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— NOTE: 100-150 characters **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 8.89X. This speedup renders the McEliece cryptosystem more realistic for actual use.**

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

I. Introduction

TODO: 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 can not 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 very large matrix operations in all steps of the program. The paper shows how a speedup of **9614X** for matrix multiplication, **9.08X** for matrix addition, and **3.46X** for matrix inverse was achieved to get an overall speedup of **13.32X**.

II. Related Work

The C based code that this project is based off is the McEliece cryptosystem created by Varad0612 [3]. TODO: add more references, especially with OpenMP

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
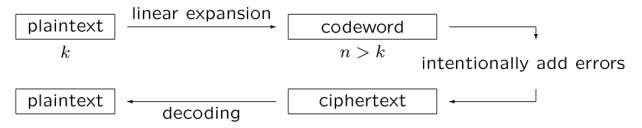
Source: nist.gov

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**.
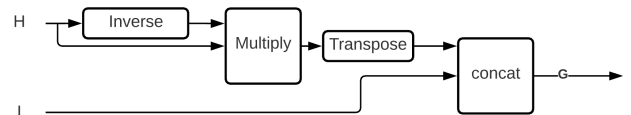
Figure 2: McEliece Keygen Diagram

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.
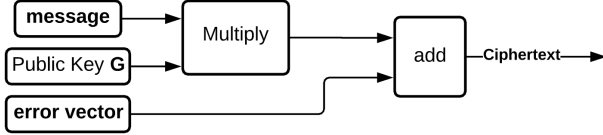

Figure 3: McEliece Encryption Diagram

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.

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[2, 1]. 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[1].

This asymptotic computational complexity is what is referenced when a certain set of input parameters is considered. TODO: input param to bits of security

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. TODO: expand

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 targetted to be sped up, and an overall end-to-end testk that includes generating a pair of keys, encrypting random data, and decrypting random data.

a) Matrix Multiplication Kernel:

TODO: add table of timing

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	0.01
4096	283.79	0.23	0.04
8092	2307.38	0.9	0.24
16384	N/A	5.99	1.71
32768	N/A	46.08	12.53

Table I: Timing results for Matrix Multiplication execution

Matrix Inverse

TODO: add table of timing

Matrix Inversion is a very serialized task, the common Gaussian elimination requires many operations that depend on previous operations. This is not ideal for a GPU when we want to be able to do all operations in parallel. Therefore, in order to do matrix inversion on a GPU, LU decomposition was created to be used instead of regular Gaussian elimination.

LU Decomspotion 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_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 kernel 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 kernel 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 kernel 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 kernel

GF2_Backward_substitute Kernel is really similar to the forward 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.

TODO: fix table timing

matrix dimension	CPU GJ (seconds)	CPU LU (seconds)	GPU LU (seconds)	speedup
512	0.19	0.43	0.57	0.34
1024	1.43	4.43	3.19	0.45
2000	11.43	26.29	12.38	0.90
3050	11.43	26.29	12.38	0.95
4000	37.00	269	44.96	1.19
4400	42.70	N/A	39.60	1.08
6000	375	1010.7	132.49	2.84
12000	862.2	8927.2	249.0	**3.46**
24000	8638	N/A	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

Transpose Kernel

TODO: add information about transpose kernel
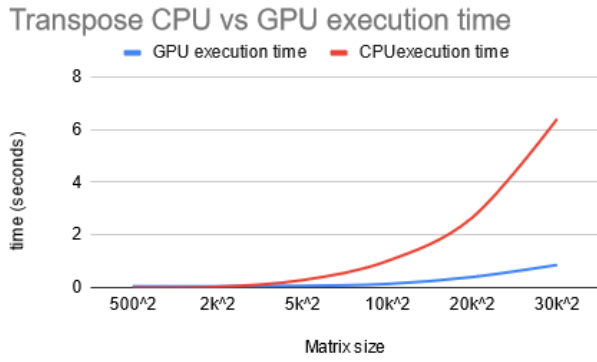
TODO: update timing diagram

Figure 4: Transpose Timing Results

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 III: 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[2, 1], 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 decomspotion would be the next step, but due to the sake of time, that algorithm was not implemented for this project.

VI. Conclusion

TODO: conclusion

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] 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.

[2] 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.

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