Hardware Accelerated QC-MDPC McEliece Cryptosystem

Dzurick, Mitchell
Eletrical and Computer Engineering
University of Arizona
Tucson, Arizona
mitchdz@email.arizona.edu

Russell, Mitchell
Electrical and Computer Engineering
University of Arizona
Tucson, Arizona
mcrussell@email.arizona.edu

Kuban, James
Electrical and Computer Engineering
University of Arizona
Tucson, Arizona
kubandj@email.arizona.edu

***Abstract*—Due to the advent of quantum computers, a need for new cryptographic algorithms that can resist this new computing type emerged. Classic logarithmic based algorithms such as AES and RSA are hard for classical computers to crack, but not as challenging for quantum computers, the McEliece cryptosystem is one of them. The McEliece cryptosystem linearly expands the message and intentionally adds errors to be recovered later. To achieve this, many matrix operations are performed on large matrices which is very demanding for a CPU. This paper focuses on using CUDA with a dedicated graphics card to accelerate the time to perform encryption and decryption with the McEliece cryptosystem. The resulting cryptosystem has an overall end-to-end speedup of 13.32X. This speedup renders the McEliece cryptosystem more realistic for actual use.**

***Index Terms*—CUDA, McEliece, cryptography, GPU**

I. Introduction

The McEliece cryptosystem is one of the many proposed post-quantum asymmetric encryption algorithms. The McEliece cryptosysem is resistant to Shor's algorithm which means that quantum computers cannot brute-force the decryption keys in polynomial time. The McEliece algorithm specifies a small list of steps in order to generate cipher text that can be recovered. This paper aims to improve the execution time of a Quasi Cyclic-Moderate Density Parity Check (QC-MDPC) based McEliece cryptosystem in key generation, encryption, and decryption. This implementation of the McEliece cryptosystem involves operations on matrices on the order of one-billion elements in all steps of the program. The paper shows how a speedup of **8874X** for matrix multiplication, **9.08X** for matrix addition, and **3.46X** for matrix inverse was achieved to get an overall speedup of **13.32X**.

The goal of this project is not to improve the cryptosystem in any way, but rather just improve the execution time using a GPU.

II. Related Work

To the best of our knowledge, full developments of the McEliece cryptosystem is currently not well documented for GPU implementation. In [4], they demonstrate the improvements seen in encryption rates by implementing the McEliece cryptosystem on the GPU. However, this research does not continue beyond encryption with any improvements in decryption as well as full end-to-end testing.

In [6], the author provides a general overview of the McEliece cryptosystem as a whole on the GPU. Yet, their analysis focuses too much on proving its security hardness with a conceptual GPU implementation.

Finally, the code we are basing our project on was developed by the Github user, Varad0612. In [9], the provides us a working end-to-end serialized McEliece cryptosystem with which we can use to compare our implementation.

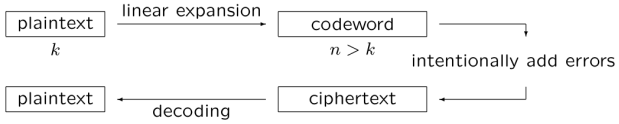
Now, the reason for this lack in documentation arises from ongoing research in improving implementations of Gaussian Elimination on the GPU. Gaussian Elimination is important to the McEliece system as it is a critical component to calculating the inverse matrix for a decryption method. This process is also majorly responsible for the higher latency and by improving it will allow our program to operate at a lower latency.

As noted in [5], efficient usage of Gaussian Elimination for matrices with binary coefficients is currently still not documented well. Their algorithm focuses on a complex mix of steps that perform either cyclic-bit shifting or matrix element elimination. However, their methodology would require that each element in the matrix utilize more memory as those elements need to have "used flag"; this extra flag would effectively double the memory requirements , which, for our target bit-security of 256-bit, our matrices will contain greater than 1 billion elements per key.

III. Background

The Mceliece cryptosystem is a relatively simple system to understand. Figure 1 shows a top level view of the cryptosystem. There are two main steps to the system, linearly expanding the plaintext, and then intentionally adding errors that can be recovered later.

A. Top Level View



Source: nist.gov

Figure 1: McEliece top level view

There are a few versions of the McEliece cryptosystem, and for this paper we are reviewing the QC-MDPC (Quasi Cyclic - Moderate Density Parity Check) variant of the cryptosystem. The QC-MDPC McEleice cryptosystem takes four inputs, n, p, t, and w where

1) n is the length of the error vector
2) p is the dimension of the private/public key
3) t is the weight of the error matrix
4) w is the weight of the QC MDPC code

The effective keyspace, k is $\{0,1\}^{(n-1)*p}$. This keyspace is what determines how much data you can encrypt. If the size of your data is less than k then you will not be able to use this cryptosystem. Therefore, for larger sized files you are required to adjust your input parameters appropriately.

B. McEliece Operations

The McEliece cryptosystem offers support to **generate a pair of asymmetric keys**, **encrypt**, and **decrypt** data. This paper does not aim to explain the rational or the math behind each operation, but rather to show the the process involved with each operation.

Key Generation:
Key generation starts with creating the private key, **H** which is a parity-check matrix. This paper will not go into how **H** is generated, because the process is very serialized, but does not take that long on a CPU. The key that is worth looking into is the public key **G**.

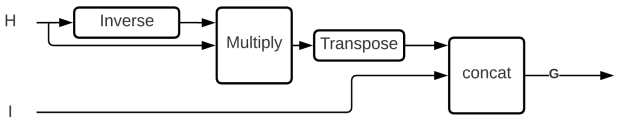


Figure 2: McEliece Keygen Diagram

Figure 2 shows the resulting operations needed the generate the public key **G**. It can be seen that 3 expensive matrix operations are needed: inverse, multiply, and transpose. The matrix **I** is the identity matrix of size H.

Encryption:
compute the ciphertext c as

$$c = uG + e \tag{1}$$

where u is the plaintext, G is the generator matrix, and e is an error vector.

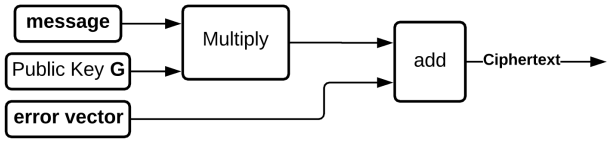


Figure 3: McEliece Encryption Diagram

Figure 3 shows a diagram of the operations needed to encrypt a message. The two multiply and add operations are matrix operations, thus they slow down a lot when the key sizes are very large.

Decryption:
To decrypt a ciphertext c, the user has to obtain the private key **H**. The decryption function has to be some type of decoding function that can recover the errors intentionally added during the encryption process. This implementation of the McEliece cryptosystem uses *syndrome decoding*. This paper does not aim to explain syndrome decoding, so a function $DEC_H(c)$ is defined as the syndrome decoding function, which requires the private key **H** and the ciphertext c

$$u = DEC_{\mathbf{H}}(c) \tag{2}$$

Where u is the plaintext, **H** is the private key, and c is the ciphertext generated from Equation 1. This paper will not go into syndrome decoding as that is out-of-scope (for the sake of time), but do know that in the process of setting up syndrome decoding, matrix multiplication and matrix addition are used.

C. Binary Arithmetic

Because McEliece is a binary encoding based cryptosystem, every matrix operation is not floating point arithmetic, but rather binary arithmetic. This causes a few annoyances because most standard libraries do not handle binary arithmetic. For example, CuBLAS only handles floating point arithmetic for matrix inverse operations. There are libraries such as M4RI[7] which deal only with dense matrix operations over F2 (F2 meaning a field of two elements). However, the parallelization strategies this library uses is OpenMP, not CUDA.

D. Functional Security

It is important to pick the right input parameters to this cryptosystem to try and achieve the desired security. It is often referenced that a certain set of input parameteres offer a level of security in bits[3, 2]. what this means is that for example if you have a level of security of 80 bits, it will take 2^{80} computations to brute-force the secret key. Determining the realistic security of this cryptosystem is more nuanced than simple thinking about the input parameters, but for this paper we aim to find a generalized baseline to hopefully achieve a realistic implementation in a reasonable amount of time.

To determine the security of the system, we must determine the most effective way of attacking the cryptosystem currently known. At the time of writing this paper, it is believed that Information Set Decoding (ISD) is the most effective attack against this cryptosystem. Anja Becker, et. al. Explain the challenges of ISD and propose an algorithm to attack this cryptosystem in asymptotic computational complexity[2]. This asymptotic computational complexity is what is referenced when a certain set of input parameters is considered. What's Anja Becker, et. al proposes is that a matrix size of 32,771 is needed for 256 bits of security for this cryptosystem.

IV. Methodology

The goal of this project is to convert a CPU-based cryptosystem into one that can leverage hardware acceleration. Therefore, we want the operations of the system to remain very close to the source code. The obvious first choice is to review

the source code, and find functions that are optimal for GPU based execution. After doing initial research, the following kernels were determined to be the most time consuming and worth-while to convert to CUDA:

1) Matrix Multiplication
2) Matrix Transpose
3) Matrix Inverse
4) Matrix Addition

The matrix operations mentioned above are the largest bottleneck for CPU execution, as the matrix operations required for this cryptosystem are very large and take an unreasonable amount of time for a CPU based execution. The goal is to mimic the same functionality of the reference cryptosystem, but interchange the CPU based matrix operations with GPU based matrix operations to hopefully realize a a very large performance gain.

Algorithm 1: CPU based Matrix Multiplication

```
Result: AB = C
check for compatible matrix dimensions
transpose matrix B
for i ∈ ARows do
    for j ∈ BColumns do
        val = 0;
        for k ∈ BRows do
            val = val ⊕ (A[i,k] & B[j,k]);
        end
        C[i,j] = val;
    end
end
Return C;
```

V. Evaluation and Validation

Evaluation on the created system is directly compared to the C based implementation. The system that all tests were ran on has 16GB of DDR4 RAM, Intel(R) Core(TM) i9-9900KF CPU @ 5.0GHz, and a GTX 980Ti. Evaluation was done on each major matrix operation via timing analysis, comparing directly to the C based implementation. Validation was relatively straight-forward where we created a kernel (or set of kernels) to achieve the same functionality as a function in the baseline code. If the kernel or sets of kernels produced the same output as the baseline C code, it is validated to work properly. Each test-case requires different input parameters, but testing was done with the goal of realistic operations that the cryptosystem would encounter. For example, there would not be any need to test a matrix operation on rectangular test inputs when the code only works on square matrices.

A. Experimental Results and Analysis

Experimental results consist of timing analysis for each major matrix operation that was targeted to be sped up, and an overall end-to-end test that includes generating a pair of keys, encrypting random data, and decrypting random data.

Matrix Multiplication Kernel

Matrix multiplication is prevalent throughout this cryptosystem, this include encryption/decryption as well as key generation and decoding, making it a prime target for optimization. Furthermore, matrix multiplication has long been a target for GPU optimization due to is paralizable nature. However little prior work was found for optimizing binary matrix multiplication. Due to the use of uint8_t memory accesses were not able to utilize the available bandwidth effectively. As such the first optimization targeted consolidating global memory accesses. This was achieved for matrix A by reading 4 data points concurrently by typecasting the global array to uint32_t, however this optimization was not suitable to matrix B due to the to transpose it. This had the unexpected outcome of allowing 4 binary multiplications sums to be completed at once due to the fact that binary multiplication is bit-wise.

For optimization of GPU occupancy, shared memory bank conflicts were eliminated and a sweep of tile size was performed. The bank conflicts were found in the shared B array due to transposing upon write to shared memory and could be resolved by increasing the column size of our shared B array and adjusting indices. Optimizing the tile size was straightforward, a 32 x 32 tile proved to utilize as much of the GPU as possible since this was the max number of threads a block can handle. Furthermore, this allowed for easy memory coalescence and used approximately half of the available resources per SM such as registers.

Further optimization using the outer product method of matrix multiplication to reduce shared memory accesses. By using the outer product an element of B is instead read into the registers and used by a single thread. This has shown a substantial improvement for integer and floating point matrix multiplication, however has not been proven to work with binary matrices.

Matrix Dimension	CPU (seconds)	GPU (seconds)	Optimized GPU (seconds)
16	0	0.16	0
32	0	0.12	0
64	0	0.12	0
128	0.01	0.12	0
256	0.07	0.12	0
512	0.56	0.13	0
1024	4.46	0.11	0
2048	35.55	0.14	0
4096	283.79	0.23	0.05
8092	2307.38	0.9	0.26
16384	N/A	5.99	1.84
32768	N/A	46.08	13.73

Table I: Timing results for Matrix Multiplication execution

Matrix Inverse

Matrix inverse is one of the large time sinks in this cryptosystem, and it's only observed during generating the public key. Therefore, the inverse matrix does not effect encryption and

decryption. With that said, it can take over a day to produce a relatively strong key pair. The main optimization this paper proposes is a change in the algorithm, and to understand why that was necessary, the reference algorithm needs to be explained.

Algorithm 2: CPU based Inverse

```
   Result: Inverse of matrix A
1  generate identity matrix B of size A
2  for i ∈ AColumns do
3      if A[i,i] == i then
4          for i ∈ AColumns do
5              if i != j and A[j,i] == 1 then
6                  addRowsB();
7                  addRowsA();
8              end
9          end
10     else
11         for k = i + 1 to ARows do
12             if A[k,i] == 1 then
13                 addRowsB();
14                 addRowsA();
15                 i = i - 1;
16             end
17         end
18     end
19 end
20 Return B;
```

As shown in algorithm 2, the algorithm is recursive by nature due to the i = i - 1 in line 15. This is not GPU friendly at all because every operation is extremely dependent on the previous operation. It is hard impossible to do all operations in parallel with this algorithm. Therefore, in order to do matrix inversion on a GPU, LU decomposition is used instead of regular Gaussian Elimination because LU decomposition is not recursive by nature[1]. However, this does not mean LU decomposition is perfect. As we will see, there are portions that have to be run in serial, and at the end we will see the solution to this problem.

Algorithm 3: GPU based Inverse

```
   Result: Inverse of matrix A
1  make_GF2_identity_gpu();
2  for i ∈ AColumns do
3      GF2_LU_decompose_find_max_row();
4      GF2_LU_decompose_update_trailing_row();
5      GF2_LU_decompose_pivot_row();
6  end
7  GF2_Forward_substitute();
8  GF2_Backward_substitute();
9  GF2_swap_cols();
```

The GPU based inverse algorithm 3 on the other hand has no if conditionals, making the algorithm not recursive. this offers a huge benefit as a majority of the computations can be done without having to rely on previous computations. Unfortunately, the order that the algorithms are called is important and forward/backward substitution can not be done in parallel.

LU Decomposition is not a single kernel, but rather a string of kernels doing specific tasks. The following kernels were created to allow GPU based LU decomposition:

- GF2_LU_decompose_find_max_row
- GF2_LU_decompose_pivot_row
- GF2_LU_decompose_update_trailing_row
- make_GF2_identity_gpu
- GF2_Forward_substitute
- GF2_Backward_substitute
- GF2_swap_cols

Each kernel has a specific task, and the order of which they are called is important. To reduce the latency of generating an identity matrix, a single threadblock is dedicated to run in parallel with all of the decomposition steps. This means that once decomposition is done, the program can immediately start performing forward substitution. This is achieved through the use of CUDA streams.

GF2_LU_decompose_find_max_row

GF2_LU_decompose_find_max_row is unfortunately single threaded because it just does a linear search and returns as soon as it finds a one in the current row's sub-diagonal. This means that if the parity check matrix is more dense, then this function returns a lot faster.

GF2_LU_decompose_update_trailing_row

GF2_LU_decompose_update_trailing_row is the longest kernel in the decompose kernels. This kernel has to update the current row and every row below it. What makes this kernel really inefficient is that each row that the program goes down, the program updates less values. That means that more threads are idle on the later rows to be updated. With that said, this kernel is being sped up greatly by having a single thread handle each column element.

make_GF2_identity_gpu

make_GF2_identity_gpu is a kernel mentioned earlier that is meant to be called with a single threadblock. This kernel is designated to use a different cuda stream than the rest of the decompose kernels, and therefore gets generated slowly while decomposing the matrix. Usually, there is a delay after you are done decomposing to generate the matrix, and this speeds that up. The only downside to this method is that if the decompose steps requires all available threadblocks to operate, then this kernel could slow the program down.

GF2_Forward_substitute

GF2_Forward_substitute kernel requires the matrix A to be modified by the decomposition loop, and the matrix B which is the identity matrix of size A. In this kernel, each thread handles a column element, but unfortunately each thread has two nested for loops that can not be eliminated.

GF2_Backward_substitute

GF2_Backward_substitute kernel is really similar to the for-

ward substitution kernel. Each thread handles a separate column value.

Profiling was done on each kernel, but unfortunately no interesting information was gained through the use of profiling. The weak points in each kernel were already known before profiling. In addition to that, profiling the overall inverse function is an extremely timely process as nvprof has the rewind each kernel call. There are 4 + N*3 kernel calls where N is the dimension of the square matrix. Therefore, it could take hours to profile an execution time that only took a few minutes.

matrix dimension	CPU GJ (seconds)	CPU LU (seconds)	GPU LU (seconds)	speedup
512	0.19	0.425	0.523	0.37
1024	1.4	4.4	2.5	0.57
2000	10.0	26.3	12.5	0.80
3050	49.6	103.4	29.7	1.67
4000	125.2	269.4	58.1	2.16
4400	147.9	374.4	39.60	2.14
6000	375	1010.7	132.49	2.83
12000	862.2	8927.2	249.0	**3.46**
24000	8638	74625.4	4172	2.07

Table II: Timing results for Inverse Matrix execution

Table II shows the timing results, with the execution time of CPU based LU decomposition added in. It can be seen that the CPU based LU decomposition is a lot slower than the CPU based algorithm, which is to be expected. The real benefits comes in when the matrix is large enough and the GPU can handle a lot more operations in parallel.

Improving LU Decomposition

LU Decomposition suffers from one major drawback; that the algorithm has to finish executing one row before executing operations on the next row. To be able to speed up the execution, it is optimal to not have to wait for any operation, and be able to execute all necessary operations in parallel. To achieve this effect, modern libraries such as CuBLAS[8] use batched inverse operations which means breaking up the large matrix into block matrices, and performing the computation on a lot of smaller matrices.

Transpose Kernel

Matrix Transpose is utilized during key generation and decoding. Therefore, speeding up this operation will have no impact on encrypting the data. However, with that said, a simple matrix transpose kernel was created, one with a naive implementation, and one that avoids bank conflicts.

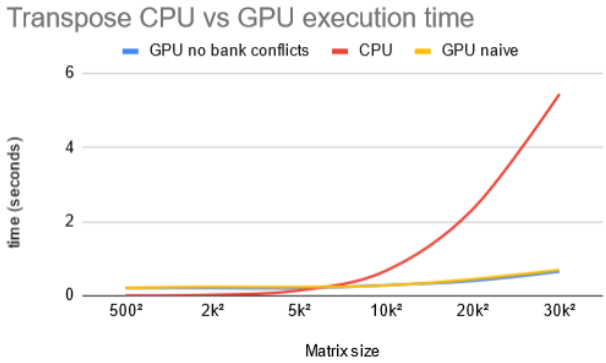


Figure 4: Transpose Timing Results

As shown in Figure 4 A large speedup can be achieved with very large matrices, where the no bank conflicts optimization only adds a very tiny improvement to the execution time. Adding the no bank conflicts optimization is not too much extra work, so it is worth it. Overall, this kernel does not have a huge impact on the performance of the cryptosystem, but it is nice to have.

Matrix Add Kernel

Matrix Dimension	CPU time (seconds)	GPU V1 time (seconds)	GPU V2 time (seconds)
200	0	0.2	0.21
512	0	0.17	0.18
1024	0.01	0.18	0.17
3050	0.1	0.19	0.19
4000	0.17	0.21	**0.18**
5000	0.26	**0.23**	0.2
6000	0.37	0.25	0.21
10000	1.07	0.37	0.28
15000	2.3	0.61	0.4
20000	4.7	0.93	0.57
32000	10.45	2.09	**1.15**

Table III: Timing results for Add Matrix execution

Table III shows the timing results for the add matrix kernel in the [9] serialized CPU code, GPU implementation version one and GPU implementation version two. The versions of GPU implementation differ in variable data type usage as we optimized the kernel by reducing the size of the data type from the original serialized data type of unsigned short to uint_8. This reduction provided negligible improvements for smaller matrix sizes of less than 4000 square-elements, yet resulted in a **9X** speedup for our target matrix size of 32,000.

End to End Test

Doing an end to end test of hamc is very simple. You can run the generate executable with the flag ”-a test” to run the test suite, and specify each input parameter like such ”-n 2 -p 2000 -w 10 -w 120” To run the program with CPU based execution, you pass the ”-c” flag. The test suite simply encrypts data, and then decrypts it right away. No input data needs to be given when testing, because randomly generated bits of data are generated for both the GPU and CPU based execution,

using the same seed. The seed is by default 10 but can be changed by passing the "-s X" flag where X is the seed value you want. To understand the impact that input parameters have on the workload, the most important parameter is p which determines the dimensions of the square matrix used during all the operations mentioned in Section III-B.

input parameters (n, p, t, w)	CPU time (minutes)	GPU time (minutes)	speedup
2, 2000, 10, 120	0.07	0.04	1.7
2, 4800, 20, 60	7.57	1.36	5.55
2, 6000, 20, 60	14.78	2.26	6.53
2, 1200, 20, 60	117.73	13.24	8.89
2, 1200, 20, 60	938.88	70.46	**13.32**
2, 32771, 264, 274	938.88+	N/A	N/A

Table IV: Timing results for End To End execution

The goal for the end-to-end test was to be able to run the entire program on a matrix of size 32,771 which would offer 256 bits of security[3, 2], but unfortunately there is not enough memory on the GPU to store the necessary matrices to do the LU decomposition operations for the inverse. To combat this, blocking LU decomposition would be the next step, but due to the sake of time, that algorithm was not implemented for this project.

VI. Conclusion

Overall, the McEliece cryptosystem could be improved greatly by GPU based acceleration. Although, not as much as was hoped, there were some great learnings from this implementation. The largest one was the lack of binary arithmetic support in a lot of CUDA standard libraries. The cryptosystem does rely on very large keys for proper security, thus making it difficult to use GPUs for this system due to lack of memory on the GPU. Rather than trying to create CUDA programs, effort should be made in increasing security of the cryptosystem with smaller key sizes.

VII. Acknowledgement

We would like to thank Dr. Ali Akoglu for his tremendous support in this project. Without Akoglu's guidance and help, this project would not be where it is. Special thanks to Michael Chuvelev from the Intel OneAPI MKL team for help on the LU decomposition algorithm for inverse.

References

[1] James R. Bunch and John E. Hopcroft. *Triangular factorization and inversion by fast matrix multiplication*. 1974. DOI: 10.1090/S0025-5718-1974-0331751-8.

[2] Anja Becker et al. "Decoding Random Binary Linear Codes in 2n/20: How 1 + 1 = 0 Improves Information Set Decoding". In: *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT'12. Cambridge, UK: Springer-Verlag, 2012, pp. 520–536. ISBN: 9783642290107. DOI: 10.1007/978-3-642-29011-4_31. URL: https://doi.org/10.1007/978-3-642-29011-4_31.

[3] Rafael Misoczki et al. "MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes". In: July 2013, pp. 2069–2073. DOI: 10.1109/ISIT.2013.6620590.

[4] Alaa Mahmoud Elsobky, Abdelalim Kamal Farag, and Arabi Keshk. *Efficient Implementation of McEliece Cryptosystem on Graphic Processing Unit*. Giza, Egypt, 2016. DOI: 10.1145/2908446.2908491. URL: https://doi-org.ezproxy1.library.arizona.edu/10.1145/2908446.2908491.

[5] C. Paar A. Bogdanov M.C. Mertens. *SMITH - A Parallel Hardware Architecture for fast Gaussian Elimination over GF2*. URL: http://www.hyperelliptic.org/tanja/SHARCS/talks06/smith_revised.pdf.

[6] Hnin Pwint Phyu Louis Fogel. *Cryptanalysis of the McEliece Cryptosystem on GPGPUs*. URL: https://web.wpi.edu/Pubs/E-project/Available/E-project-042915-123714/unrestricted/MQP_FINAL.pdf.

[7] malb. *malb/m4ri*. URL: https://github.com/malb/m4ri.

[8] NVidia. *CUDA Basic Linear Algebra Subroutine*. URL: https://docs.nvidia.com/cuda/cublas.

[9] Varad0612. *Varad0612/The-McEliece-Cryptosystem*. URL: https://github.com/Varad0612/The-McEliece-Cryptosystem.