

The data suggests there are some performance gains from increasing the number of processes. This makes sense since the total size of the lists to sort is the same regardless of the number of processes we use. But, it is still very limited and the speedup gains aren't very significant against the computational power we added, never getting to a speedup of 2 in our tests. The results with these specific inputs also suggests that there is a sweet spot for the maximum gains we can have before the amount of processes and their associated performance cost makes it not worth it to add more of them. In this case, it seems to be at 7 processes. The other parallelization methods presented seem to be much better suited for this algorithm.

References

[1] "Discrete Log - Algorithms for Competitive Programming", Cp-algorithms.com, 2022. [Online]. Available: https://cp-algorithms.com/algebra/discrete-log.html#implementation. [Accessed: 06- Sep- 2022].

[2] "Baby-step giant-step - Wikipedia", En.wikipedia.org. [Online]. Available: https://en.wikipedia.org/wiki/Baby-step_giant-step. [Accessed: 06-Sep- 2022].

[3] Daniel Shanks (1971), "Class number, a theory of factorization and genera", In Proc. Symp. Pure Math., Providence, R.I.: American Mathematical Society, vol. 20, pp. 415–440

[4] M. E. Hellman, "An overview of public key cryptography," IEEE Communications Magazine, vol. 40, no. 5, pp. 42–49, 2002.

[5] "Nvidia GeForce RTX 4090 graphics cards," NVIDIA. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/. [Accessed: 10-Nov-2022].

[6] C. Hoekstra, K. Shukla , and M. Harris, "Implementing high-precision decimal arithmetic with Cuda int128," NVIDIA Technical Blog, 10-Feb-2022. [Online]. Available: https://developer.nvidia.com/blog/implementing-high-precision-decimal-arithmetic-with-cuda-int128/. [Accessed: 30-Nov-2022].