Cryptanalysis of the McEliece Cryptosystem on GPGPUs

A Major Qualifying Project Report
submitted to the Faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Louis Fogel, Hnin Pwint Phyu

on April 29, 2015

Approved:

Professor Thomas Eisenbarth,
Advisor

Contents

List of Figures

List of Tables

Abstract

The linear code based McEliece cryptosystem is potentially promising as a so-called "post-quantum" public key cryptosystem because thus far it has resisted quantum cryptanalysis, but to be considered secure, the cryptosystem must resist other attacks as well. In 2011, Bernstein et al. introduced the "Ball Collision Decoding" (BCD) attack on McEliece which is a significant improvement in asymptotic complexity over the previous best known attack. We implement this attack on GPUs, which offer a parallel architecture that is well-suited to the matrix operations used in the attack and decrease the asymptotic run-time. Our implementation executes the attack more than twice as fast as the reference implementation and could be used for a practical attack on the original McEliece parameters.

Chapter 1

Introduction

1.1 Project Motivation

The arrival of quantum computers in the near future would render all commonly used public key cryptographic algorithms easily breakable. This is because number theory based cryptosystems such as RSA and Diffie-Hellman are vulnerable to quantum cryptanalysis [6]. Shor's algorithm for integer factorization breaks the security assumptions of RSA, Diffie-Hellman, and related schemes such as ElGamal, DSA, and ECC. To defend against quantum cryptanalysis using Shor's algorithm, it is necessary to identify public key cryptosystems that are not vulnerable.

There are other classes of systems such as hash based cryptography, code based cryptography and lattice based cryptography which are believed to be resistant to attacks by quantum computers. Such systems make promising candidates for post quantum cryptography, but are less researched than their number theory based cousins. The McEliece cryptosystem is such a public key cryptosystem. McEliece is based on hidden-Goppa-code encryption which was introduced almost forty years ago. The scheme has demonstrated impressive resistance to quantum attacks [17] which makes it an attractive candidate as a post-quantum public-key scheme.

Although the McEliece system is seemingly immune to attacks by quantum computers, the key sizes that it uses for 128 bits of security are close to a million bits which poses issues regarding its efficiency for practical usage. These large key sizes are especially problematic when compared to the key sizes for more commonly used algorithms, such as RSA (a few thousand bits), or ECC (less than one thousand bits). To make matters worse, the cryptosystem parameters proposed by McEliece in 1978 are no longer considered secure. In fact, in 2008 Bernstein et al. successfully executed an attack on McEliece's smaller parameters (which were intended to provide 65 bits of security) in a matter of days [8].

In 2011, Bernstein et al., introduced the "ball-collision decoding algorithm"

[9] which has smaller decoding exponents compared to the previous best known upper bound on these exponents. Recent attacks on the system only use about $2^{n/20}$ operations [3]. These attacks have increased the necessary secure key size even more, further limiting the practicality of the McEliece scheme.

Cryptanalytic improvements are not the only threat to the security of McEliece. The use of Graphical Proccessing Units in public key cryptography has become more widespread and much better studied since 2008 [21]. With the advent of general purpose GPU programming, a parallelized and distributed implementation of attacks on on GPUs is likely to achieve a higher speed up factor for attacks on McEliece at the same economic cost.

The purpose of this project is to, for the first time, create a practical implementation of Bernstein et al.'s Ball-Collision Decoding attack on Graphical Proccessing Units to demonstrate improvements in both hardware any cryptanalysis. This implementation should show the benefits and limitations of GPUs in attacking McEliece and demonstrate the effectiveness of modern McEliece key sizes in resisting decoding attacks.

1.2 General-Purpose Computing on GPUs

Graphics processing units are chains of special-purpose hardware designed for rapid memory manipulations. Their parallel throughput architecture emphasizes executing many concurrent threads slowly, which is more effective when implement algorithms where processing large blocks of data can be done concurrently. GPUs were initially intended for use in accelerating memory-intensive calculations related to three dimensional computer graphics such as texture mapping, rendering polygons, and high-precision color spaces. However, with the realization of the matrix and vector operations underlying the graphical computations, the uses of GPUs now include non-graphical calculations especially in engineering and science applications. This use of GPUs for non-graphical applications is called general-purpose computing on graphics processing units or 'GPGPU'. Put differently, GPGPU is the utilization of a graphics processing unit to perform traditional CPU computations. GPGPU has gained popularity as a result of the addition of floating point support to graphics processors. The increase in popularity is also a result of additional support for GPGPU programming languages such as CUDA (which is discussed below).

GPU-accelerated applications are designed to run the sequential part of their workload on the CPU – which is optimized for single-threaded performance – while accelerating parallel processing on the GPU. OpenCL (Open Computing Language) is one framework that makes it easier to reformulate computations for parallel programming across different kinds of hardware (using both CPU and GPU). NVIDIAs CUDA (Compute Unified Device Architecture) is a parallel computing platform that allows programmers to ignore underlying graphical concepts in favor of more common high-performance computing concepts through extensions to industry-standard programming languages such as C, C++, and CUDA accelerated libraries.

The processing power of modern GPUs has been increasingly utilized among different disciplines of academic research, including cryptography. In 2008, Szerwinski and Güneysu [32] employed GPU acceleration to asymmetric cryptosystems RSA, DSA and Elliptic Curve Cryptography (ECC). In 2009 Harrison and Waldron [21] also presented performance improvements on GPU implementations of public key cryptography (specifically RSA) and the necessary criteria to achieve those improvements. An IT security research group in RWTH Aachen University Aachen, Germany has worked on accelerating block ciphers using the CUDA framework in 2012 [20]. While the area of cryptography on GPUs has experienced significant growth in the last decade, to our knowledge little work has focused on code-based cryptography, and we do not know of any implementations of cryptanalytic attacks on code-based systems on GPUs.

1.3 Project Overview

The goal of this project is to expand on the work in [9] to create an efficient, parallelized implementation of the ball collision decoding attack which could be used against weak McEliece parameters. This project will not only use the improvements in cryptanalysis since Bernstein et al.'s 2008 attack on the original McEliece parameters was published, but will also take advantage of improved computing hardware [8]. Unlike previous implementations of attacks on McEliece, our implementation will take advantage of Graphical Processing Units, which allow more parallel computation within iterations of the attack. This will allow more iterations of the attack than a CPU-only implementation.

In this paper we present, for the first time, a parallel implementation of Bernstein et al.'s ball collision decoding attack. Unlike previous work, we attempt to parallelize computation within iterations of the attack to take full advantage of the GPU architecture and decrease the cost of the attack. We attempt to speed up the ball collision decoding attack by parallelizing the Gaussian elimination and set matching operations of the algorithm. Using the results from our implementation, we estimate the practical cost of an attack on the original McEliece parameters would be $400 million using rented infrastructure. We then discuss the practical implications of GPU implementations of information set decoding attacks on the security of McEliece.

Using our parallel implementation we are able to perform individual iterations of the ball collision decoding attack in less than half the time of the reference implementation. This implementation would allow us to break the original McEliece parameters. The format of this paper as follows: In this chapter we introduce the project and its motivation. In Chapter 2 we provide background information about the McEliece cryptosystem, the Niederreiter cryptosystem, and techniques for accelerating applications using GPUs. In Chapter 3 we summarize the cryptanlysis of the McEliece cryptosystem and describe the Ball-Collsion decoding algorithm. In Chapter 4 we describe our implementation of the Ball-Collsion decoding algorithm. In Chapter 5 we compare the performance of our implementation to previous attacks on McEliece

and propose new economic costs for breaking the McEliece cryptosystem. In Chapter 6 we conclude the paper and discuss potential future work on attacking the McEliece cryptosystem.

Chapter 2

Background Information

2.1 Linear Codes

Linear codes are error-correcting codes with the property that any linear combination of codewords can serve as a codeword itself. Linear codes have more structure added to the codespace compared to other codes. A linear code C of length n and dimension k over the finite field F is a linear subspace of F. Thus, C is an $[n, k]$ linear code over F and the codewords of the codespace F are vectors. Linear codes are can be chosen such that they are more efficient for encoding and decoding algorithms than other types of codes. Hamming codes are an example of linear codes that are commonly used for error-correction in digital communications.

The problem of decoding a generic linear code (i.e. a code without a known efficient decoding mechanism) is known to be NP-hard. Code-based public-key cryptosystems such as McEliece base their security on the difficulty of this problem by using a generic linear code as the public key and a more efficient code as the private key. The difficulty of the generic linear decoding problem is discussed in more detail in Chapter 3. As mentioned before, there are no known quantum algorithms that solve the generic linear decoding problem.

2.2 The McEliece Cryptosystem

The McEliece Cryptosystem, hence McEliece, is a public key cryptosystem designed and published by Robert J. McEliece in 1978 [27]. Unlike more popular public key cryptosystems such as RSA and Diffie-Hellman, no known efficient attacks are present for McEliece that make use of Shor's algorithm for integer factorization, or any other quantum algorithm.

The McEliece scheme makes use of a randomly selected code from a family of codes which can be decoded efficiently, and sets the description of this code as a private key [30]. The private key then undergoes secret transformations to emerge as a general linear code in order to produce the public key. As previously

mentioned, the problem of decoding these general linear codes is NP-hard. The transformations are meant to hide any visible structure of the private key which can be used to reveal the underlying code.

The McEliece public key is derived from the private key after a scrambling transformation and a permutation. The McEliece system uses binary Goppa codes as private key since they are known to be easy to decode. Other error-correcting codes have been proposed as alternatives to binary Goppa codes, but most of these variants have been broken [6].

One advantage to the McEliece scheme is that the encryption and decryption algorithms are not computationally expensive. In fact, they can actually be substantially faster than RSA [7]. This performance, in addition to the resistance to quantum cryptanalysis, makes the McEliece system a promising candidate for post quantum public-key cryptography. However, a few drawbacks have kept McEliece cryptosystem from being widely adopted to be in use today. One major factor being the very large key size of several hundred thousand bits (compared to ECC key sizes of less than a thousand bits) and the other being that it is not semantically secure against adaptive chosen ciphertext attacks [30].

2.2.1 Key Generation

The first part of McEliece system is a probabilistic key generation algorithm which produces a private and public key. Parameters of the underlying $[n, k, d]$ binary Goppa code (linear code) are defined by an irreducible polynomial of degree t over $GF(2^m)$ called the Goppa polynomial and they act as common system parameters for the McEliece system. A binary code of length $n = (2^m)$ each corresponds to the former mentioned polynomial. The code is of dimension k where k exceeds n-mt and has minimum distance d where d is one more than twice the number of errors efficiently correctable by a decoding algorithm. Note that the public parameters (which control the key size and security of the scheme) for the cryptosystem are (n, k, t).

The steps of the key generation are as follows:

1. Select a binary (n, k, d)-linear Goppa code C capable of correcting t errors. The code must possess an efficient decoding algorithm and generates a $k \times n$ generator matrix G for the code C.
2. Select a random $k \times k$ invertible binary matrix S.
3. Select a random $n \times n$ permutation matrix P.
4. Compute the $k \times n$ matrix $\hat{G} = SGP$.
5. The public key (which will be used in 2.2.2) is $(\hat{G}, t)$ and the private key (which will be used in 2.2.3) is (S, G, P) .

The first step of the Mceliece key generation algorithm is choosing a random binary code C (as mentioned before). The private key is the generator matrix G for C. Next, a random scrambling matrix of size $k \times k$, which we call S, is

selected and transformation of G takes place. The result gets sent into a G' matrix obtained from SG. G' becomes another generator matrix for the same C since S is invertible. Then an $n \times n$ permutation matrix P is also randomly selected for the purpose of reordering G' columns and producing $\hat{G}$, which is SGP. Since P is a permutation, $\hat{G}$ is a generator matrix for a C equivalent linear code, with the same rate and minimum distance but no existing efficient decoding algorithm. $\hat{G}$ therefore serves as the public key [30].

2.2.2 Encryption

The encryption is carried out by multiplication of a k-bit message vector with the generator matrix $\hat{G}$, the public key of the message's intended recipient. This encoded message is then added to a random error vector with a Hamming weight value no greater than t [30]. The time complexity of this encryption is $O(k/2n + t)$ [18]. To send a message m to a person with public key $(\hat{G}, t)$:

1. Encode the message m as a binary string of length k.
2. Compute the vector $c' = m\hat{G}$.
3. Generate a random n-bit vector z containing exactly t ones (i.e. the vector z has length n and weight t).
4. Compute the ciphertext as $c = c' + z$.

2.2.3 Decryption

Knowledge of the P permutation is needed to decode a linear code $\hat{C}$ equivalent to a binary Goppa code C. First, the permutation transformation needs to be reversed then, the decoding algorithm for C is used to also decode the permuted cipher text $\hat{c}$ to a message equivalent to Sm. The original m can be retrieved by reversing the scrambling transformation S, and using the inverse, S^{-1}, from the key generation step [30]. To recover the message m from the ciphertext c using the private key:

1. Compute the inverse of P, P^{-1}.
2. Compute $\hat{c} = cP^{-1}$.
3. Use the decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m}$.
4. Compute $m = \hat{m}S^{-1}$.

2.3 Niederreiter Cryptosystem

The Niederreiter Cryptosystem is a dual variant of the McEliece system proposed by Harald Niederreiter in 1986 [28]. Like the original McEliece system, the system is based on linear codes and relies on the same NP-hard problem for

its security. Both systems undergo a scrambling transformation and a permutation transformation to hide the underlying structure. However, Niederreiter differs from McEliece in that the encryption scheme of Niederreiter describes codes through parity-check matrices and uses a parity check matrix of length n as its public key. The encryption algorithm takes as input words $W_{q,n,t}$ of weight t where t is the number of errors that can be decoded [4].

2.3.1 Key Generation

1. Choose (n, k, t) such that $W_{2,n,k,t} \geq 2^{kd}$
2. Pick a random $(n-k) \times n$ parity-check matrix H_0 of linear Goppa code G
3. Select a random $n \times n$ permutation matrix P
4. Randomly select a $(n-k) \times (n-k)$ invertible matrix S
5. Calculate public key matrix H as $H = SH_0P$
6. Public key output is (H, t) and private key is (S, H_0, P, γ). γ is a t-bounded decoding algorithm for the binary Goppa code G.

2.3.2 Encryption

1. Calculate for a given message m a binary string of length n and weight $wt(m) \leq t$
2. The ciphertext output c is calculated as $c = H \times m^T$

2.3.3 Decryption

1. Compute $S^{-1}c = HPm^T$
2. Apply an efficient syndrome decoding algorithm for G and recover Pm^T.
3. Retrieve message m from $m^T = P^{-1}Pm^T$

2.3.4 Digital Signature Scheme

Before the emergence of the dual variant Niederreiter Cryptosystem, it was thought that the McEliece system (and systems like it) could not be used for producing digital signatures because the encryption scheme of the original McEliece is not invertible. In 2011, Courtois, Finiasz, and Sendrier [19] introduced a method to find parameters (n, k, t) that would allow the Niederreiter scheme to be practically invertible. From this method they derive a digital signature scheme, which is often referred to as the CFS signature scheme.

2.4 GPU Acceleration

GPU-accelerated general purpose computing became popular around 2007 with the introduction of NVIDIA's CUDA programming language. Unlike the traditional approach of letting the CPU handle an application from the beginning to completion, a GPGPU application uses a GPU alongside the CPU to speed up computation intensive tasks. A visualization of the process of accelerating CPU tasks is shown in Figure 2.1. The GPU architecture is massively parallel, with thousands of cores that are designed to efficiently handle simultaneous tasks while CPUs are more optimized for sequential processing and have relatively few cores. In GPGPU programming, once the compute-heavy parts of an application code have been identified, these portions are offloaded on to the GPU for faster processing while the rest of the code continues on sequentially.

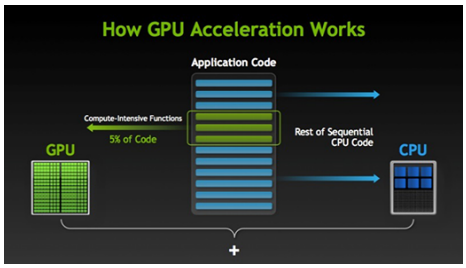

Figure 2.1: How GPU Acceleration works, figure adapted from [16]

NVIDIA's Compute Unified Device Architecture, or CUDA, is a parallel computing platform that makes GPU accelerated computing possible. CUDA provides extensions to C and C++ and allows development of a parallelized version of standard applications. CUDA also provides three key abstractions, namely thread groups, shared memory and barrier synchronization to help achieve fine-grained data parallelism [15]. Through CUDA, a problem can be repeatedly partitioned into sub-problems that can be solved independently and in parallel. Problems that are well suited to GPUs involve using the same process repeatedly on different data. This paradigm is known as SIMD, or Single Instruction Multiple Data.

2.4.1 Turing Cluster Architecture

Our version of the ball collision decoding algorithm is run on GPU compute nodes on a high performance computing cluster at WPI called the Turing Cluster. The Turing Cluster is a good example of what a GPU cluster designed to break would look like.

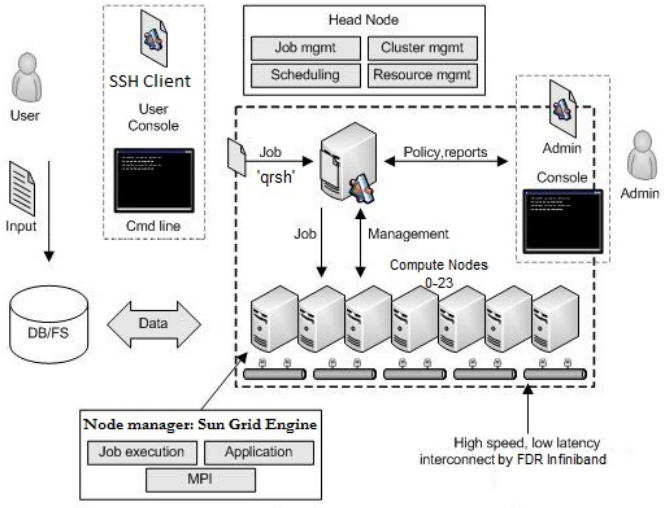

Figure 2.2: Turing Cluster Overall Architecture

Each of the 24 nodes of the cluster consists of 2 Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz,each with 10 CPU cores for a total of 20 cores and 2 NVIDIA K-20 modules [1]. It is equipped with 128 GB of RAM for the CPU. However, the GPUs only have 5GB of memory each which is what will determine the upper bound for the problem size that our implementation will be able to solve. It uses Rocks, a Linux distribution designed for use with clusters. A 56GB/s FED Infiniband adapter serves the interconnect between the compute nodes.

The nodes are managed by Sun Grid Engine. When the user logs in to turing.wpi.edu through SSH, they get to the head node (front end node) which manages the cluster and its resources. The head node sends job requests to Sun Grid Engine which manages job execution on the nodes using a queuing system. The `qrsh` command can be used to obtain an interactive compute session on one of the 24 compute nodes. The executable file can be produced on the head node using `nvcc` compilation commands and then through textttqrsh a compute session should be activated in order for the code to execute on one of the GPU compute nodes.

2.4.2 NVIDIA K20 Architecture

The NVDIA Tesla K20 module uses the GK110 Kepler architecture [13] which is the newest, high performance computing architecture after Fermi. The Kepler architecture uses compute capability 3.5 and features a new generation of streaming multiprocessors known as 'SMX', which allows more space for processing cores than control logic. Kepler's new ISA encoding has also brought the number of registers per thread up to 255. Some of the highlights of the new

architecture include the following:

- Dynamic parallelism which allows the GPU to spawn new threads on its own, and possess more control capabilities without involving the CPU.
- Hyper-Q simultaneously connects up to 32 CPU cores to the GPU for a higher GPU utilization.
- Grid Management Unit enables dynamic parallelism.
- Shuffle instruction allows data to be shared among threads in the same block/warp and reduces the demand for shared memory.

2.4.3 Inside the Tesla K-20 module

The K-20 has a transistor count of 7.1 billion, and is equipped with 13 functional SMX units and a 5GB GDDR5 memory. Each SMX has 192 single precision CUDA cores, 64 double precision units, 32 special function units and 32 load store units, which makes up to 2496 CUDA cores in total. A visualization of the GK1110 SMX is shown in Figure 2.3. Each SMX has 48KB of read-only data cache accessible by a thread block and 256KB of register file space, 64KB of L1 cache, 48KB of uniform cache, and 1536KB of dedicated L2 cache memory. A visualization of the memory hierarchy for a CUDA core is shown in Figure 2.4

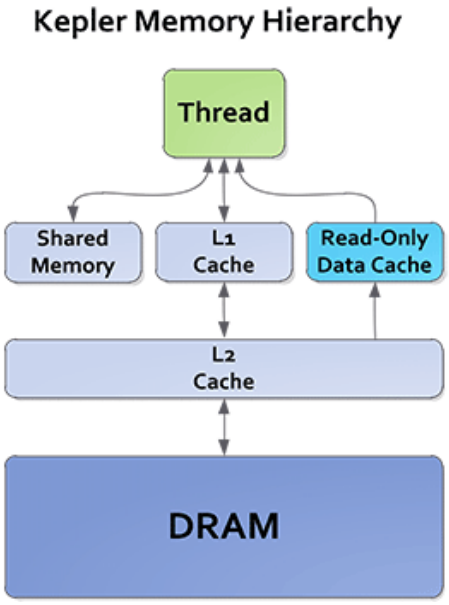

Figure 2.4: Memory hierarchy inside the GPU, figure adapted from [13]

CUDA threads are grouped into instruction units called warps. Each warp contains up to 32 threads. The Kepler quad warp scheduler selects four warps,

and two independent instructions per warp can be dispatched each cycle. A visualization of the Warp scheduler is shown in Figure 2.5.

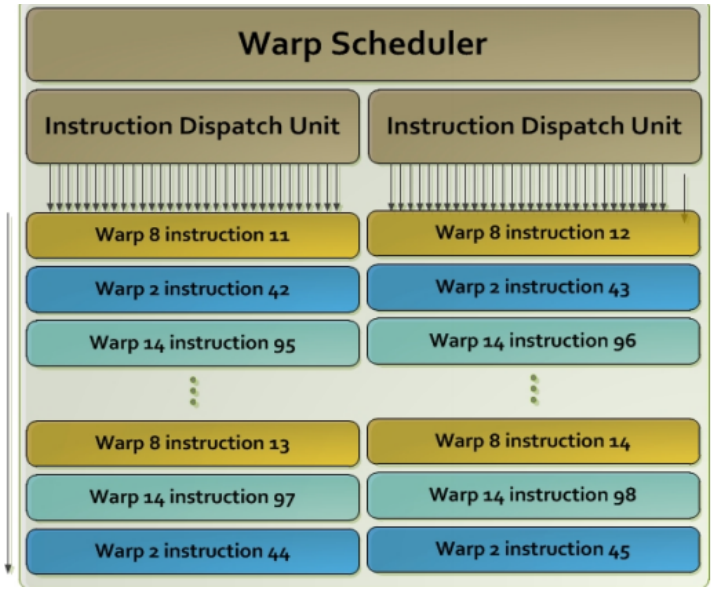

Figure 2.5: Warp scheduling in the GPU, figure adapted from [13]

GPU activity is managed by a hardware unit called the CUDA Work Distributor (CWD). The CWD assigns work to the individual multiprocessors and communication is done through the MPI (message passing interface). 32 simultaneous MPI tasks can be performed with Hyper-Q in Kepler.

2.4.4 CUDA Basics: Vector addition parallelized

The following is an example implementation of parallelized vector addition using threads and blocks [35].

Algorithm 1 GPU Kernel for parallel vector addition: Part 1

$\_global\_void\ add(int * a, int * b, int * c)$
$int\ index = threadIdx.x + blockIdx.x * blockDim.x;$
$c[index] = a[index] + b[index];$

The GPU kernel (function) that operates on the GPU to perform the addition of two vectors is shown in Algorithm 1. The keyword `__global__` is required for all GPU functions that will be called by the CPU. This keyword separates code that needs to be processed by the NVIDIA compiler `nvcc` from the rest that can be processed by standard compilers. A block is a parallel invocations of the kernel. These blocks can be defined for up to 3 dimensions (x,y,z) and

can be accessed inside the kernel using `blockIdx`. A block can again be spilt up into threads that run in parallel and those are accessed using `threadIdx`. A built-in variable `blockDim` for referring to the number of threads per block is also provided to allow indexing into parallel threads and parallel blocks. By indexing into the integer arrays using the thread and block indexes `threadIdx` and `blockIdx`, a specific number of concurrent additions of different values can take place as the same time on different threads running on the GPU's processing cores.

Algorithm 2 GPU Kernel for parallel vector addition: Part 2

$int\ *a, *b, *c;$ // host copies of a, b, c
$int\ *d_a, *d_b, *d_c;$ //device copies of a, b, c
$int\ size = N * sizeof(int);$
// Alloc space for device copies of a, b, c
$cudaMalloc((void**)\&d\_a, size);$
$cudaMalloc((void**)\&d\_b, size);$
$cudaMalloc((void**)\&d\_c, size);$
// Copy inputs to device
$cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice);$
$cudaMemcpy(d\_b, b, size, cudaMemcpyHostToDevice);$

In CUDA terminology, the CPU is referred to as the host, and the GPU the device. For use on the GPU, separate device variables have to be declared and then allocated using `cudaMalloc` as seen above. The default values from the host are copied onto the device using a cuda built-in `cudaMemcpy` function.

Algorithm 3 GPU Kernel for parallel vector addition: Part 3

// Launch add() kernel on GPU
$add <<< N/BLOCK\_SIZE, BLOCK\_SIZE >>> (d\_a, d\_b, d\_c);$
// Copy result back to host
$cudaMemcpy(c, d\_c, size, cudaMemcpyDeviceToHost);$

After the memory copy and the set up of variables, the kernel can be called from main, using a special kernel launch format with triple angle brackets. The first element inside the angle brackets refer to the number of blocks. The second parameter for the kernel refer to the number of threads. At the end of the kernel execution, results are fetched back again with `cudaMemcpy`.

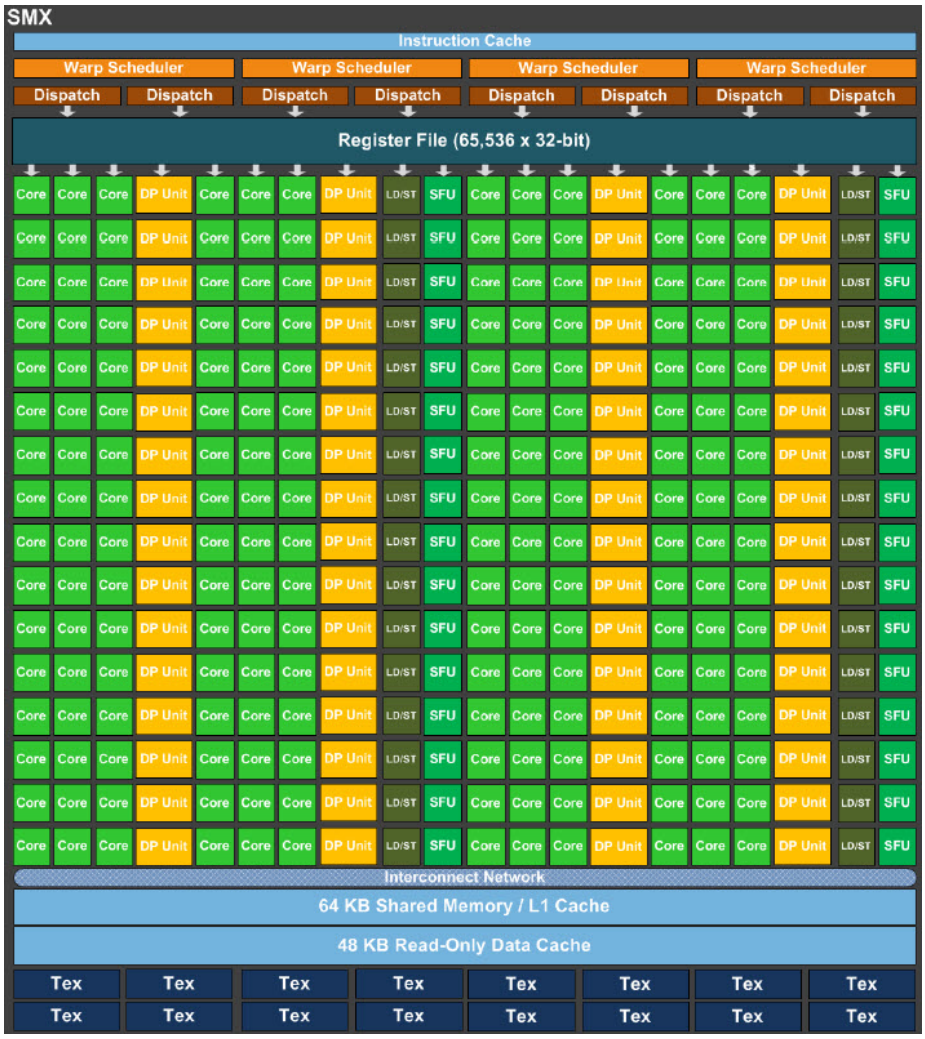

Figure 2.3: Streaming multiprocessor architecture of the gk110 GPU, figure adapted from [13]

Chapter 3

Previous Cryptanalysis

3.1 Original Parameters

The security of the McEliece cryptosystem is based around the assumed difficulty of the general decoding problem for nonlinear codes. This problem was proven NP-complete in [5]. Because the assumption of difficulty of the general decoding problem remains unchallenged, the scheme remains, in theory, secure. However, the parameters proposed by McEliece in [27] fail to account for improvements in the general decoding problem, as well as improvements in computing hardware. Even without improvements to the complexity of the general decoding problem, McEliece's proposed "secure" work factor of 2^{65} would not be considered sufficient for long-term security today.

But even if 2^{65} were enough to guarantee long term security, the formula McEliece proposed for calculating the work factor no longer represents the best known attack. In the original proposal of the cryptosystem McEliece proposed the following formula for approximating the work factor:

$$k^3 \left(1 - \frac{t}{n}\right)^{-k} \tag{3.1}$$

but improvements in general linear decoding algorithms have rendered this formula obsolete.

3.2 Information Set Decoding

Although the general decoding problem is NP-complete, there are still probabilistic algorithms that can improve the performance of special cases of the problem, decreasing the security of cryptographic schemes based around the problem [11]. The method for finding low-weight codewords described in [31] and then applied to the McEliece cryptosystem in [11] is the first of the so called "information set decoding" attacks.

Like many cryptanalytic attacks, parts of Stern's algorithm, and related algorithms, are "embarassingly parallel", that is, they can trivially be split into multiple threads that are executed simultaneously. McEliece can be broken exponentially faster than McEliece's original work factor expression shown in Equation 3.1 by exploiting the embarrassingly parallel nature of Stern's algorithm. In 2008 Bernstein, Lange, and Peters [8] published the first major breakthrough in over ten years. Their optimized implementation of an information set decoding algorithm similar to Stern's was able to break McEliece's original parameters in less than a week.

3.3 Ball-Collision Decoding

In 2011 Bernstein, Lange, and Peters [9] presented a new algorithm which improves on the algorithm presented in [31]. They claim this algorithm allows attackers to decode random linear codes in $\tilde{\mathcal{O}}(2^{0.05558n})$.

Ball-collision decoding has its roots in information-set decoding, like previous attacks. But unlike those attacks, which select a random information set in the parity-check matrix and then search for vectors having a particular pattern of non-zero entries, Ball-collision decoding searches for a more complicated, and more likely pattern [9]. This approach is what makes BCD more efficient that previous attacks, and accounts for the slightly lower upper bound on asymptotic complexity that of the BCD attack.

The steps of the BCD algorithm, taken from [9]:

1. Choose a uniform random information set Z. "F_2^Z" will be used to denote to the subspace of F_2^n supported on Z.

2. Choose a uniform random partition of Z into part of sizes k_1 and k_2. The subspaces formed by these partitions will be denoted as "$F_2^{k_1}$" and "$F_2^{k_2}$".

3. Choose a uniform random partition of $\{1, 2, \ldots, n\} \setminus Z$ into parts of sizes l_1, l_2, and $n - k - l_1 - l_2$. The subspaces formed by these partitions will be denoted as "$F_2^{l_1}$", "$F_2^{l_2}$" and "$F_2^{n-k-l_1-l_2}$".

4. Find an invertible matrix U such that $U \in F_2^{(n-k)\times(n-k)}$ and the columns of UH indexed by $\{1, 2, \ldots, n\} \setminus Z$ form the $(n-k) \times (n-k)$ identity matrix I_{n-k}.

5. Write Us as $\binom{s_1}{s_2}$ with $s_1 \in F_2^{l_1+l_2}$,and $s_2 \in F_2^{n-k-l_1-l_2}$.

6. Compute the set S consisting of all triples $(A_1x_0 + x_1, x_0, x_1)$ where $x_0 \in F_2^{k_1}$, $wt(x_0) = p_1$, $x_1 \in F_2^{l_1}$, and $wt(x_1) = q_1$.

7. Compute the set T consisting of all triples $(A_1y_0 + y_1 + s_1, y_0, y_1)$ where $y_0 \in F_2^{k_2}$, $wt(y_0) = p_2$, $y_1 \in F_2^{l_2}$, and $wt(y_1) = q_2$.

8. For each $(v, x_0, x_1) \in S$:
   For each y_0, y_1 such that $(v, y_0, y_1) \in T$:

If $wt(A_2(x_0 + y_0) + s_2) = w - p_1 - p_2 - q_1 - q_2$:
Output $x_0 + y_0 + x_1 + y_1 + A_2(x_0 + y_0) + s_2$.

The new work factor, proposed in [9] is:

$$min\left\{\binom{n}{w}\binom{n-k}{w-p}^{-1}\binom{k}{p}^{-1/2} : p \geq 0\right\} \tag{3.2}$$

Additionally, that work factor can be split between as many processors as an attacker can afford. Not only can many iterations of the algorithm detailed above be executed in parallel, but different steps of the algorithm can also be parallelized. In particular, both of the loops inside of step 8 can be unrolled and evaulated simultaneously. The gaussian eliminaton from step 4 can also be executed in parallel for a significant decrease in attack time. Our approach for optimizing individual steps of the BCD algorithm can be found in Section 4.1.

3.4 Practical Attacks

Although significant improvements have been made to the cryptanalysis since 2008, none of these new attacks have been implemented for systems on the scale necessary to break anything but toy parameters since [8]. The purpose of this project is to show that not only has the hardware necessary for the attack improved, but the cryptanalysis has also improved substantially since 2008, and practical attacks on McEliece are even easier than those in [8]. In addition, the ability to rent hardware in the cloud (so-called Infrastructure as a Service) has made the capital investment necessary to perform an attack much lower because attackers no longer need to build and maintain their own supercomputers.

In Chapter 4, we will describe an implementation of the ball-collision decoding algorithm on massively parallel graphical processing units. Then, in Section 5.3 we will propose new estimates for the economic cost of practical attacks on McEliece's original parameters as well as the more secure parameters proposed in [9].

Chapter 4

Ball Collision Decoding on GPUs

4.1 Analyzing the Ball Collision Decoding Algorithm

4.1.1 Introduction

In 2011 Bernstein, Lange, and Peters published a new attack on McEliece that they called Ball Collision Decoding. They claim that this algorithm can decode random linear codes in $\tilde{\mathcal{O}}(2^{0.05558n})$ [9] (the algorithm is discussed in further depth in 3.3). Provided with the paper is a reference implementation of Ball Collision decoding which counts the number of bit operations it performs to confirm the predictions of the paper. Unlike in their previous work in [8], the reference code for Ball Collision Decoding is almost entirely unoptimized.

4.1.2 Profiling the Reference Implementation

By far the most common operation used in bcd.cpp is a bit addition the + operator for the bit class). This operation is actually a bitwise XOR (on a single bit). The fact that the majority of execution time is spent performing bit operations implies that the task is compute-intensive and could be sped up if executed in parallel.

For attacks on McEliece's original parameters, most of these bit operations happen during the Gaussian elimination that occurs during step 4 of the algorithm. The Gaussian elimination step is a good target for parallelization because of the large number of bit operations involved and because many of these operations can be done in parallel.

Table 4.1: Cryptosystem Parameters

Parameters	n	k	w	Security (bits)
Original	1024	524	50	49.69
Improved [9]	3178	2384	68	128
Set 1	30	11	8	negligible
Set 2	768	300	30	21

4.1.3 Attacking Cryptosystem Parameters

The reference implementation is provided with hardcoded cryptosystem parameters (Set 1 Table:4.1) to allow the program to run to completion fully as a fast check on the results of [9]. These toy parameters are also good as an illustration of the operation of the algorithm, and as a fast check that it has been implemented correctly. Because of the iterative nature of the algorithm, there is not much use for an intermediate key size (i.e. a key size between the toy parameters and the original proposed parameters) because the complexity of the total attack can be verified on the toy parameters and the complexity of a single iteration of the attack is relatively trivial even for the largest proposed key sizes. Even with a faster algorithm and improved hardware, we would not expect to be able to break any of the parameters proposed in [8].

Using the work factor equation 3.2 proposed in [9], we chose another set of toy parameters (Set 2 Table 4.1) which are slightly larger than the reference implementation's toy parameters and provide up to 21 bits security.

Without any significant breakthroughs in cryptanalysis, the upper bound of a realistic attack on commonly used McEliece parameters using university resources is still the original proposed parameters, $(n, k, w) = (1024, 524, 50)$ [27]. Although we do not expect to be able to break much larger parameters than [8], we do expect to be able to implement a practical attack with better asymptotic bounds.

Though a successful attack on both the 128-bit and 256-bit equivalent parameters is impossible, testing attacks on $(n, k, w) = (3178, 2384, 67)$ and $(n, k, w) = (6944, 5208, 136)$ should provide useful information on the memory requirements of performing attacks similar to ball collision decoding on GPUs. If the memory resources required to perform such an attack are realistic, then it is possible an improvement to the cryptanalysis could make such an attack feasible.

4.2 CUDA tools for optimization

The GK110 graphical processing unit that lies inside our NVIDIA K20 modules has the compute capability of 3.5, which is the most recent version for devices based on Kepler architecture. Thus, it offers the highest instruction through-

put among different CUDA-capable devices. We follow the CUDA performance guidelines to achieve maximum instruction throughput. Our optimized algorithm mainly makes use of the bitwise integer operations such as AND, OR and XOR, which can perform up to 160 operations per clock cycle per multiprocessor. Another optimization strategy we follow is maximization of utilization of the hardware at all three levels :

- Application level
- Device level
- Multiprocessor level

We intend to employ as much parallelism as possible and efficiently map this parallelism to various components of the system so that we can keep it busy most of the time. We maximize utilization with different mechanisms such as scheduling concurrent kernels, or parallel threads and choosing the optimal block and grid sizes for kernel execution.

4.3 Finding an Information Set in Parallel

Step 4 of the algorithm requires finding an invertible matrix U such that $U \in F_2^{(n-k)\times(n-k)}$ and the columns of UH indexed by $\{1, 2, \ldots, n\} \setminus Z$ form the $(n-k)\times(n-k)$ identity matrix I_{n-k}. This is achieved through Gaussian Elimination of the matrix UH. The algorithm for this step was first presented by Stern in [31].

Implementation of this step in parallel has implications for the performance of other information set decoding attacks, not just ball collision decoding [3, 8, 9, 26, 31]. A practical implementation of an attack on the McEliece cryptosystem must include an efficient Gaussian elimination.

4.3.1 GF(2) Gaussian Elimination on GPUs

Gaussian elimination on GPUs is a relatively obvious and well studied problem. Efficient implementations of Gaussian Elimination for solving linear systems have been proposed in [33]. However, these implementations are typically focused on scientific computing, and are therefore designed to be efficient for matrices of floating point numbers. Efficient algorithms for operations on matrices in finite fields can differ significantly from those for floating point matrices [25, 29].

The implementation of $GF(2)$ Gaussian Elimination that we use is a modified version of the parallel algorithm presented in [25]. The modifications stem from the fact that our implementation is targeted at GPUs, whereas the algorithm proposed in [25] is for purpose-built linear algebra parallel processors. Because the architectures of these processors are very similar, few changes were needed to make the algorithm efficient on GPUs.

4.3.2 Parallelized Generation of Information Set

Using a parallelized modification of the algorithm presented in [31] we have implemented step 4 of the ball-collsion decoding algorithm on GPUs.

As previously mentioned, the information set Z is found using a Gaussian elimination operation on the matrix H until the columns of H indexed by $\{1, 2, \cdots, n\} \setminus Z$ are linearly independent. Fruitless Gaussian elimination steps are omitted using the optimization described in [31].

Using the GPU we are able to process the matrix operations in parallel. For each of the $n - k$ linearly independent row of the matrix we perform the following operations:

1. Find an element j in the information set Z corresponding to a nonzero (and unreduced) column of the row.
2. Swap rows such that the reduced matrix will form an $(n - k) \times (n - k)$ identity matrix.
3. Add the current row to all rows that are nonzero in the jth column.

In our parallel implementation we use up to $n - k$ threads to construct a sparse representation of the current row (using indexes into Z) and randomly select an element of Z from the sparse representation as our value of j. This approach is equally efficient for dense and sparse rows, which allows its performance to stay constant as the rows become progressively more and more reduced.

The row swapping operation is executed in a single thread on the GPU because it is simply a swap of two elements of Z. This single threaded operation would, of course, be faster on the CPU, but that would require keeping CPU and GPU copies of Z in sync during the Gaussian elimination, which would cancel out any performance gain from performing it on the CPU.

The row additions are executed in parallel, with a separate block of threads for each row ($n-k$ total blocks) and a separate thread in each block for each row ($(n - k) \times (n - k)$ total threads). The row operations are entirely independent of each other once j has been found, thus, we should experience a speedup by a factor of up to $n - k$. This approach lets us utilize as many simultaneous threads of execution as possible on the CPU and it ensures that because threads within the same block will be accessing the same row of memory. When the operations are done with one row per block, we can take full advantage of the GPU's memory pipeline and achieve a high level of performance.

4.4 Parallelized Generation of Output

Step 8 of the algorithm serves as a good example of different types of parallelism in our implementation. The step 8 in the reference implementation involves many levels of nested for loops which results in the high operation counts as observed in the cost comparison table (Table 5.2). Our implementation of step

8 breaks the loops into parallel threads and blocks of sizes as big as the sizes of the output matrices.

We make use of different variable qualifiers to define the scope of GPU only variables such as `device` whose scope is within a grid, `shared` for variables accessible within a block and `local` for a thread. The `blockIdx` and `threadIdx` are three dimensional variables which are used to index into each parallel invocation of the device kernel being executed. The nested for loops are replaced with such threads and blocks by setting the loop variables to the thread and block indices to maximize the computation speed.

The device function `sycnthreads()` is used to make sure that all the threads finish their work before proceeding to the next portion of the code which will likely require the values produced in the preceding lines. The 2-dimensional arrays or matrices are allocated using `cudaMallocpitch()` function in cuda which is optimized for 2-D arrays and make use of the pitch of the array for indexing. `cudaMallocpitch()` allocates 2-D arrays with a memory pitch that is optimized for better caching for row access.

4.4.1 Step 8 Setup

Before actually generating possible outputs of the algorithm, we must first determine which elements of S and T should actually be compared. To do this, we have a lightweight kernel (shown in Algorithm 4) that runs a thread for each combination of two elements of S and T. Using the data structure provided by the reference implementation, we determine if the two elements corresponding to the thread satisfy the equation $A_1x_0 + x_1 = A_1y_0 + y_1 + s_1$ (from Section 5 of [9]). This kernel yields a list of structs with pointers to pairs elements of S and T that constitute matches.

This step does not result in a significant speedup because there are a lot of threads with not very much to do, but it does ensure that memory accesses are mostly adjacent, which significantly increases the cache hit rate in the step 8 kernel (described in Section 4.4.2).

Algorithm 4 GPU Kernel for Step 8 setup

if $thread \geq Tlen$ **then**
  return
end if
$this\_match.t = \&T[thread]$
for $(inti = head[this\_match.t- > sum.gpu\_index()]; i >= 0; i = S[i].next)$ **do**
  $this\_match.s = \&S[i]$
  $intmatch\_idx = atomicInc(\&g\_match\_len,\ 0)$
  $matches[match\_idx] = this\_match$
end for

4.4.2 Step 8 GPU Kernel

The step 8 kernel runs on the matches generated by the step 8 setup kernel (described in Section 4.4.1). The algorithm for generating output from elements of S and T is described in Section 3.3. We parallelize it to create a new thread for every match, which allows for many operations to be executed simultaneously. In addition, the structure of the kernel allows all of the loops to be fully unrolled, which allows each thread to take advantage of the pipelined architecture of the streaming processor. Finally, the order in which the list of matches is generated ensures that accesses to the set T will be local (and near sequential), which allows the GPU to take advantage of its cache, and better utilize its instruction-level parallelism.

The result of these optimizations is $> 90\%$ thread occupancy on the GPU, with a per-thread IPC of ≈ 2.5. The main bottleneck of this kernel is probably cache misses from accesses to the set S, which are unpredictable. Pre-sorting S in the setup step could increase the cache hit rate, but would incur significant additional overhead, which would likely offset any perfomance increase from the better cache hit-rate.

4.5 Algorithm Memory Requirements

The memory requirements for most cryptosystem parameters scale either linearly or quatdratically with the cryptosystem parameters. The major exception to this rule are the sets S and T, whose size are the product of two binomials. Note however, that the attacker can optimize their attack for the amount of memory they have by scaling the attack parameters p, q, and l to adjust the size of S and T. Note also that the size of head scales exponentially with l_1 and l_2. While this seems like it could also be a limiting factor, in reality the value of $l_1 + l_2$ is always less than 30, which keeps the size of head relatively manageable. The choice of attack parameters allows the attacker to make a time/memory tradeoff that matches the resources they have available.

On a GPU the size of S and T is the most significant consideration in the allocation of global memory. The NVIDIA Tesla K20 has 5GB of memory, which is easily enough for iterations of an attack on the original McEliece parameters, and possibly even the 128-bit eqivalent parameters proposed in [9]. The 256-bit equivalent parameters cause S and T to be incredibly large, making the attack unfeasible before even considering the enormous computational requirements.

It is important to note that because step 8 of the ball collision decoding algorithm (described in section 3.3) must be run for every possible pairing of S and T, it is nontrivial to distribute both S and T over nodes in a cluster. Ideally, at least one of the sets should fit in the memory of a single node – or better yet, a single GPU – so that one of the sets can be partitioned across nodes (and/or GPUs) and matched against the set that fits in memory. For cryptosystem parameters that approach the limits of what can practically fit in memory it may be possible to choose the values of k_1 and k_2 such that one of the sets fits in

Table 4.2: Memory Allocation Requirements for Named Variables

Name	Type	Size (bytes)		Dimension
		Set 1	Set 2	
head	int	8	4	$2^{l_1+l_2}$
H	bit	2280	1437696	$(n-k) \times n$
target	bit	120	3072	n
roworder	int	76	1872	$n-k$
U	bit	1444	876096	$(n-k) \times (n-k)$
s	bit	76	1872	$(n-k)$
UH	bit	2280	1437696	$(n-k) \times n$
Us	bit	76	1872	$(n-k)$
Z	int	120	3072	n
rowcol	int	76	1872	$(n-k)$
Fk1	int	24	600	k_1
Fk2	int	20	600	k_2
Fl1	int	16	32	l_1
Fl2	int	20	32	l_2
Fnkl1l2	int	40	1808	$(n-k-l_1-l_2)$
lprep	bit	36	64	l_1+l_2
A1	lbits	44	600	k
A2	bit	440	542400	$k \times (n-k-l_1-l_2)$
s1	lbits	4	4	1
s2	bit	40	1808	$n-k-l_1-l_2$
S	custom struct	960	1430400	$\binom{k_1}{p_1}\binom{l_1}{q_1}$
T	custom struct	12	1072800	$\binom{k_2}{p_2}\binom{l_2}{q_2}$
A2x0y0	bit	40	1808	$n-k-l_1-l_2$
output	bit	120	3072	n
soutput	bit	76	1872	$n-k$

memory and the other does not. When possible however, the parameters should be roughly equivalent to decrease the computational complexity of the attack.

Different types of memory spaces exist on the device such as register, local, global, shared, constant and texture. Only the register and shared memory spaces are actually located on the GPU chip and the rest are located off-chip, although local, constant and texture memory are cached. Global and constant variables are accessible from both device and host while variables in shared and local memory are only visible to the device. Any global variable to be processed on the device requires to be allocated separately on the device through `cudaMalloc` and the data from host space is copied on to the device using `cudaMemcpy`. Also after the kernel has been executed, another `cudaMemcpy` call is required to transfer data back from device to host memory space.

Our GPU kernels mainly make use of global, local and shared memory. Most of the matrices in the algorithm are declared as global variables since they require both GPU and CPU processing. The overhead of data transfer back and forth between the host and the device for these global values only takes up about less than 1% of the total execution time as seen in the performance analysis table in the results section. The memory allocation of device variables also takes up a negligible constant overhead. Shared memory space is where most of the action takes place with every thread in a block having access to the variables in this space. Our implementation makes use of this memory space for variables that are to be updated by every thread. A synchronization point is also required after writing to a shared memory space if the next operations are dependent on the shared memory. We utilize a `syncthreads()` function to achieve this synchronization.

The CUDA library contains a function that allows host memory to be pinned for use with the device called `cudaMallocHost`. The function allows the allocated data to be directly accessible from the device thus allowing a higher bandwidth with read/write operations. It also provides a speed-up when used along side `cudaMemcpy` operations. In our implementation, instead of calling a `cudaMemcpy` for each value, integer variables are allocated using `cudaMallocHost()` which speeds up the total execution time by approximately 15% compared to using normal cudaMemcpy.

Chapter 5

Results

5.1 Experimental Methodology

The reference implementation has a cost variable which updates every time an arithmetic operation is performed on one of its defined data classes. This 'cost' variable tracks the total number of operations at each point of the algorithm. After analyzing the reference implementation, and identifying parts of the algorithm which have heavy costs, we started by moving those certain parts onto the GPU. We then proceeded with the parallelization by defining an optimal number of grid and block sizes which execute our GPU kernels concurrently on the device. With each new kernel implemented, we keep frequent timing measurements. We use the Linux 'time' command which outputs a number of timing statistics such as elapsed real time read from the wall clock, user CPU time for the process and CPU time used by the system. We also make use of the GNU profiler 'gprof' and the NVIDIA profiler 'nvprof' for our performance analysis. The profilers can produce several different styles of outputs including

- Flat Profile, which shows the total amount of time the program spent to execute each function.
- The call graph, which shows function calls and time spent in each subroutines.
- The line-by-line option to profile each line of code.
- The annotated source option to produce profiled information along with respective source code from the program.

5.2 Measured Perfomance Comparison

The following tables summarize timing and speed-wise comparison of how our version of the algorithm performs alongside the reference implementation from 2011 with the toy parameter sets defined in Table 4.1.

Table 5.1: Results with toy parameters

	GPU		Reference	
Parameter	Set 1	Set 2	Set 1	Set 2
Avg. iterations to target	43.27	11128	43.2722	11128
Gauss cost /iteration	5269.84	8.40e07	5269.84	8.40e07
S cost/iteration	195	268200	195	268200
T cost/iteration	115	270584	115	270584
Match cost/iteration	103.56	2.44e07	103.56	2.44e07

In 2008, Bernstein implemented a successful attack on the original McEliece parameters. Their attack recovers the plaintext from a chosen ciphertext by decoding 50 errors. The attack takes about 1400 days or 2^{58} CPU cycles when run on a single computer with a 2.4GHz Intel Core 2 Quad CPU, which results in about a speed up factor of 150 after taking out the hardware improvement contributions. One attack iteration takes 6.3 million CPU cycles on a Core 2 Quad.

The graph below summarizes of our results from the practical attack.

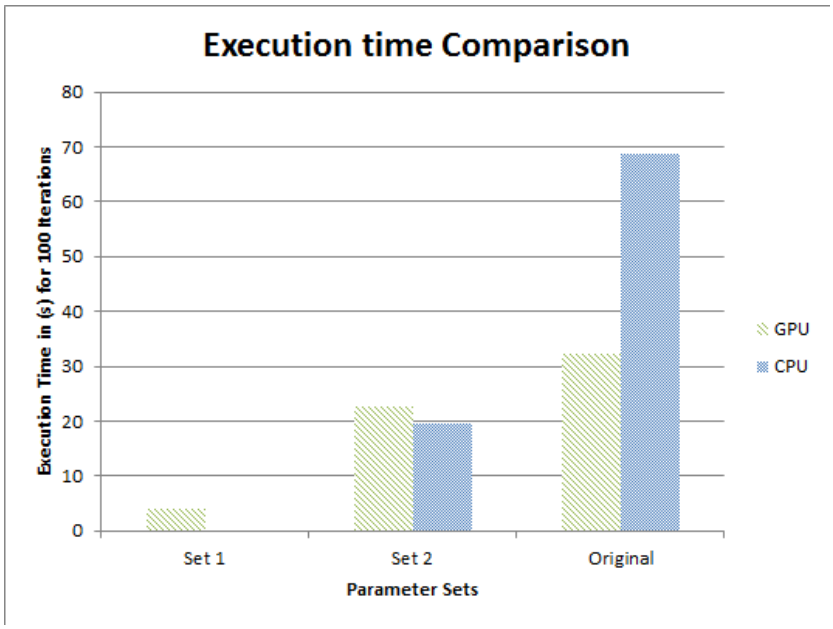

Figure 5.1: Execution Time Comparison between CPU and GPU

5.2.1 Performance Analysis of the GPU implementation of Ball Collision Decoding

Table 5.2: Performance Analysis

Kernel Name	Total Time	Avg. Time/Call	# of calls	% Time
Step8	129.82s	129.82 ms	1000	49.11
row_elim	78.46s	156.92 μs	500000	29.68
findj	47.77s	95.53 μs	500000	18.07
memcpy HtoD	2.55s	170.18 μs	15000	0.97
PreStep8	2.03s	2.03 ms	1000	0.77
Step4_Gen_A1_A2	1.80s	1.80 ms	1000	0.68
update_cols	1.76s	3.52 μs	500000	0.67
memcpy DtoH	164.38ms	32.88 μs	5000	0.06

Table 5.2.1 summarizes the profiling results of the GPU implementation of Ball Collision Decoding for 1000 iterations of an attack on the original McEliece parameters. The first column shows a list of all GPU calls in the algorithm including the memory copy operations to and from the device to the host. The GPU device functions (kernels) are named according to both their functions and the corresponding steps in the original BCD algorithm.

These results are consistent with the expected costs of different parts of the algorithm. The most significant conclusion we draw from them is that the memory bandwidth of the GPU is not a limiting factor on our implementation, because the percentage of GPU execution time spent copying memory between GPU and CPU memory is negligible. This means that further optimization of GPU kernels, especially the `Step8` kernel, would significantly improve performance.

5.3 Economic Cost Analysis

The cost analysis in [8] assumes a 2.4GHz Intel Core 2 Quad Q6600 CPU, which was the state of the at the time.

Since 2008, not only has hardware improved substantially due to Moore's law, but GPGPU has exploded, making massively parallel GPUs much cheaper. In the 7 years since [8] was pubished, hardware that is cheaper, faster, more parallel, and less power-hungry has become available.

For embarassingly parallel problems such as the generic decoding attack, GPUs can offer impressive speedups over CPUs for a fraction of the cost due to the parallelism of their architecture. The NVIDIA Tesla K20s used in the Turing cluster cost $3,018.99 each. They have a 706MHz core clock and 2496 streaming processors. They also have 5120 MB of total memory. The Tesla K20

can compute up to 3520 GLOPS, two orders of magnitude more than the 38.40 GFLOPS for the Q6600 CPU.

But GPUs aren't the only hardware improvements that have been made in the last 7 years. The technology in CPUs has also made impressive improvements over the state of the art in 2008. The CPUs in the Turing cluster are 2.80GHz Intel Xeon E5-2680 v2. They cost $2,091.12 each. They have 10 cores. The Xeon E5-2680 v2 can compute up to 199.43 GFLOPS, over 5 times more than the processor from 2008. There are two per node in the cluster. For the $200000 cost in [8] we could buy 40 pairs of CPUs and GPUs, which would be a substantial improvement over the 2008 hardware.

The cost of hardware isn't the only cost associated with running a high performance computer. In terms of hardware utilized for this project, the NVIDIA k2 GPU uses about 225 Watts of board power and 25 Watts when idle [14]. The Intel Xeon E5-2680 v2 dissipates about 115 Watts of power with all cores active [12].

Attackers don't need to build their own GPU cluster to perform this attack. In fact, the attack could be performed entirely in the cloud using rented infrastructure. The idea of evaluating cryptograpic key strengths based on the cost to break them has been explored before [24]. But previous work on breaking cryptographic keys does not address McEliece. In addition, since 2012 the cost of infrastructure as a service (IaaS) has decreased significantly (with the cost of hardware).

An Amazon EC2 g2.2xlarge instatnce has 8 CPU cores, 15 GiB of RAM, and a "High-performance NVIDIA GPU with 1,536 CUDA cores and 4GB of video memory" [22]. These instances can be rented hourly for $0.650 per Hour, for an entire year for $3478, or for three years for $7410 [23]. So-called "spot" instances can also be rented at prices that are tied to market demand, but predicting the market value of GPU instances is outside the scope of this paper.

Based on our results on the turing cluster we expect that a single g2.2xlarge instance would be capable of performing at least 1000 iterations per hour on the original McEliece parameters. This means to execute an attack on the cryptosystem parameters from [8] would require roughly $400 Million using infrastructure rented by the hour using the attack parameters from our tests. It is likely that more optimized attack parameters exist that would bring this cost down in terms of both time and money. Note that this attack is more expensive than Bernstein et al.'s 2008 attack. Although BCD's asymptotic complexity is much better than that of the 2008 attack, the iterations are much more complex, which makes the attack less efficient for smaller parameters.

Chapter 6

Conclusions

6.1 Summary

The McEliece system is not only promising for post-quantum cryptography with its immunity to Shor's algorithm but also a highly secure system with the security growing along with the key size. The original system using Goppa codes is still resistant to cryptanalysis.

Information set decoding attack and its variants are the most effective attacks against the system. In 2008, Bernstein et.al [8], introduced a practical attack on original parameters with a series of improvements to the 1989 Stern's algorithm [31]. In 2011, Bernstein et.al [9] came up with ball-collision decoding which reduces the upper bound of the decoding exponent and provides a speedup exponential in n to [31]. This attack is parallel in nature with most matrix operations carried out in iterations and thus is a suitable candidate for GPU acceleration.

Compared to clusters with 10 cores per CPU, a single NVIDIA K20 GPU contains about 2500 processing cores, on which threads can run concurrently thus decreasing the asymptotic runtime of the attack.

After analysis of the reference implementation of ball-collision decoding, the Gaussian elimination steps (Step 1+4) and the final join operation step between S and T (Step 8) are identified to involve compute-heavy operations and possible non-sequential processing. These two steps are the main targets of the GPU acceleration and respective GPU kernels were implemented to allow for parallel processing of these steps while the rest of the implementation mainly runs sequentially on the GPU.

CUDA profiling tools for GPU are utilized to analyze the performance of the kernels and look for possible improvements and further optimizations. With the help of the profiler, certain points in the algorithm that were slowing things down were easily identified and improved upon. The proper and effective allocation of block and grid sizes for the kernels also play a big role in ensuring that the implementation make use of all available resources and provides the highest

parallel throughput the device can achieve.

We have succeeded in speeding up the Ball-Collsion decoding algorithm by implementing it on GPUs. Our implementation is two times faster than the reference implementation provided by Bernstein et al. Although our implementation does not improve cost of the attack against the orginal McEliece parameters from [8] it does show that Ball-collsion decoding can be used in practical attacks.

We have also shown that the matrix operations associated with information-set decoding attacks are well-suited to the parallel architecture of GPUs. Unlike previous attacks, we focus on parallelizing within iterations of the attack to decrease the cost of individual iterations. In particular, we show that the algorithm presented in [31] is particularly well suited to GPU hardware and a parallel implementation can easily outperfom a single-threaded implementation by at least a factor of two.

While our work represents a significant optimization of existing cryptanalysis, it does not change the asymptotic bounds of attacks on McEliece. The use of GPU hardware can make difficult attacks more feasible, but these attacks will only be possible on cryptosystem parameters that were already considered insecure. The McEliece cryptosystem is far from broken; it is still a strong candidate for post-quantum public key cryptography. But future users of the cryptosystem will need to ensure that secure key sizes, such as those proposed in [9], are used.

6.2 Future Work

While the work that we present represents a significant increase in the performance of BCD attacks, we believe that there is still much potential for future work on practical attacks on McEliece. We believe that future work could explore the use of distributed computation to scale attacks better. We also believe that the BCD attack can be further optimized to perform even better on GPU and CPU hardware. In addition, we believe that future work could parallelize the algorithm even further, allowing for better utilization of GPU resources. Finally, we recognize that newer cryptanalysis could provide even better results for attacks on McEliece and could also be targeted for GPU implementations.

6.2.1 Distributed Computation

Although we used a computational cluster for testing our implementation, we did not attempt to distribute iterations between nodes of the cluster. In addition, we did not attempt to divide work between available resources on the same node. Future work could divide iterations of the BCD attack between different nodes of a cluster, distributing the work across many compute nodes to allow for even greater parallelism in the attack.

6.2.2 Further Optimization

Although we have achieved a significant speedup over the reference implementation by moving computation to GPUs, we do not believe that our work represents the best possible implementation of BCD. The number of load, store, add, and multiply operations performed on individual bits could be reduced significantly if bits were packed into integers rather than stored individually. The advantages to this approach would be more efficient storage (an improvement by a factor of 32) of matrices and vectors as well as reduced cost of addition and multiplication of vectors (again, by a factor of 32). The main disadvantage would be additional cost to index into a vector because of the need for additional operations to extract a particular bit.

This technique would likely be necessary to mount an attack against parameters significantly larger than the original McEliece parameters because of the limited memory and threads available on a single GPU.

We chose not to attempt this optimization because we believed it would make the code error-prone and difficult to understand. In the best case, it could provide a decrease in time and memory requirements by a constant factor of 32.

6.2.3 Parallel Set Generation

In our analysis of the BCD algorithm we determined that the generation of the sets S and T was the most difficult part of the algorithm to parallelize. Unfortunately, if the sets are generated in a single thread we must choose attack parameters that keep the complexity of these operations comparable to the Gaussian elimination and matching steps. This means that to take advantage of the GPU we must choose attack parameters that require a very large number of iterations to solve.

If the set generation were paralleled, better attack parameters could be used, and the number of iterations required for a successful attack would drop significantly. This would likely decrease the overall cost of the attack significantly and make the attack more practical.

6.2.4 Further Cryptanalysis

The most recently published cryptanalysis by May, Meurer, and Thomae [26] claims the ability to decode random linear codes in $\tilde{\mathcal{O}}(2^{0.05363n})$. Unlike the papers published by Bernstein et. al. [8, 9], May, and Thomae did not make an effort to optimize their code, although they did make an implementation of their algorithm available. Because their attack is also related to information set decoding, much of the work presented in this paper should be applicable to a GPU implementation of the attack.

Bibliography

[1] IT Academic and WPI Research Computing. C8220 turing cluster worker nodes, 2015.

[2] Martin Albrecht, Gregory Bard, and William Hart. Efficient multiplication of dense matrices over gf (2). *arXiv preprint arXiv:0811.1714*, 2008.

[3] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in 2 n/20: how 1+ 1= 0 improves information set decoding. In *Advances in Cryptology–EUROCRYPT 2012*, pages 520–536. Springer, 2012.

[4] Thierry P. Berger, Pierre-Louis Cayrel, Philippe Gaborit, and Ayoub Otmani. *Progress in Cryptology AFRICACRYPT 2009*, chapter Reducing Key Length of the McEliece Cryptosystem. Springer Berlin Heidelberg, 2009.

[5] Elwyn R Berlekamp, Robert J McEliece, and Henk CA Van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[6] Daniel J Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-quantum cryptography*. Springer, 2009.

[7] Daniel J Bernstein and Tanja Lange. ebacs: Ecrypt benchmarking of cryptographic systems, 2015.

[8] Daniel J Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In *Post-Quantum Cryptography*, pages 31–46. Springer, 2008.

[9] Daniel J Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: ball-collision decoding. In *Advances in Cryptology–CRYPTO 2011*, pages 743–760. Springer, 2011.

[10] Anne Canteaut, Hervé Chabanne, et al. A further improvement of the work factor in an attempt at breaking mceliece's cryptosystem. 1994.

[11] Florent Chabaud. On the security of some cryptosystems based on error-correcting codes. In *Advances in CryptologyEUROCRYPT'94*, pages 131–139. Springer, 1995.

[12] Intel Coporation. Intel xeon processor e5-2680 v2, 2015.

[13] NVIDA Coporation. Nvida kepler gk110 architecture whitepaper. 2012.

[14] NVIDIA Coporation. Tesla k20 gpu accelerator board specification, 2012.

[15] NVIDIA Coporation. Cuda c programming guide, 2015.

[16] NVIDIA Coporation. What is gpu computing, 2015.

[17] Hang Dinh, Cristopher Moore, and Alexander Russell. The mceliece cryptosystem resists quantum fourier sampling attacks. *arXiv preprint arXiv:1008.2390*, 2010.

[18] D. Engelbert, R. Overbeck, and A. Schmidt. A summary of mceliece-type cryptosystems and their security, 2006.

[19] Matthieu Finiasz and Nicolas Sendrier Dr. Digital signature scheme based on mceliece. *Encyclopedia of Cryptography and Security*, 2011.

[20] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. Gpu-acceleration of block ciphers in the openssl cryptographic library. In *Information Security*, pages 338–353. Springer, 2012.

[21] Owen Harrison and John Waldron. Public key cryptography on modern graphics hardware. *Booklet of posters, Eurocrypt*, 200, 2009.

[22] Amazon Inc. Amazon ec2 instances, 2015.

[23] Amazon Inc. Amazon ec2 reserved instances, 2015.

[24] Thorsten Kleinjung, Arjen K Lenstra, Dan Page, and Nigel P Smart. Using the cloud to determine key strengths. In *Progress in Cryptology-INDOCRYPT 2012*, pages 17–39. Springer, 2012.

[25] Çetin K Koç and Sarath N Arachchige. A fast algorithm for gaussian elimination over gf (2) and its implementation on the gapp. *Journal of Parallel and Distributed Computing*, 13(1):118–122, 1991.

[26] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{\wr}(2^{0.054n})$. In *Advances in Cryptology–ASIACRYPT 2011*, pages 107–124. Springer, 2011.

[27] Robert J McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.

[28] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *PROBLEMS OF CONTROL AND INFORMATION THEORY-PROBLEMY UPRAVLENIYA I TEORII INFORMATSII*, 15(2):159–166, 1986.

[29] Dennis Parkinson and Marvin Wunderlich. A compact algorithm for gaussian elimination over gf (2) implemented on highly parallel computers. *Parallel Computing*, 1(1):65–73, 1984.

[30] Olga Paustjan. Post quantum cryptography on embedded devices; an efficient implementation of the mceliece public key scheme based on quasi-byadic goppa codes. Master's thesis, Ruhr-University Bochum, 2010.

[31] Jacques Stern. A method for finding codewords of small weight. In *Coding theory and applications*, pages 106–113. Springer, 1989.

[32] Robert Szerwinski and Tim Güneysu. Exploiting the power of gpus for asymmetric cryptography. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 79–99. Springer, 2008.

[33] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[34] Wade Trappe and Lawrence C Washington. *Introduction to cryptography with coding theory*. Pearson Education, Inc., 2006.

[35] Cyril Zeller. *CUDA C/C++ Basics*. NVIDIA Coporation, 2011.