Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Coding Theory and Cryptology Group

An Adaption of the Crossbred Algorithm for Solving Multivariate Quadratic Systems over $\mathbb{F}_2$ on GPUs

Master Thesis

Kai-Chun Ning

Supervisors:
prof.dr. Tanja Lange (Technische Universiteit Eindhoven)
prof.dr. Daniel J. Bernstein (University of Illinois at Chicago)
dr. Ruben Niederhagen (Fraunhofer SIT)

version 1.03

Eindhoven, 13th November, 2017

Abstract

Solving multivariate quadratic systems is of great importance for many cryptographic applications. In this master thesis, the state-of-the-art algorithms of solving multivariate quadratic systems are reviewed. Subsequently an adaption of an existing algebraic approach is proposed and implemented. The adapted algorithm is highly parallelizable and is suitable for solving multivariate quadratic systems on the GPU architecture. With one single commercial Nvidia GTX 980 graphics card, the proposed algorithm is able to solve a multivariate quadratic system of 134 equations in 67 variables in 98.39 hours, while the fastest existing algorithm requires 6200 hours for the same problem with one CPU. The experimental results of solving all the Fukuoka type I MQ challenges, where $n = 55 \sim 74$, are also given in this thesis. Based on the result, a multivariate quadratic system of 184 equations in 92 variables is shown to provide 80 bits of security with respect to the proposed algorithm and can be solved with 61617 GTX 980 graphics cards in one year. In addition, this thesis shows that with a cluster of 3500 GTX 980 graphics cards, the underlying multivariate quadratic system of a post-quantum cryptographic scheme which consists of 80 equations in 84 variables could be solved on average in one year.

Acknowledgement

I would like to thank all the people that helped me during this master thesis project. First of all, I would thank my supervisor Tanja for her guidance and her unexpected feedback in the middle of the night. I would also thank Dan, who gave me access to the Saber clusters, which is probably the most luxurious toy that I have ever played with. I would also thank Ruben, who provided me great support and has been the source of inspiration during my stay in Darmstadt. I would also thank the people that I met at Fraunhofer, who have been kind to me. I would also thank the family at *Istanbul*, whose tasty meals made summer 2017 especially productive. I would thank my family for their support for my master study, and my friends, who are always there when I need to take the day off.

Eindhoven
November 2017

Kai-Chun Ning

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

With the advent of quantum computing, an adversary can efficiently break universally adopted public key cryptographic schemes, e.g. RSA and elliptic curves, with a sufficiently large quantum computer. In order to mitigate this imminent threat, cryptographic schemes which are resistant against quantum computers have drawn great attention from academia. These schemes are collectively referred to as post-quantum cryptography.

Multivariate cryptography is considered a potential candidate for post-quantum cryptography. Throughout the past few decades, various asymmetric encryption schemes [53, 24, 55] based on Hidden Field Equations (HFE) [39] as well as signature schemes [22, 25, 52, 42, 17, 28] have been proposed. As for symmetric encryption scheme, a stream cipher QUAD [14] has been proposed and analyzed [59].

Multivariate cryptography relies on the difficulty of solving a system of m polynomial equations in n variables over a finite field. The complexity of solving a multivariate polynomial system ($\mathcal{MP}$ problem) or a multivariate quadratic system ($\mathcal{MQ}$ problem) where coefficients of the monomials are independently and uniformly distributed (i.e. random) is well-known to be NP-hard. In fact, even testing for the existence of one solution is NP-complete [34]. Note that an arbitrary $\mathcal{MP}$ system can be transformed into an equivalent $\mathcal{MQ}$ system by substituting monomials of degree larger than 2 with new variables and introducing extra equations to the system [40].

The difficulty of solving the $\mathcal{MP}$ or $\mathcal{MQ}$ problem can be used to create a one-way trapdoor function. For example, it can be constructed by choosing a matrix P' whose n rows represent an invertible polynomial in one unknown over a finite field $\mathbb{F}_{q^n}$, which is the extension field of $\mathbb{F}_q$. Subsequently, two linear transformation matrices S and T, each of which consists of n polynomials in n variables over $\mathbb{F}_q$, are created. The secret key is the set $\{S, P', T\}$ while the public key is simply the product of those three matrices $P = S \cdot P' \cdot T$ [39]. Note that since higher degree implies more coefficients to store, $\mathcal{MQ}$ systems are preferred over $\mathcal{MP}$ systems for smaller public key size. The invertibility of the secret matrices allows the secret key owner to efficiently compute the inverse while their product P, with proper choice of S and T, appears as a random multivariate system. Nevertheless, the public key is created based on the chosen private key hence inevitably reflects the set $\{S, P', T\}$ as a hidden structure in P, which is often exploited by cryptanalysis on multivariate cryptography schemes [41, 32]. Those scheme specific attacks can therefore solve these seemingly random systems with much lower complexity than completely random systems.

This thesis is not restricted to any particular cryptography scheme and focuses on generic attacks for solving completely random $\mathcal{MQ}$ systems over $\mathbb{F}_2$. Solving $\mathcal{MQ}$ systems over $\mathbb{F}_2$ is essential for cryptanalysis of various cryptographic schemes [39, 42, 17, 18, 54] as these schemes are based on $\mathcal{MQ}$ systems over $\mathbb{F}_2$. Despite the impracticality, it is also possible to recover an AES key by modeling it as an overdetermined $\mathcal{MQ}$ system of 8000 equations and 1600 variables over $\mathbb{F}_2$ [47, 27]. In addition, a polynomial system over any extension field $\mathbb{F}_{2^n}$ can be reduced into an equivalent system over $\mathbb{F}_2$ using Weil descent [36] and, as mentioned above, any $\mathcal{MP}$ system can be transformed into an equivalent $\mathcal{MQ}$ system. Solving $\mathcal{MQ}$ problems over $\mathbb{F}_2$ is the core of aforementioned cryptographic applications and therefore is of great importance.

1.2 Contribution

The contribution of this thesis includes a review of the state-of-the-art algorithms of solving $\mathcal{MQ}$ problems and an adaption of an existing algebraic approach which is suitable for solving $\mathcal{MQ}$ problems on the GPU architecture. In addition, programming techniques and caveats for implementing the proposed algorithm are also documented. With one single commercial Nvidia GTX 980 graphics card, a program called MQsolver implemented according to the proposed algorithm is able to solve an $\mathcal{MQ}$ system of 134 equations in 67 variables in 98.39 hours, while the fastest existing algorithm requires 6200 hours for the same problem instance with one CPU [40]. The experimental results of solving all the Fukuoka type I MQ challenges where $n = 55 \sim 74$ are also given in this thesis. Based on the result, an adversary supported by a nation or a multinational conglomerate is shown to be able to solve an $\mathcal{MQ}$ system of 184 equations in 92 variables, which provides 80 bits of security against the proposed algorithm, with 61617 GTX 980 graphics cards in one year. In addition, this thesis shows that with a cluster of 3500 GTX 980 graphics cards, the expected computation time of solving an $\mathcal{MQ}$ system of 80 equations in 84 variables which was claimed to provide 80 bits of security [54] and is the underlying problem of a post-quantum cryptographic scheme [54] is one year.

1.3 Thesis Organization

The rest of the thesis is structured as follows. Chapter 2 introduces the preliminaries, including mathematical notations and computer architectures. Chapter 3 explains state-of-the-art algorithms for solving random $\mathcal{MQ}$ systems and proposes an adaption based on one of the existing algorithms. Thereafter implementation details and design choices for the proposed algorithm are documented in Chapter 4. Finally, Chapter 5 presents the experimental results, discusses proper choice of various parameters for the proposed algorithm and analyzes their implications.

Chapter 2

Preliminaries

2.1 Mathematical Preliminaries

This section introduces mathematical notations and definitions relevant to this thesis.

2.1.1 Monomial

A monomial in $x_1, x_2, \dots, x_n$ is a product in the form $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$, where the exponents $\alpha_1, \alpha_2, \dots, \alpha_n$ are non-negative integers. This notation can be further simplified to

$$x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n} = x^\alpha, \text{ where } \alpha = (\alpha_1, \alpha_2, \dots, \alpha_n) \in \mathbb{Z}^n_{\geq 0}, \text{ and when } \alpha = (0, 0, \dots, 0),\ x^\alpha = 1.$$

The degree of the monomial x^α is defined as $|\alpha| = \alpha_1 + \alpha_2 + \cdots + \alpha_n$.

2.1.2 Polynomial over a Finite Field

A polynomial f in $x_1, x_2, \dots, x_n$ with coefficients in a finite field $\mathbb{F}_q$, where q is a prime power ($q = p^k$, p is a prime and k is a positive integer), is a finite linear combination of monomials.

$$f = \sum_{\alpha \in \mathbb{S}} a_\alpha x^\alpha,\ a_\alpha \in \mathbb{F}_q, \text{ where } \mathbb{S} \text{ is a finite set of } n\text{-tuples} \in \mathbb{Z}^n_{\geq 0}.$$

If $a_\alpha \neq 0$, then $a_\alpha x^\alpha$ is a term of f. The degree of f, denoted as $\deg(f)$, is the maximal $|\alpha|$ with $a_\alpha \neq 0$. The zero polynomial is the polynomial whose coefficients a_α are all zero.

2.1.3 Polynomial Ring over a Finite Field

The set of all polynomials in $x_1, x_2, \dots, x_n$ with coefficients in $\mathbb{F}_q$ is denoted as $\mathbb{F}_q[x_1, x_2, \dots, x_n]$. Clearly, the sum and product of two polynomials is again a polynomial. A polynomial g divides a polynomial f if $f = g \cdot h$, for some $h \in \mathbb{F}_q[x_1, x_2, \dots, x_n]$. The multiplicative inverse of f is defined as the polynomial g such that $1 = f \cdot g$. Note that $\mathbb{F}_q[x_1, x_2, \dots, x_n]$ is not a field but a commutative ring, which is closed under addition and multiplication. However, for a polynomial f in the ring that is not a non-zero constant polynomial, e.g. x_1, its multiplicative inverse might not exist.

2.1.4 Monomial Order

Since a monomial $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$ can be represented via the n-tuple $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n) \in \mathbb{Z}^n_{\geq 0} \subset \mathbb{Z}^n$, there exists a one-to-one correspondence between the monomials in $\mathbb{F}_q[x_1, x_2, \dots, x_n]$ and $\mathbb{Z}^n_{\geq 0}$. If an ordering in $\mathbb{Z}^n$ is established then so is a monomial order. A polynomial is an $\mathbb{F}_q$-linear combination of monomials therefore the ability to arrange its terms in descending or ascending order is naturally desired. There are many approaches to establishing an ordering in $\mathbb{Z}^n$ but most of the resulting orderings cannot be used to arrange monomials in this manner. To sort monomials in ascending of descending order, the ability to compare every pair of monomials and establish their proper relative position is necessary. The ordering must therefore be a total ordering, which means for every pair of monomials (x^α, x^β), exactly one of the following cases is true

$$x^\alpha > x^\beta,\ x^\alpha = x^\beta,\ x^\alpha < x^\beta,$$

and additionally transitivity should be satisfied

$$x^\alpha > x^\beta \text{ and } x^\beta > x^\gamma \text{ implies } x^\alpha > x^\gamma.$$

Now consider the sum and product of two polynomials. Ordering monomials of the sum is trivial. On the other hand, since multiplication in a polynomial ring distributes over addition, the product of two polynomials can be viewed as the sum of products obtained by multiplying the first polynomial by a monomial from the second polynomial

$$f \cdot g = f \cdot \sum_{\alpha \in \mathbb{S}} a_\alpha x^\alpha = \sum_{\alpha \in \mathbb{S}} f \cdot a_\alpha x^\alpha.$$

It is desirable to keep the leading term of the product $f \cdot a_\alpha x^\alpha$ as the product of the leading term of f and $a_\alpha x^\alpha$, otherwise extra effort for identifying the leading term will be needed. One additional property is hence required

$$\text{if } x^\alpha > x^\beta \text{ then for any } x^\gamma,\ x^\gamma \cdot x^\alpha > x^\gamma \cdot x^\beta,$$

which translates to

$$\text{if } \alpha > \beta \text{ in the ordering on } \mathbb{Z}^n \text{, then for any } \gamma \in \mathbb{Z}^n,\ \alpha + \gamma > \beta + \gamma.$$

Finally, to ensure multivariate division (introduced in Section 2.1.6) eventually terminates, the monomial order should be a *well-ordering*. In other words, every non-empty subset of monomials has exactly one smallest element according to the ordering.

Let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$, both $\in \mathbb{Z}^n_{\geq 0}$. The rest of this section introduces three monomial orders which satisfy the desired properties listed above.

Lexicographical Order

$\alpha >_{\text{lex}} \beta$ (i.e. $x^\alpha >_{\text{lex}} x^\beta$) if and only if the leftmost non-zero entry in the vector $\alpha - \beta \in \mathbb{Z}^n$ is positive. For example,

$$\alpha = (1, 2, 0) >_{\text{lex}} \beta = (0, 3, 4) \text{ since } \alpha - \beta = (1, -1, -4).$$

Note that for variables $x_1, x_2, \ldots x_n$

$$x_1 >_{\text{lex}} x_2 >_{\text{lex}} \cdots >_{\text{lex}} x_n \text{ since } (1, 0, \ldots, 0) >_{\text{lex}} (0, 1, \ldots, 0) >_{\text{lex}} \cdots >_{\text{lex}} (0, 0, \ldots, 1).$$

Graded Lexicographical Order

$\alpha >_{\text{glex}} \beta$ if $|\alpha| > |\beta|$ or $|\alpha| = |\beta|$ and $\alpha >_{\text{lex}} \beta$. In other words, monomials are ordered by their degree first, then ties are broken with lexicographical order.

Graded Reverse Lexicographical Order

$\alpha >_{\text{grlex}} \beta$ if $|\alpha| > |\beta|$ or $|\alpha| = |\beta|$ and the rightmost non-zero entry in the vector $\alpha - \beta$ is negative. For example,

$$\alpha = (1,5,2) >_{\text{grlex}} \beta = (4,1,3) \text{ since } |\alpha| = |\beta| \text{ and } \alpha - \beta = (-3,4,-1).$$

Both graded lexicographical order and graded reverse lexicographical order use monomial degree to order monomials first, then to break ties, graded lexicographical order looks at variables from the left and favors larger powers. In contrast, graded reverse lexicographical order looks at variables from the right and favors smaller powers.

2.1.5 Multidegree of a Polynomial

Given a polynomial f as in Section 2.1.2 and a monomial order, define

1. multidegree of f: multideg(f) is the largest non-zero α in $\mathbb{S}$ with respect to the monomial order,
2. leading coefficient of f: LC(f) is the coefficient $a_{\text{multideg}(f)}$,
3. leading monomial of f: LM(f) is the monomial $x^{\text{multideg}(f)}$, with coefficient 1, and
4. leading term of f: LT(f) is the term $a_{\text{multideg}(f)} \cdot x^{\text{multideg}(f)}$, i.e. LC($f$) $\cdot$ LM(f).

2.1.6 Multivariate Division

With the definition of multidgree and a monomial order, univariate polynomial long division and the remainder thereof can be extended to multivariate polynomials. Given a monomial order and an *ordered* m-tuple $(f_1, f_2, \ldots, f_m)$ of polynomials in a ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$, multivariate division[1] can decompose an arbitrary polynomial f in the ring into

$$f = q_1 \cdot f_1 + q_2 \cdot f_2 + \cdots + q_m \cdot f_m + r$$

where the remainder r is either 0 or a linear combination of monomials which are not divisible by any of the leading terms LT(f_1), LT(f_2), ..., LT(f_m) with coefficients in the field $\mathbb{K}$.

Note that the reminder depends on the order of the tuple hence may not be unique. For example, let $f_1 = xy - 1$, $f_2 = y^2 - 1, f = xy^2 - x$ and use lexicographical order $x > y$. Since $f = y(xy-1) + y - x = x(y^2 - 1)$, dividing f by (f_1, f_2) yields $r = -x + y$ while (f_2, f_1) yields $r = 0$.

2.1.7 Multivariate Polynomial Systems

An $\mathcal{MP}$ system is a set of m multivariate polynomial equations $f_1 = 0, f_2 = 0, \ldots, f_m = 0$ in n variables $x_1, x_2, \ldots x_n$, where coefficients of the polynomials belong to a field[2] $\mathbb{K}$. A solution to the system is an n-tuple in $\mathbb{K}^n$ that satisfies all m equations. If the maximal degree of the equations is two, then the system is referred to as an $\mathcal{MQ}$ system.

2.1.8 Variety

Given an $\mathcal{MP}$ system as in Section 2.1.7, the *variety* of the system, denoted as $\mathbb{V}(f_1, f_2, \ldots, f_m)$, is the set of all solutions for the system.

$$\mathbb{V}(f_1, f_2, \ldots, f_m) = \{(a_1, a_2, \ldots, a_n) \in \mathbb{K}^n : f_i(a_1, a_2, \ldots, a_n) = 0, \text{ for all } 1 \leq i \leq m\}.$$

Trivially, a variety is a subset of $\mathbb{K}^n$. A variety can be empty. For example, $\mathbb{V}(x^2 + y^2 + 1) = \emptyset$ when the field is $\mathbb{R}$.

[1] See Appendix A.1 for the algorithm.

[2] Not necessarily finite.

2.1.9 Ideal

A subset $\mathbb{I}$ of a polynomial ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$ is called an *ideal* if the following properties are satisfied

1. The zero polynomial belongs to $\mathbb{I}$.
2. If f, g are in $\mathbb{I}$, then $f + g \in \mathbb{I}$.
3. If f is in $\mathbb{I}$ then for any polynomial h in the ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$, $f \cdot h \in \mathbb{I}$.

An ideal is similar to a subring. Both have to be closed under addition and multiplication, except that for an ideal polynomials from the whole ring $\mathbb{K}[x_1, x_2 \ldots, x_n]$ instead of just from the subring can be used for multiplication. Given an $\mathcal{MQ}$ system as in Section 2.1.7, the set of polynomials generated by $f_1, f_2, \ldots, f_m$

$$\langle f_1, f_2, \ldots, f_m \rangle = \left\{ \sum_{i=1}^{m} h_i \cdot f_i, \text{ where } h_1, h_2, \ldots, h_m \text{ are arbitrary polynomials in the ring} \right\}$$

is an ideal of $\mathbb{K}[x_1, x_2, \ldots, x_n]$ and is referred to as the ideal generated by $f_1, f_2, \ldots, f_m$. An ideal $\mathbb{I}$ is *finitely generated*[3] if there exist generators $f_1, f_2, \ldots, f_m$ in the ring such that $\mathbb{I} = \langle f_1, f_2, \ldots, f_m \rangle$. If so, the set $\{f_1, f_2, \ldots, f_m\}$ is referred to as a basis for the ideal $\mathbb{I}$. An ideal may have multiple different bases.

A variety $\mathbb{V}(f_1, f_2, \ldots, f_m)$ is determined by the ideal $\mathbb{I} = \langle f_1, f_2, \ldots f_m \rangle$ generated by its defining polynomials. If $\mathbb{I}$ can be generated by another basis $\{g_1, g_2, \ldots g_{m'}\}$, then the varieties defined by both bases are identical. Therefore by changing basis, it may become easier to analyze a variety. This concept is essential to algebraic approaches for solving $\mathcal{MQ}$ problems, which are introduced in Chapter 3.

2.1.10 Ideal of Leading Terms

Given a monomial order, each f in a polynomial ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$ has a unique leading term $\mathrm{LT}(f)$. Let $\mathbb{I}$ be an ideal $\subseteq \mathbb{K}[x_1, x_2, \ldots, x_n]$, then

1. $\mathrm{LT}(\mathbb{I})$ is defined as the set of leading terms of non-zero elements of $\mathbb{I}$.
2. $\langle \mathrm{LT}(\mathbb{I}) \rangle$ is the ideal generated by elements of $\mathrm{LT}(\mathbb{I})$.

2.1.11 Macaulay Matrix

A Macaulay matrix represents a system of polynomials extended from a base system $\mathcal{F}$ of m polynomials of degree $\leq d$ in n variables. The *Macaulay degree*, denoted as D, is the maximal degree of the polynomials in the extended system. Each row in a Macaulay matrix is the product of a polynomial f in $\mathcal{F}$ by a monomial t such that $\deg(t \cdot f) \leq D$. The columns of a row represent coefficients of the monomials in the product and are ordered by a monomial order. The rows can be ordered arbitrarily but in the rest of this thesis, they are arranged in descending order with respect to $>_{\mathrm{grlex}}$ based on their multiplier. Rows with an identical multiplier form a group where they keep the same order as their base polynomials in $\mathcal{F}$. For example, let $\mathcal{F}$ be an $\mathcal{MQ}$ system of two polynomials in four variables x_1, x_2, x_3, x_4 over finite field $\mathbb{F}_2$

$$\mathcal{F} = \begin{cases} f_1 = x_1x_2 + x_2x_3 + x_3 \\ f_2 = x_1x_4 + 1 \end{cases}$$

[3] In fact, by Hilbert's Basis Theorem, all ideals of $\mathbb{K}[x_1, x_2, \ldots, x_n]$ are finitely generated.

then by simplifying with $x_i = x_i^2, 1 \leq i \leq 4$, the Macaulay matrix of degree $D = 4$ extended from $\mathcal{F}$ with respect to graded reverse lexicographical order is

$$
\begin{array}{l|cccccccccccccccc|}
 & x_1x_2x_3x_4 & x_1x_2x_3 & x_1x_2x_4 & x_1x_3x_4 & x_2x_3x_4 & x_1x_2 & x_1x_3 & x_2x_3 & x_1x_4 & x_2x_4 & x_3x_4 & x_1 & x_2 & x_3 & x_4 & 1 \\
x_1x_2 \cdot f_1 & & & & & & 1 & & & & & & & & & & \\
x_1x_2 \cdot f_2 & & & 1 & & & 1 & & & & & & & & & & \\
x_1x_3 \cdot f_1 & & & & & & & 1 & & & & & & & & & \\
x_1x_3 \cdot f_2 & & & & 1 & & & 1 & & & & & & & & & \\
x_2x_3 \cdot f_1 & & 1 & & & & & & & & & & & & & & \\
x_2x_3 \cdot f_2 & 1 & & & & & & & 1 & & & & & & & & \\
x_1x_4 \cdot f_1 & 1 & & 1 & 1 & & & & & & & & & & & & \\
x_1x_4 \cdot f_2 & & & & & & & & & & & & & & & & \\
x_2x_4 \cdot f_1 & & & 1 & & & & & & & & & & & & & \\
x_2x_4 \cdot f_2 & & & 1 & & & & & & & 1 & & & & & & \\
x_3x_4 \cdot f_1 & 1 & & & & 1 & & & & & & 1 & & & & & \\
x_3x_4 \cdot f_2 & & & & 1 & & & & & & & 1 & & & & & \\
x_1 \cdot f_1 & & 1 & & & & 1 & 1 & & & & & & & & & \\
x_1 \cdot f_2 & & & & & & & & & 1 & & & 1 & & & & \\
x_2 \cdot f_1 & & & & & & 1 & & & & & & & & & & \\
x_2 \cdot f_2 & & & 1 & & & & & & & & & & 1 & & & \\
x_3 \cdot f_1 & & 1 & & & & & & 1 & & & & & & 1 & & \\
x_3 \cdot f_2 & & & & 1 & & & & & & & & & & 1 & & \\
x_4 \cdot f_1 & & & 1 & & 1 & & & & & & 1 & & & & & \\
x_4 \cdot f_2 & & & & & & & & & 1 & & & & & & 1 & \\
1 \cdot f_1 & & & & & & 1 & & 1 & & & & & & 1 & & \\
1 \cdot f_2 & & & & & & & & & 1 & & & & & & & 1
\end{array}
$$

where all missing coefficients are zero.

Clearly, the number of columns in a Macaulay matrix is $\sum_{i=0}^{D} \binom{n}{i}$, which is exactly the number of monomials of degree $\leq D$ while the number of rows is the product of the number of monomials of degree $\leq D - d$ and the number of equations in $\mathcal{F}$. As shown above, a Macaulay matrix extended from an $\mathcal{MQ}$ system $\mathcal{F}$ in n variables is highly sparse. Since the maximal number of monomials in a polynomial f in $\mathcal{F}$ is $\text{term\_num}_{max} = \binom{n}{2} + n + 1$, which does not increase when multiplying f by a monomial t, each row in a Macaulay matrix extended from $\mathcal{F}$ has at most term_num_{max} non-zero entries. In addition, if $\mathcal{F}$ is random and defined over $\mathbb{F}_2$ where half of the coefficients are expected to be zero, the expected maximal number of non-zero entries in a row becomes $\frac{\text{term\_num}_{max}}{2}$.

2.1.12 Gray Code

A Gray code is a binary enumeration system where two consecutive encodings differ only by one bit (see Table 2.1). Originally designed to prevent outburst of the number of electronic switches, which is the number of bits that flip when changing from one state to the next, Gray codes have been widely used for error correction in digital communication. Conversion between binary encoding and Gray code encoding is computationally efficient [6]. For example, converting a 32-bit binary integer b to its Gray code encoding g can be achieved by

$$g = b \oplus (b \gg 1)$$

which takes a right shift and an xor operation. On the other hand, converting g back to b can be done by

$$\begin{aligned} g_1 &= g \oplus (g \gg 2^4) \\ g_2 &= g_1 \oplus (g_1 \gg 2^3) \\ g_3 &= g_2 \oplus (g_2 \gg 2^2) \\ g_4 &= g_3 \oplus (g_3 \gg 2^1) \\ b &= g_4 \oplus (g_4 \gg 2^0) \end{aligned}$$

which takes five right shift and five xor operations.

Decimal Value	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Table 2.1: 4-bit Gray Code Encoding

2.2 Computer Architecture Preliminaries

This section first explains the architecture of CPUs and GPUs. Thereafter CUDA, the programming framework developed by Nvidia for GPUs, is briefly introduced.

2.2.1 CPU

A modern multi-core CPU resides on a IC chip connected to the motherboard via a physical connector called *socket*. A typical personal computer has only one socket while servers may have multiple ones. The chip consists of several execution units called *cores*. Each core has generally its own cache and *arithmetic logic units (ALUs)*, with which a core is able to execute instructions independently. A *thread* can be viewed as a series of instructions to be dispatched to a core. It might be advantageous to dispatch multiple threads to a physical core. For example, with the *hyper-threading* technology from Intel [7] or the *Cluster MultiThreading* [2] from AMD, a single physical core appears as two logical execution units. If one of the threads is stalled, the other one can take over the execution resources (e.g. ALUs and cache) so the core will not sit idle. In addition, if execution resources of a core are sufficient for two logical threads, they can be executed simultaneously.

Execution Model

Generally a CPU follows three steps to operate: fetch, decode, and execute. The instructions to be executed are stored in the main memory and therefore need to be loaded into the CPU first. The location of the next instruction to execute is stored in a special register called *program counter*. After the CPU retrieves the instruction, it increments the program counter and decodes the instruction into signals that control different ALUs. Finally the ALUs execute the desired operations, after which the entire procedure repeats. Usually fetch, decode, and execution of three instructions is done at a time to achieve a completion rate of up to one instruction per cycle per ALU. This technique is referred to as *instruction pipelining* and is generally applied to each core inside the CPU. Therefore, a modern CPU is able to execute more than one instructions per cycle, while the number of cycles per second is the inverse of the CPU frequency.

If one instruction requires the result of a previous instruction, which is called *data dependency*, then the pipeline might have to be delayed in order to wait for the result. In addition, the pipeline might need to start over if a jump instruction was executed, which renders instructions fetched and decoded in the current cycle useless. The first problem can be mitigated by issuing instructions that are independent from each other's results, which is referred to as *instruction-level parallelism*, while the second problem can be avoided by enforcing sequential execution or alleviated with branch prediction.

Memory Hierarchy

Since instructions and data are stored in the main memory, which can require hundreds of cycle to access, modern CPUs have multiple levels of caches to reduce memory access latency. Usually, each core has its own private instruction and data cache, known as *L1 instruction* and *L1 data cache*, respectively. For hyper-threading, those two types of cache memory are shared by two logical threads deployed on a physical core (see Figure 2.1). Typically there are two lower levels of cache memory known as *L2* and *L3*, which are generally shared by all the cores. L2 and L3 cache are much larger but slower than the top level L1 cache and are only accessed when there is a cache miss.

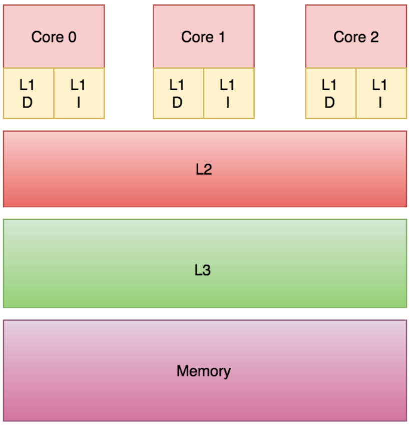


Figure 2.1: Memory Hierarchy on a CPU [3]

SIMD

Generally modern CPUs operate on 64-bit registers. Therefore, an operand can only be at most 64 bits. Some CPUs are however able to apply a single instruction on multiple data (SIMD) in order to exploit data level parallelism. SIMD CPUs pack multiple operands into a vector (known as *vectorization*) and store it into a special large register, whose size typically ranges from 128 bits to 512 bits. Special instructions can then be issued to operate on these large registers to apply the same operation on all operands at once. The number of operands packed into a vector is referred to as the length of the vector or the *SIMD width*.

2.2.2 GPU

A graphics processing unit (GPU) is a specialized circuit originally designed to accelerate graphics processing. GPUs are designed to be efficient at processing data in parallel, which is the nature of graphics processing. Over the years, usage of GPUs has been extended to applications other than graphics, e.g. molecular modeling and artificial intelligence where algorithms exhibit this parallel characteristic [4]. However, to fully utilize a GPU, a large degree of parallelism is necessary. A serial program generally will run faster on a CPU. Therefore, a GPU is never used alone but as an auxiliary computing device for a CPU. The main program is executed on the CPU and only parts of the program that exhibit parallelism should be delegated to the GPU.

A GPU consists of an array of *Streaming Multiprocessors (SM or SMX)*, each of which contains hundreds of *CUDA cores* (see Figure 2.2). Despite the name, a CUDA core is in fact similar to a CPU ALU, while an SM is analogous to a CPU core and can execute instructions independently. In addition, a CUDA core does not have dedicated cache. Instead, a unified L1 cache in each SM is shared by all CUDA cores in the SM (see Figure 2.3).

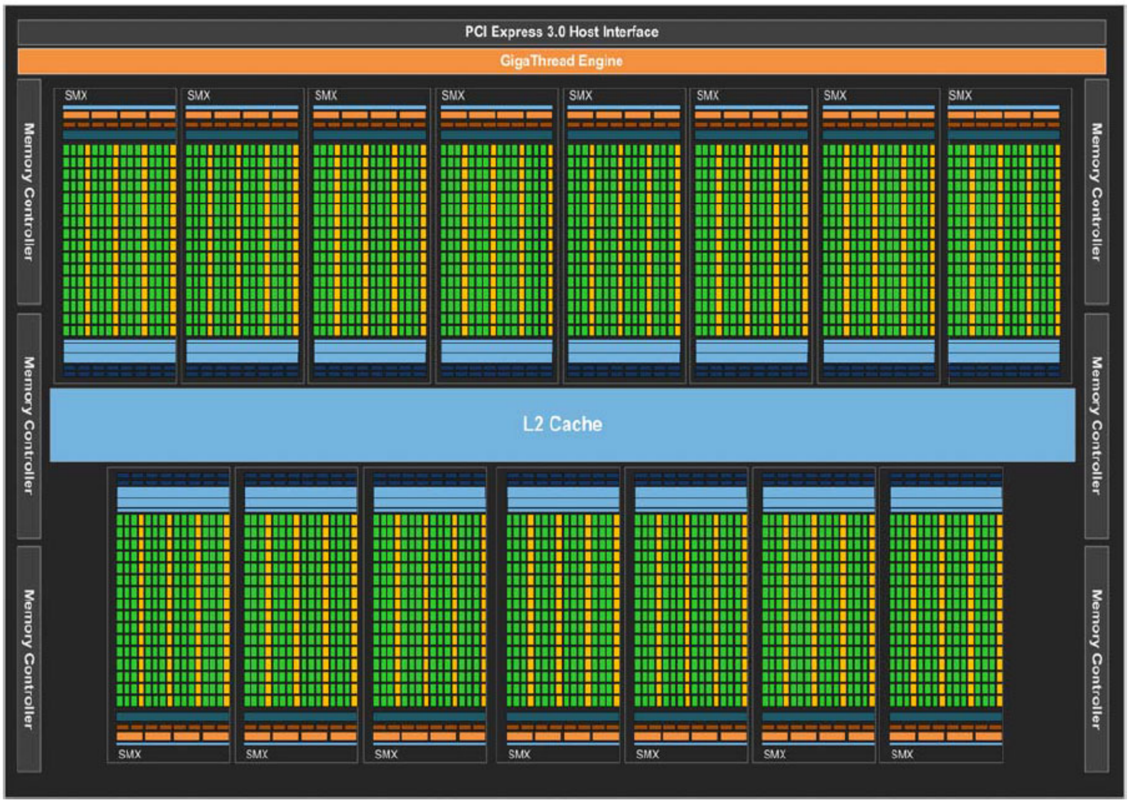


Figure 2.2: Architecture of a GPU [48]

Execution Model

Similar to CPUs, instructions are fetched, decoded, then finally executed and instruction pipelining is applied to hide the latency. By design, all instructions are executed on a GPU in a SIMD manner. One key difference between a SIMD CPU and a GPU is that the SIMD width (see Section 2.2.1) is much larger on a GPU. Currently all Nvidia GPUs operate on an array of 32 registers. Therefore, an add-instruction adds 32 numbers to another 32 numbers respectively, and writes the result back to a set of 32 registers.

The size of all registers on a GPU is 32 bits. Hence, to process data types larger than 32 bits, e.g. double precision, multiple registers are required. CUDA cores operate on 32-bit data so

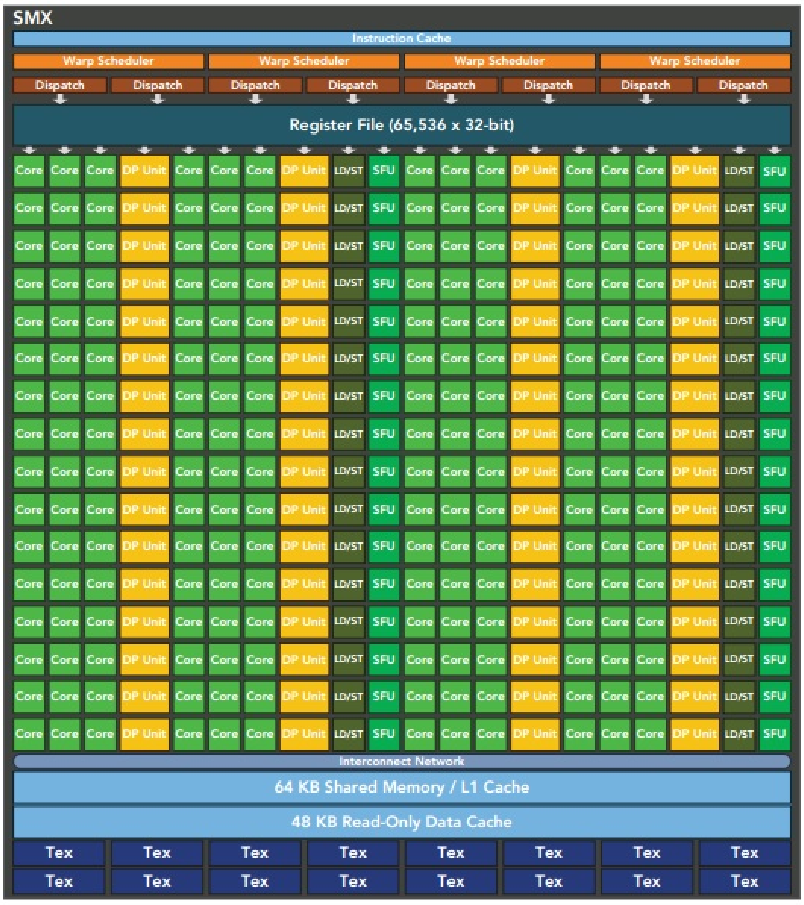


Figure 2.3: Architecture of a Streaming Multiprocessor [48]

instructions that operate on 64-bit data types have to be emulated by issuing multiple instructions. However, for some commonly used 64-bit operations in scientific computing, e.g. double-precision multiplication, GPUs provide direct hardware support. Therefore for these 64-bit operations, only one single instruction needs to be executed.

Similar to CPUs, an actual thread is a series of SIMD instructions and is dispatched to an SM which is able to execute these instructions independently. An actual thread is however exposed to programmers as a group of 32 logical threads. A GPU program is therefore written to operate on individual registers, instead of an array of them. To differentiate the two, an actual thread is called a *warp* while *thread* only refers to a logical thread. A warp consists of 32 threads since the SIMD width is 32 on a GPU. Several warps form a block, which is the basic unit for distributing workload to SMs. A GPU program is referred to as a *kernel*. To launch a kernel, its configuration including the number of warps per block and the number of blocks, must be specified beforehand. Once a block of warps acquires execution resources on an SM, it becomes *active*. Otherwise the block stays *resident*. Each SM keeps a pool of active blocks. The maximum number of active blocks depends on the amount of resources needed for a block and how many resources are available on an SM. Once all warps in a block have finished, a new resident block takes over its resources and becomes active. At each cycle, a warp scheduler in an SM selects some *eligible* warps ready for execution from the active pool and issues instructions for them. If a warp is stalled due to, e.g. memory access latency, the warp is marked and cannot be picked by the scheduler until the stall is resolved. To utilize a GPU to the fullest, it is a common and recommended practice to launch more blocks than the maximum number of active blocks for an SM so the hardware can be kept busy all the time.

SIMT

Another key difference between a SIMD CPU and a GPU is the hardware support of SIMD divergence. Different threads in a warp are allowed to take divergent execution paths, which is not possible for SIMD CPUs. A GPU achieves this by disabling some of the threads and executing instructions for one execution path. For example, if a branch is to occur, the GPU disables the threads that are to take the else-clause and executes instructions for the if-clause. Subsequently it disables the threads that took the if-clause and executes instructions for the else-clause. The SIMD divergence is handled automatically by the hardware and is completely transparent to programmers. Nvidia calls this architecture *Single Instruction Multiple Threads (SIMT)*, which can be viewed as a variant of SIMD. Even though with SIMT programmers do not need to handle SIMD divergence explicitly, it degrades performance and thus should still be avoided.

Memory Hierarchy

An integrated GPU that is often used for notebooks shares the system main memory with the CPU while a discrete GPU generally has its own dedicated *off-chip* memory. A discrete GPU can often achieve better memory throughput than a CPU because it uses wider system bus interface as well as newer types of memory, e.g. GDDR5, GDDR5X, GDDR6.

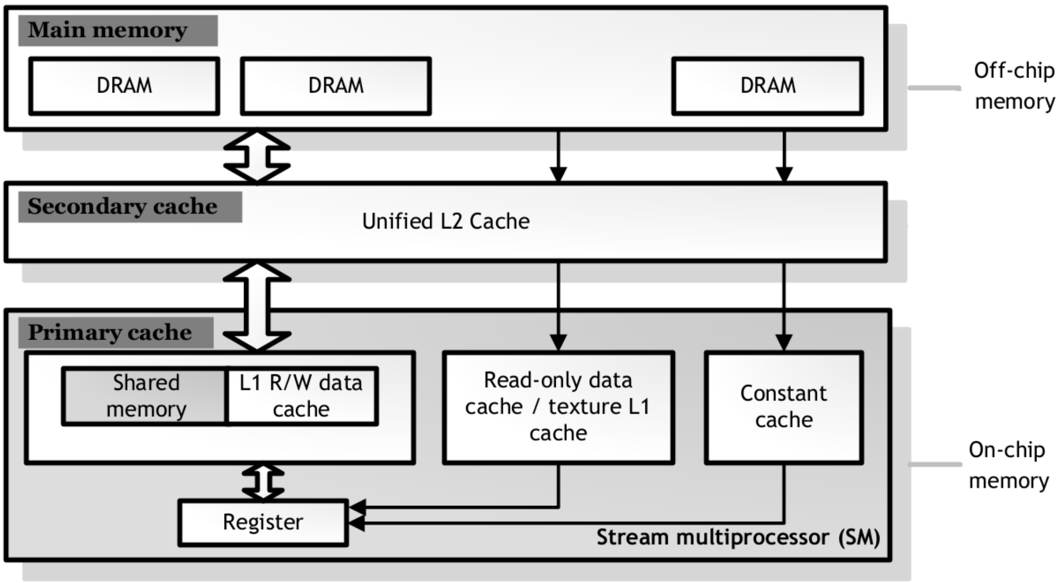


Figure 2.4: Memory Hierarchy on a GPU [45]

Besides the off-chip memory, an SM can access the *on-chip* memory which locates directly inside the SM. As Figure 2.4 shows, there are only two types of on-chip memory in terms of access latency: registers and cache. Accessing registers requires a few cycles but data stored in registers lasts only for the lifetime of the thread. Besides registers, an SM can access a pool of 64KB on-chip cache which is partitioned into *L1 cache* and *shared memory*. Data stored in L1 cache lasts until it is evicted and requires tens of cycles to access. On the other hand, shared memory serves as a means for communication and sharing data between threads in a block. Therefore data stored in shared memory lasts for the lifetime of the whole block. The size of the L1 cache and the shared memory can be configured before a kernel launch, ranging from (16KB, 48KB), (32KB, 32KB), to (48KB, 16KB) [50]. Note that L1 cache is disabled by default in some newer GPU architectures, e.g. Maxwell, but can still be activated.

In addition to cache memory inside an SM, a larger but slower unified L2 cache is provided and shared by all SMs (see Figure 2.2), which is only accessed when a cache miss occurs.

Because of the limited amount of cache memory, most of the data must be stored on the off-chip memory, which is split into global, local, constant and texture memory based on their purpose and properties.

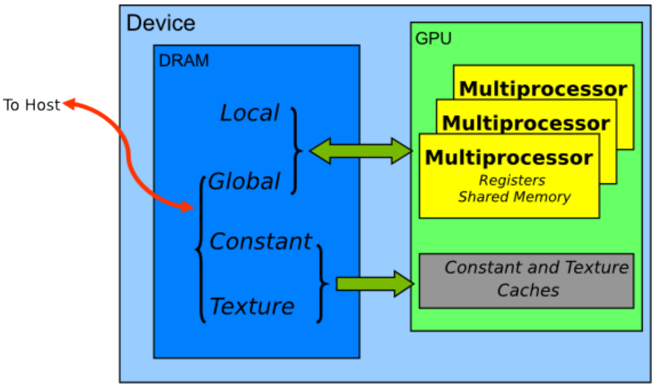


Figure 2.5: Different Types of Off-chip Memory on a GPU [50]

Global memory, which requires hundreds of cycles to access, stores most of the data for a kernel. When data storage is allocated for a kernel by either the CPU or the kernel itself, a block of the off-chip memory is carved out and becomes part of global memory.

As described in Section 2.2.2, the amount of execution resources allocated to a thread is subject to the amount of resources available in an SM. If a thread requires more registers than the amount that can be allocated, *register spill* occurs and temporary storage known as *local memory* is allocated for the thread. Despite the name, local memory is allocated from the off-chip memory as global memory and hence has the same access latency. Therefore, register spill should be avoided.

Constant memory is a special read-only region in the off-chip memory. The size of constant memory is restricted to 64KB but its content is cached inside an SM (see Figure 2.4). Only the first load-operation from a constant memory address incurs memory access to the off-chip memory. All subsequent accesses can read the data directly from the on-chip *constant cache*. However, even though the data is cached, if threads in a warp require data at different addresses then the requests must be serialized (see Section 2.2.2), which increases access latency linearly to the number of requests.

Similar to constant memory, *texture memory* is a region in the off-chip memory declared as read-only by programmers. When memory access pattern exhibits *2D spatial locality*, i.e. threads read data from addresses that are physically adjacent as shown in Figure 2.6, accessing data in texture memory requires less memory bandwidth than global memory. In addition, the content in texture memory is also cached in an SM. As shown in Figure 2.4, the on-chip *texture cache* is used to store data in texture memory. Texture cache is also used to store data in global memory that is marked as read-only either explicitly by programmers or implictly by the compiler. Therefore, texture cache is also referred to as *read-only data cache*.

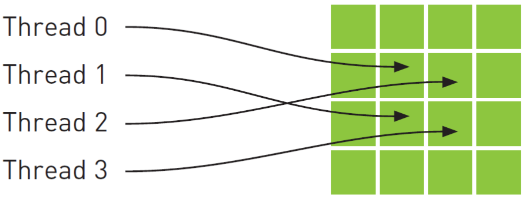


Figure 2.6: 2D Spacial Locality [9]

To sum up, global, texture, and constant memory, which are part of the off-chip memory, have the largest access latency. However, access to texture memory may require less memory bandwidth than global memory when 2D spatial locality is satisfied. In additon, the content of texture and constant memory is cached. Inside an SM, cache memory and registers are available and their access latencies are the lowest ones.

Note that the main memory on the computer can also be accessed by a discrete GPU via the system bus interface but the speed is extremely slow. Therefore accessing the system main memory should be avoided at all cost.

Coalesced Memory Access

One caveat of accessing data that is located in global memory is to avoid *non-coalesced memory access*. A memory access is referred to as non-coalesced if threads in a warp do not access consecutive addresses in global memory. For example, given an array of 1024 32-bit integers that is stored in row-major order on a GPU whose memory bus width is 128-bit, if threads in a warp access array elements at indices 0, 127, 255, 383 and so on, as shown in Figure 2.7, the access is non-coalesced and in total 32 global memory transactions are required since the elements are 128 bytes away from each other.

thread 1	→	0x000	0x004	...
thread 2	→	0x080	0x084	...
thread 3	→	0x100	0x104	...
⋮			⋮	
thread 31	→	0xF00	0xF04	...
thread 32	→	0xF80	0xF84	...

Figure 2.7: Non-coalesced Memory Access

On the other hand, the access becomes coalesced when threads in a warp access the first 32 array elements at once (see Figure 2.8). In this manner, the whole chunk of 128 bytes of data can be loaded with $\frac{128 \cdot 8}{128} = 8$ memory transactions, which use the available memory bus width optimally. Note that for older GPU architectures, consecutive threads in a warp have to access consecutive memory addresses as in Figure 2.8 but for newer generations of GPUs the order of the threads does not matter anymore. An access remains coalesced as long as the addresses are consecutive.

thread 1	thread 2	thread 3	thread 4	...	thread 31	thread 32
↓	↓	↓	↓		↓	↓
0x000	0x004	0x008	0x00C	...	0x078	0x07C
0x080	0x084	0x088	0x08C	...	0x0F8	0x0FC
				⋮		
0xF80	0xF84	0xF88	0xF8C	...	0xFF8	0xFFC

Figure 2.8: Coalesced Memory Access

Shared Memory Bank Conflict

Similar to global memory, one caveat of accessing data located in shared memory is to avoid *shared memory bank conflict*. Shared memory is only accessible to threads in a block and is divided into equally sized chunks which are mapped into memory modules called *banks* to allow concurrent access. Each bank can only serve one request at a time. When a thread accesses data at n addresses that are mapped into n memory banks, the data can be served at the same time, which yields an effective bandwidth that is n times higher than the bandwidth of a single bank. Nevertheless, if multiple threads in a warp access addresses mapped to the same bank, the accesses must be serialized, which increases the memory access latency linearly to the number of conflicting access requests. One exception is when these threads access *the same* address, in which case the data can be broadcast and therefore served to the threads simultaneously.

For GPU architectures since Kepler, there are 32 banks in total. Successive 32-bit words in shared memory are mapped to successive memory banks whose size is also 32-bit. Consequently, if threads in a warp access consecutively a char array, where the size of each element is one byte, the first four elements are mapped to the same bank therefore the accesses of the first four threads must be serialized. An approach to avoiding shared memory bank conflict is to interleave the array elements in shared memory and change the access pattern of the threads accordingly. Note that the size of the bank can be configured to 64-bit, which can help avoid shared memory bank conflict when accessing 8-byte data types, e.g. double precision floating point values.

2.2.3 CUDA

CUDA[4] is a programming framework and runtime environment created by Nvidia for general purpose computing on GPUs. CUDA exposes the GPU execution model and memory hierarchy, e.g. threads, blocks, shared and constant memory, as a set of language extensions. For the C programming language, programmers use the C extension [49], which is referred to as *CUDA C*, to write source code for GPU kernels. Nvidia provides a front-end compiler *NVCC* which isolates, preprocesses the language extensions, subsequently generates binary code for GPU kernels, and finally forwards the rest of the C code to a supported back-end compiler. Currently only GCC, Clang, Intel C++ compiler and Microsoft Visual C compiler are supported [51].

In addition to the C language extensions, CUDA also provides a set of virtual machine instructions, known as *PTX*, which is independent from GPU architectures. In fact, NVCC first translates the C source code for a GPU kernel from CUDA C to PTX instructions, which are subsequently compiled into native instructions for the target GPU architecture.

[4] Originally an acronym for Compute Unified Device Architecture.

Chapter 3

Solving Multivariate Quadratic Systems

The number of solutions for a random $\mathcal{MQ}$ system depends on the number of variables n and the number of equations m. If $m < n$, the system is referred to as underdetermined and on average has $\mathcal{O}(2^{n-m})$ solutions [35]. If $m = n$, the system is determined and has at least one solution for large n with probability close to $1 - \frac{1}{e} \approx 0.63$ [35]. Otherwise the system is overdetermined and has either one or with overwhelming probability no solution.

It has been shown that extremely underdetermined ($n > m(m + 1)$) and overdetermined ($m > \frac{n(n+1)}{2}$) systems can be solved in polynomial time with the Kipnis-Patarin-Goubin algorithm and linearization, respectively [43, 42, 56]. Hence this thesis does not take systems beyond those extreme points into consideration.

The common definition of solving an $\mathcal{MQ}$ system requires all the solutions to be enumerated. This is possible when the system has a finite number of solutions, i.e. the variety defined by the equations is zero-dimensional. If the solution set is too large or is infinite, which is often the case with underdetermined systems, an alternative approach is to establish a description of the solution set. In many cryptographic applications, however, it is sufficient to find one solution to the system. For example, in the Unbalanced Oil and Vinegar (UOV) signature scheme [42], being able to obtain a solution to the system allows one to forge a valid signature [33]. Therefore this thesis aims to test the existence of solutions to a random $\mathcal{MQ}$ system and to provide one if the system is solvable.

With this objective, this thesis focuses only on $\mathcal{MQ}$ systems where $m \geq n$. In order to find a solution to an underdetermined system, one can simplify the system with arbitrary values for $n - m$ variables (i.e. fixing $n - m$ variables) to yield a determined system, which has at least one solution with probability roughly 0.63. Therefore in terms of testing the existence of solutions, an underdetermined system is no more difficult than the determined systems obtained by fixing variables.

Finally, a common practice to simplify a system in n variables $x_1, x_2, \ldots, x_n$ over a finite field $\mathbb{F}_q$ is to take advantage of the cyclic nature of multiplication. By applying the equations $x_i = x_i^q$, $1 \leq i \leq n$, all the individual variables that appeared in the system must have a degree smaller than q. This technique can effectively reduce the number of columns in a Macaulay matrix (see Section 2.1.11) so it is adopted by this thesis as well.

3.1 State of the Art

3.1.1 Gröbner Basis

Given a monomial order and an ideal $\mathbb{I}$ in a polynomial ring $\mathbb{K}[x_1, x_2, \ldots, x_n]$, a finite non-empty generating subset $\mathbb{G} = \{g_1, g_2, \ldots, g_t\}$ of the ideal $\mathbb{I}$ is referred to as a *Gröbner basis* if the ideal of leading terms of $\mathbb{I}$ can be generated by the leading terms of elements in $\mathbb{G}$, i.e.

$$\langle LT(\mathbb{I})\rangle = \langle LT(g_1), LT(g_2), \ldots, LT(g_t)\rangle.$$

Recall that in Section 2.1.6 it is shown that the remainder of multivariate division depends on the order of the divisors. One important property of a Gröbner basis is that applying multivariate division on a polynomial by a Gröbner basis yields a *unique* remainder. Therefore one can test whether or not a polynomial f belongs to $\mathbb{I}$ by checking if the remainder of f on division by $\mathbb{G}$ is zero. Note that the quotients however still depend on the order of the divisors in the tuple.

By Hilbert's Basis Theorem, every ideal has a Gröbner basis. To compute it, one can apply Buchberger's algorithm [21], whose basic idea is to extend the original generating set by inducing cancellation of leadings terms on existing generators and subsequently dropping generators that became redundant during the process. Since computing a Gröbner basis involves canceling leading terms which simplifies the polynomials and successively eliminates variables according to a monomial order, it can be applied to solve $\mathcal{MQ}$ systems. For example, in order to test the existence of at least one solution to the $\mathcal{MQ}$ system over $\mathbb{C}[x, y, z]$

$$\begin{cases} f_1 = x^2 + y^2 + x^2 - 1 \\ f_2 = x^2 + z^2 - y \\ f_3 = x - z \end{cases}$$

which defines the ideal $\mathbb{I} = \langle f_1, f_2, f_3\rangle$, one can compute a Gröbner basis

$$\begin{cases} g_1 = x - z \\ g_2 = y - 2z^2 \\ g_3 = z^4 + \frac{z^2}{2} - \frac{1}{4} \end{cases}$$

with respect to lexicographical order $x > y > z$, which provides a description of the solution set (i.e. the variety defined by the ideal $\mathbb{I}$). In particular, the set is finite if and only if for each variable there exists at least one generator g in the Gröbner basis whose leading term $\mathrm{LT}(g)$ contains only that variable. If that is the case (as this example), then by solving equations involving only one variable (g_3) and backward substitution, a solution can be obtained.

Clearly, with this approach solving an $\mathcal{MQ}$ system is no more difficult than computing a Gröbner basis for the system, whose complexity is related to the number of leading terms to eliminate. It is therefore desirable to estimate the maximal degree of polynomials in a Gröbner basis, which is referred to as the *degree of regularity* and is analyzed in [11].

Some variants of Buchberger's algorithm [30, 31] have been proposed, which achieved better performance by taking advantage of sparse linear algebra. It has been shown that these algorithms are suitable for overdetermined systems or systems with hidden algebraic structure, but ineffective for general random systems [11, 13]. In addition, all those variants as well as the original algorithm require exponential storage. Therefore, solving a systems of 40 variables and equations is still out of reach for most modern computers [38].

3.1.2 XL

The XL algorithm, which stands for *eXtended Linearization*, and its variants are a family of algorithms [57] related to Gröbner basis based algorithms [10]. Given a degree D and an $\mathcal{MQ}$ system $\mathcal{F}$ of m polynomials in n variables over a field $\mathbb{K}$, the algorithm first extends the system to degree D by multiplying the polynomials with all monomials whose degree is no larger than $D - 2$

$$\mathcal{F}' = \{l_j \cdot f_i, \text{ where } f_i \in \mathcal{F} \text{ and } l_j \in \mathbb{K}[x_1, x_2, \ldots, x_n], \deg(l_j) \leq D - 2\}$$

then it solves the extended system as if it were a linear system by treating each monomial in $\mathcal{F}'$ as a new variable. With a monomial order that considers the original variables as the smallest ones, the extended system may yield solutions to some original variables or non-linear equations in one single original variable, in which case Berlekamp's algorithm [15] can be applied to compute the solution. If so, the original $\mathcal{MQ}$ system is simplified with the newly obtained solutions. Thereafter the algorithm repeats this procedure until all variables have been solved. Clearly, termination of the algorithm depends on the number of independent equations in the extended system. As shown in [57], it depends on the parameter D and the minimal degree D for XL to terminate can be computed based on the number of variables n and equations m. Note that instead of Gaussian elimination, sparse linear algebra, e.g. block Lanczos algorithm [46] or block Wiedemann algorithm [23] can be used to solve the extended system since it is inherently sparse.

XL cannot operate with underdetermined systems. In addition, it has been observed that XL requires $m - n \geq 2$ for reasonable performance [26]. Therefore a simple adaption is to fix some variables in the original $\mathcal{MQ}$ system. The adapted XL algorithm is referred to as *FXL* [26] and introduced in Section 3.1.4. In fact, fixing variables to simplify the original system is considered a good approach in general to reduce the degree of regularity before applying the main algorithm [57].

3.1.3 Fast Exhaustive Search

If an $\mathcal{MQ}$ system of m equations in n variables is defined over a rather small finite field, e.g. $\mathbb{F}_2$, testing all possible solutions is a viable method to solve the system. Clearly, when testing a solution there is no need to check with the rest of the equations once an equation does not evaluate to zero. The expected number of equations to evaluate for each possible solution is only $\sum_{i=0}^{m-1} \frac{1}{2^i} \approx 2$. Therefore, a full naïve exhaustive search for an $\mathcal{MQ}$ system of 64 variables and equations over $\mathbb{F}_2$ is expected to evaluate degree-2 equations for $2 \cdot \frac{1}{2} \cdot 2^{64}$ times, since on average only half of the search space $\mathbb{F}_2^n$ needs to be explored.

Nevertheless, even with early aborts such complexity is still far from being practical on most modern computers [20]. A more sophisticated exhaustive search algorithm based on partial derivatives and Gray code enumeration (see Section 2.1.12) has been proposed [20] and implemented [8], which is able to iterate over all solutions in the search space $\mathbb{F}_2^n$ with $\mathcal{O}(\log_2 n \cdot 2^{n+2})$ elementary bit operations. When testing for the existence of a solution, the algorithm can terminate once a solution is found therefore even fewer bit operations are required.

The algorithm relies on the observation that the value of a function f at a point $\vec{a}$ can be computed from the value of f and its partial derivatives $\frac{\partial f}{\partial x_i}$ at another point $\vec{a'}$

$$f(\vec{a}) = f(\vec{a'}) + \frac{\partial f}{\partial x_i}(\vec{a'})$$

when only the i^{th} coordinates (i.e. the value for x_i) of $\vec{a}$ and $\vec{a'}$ differ. This technique can be applied recursively on the first order partial derivatives. When applying to an $\mathcal{MQ}$ system, the first order partial derivatives become linear and the second order partial derivatives $\frac{\partial^2 f}{\partial x_i \partial x_j}$ reduce to constants, in which case the recursion reaches the base case.

Gray code enumeration iterates over the search space $\mathbb{F}_2^n$ in a manner where two consecutive points differ only by one coordinate. Therefore, it can be combined with the observation above to efficiently evaluate the functions in the system over the search space. Note that the algorithm does not need to evaluate all equations in the system at once. Instead only a subset of the equations are checked with Gray code enumeration while the rest of the equations are only evaluated using a naïve approach when a solution candidate is found. With this approach the complexity of the algorithm does not rely on the number of equations in the system. The algorithm may however terminate earlier for system with fewer equations since the solution set is expected to be larger. In [20] the implementation chooses to evaluate a subset of 32 equations with Gray code enumeration and leaves the rest of the system to naïve evaluation in order to fully utilize the 32-bit registers provided by the hardware.

This algorithm trades memory space for computation time. For an $\mathcal{MQ}$ system $\mathcal{F}$ in n variables over $\mathbb{F}_2$, a single function in $\mathcal{F}$ requires $n + 1$ bits to store the evaluation of the function and its partial derivatives, and $\binom{n}{2}$ bits for the last order partial derivative where the function reduces to constants. If the amount of memory is limited one may stop the recursion earlier at the first order partial derivative and switch to naïve enumeration. Note that because of the cyclic nature of Gray code enumeration, the evaluation of a partial derivative $\frac{\partial f}{\partial x_i \partial x_j}$, where $j > i$, is accessed with period 2^j [20]. Therefore the algorithm can be further optimized by deliberately keeping frequently accessed evaluation of partial derivatives in registers or cache, which will never be evicted under optimal cache replacement policy [20].

3.1.4 FXL and BooleanSolve

As shown in Section 3.1.2, to solve an $\mathcal{MQ}$ system with an algebraic approach, instead of computing a Gröbner basis directly as described in Section 3.1.1, one can extend the system by multiplying equations therein with monomials. The extended system can be represented by a Macaulay matrix (see Section 2.1.11). It has been shown that for an $\mathcal{MQ}$ system $\mathcal{F}$ in n variables which defines an ideal $\mathbb{I}$, the reduced row echelon form of the Macaulay matrix extended from $\mathcal{F}$ contains the coefficients of a Gröbner basis of $\mathbb{I}$ [44]. However, to obtain a whole Gröbner basis, a Macaulay matrix with degree no less than the degree of regularity D_{reg} of the system is required. Assuming a sufficient amount of memory is available, computing the reduced row echelon form of the Macaulay matrix then requires $p(n) \cdot \binom{n}{D_{\text{reg}}-1}^2$ bit operations, where $p(n)$ is a polynomial in n [20]. Such complexity implies that with the hardware available nowadays, a pure algebraic approach will not outperform exhaustive search when $n < 200$ [20].

FXL [58] and BooleanSolve [12] are algorithms that combine the algebraic approach with exhaustive search whose basic idea is to prune branches of the search tree with Macaulay matrices. Given an $\mathcal{MQ}$ system $\mathcal{F} = \{f_1, f_2, \dots, f_m\}$ in n variables $x_1, x_2, \dots, x_n$, both algorithms only keeps k variables $x_1, x_2, \dots x_k$ by first fixing $n-k$ variables in order to reduce the degree of regularity to D_{reg} as mentioned in Section 3.1.2.

After fixing variables, FXL continues with the same steps as the XL algorithm introduced in Section 3.1.2. On the other hand, BooleanSolve computes the Macaulay matrix of the witness degree D_{wit}, which is an upperbound of D_{reg} and can be calculated based on n, m, k [12], based on the base system $\mathcal{F}'$

$$\mathcal{F}' = \{\tilde{f}_1 \tilde{f}_2, \dots, \tilde{f}_m, x_1(x_1-1), x_2(x_2-1), \dots, x_k(x_k-1)\}$$

where $\tilde{f}_i, 1 \leq i \leq m$ is the polynomial obtained by fixing those $n-k$ variables in f_i while the rest of the polynomials are deliberately introduced for the next step. After obtaining the Macaulay matrix, BooleanSolve proceeds to test the existence of the polynomials $h_1, h_2, \dots, h_{m+1}, \dots, h_{m+k}$ that satisfy the equation

$$h_1\tilde{f}_1 + h_2\tilde{f}_2 + \cdots + h_m\tilde{f}_m + h_{m+1}x_1(x_1-1) + \cdots + h_{m+k}x_k(x_k-1) = 1$$

with a Las Vegas variant of Wiedemann's algorithm [37]. Clearly, since the deliberately introduced polynomials $x_i(x_i-1)$, $1 \leq i \leq k$ always evaluate to zero, the existence of $h_i, 1 \leq i \leq m+k$ implies the solution set for the original $\mathcal{MQ}$ system $\mathcal{F}$ is empty. If this is the case, BooleanSolve chooses another set of values for $n-k$ variables and repeats from the beginning, otherwise exhaustive search is applied to compute the solution for the remaining k variables after which the algorithm terminates.

Both FXL and BooleanSolve achieve better asymptotic performance ($\mathcal{O}(2^{0.785n})$ and $\mathcal{O}(2^{0.792n})$, respectively) than Fast Exhaustive Search (see Section 3.1.3) under the assumption that $n = m$ and the systems obtained from fixing variables behave like random systems (i.e. remain semi-regular [11]). By fixing $n-k$ variables before computing a Macaulay matrix, both the degree of regularity and the number of variables decrease which in turn reduces the memory requirement. Nevertheless, working with Macaulay matrices is still the most time-consuming and memory intensive step for both algorithms, which on average has to be done $\frac{1}{2} \cdot 2^{n-k}$ times. Therefore, it is the main focus of optimization for the algorithm introduced in Section 3.1.5.

3.1.5 Crossbred

As mentioned in Section 3.1.4, the most computation intensive step of the BooleanSolve algorithm is testing the solvability of Macaulay matrices with a variant of Wiedemann's algorithm. To avoid this costly step, the Crossbred algorithm has been proposed [40], whose basic idea is to fix variables *after* computing Macaulay matrices. Given an $\mathcal{MQ}$ system $\mathcal{F}$ of m equation in n variables, a degree-D Macaulay matrix with respect to a monomial order can first be computed based on $\mathcal{F}$. Thereafter the Macaulay matrix is reduced to its row echelon form, whose last rows might represent equations where $n - k$ variables have been eliminated. These equations can then be extracted as a sub-system in k variables, which can be solved with exhaustive search or with Gaussian elimination if the sub-system happens to be linear. Nevertheless, as mentioned in Section 3.1.4, to obtain a linear sub-system in n variables, the degree D of the Macaulay matrix must be no smaller than the degree of regularity D_{reg} of the $\mathcal{MQ}$ system $\mathcal{F}$. On the other hand, to eliminate a significant number of variables in the sub-system, D might grow too large so working with the resulting Macaulay matrix becomes impractical as well [40].

The key observation of the Crossbred algorithm is that it is not necessary to completely eliminate $n - k$ variables in the extracted sub-system. Instead, one simply needs to eliminate the monomials that do not become constant or linear in the remaining k variables after fixing these $n - k$ variables, which is referred to as being *non-linear* for these k variables (non-linear$_k$). For example, by fixing the last two variables x_3 and x_4, the sub-system

$$\mathcal{S} = \begin{cases} x_1x_4 + x_2x_3 + x_1 + x_3 + x_4 = 0 \\ x_1x_3 + x_3x_4 + x_2 + 1 = 0 \\ x_2x_3 + x_2x_4 + x_3x_4 + x_1 + x_4 = 0 \end{cases}$$

becomes a linear system in x_1 and x_2 since $\mathcal{S}$ consists of only monomials that *become linear* after fixing variables (to-be-linear$_k$). Clearly, the resulting linear system can be directly solved with Gaussian elimination, with which solutions to the system can be derived efficiently.

For a monomial $x^\alpha = x_1^{\alpha_1}x_2^{\alpha_2}\dots x_k^{\alpha_k}x_{k+1}^{\alpha_{k+1}}\dots x_n^{\alpha_n}$, the total degree of the first k variables is denoted as $\deg_k x^\alpha = \sum_{i=1}^{k}\alpha_i$. The Crossbred algorithm first computes a degree-D Macaulay matrix with respect to a monomial order $>_{\deg_k}$where monomials are sorted according to $\deg_k$ in descending order. Subsequently the algorithm extracts at least k equations where the monomials of $\deg_k$ larger than one (non-linear$_k$) are eliminated and only keeps monomials of $\deg_k \leq 1$ (to-be-linear$_k$) to construct a sub-system that can be transformed into a linear system in the first k variables by fixing the remaining $n - k$ variables. After one such sub-system $\mathcal{S}$ is obtained, Crossbred performs exhaustive search by fixing the last $n-k$ variables and testing whether or not the resulting linear system $\mathcal{S}'$ is solvable. If so, solutions to $\mathcal{S}'$ are checked with the original $\mathcal{MQ}$ system $\mathcal{F}$. The algorithm then terminates if a solution is found, otherwise it fixes $n - k$ variables in $\mathcal{S}$ with another set of values and continues the exhaustive search procedure.

To obtain a linear system $\mathcal{S}'$ from the extracted sub-system $\mathcal{S}$, the Crossbred algorithm uses the FastEvaluate recursive algorithm (see Algorithm 2) to fix $n - k$ variables in $\mathcal{S}$, whose basic idea is to split each polynomial into two groups of monomials. An arbitrary polynomial p can be written as $p = p_0 + x_ip_1$, where x_ip_1 are monomials that involve a specific variable x_i while p_0 are monomials that do not. It is clear from this form that p_0 is exactly the result of fixing $x_i = 0$ in p while $p_0 + p_1$ is the result of fixing $x_i = 1$ in p. This idea can be applied recursively to fix $n-k$ variables. FastEvaluate however requires exponential storage for parallelization. In addition, the recursive tree is very unlikely to be balanced as there are at most $\sum_{i=1}^{D}\binom{n-1}{i-1}$ monomials that involve x_i while there are $\sum_{i=0}^{D}\binom{n}{i}$ monomials in total, therefore the maximal storage requirement of a tree node decreases only polynomially. Finally, the recursive nature of FastEvaluate makes it difficult to apply the algorithm in parallel, hence it is not suitable for the GPU architecture.

Note that one can further fix some variables in the original $\mathcal{MQ}$ system before computing Macaulay matrices, which is referred to as *external hybridation* by the authors [40]. Nevertheless, since the coined terminology might contain a typo as hybridation is not an English word, this thesis refers to the technique as *$\mathcal{MQ}$ preprocessing*. Also note that the authors of the Crossbred algorithm consider $\mathcal{MQ}$ preprocessing merely as a method to distribute the workload between computers and do not expect it to be asymptotically useful [40].

Algorithm 1 The Original Crossbred Algorithm

```
procedure CROSSBRED
    Input:
        an $\mathcal{MQ}$ system of $m$ equations in $n$ variables $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$
        Macaulay degree: $D$
        number of variables to keep: $k$
        number of variables to fix during $\mathcal{MQ}$ preprocessing: $p$

    for each $(x_{n-p+1}, \ldots, x_n)$ in $\{0,1\}^p$ do
        1. Fix the last $p$ variables in $\mathcal{F}$ according to the tuple to obtain an $\mathcal{MQ}$ system $\mathcal{F}'$.
        2. Compute the degree-$D$ Macaulay matrix $\mathrm{Mac}_D^k$ with respect to a monomial order
            where monomials are sorted by $\deg_k$ based on $\mathcal{F}'$.
        3. Extract $r$ linearly independent equations $\mathcal{S} = \{s_1, s_2, \ldots, s_r\}$ from $\mathrm{Mac}_D^k$ where
            monomials of $\deg_k > 1$ have been eliminated.

        Call FastEvaluate($\mathcal{S}, k, n-p$) and
        for each output linear system $\mathcal{S}'$ do
            4. Test if $\mathcal{S}'$ is solvable. If so, extract solutions from $\mathcal{S}'$ and verify them with $\mathcal{F}$.
            5. Continue if no solution is found. Otherwise output the solution and terminate.
        end for
    end for
end procedure
```

Algorithm 2 Fast Evaluation of Polynomials over $\mathbb{F}_2$

```
procedure FASTEVALUATE
    Input:
        a system $\mathcal{S} = \{s_1, s_2, \ldots, s_r\}$
        number of variables to keep: $k$
        number of variables left in the system $\mathcal{S}$: $l$

    if $l = k$ then
        Output $\mathcal{S}$
    else
        Separate each $s_i$, $1 \leq i \leq r$ into monomials that involve $x_l$ and monomials
        that do not: $s_i = s_{i0} + x_l s_{i1}$

        $\mathcal{S}_0 \leftarrow \{s_{i0} \mid 1 \leq i \leq r\}$
        $\mathcal{S}_1 \leftarrow \{s_{i0} + s_{i1} \mid 1 \leq i \leq r\}$
        Call FastEvaluate($\mathcal{S}_0, k, l-1$)
        Call FastEvaluate($\mathcal{S}_1, k, l-1$)
    end if
end procedure
```

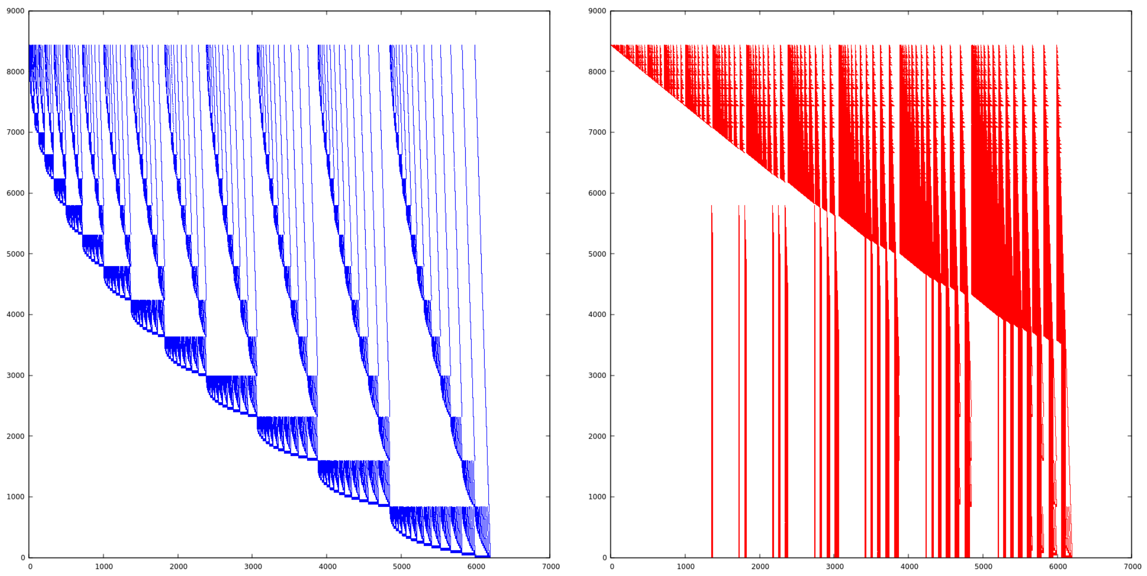
3.2 Parallel Crossbred

Although the main idea of the Crossbred algorithm was explained in [40], some critical details were left out. In particular, step 3 and 4 of the Crossbred algorithm (see Algorithm 1) were not specified. In addition, as pointed out in Section 3.1.5, the FastEvaluate algorithm for fixing variables in the sub-system is not suitable for parallelization. Therefore this thesis proposes the *Parallel Crossbred* algorithm, which is an adaption of the Crossbred algorithm that is highly parallelizable, and provides the missing critical steps in the original algorithm.

3.2.1 Extracting a Sub-system from the Macaulay Matrix

To extract r linearly independent equations where monomials of $\deg_k > 1$ are eliminated from the Macaulay matrix, intuitively one can perform Gaussian elimination on the Macaulay matrix to extract all equations that meet the criteria at once and choose r from them. Since computing Macaulay matrices with respect to graded reverse lexicographical order $>_{\text{grlex}}$ can be done more efficiently than computing with respect to the monomial order $>_{\deg_k}$ (defined in Section 3.1.5), one might be tempted to compute the Macaulay matrix with respect to $>_{\text{grlex}}$ first then perform Gaussian elimination with the columns of the matrix permuted. Note that it is not necessary to actually permute the columns according to $>_{\deg_k}$ as one can compute the indices of the columns that must be eliminated and apply Gaussian elimination on these columns directly.

For example, the degree-4 Macaulay matrix of an $\mathcal{MQ}$ system of 40 equations in 20 variables with respect to $>_{\text{grlex}}$ can be reduced into its permuted row echelon form as shown in Figure 3.1(a) and Figure 3.1(b), respectively. The dimension of the original Macaulay matrix is 8440 × 6196. To keep 12 variables, 4917 out of 6196 monomials are categorized as non-linear$_k$ and therefore need to be eliminated. Since the remaining 1279 to-be-linear$_k$ monomials that can be kept are scattered along the x-axis, one cannot optimize the algorithm by ignoring monomials before a pivot monomial. In particular, when performing the row reduction step in Gaussian elimination, one has to specifically perform reduction on the to-be-linear$_k$ monomials before the pivot monomial as well since they might not be zero. This non-consecutive memory access pattern makes naïve Gaussian elimination on the original Macaulay matrix cache-unfriendly and consequently inefficient.

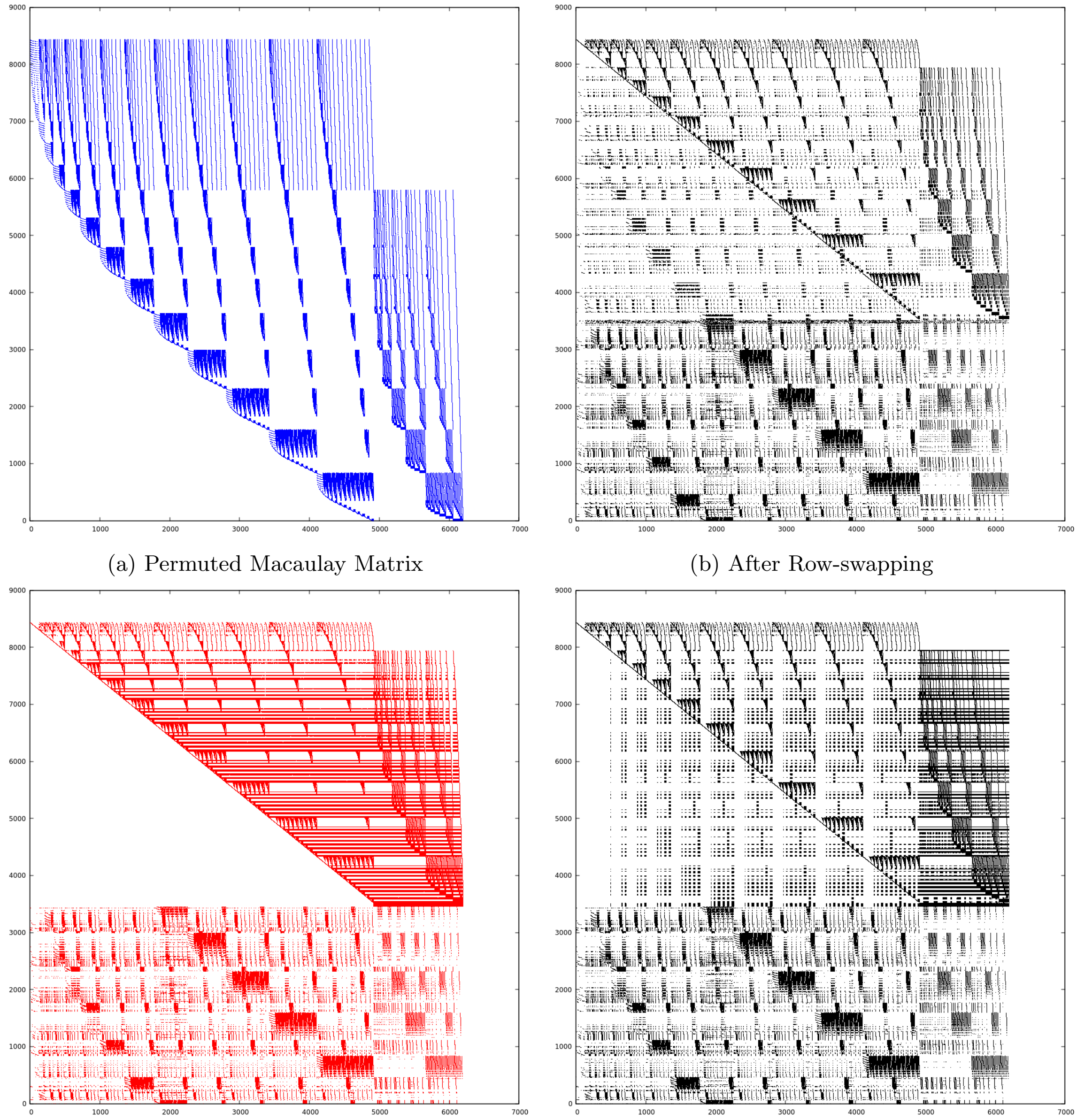


(a) Original Macaulay Matrix

(b) After Gaussian Elimination

Figure 3.1: Gaussian Elimination on the Original Macaulay Matrix

To avoid non-consecutive memory access, one can directly compute the Macaulay matrix with respect to $>_{\deg_k}$ instead of $>_{\text{grlex}}$ as shown in Figure 3.2(a), which can also be achieved by permuting the columns in the original Macaulay matrix with respect to $>_{\deg_k}$. However for large matrices the second approach might incur considerable extra computation cost due to memory access latency and therefore is not recommended.



(a) Permuted Macaulay Matrix

(b) After Row-swapping

(c) After Gaussian Elimination

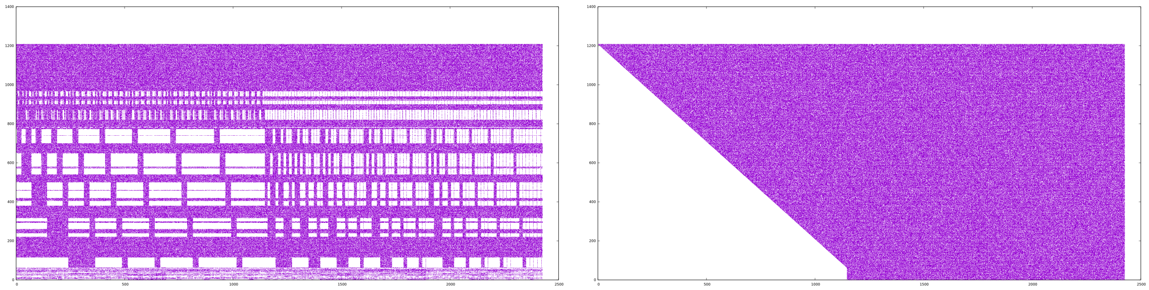
(d) After Eliminating Pivot Monomials

Figure 3.2: Gaussian Elimination on the Permuted and Row-swapped Macaulay Matrix

Nevertheless, even with the permuted Macaulay matrix, applying Gaussian elimination directly is still far from being optimal since a Macaulay matrix exhibits a special structure that can be taken advantage of. As shown in Figure 3.2(a), the leading monomial of each row in a Macaulay matrix can be computed based on the polynomial f and the multiplier t for that row (see Section 2.1.11), which determines the point where all monomials ahead must be zero. A Macaulay matrix therefore has a skew diagonal line below which coefficients are guaranteed to be zero. Hence, one can perform *row-swapping* on the permuted Macaulay matrix before applying Gaussian elimination, which searches for rows that have only zero coefficients before a desired pivot monomial and swap them into their final position after Gaussian elimination. As shown in Figure 3.2(b), by exploiting the special structure of a Macaulay matrix, most of the pivot monomials along the diagonal line can be found. Therefore, the majority of the rows will be in their final position even before Gaussian elimination begins. Consequently, during Gaussian elimination there is no need to search for the pivot rows for pivot monomials that are present nor perform row reduction on the rows that are in their final position, both of which reduce memory access considerably.

Another optimization technique is to ignore the lower part of the Macaulay matrix during Gaussian elimination. Since it is sufficient to extract r linearly independent equations from the Macaulay matrix, there is no need to perform Gaussian elimination on the whole Macaulay matrix. After row-swapping, one can further swap $a \cdot r$ ($a \geq 1$) random rows below the last pivot row row_{last} to the $a \cdot r$ consecutive rows immediately below row_{last}. For example, in the aforementioned example there are 4917 non-linear_k monomials to eliminate therefore after Gaussian elimination the first 4917 rows will contain at least one pivot monomial which is non-linear_k if the matrix is not singular. To extract 32 independent equations, one can randomly choose 64 ($a = 2$) rows below the 4917^{th} row as the *sub-system candidates* and swap them into the $4918^{\text{th}} \sim 4981^{\text{st}}$ rows. Clearly, the parameter a can be adjusted based on the probability of obtaining linearly dependent equations. Thereafter Gaussian elimination can be applied simply on those 4917 pivot rows and 64 sub-system candidates, as shown in Figure 3.2(c).

In addition to row-swapping and ignoring the lower part of the Macaulay matrix, more optimization techniques can be applied. In particular, since the majority of the pivot monomials along the diagonal line will be present after row-swapping, one can reduce the dimension of the matrix to process for Gaussian elimination by first applying Gauss-Jordan elimination based on the available pivot monomials. In other words, for each pivot monomial that is found during row-swapping, row reduction is applied to rows above and below the corresponding pivot row to eliminate that specific pivot monomial from the matrix (see Figure 3.2(d)). Thereafter, a reduced Macaulay matrix where each row consists of only pivot monomials that are still missing and to-be-linear_k monomials can be extracted, as shown in Figure 3.3(a).



(a) Initial Reduced Macaulay Matrix

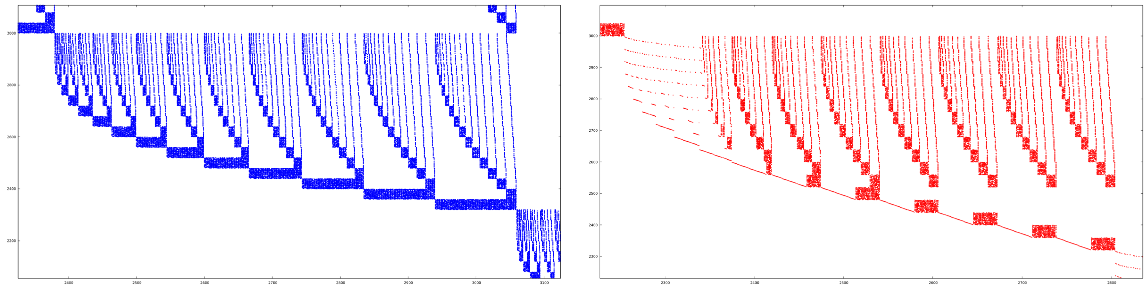
(b) After Gaussian Elimination

Figure 3.3: Gaussian Elimination on the Reduced Macaulay Matrix

As a rule of thumb, in order to increase the probability of finding rows with the desired property during row-swapping, Gauss-Jordan elimination can be applied to the $\mathcal{MQ}$ system based on which the Macaulay matrix is computed. In this manner, more leading monomials of adjacent rows in the permuted Macaulay matrix are off by one (see Figure 3.4) which reduces the number of missing pivot monomials after row-swapping.

To sum up, by permuting the columns either before or after computing the Macaulay matrix, non-consecutive memory access can be avoided. With row-swapping, the majority of the pivot rows are placed in their final position therefore there is no need to perform row reduction on them. In addition, by ignoring the lower part of the Macaulay matrix, the number of rows to process may drop substantially. Finally, by performing Gauss-Jordan elimination with pivot monomials that are present after row-swapping, both the number of rows and columns can be considerably reduced. Gaussian elimination only needs to be applied on the reduced Macaulay matrix, as shown in Figure 3.3(b).

Note that sparse linear algebra, e.g. block Lanczos algorithm [46] and block Wiedemann algorithm [23], can also be used to extract a sub-system from the Macaulay matrix and might achieve better performance.



(a) Without Gauss-Jordan Elimination

(b) With Gauss-Jordan Elimination

Figure 3.4: Effect of Applying Gauss-Jordan Elimination on the Initial $\mathcal{MQ}$ System

Algorithm 3 Reduce the Dimension of a Macaulay Matrix over $\mathbb{F}_2$

```
procedure ReduceMacDim
    Input:
        a degree-$D$ Macaulay matrix $\mathcal{M}$
        number of variables to keep: $k$
        number of variables in $\mathcal{M}$: $n$
        number of equations to keep as the sub-system candidates: $a \cdot r$

    Initialize:
        $\mathcal{M}' \leftarrow$ Permute the matrix $\mathcal{M}$ according to the monomial order $>_{\mathrm{deg}_k}$
        $N \leftarrow$ the number of non-linear$_k$ monomials

    for $i = 1$ to $N$ do                                                    ▷ Row-swapping
        $r_i \leftarrow$ Starting from the top, find a row in $\mathcal{M}'$
                which has exactly $i - 1$ zeros followed by a one as the leading coefficients
        if a row is found and stored in $r_i$ then
            Swap $r_i$ with the $i^{\mathrm{th}}$ row in $\mathcal{M}'$ and mark the $i^{\mathrm{th}}$ row as final
        end if
    end for

    for $i = 1$ to $a \cdot r$ do                                            ▷ Ignore the lower part
        $r_i \leftarrow$ Pick a random row below the $N^{\mathrm{th}}$ row in $\mathcal{M}'$
        Swap $r_i$ with the $(N + i)^{\mathrm{th}}$ row in $\mathcal{M}'$
    end for
                                                                             ▷ Reduce the dimension
    for each pivot monomial that are already in place after row-swapping do
        $r_{\mathrm{piv}} \leftarrow$ the row corresponding to the pivot monomial
        $j \leftarrow$ the index of the pivot monomial, starting from 1
        for $i = 1$ to $(N + a \cdot r)$ do
            $r_i \leftarrow$ the $i^{\mathrm{th}}$ row in $\mathcal{M}'$
            if $r_i$ is not marked as final and the $j^{\mathrm{th}}$ monomial in $r_i$ is 1 then
                Perform row reduction on $r_i$ with $r_{\mathrm{piv}}$
            end if
        end for
    end for

    $\mathcal{RM} \leftarrow$ Extract the rows that are not marked as final from the first $N + a \cdot r$ rows of $\mathcal{M}'$
            and drop the pivot monomials that are in place after row-swapping therein
            to create the reduced Macaulay matrix
    Output $\mathcal{RM}$
end procedure
```

3.2.2 Testing the Solvability of a Linear System

A trivial approach for testing the solvability of a linear system is to simply solve it with Gauss-Jordan elimination. However, since acquiring a solution to the system is neither necessary nor possible unless the system is solvable, some optimization techniques can be applied. This thesis proposes a variant of the Gauss-Jordan elimination algorithm that can test the solvability of a linear system $\mathcal{S}$ of m equations in k variables and subsequently extract a solution if possible in $\mathcal{O}(k^2)$ and $\mathcal{O}(k)$ machine instructions respectively, under the condition that m is no larger than the size of a machine word, which for example is 64 for 64-bit architectures.

In the original Gauss-Jordan elimination algorithm, once a pivot row for the i^{th} pivot element is located, it is moved to its final position by swapping with the i^{th} row. For example, when solving the linear system

$$\mathcal{S} = \begin{array}{c} \\ eq_1 \\ eq_2 \\ eq_3 \\ eq_4 \\ eq_5 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & 1 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \xrightarrow[\text{no swapping}]{1^{\text{st}}\text{ pivot element}} \begin{array}{c} eq_1 \\ eq_2 \\ eq_3 \\ eq_4 \\ eq_5 \end{array} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \xrightarrow[eq_3,\ eq_4]{\text{reduce with}} \begin{array}{c} eq_1 \\ eq_2 \\ eq_3 \\ eq_4 \\ eq_5 \end{array} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

$$\xrightarrow[\text{swap } eq_2 \text{ and } eq_3]{2^{\text{nd}}\text{ pivot element}} \begin{array}{c} eq_1 \\ eq_3 \\ eq_2 \\ eq_4 \\ eq_5 \end{array} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \xrightarrow[eq_1,\ eq_4,\ eq_5]{\text{reduce with}} \begin{array}{c} eq_1 \\ eq_3 \\ eq_2 \\ eq_4 \\ eq_5 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$\xrightarrow[\text{swap } eq_2 \text{ and } eq_4]{3^{\text{rd}}\text{ pivot element}} \begin{array}{c} eq_1 \\ eq_3 \\ eq_4 \\ eq_2 \\ eq_5 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \xrightarrow[eq_3,\ eq_5]{\text{reduce with}} \begin{array}{c} eq_1 \\ eq_3 \\ eq_4 \\ eq_2 \\ eq_5 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\xrightarrow[\text{no swapping}]{4^{\text{th}}\text{ pivot element}} \begin{array}{c} eq_1 \\ eq_3 \\ eq_4 \\ eq_2 \\ eq_5 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \xrightarrow[eq_1,\ eq_5]{\text{reduce with}} \begin{array}{c} eq_1 \\ eq_3 \\ eq_4 \\ eq_2 \\ eq_5 \end{array} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

in total four rows have to be swapped. However, row swapping can be completely avoided by maintaining a mask that tracks which rows are in their final position.

$$\begin{array}{l} \text{mask} \\ \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{array} \end{array} \quad \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \xrightarrow[\text{reduce with } eq_3,\ eq_4]{1^{\text{st}}\text{ pivot row: } eq_1} \begin{array}{c} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{array} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \xrightarrow[\text{reduce with } eq_1,\ eq_4,\ eq_5]{2^{\text{nd}}\text{ pivot row: } eq_3} \begin{array}{c} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$\xrightarrow[\text{reduce with } eq_3,\ eq_5]{3^{\text{rd}}\text{ pivot row: } eq_4} \begin{array}{c} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \xrightarrow[\text{reduce with } eq_1,\ eq_5]{4^{\text{th}}\text{ pivot row: } eq_2} \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{array} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

After as many rows as the number of variables k in the linear system $\mathcal{S}$ have been marked as final, the algorithms stops. The remaining unmarked rows are redundant equations and their first k coefficients which represent the variables $x_1, x_2, \ldots, x_k$ are guaranteed to be zero. Therefore, testing the solvability of $\mathcal{S}$ is as simple as checking if the constant term of any of the redundant equations is non-zero, which lies on the last column.

Algorithm 4 Test the Solvability of a Linear System over $\mathbb{F}_2$

```
procedure TestSolvable
    Input:
        a linear system $\mathcal{S} = \{c_1, c_2, \ldots, c_{k+1}\}$ stored in column-wise format. In other words,
            the $i^{\text{th}}$ term of the $j^{\text{th}}$ equation in $\mathcal{S}$ is stored as the $j^{\text{th}}$ bit in $c_i$.
        number of variables in $\mathcal{S}$: $k$

    Initialize:
        mask ← all 1's

    for $i = 1$ to $k$ do
        $c'_i = c_i$ & mask                                   ▷ Bitwise AND instruction
        if $c'_i$ is all 0's then
            continue                                          ▷ Missing pivot element
        end if
        $x$ ← count_trailing_zero($c'_i$) ▷ Commonly supported by the hardware as an instruction
        xor_mask ← Flip the $(x+1)^{\text{th}}$ bit in $c'_i$          ▷ At most two instructions

        for $j = i$ to $k+1$ do
            if the $(x+1)^{\text{th}}$ bit of $c_j$ is 1 then
                $c_j$ ← $c_j \oplus$ xor_mask                      ▷ Bitwise XOR instruction
            end if
        end for

        mask ← Flip the $(x+1)^{\text{th}}$ bit in mask
    end for

    if $c_{k+1}$ & mask is all 0's then
        Output true                ▷ The system can be determined or underdetermined
    else
        Output false
    end if
end procedure
```

Clearly, if the system is solvable, a solution can be extracted from the last column based on the first k columns. In particular, the position of 1 in the i^{th} column points to the value for x_i in the last column. For example, since 1 is the first element in the first column, the value for x_1 is therefore the first element in the last column. In the manner, the solution for the linear system above can be extracted as $(x_1, x_2, x_3, x_4) = (1, 1, 1, 0)$.

Note that before extracting a solution one has to test whether or not the system is underdetermined. To achieve this, one can simply verify that none of the first k columns is completely zero since one such column implies a missing pivot element. Obviously this verification can be done simultaneously while extracting a solution therefore it requires no extra computation.

Algorithm 5 Extract a Solution from a Linear System over $\mathbb{F}_2$

```
procedure ExtractSolution
    Input:
        a linear system $\mathcal{S} = \{c_1, c_2, \dots, c_{k+1}\}$ that has been processed by TestSolvable
        number of variables in $\mathcal{S}$: $k$

    for $i = 1$ to $k$ do
        if $c_i$ is all 0's then
            abort                                        ▷ The system is underdetermined
        else
            $j \leftarrow$ count_trailing_zero($c_i$)
            Output $x_i$ = the $(j+1)^{\text{th}}$ bit of $c_{k+1}$
        end if
    end for
end procedure
```

3.2.3 Fixing Variables in the Sub-system during Exhaustive Search

As pointed out in Section 3.1.5, the FastEvaluate algorithm used by the original Crossbred algorithm to fix variables in an extracted sub-system is not suitable for parallelization. For a sub-system $\mathcal{S}$ of m degree-D equations in n variables, this thesis proposes to use the Gray code enumeration algorithm (see Section 2.1.12) to fix $n-k$ variables in $\mathcal{S}$, which requires $\mathcal{O}(D \cdot k)$ machine instructions under the condition that m is no larger than the size of a machine word (same as the condition for Algorithm 4).

Gray code enumeration was first proposed by the authors of the Fast Exhaustive Search algorithm (see Section 3.1.3) to efficiently evaluate a function $f(x_1, x_2, \dots, x_n)$ whose output is a constant over $\mathbb{F}_2$. To obtain the result of evaluating f on the next point $\vec{a}$ from the current result $f(\vec{a'})$ where only the i^{th} coordinates of $\vec{a}$ and $\vec{a'}$ differ, $\mathcal{O}(1)$ machine instructions are executed [20] to combine $f(\vec{a'})$ with the result of evaluating the first order partial derivative $\frac{\partial f}{\partial x_i}$ on $\vec{a'}$. In particular,

$$f(\vec{a}) = f(\vec{a'}) + \frac{\partial f}{\partial x_i}(\vec{a'}).$$

This technique can be applied recursively to evaluate $\frac{\partial f}{\partial x_i}(\vec{a'})$ and its higher order partial derivatives until the partial derivative reduces to a constant. Therefore, if f is of degree D, $\mathcal{O}(D)$ machine instructions are required to compute $f(\vec{a})$. Nevertheless, the same technique can also be applied to evaluate a function f whose output is a *linear function* instead of a constant over $\mathbb{F}_2$. For example, given a function $f(x_4, x_5, x_6, x_7)$ of degree 4 whose output is a linear function in x_1, x_2, x_3

$$\begin{aligned} f = {} & x_1x_4x_5x_6 + x_1x_4x_5x_7 + x_4x_5x_6x_7 + x_1x_4x_5 + x_2x_4x_6 + x_4x_6x_7 + \\ & x_1x_4 + x_1x_5 + x_5x_7 + x_6x_7 + x_1 + x_2 + x_4 + 1 \end{aligned}$$

as well as some of its higher order partial derivatives

$$\begin{aligned} \frac{\partial f}{\partial x_4} &= x_1x_5x_6 + x_1x_5x_7 + x_5x_6x_7 + x_1x_5 + x_2x_6 + x_6x_7 + x_1 + 1 \\ \frac{\partial^2 f}{\partial x_4 \partial x_7} &= x_1x_5 + x_5x_6 + x_6 \\ \frac{\partial^3 f}{\partial x_4 \partial x_6 \partial x_7} &= x_5 + 1 \end{aligned}$$

the result of evaluating f at the point $(0, 0, 0, 0)$ can be computed with Gray code enumeration as

$$\begin{aligned}
f(0,0,0,0) &= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,0) \\
&= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,1) + \frac{\partial^2 f}{\partial x_4 \partial x_7}(1,0,0,1) \\
&= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,1) + \frac{\partial^2 f}{\partial x_4 \partial x_7}(1,0,1,1) + \frac{\partial^3 f}{\partial x_4 \partial x_6 \partial x_7}(1,0,1,1) \\
&= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,1) + \frac{\partial^2 f}{\partial x_4 \partial x_7}(1,0,1,1) + \frac{\partial^3 f}{\partial x_4 \partial x_6 \partial x_7}(1,0,1,0) + \frac{\partial^4 f}{\partial x_4 \partial x_6 \partial x_7 \partial x_7} \\
&= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,1) + \frac{\partial^2 f}{\partial x_4 \partial x_7}(1,0,1,1) + \frac{\partial^3 f}{\partial x_4 \partial x_6 \partial x_7}(1,0,1,0) + 0 \\
&= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,1) + \frac{\partial^2 f}{\partial x_4 \partial x_7}(1,0,1,1) + 1 \\
&= f(1,0,0,0) + \frac{\partial f}{\partial x_4}(1,0,0,1) + 1 + 1 \\
&= f(1,0,0,0) + x_1 + 1 + 1 + 1 \\
&= x_2 + x_1 + 1 + 1 + 1 \\
&= x_1 + x_2 + 1
\end{aligned}$$

because the 4-bit Gray code encoding changes in the following manner (see Table 2.1)

$$1010 \to 1011 \to 1001 \to 1000 \to 0000.$$

Note that the index of the bit that differs between the current and the next Gray code encodings is exactly the index of the last bit set to 1 in the binary representation of the value represented by the latter encoding. For example, when changing from $g_{\text{cur}} = 0010$ to $g_{\text{next}} = 0110$ (see Table 2.1), the binary representation of the decimal value represented by g_{next} is 0100, whose last bit set to 1 has the same index as the bit that flips between two Gray code encodings.

Clearly, since the result of evaluating f or any of its partial derivatives on a point $\vec{a}$ in $\mathbb{F}_2^4$ is a linear function that can be represented by four $\mathbb{F}_2$ elements (three variables and the constant term) and the last order partial derivatives reduce to constants, evaluating $f(\vec{a})$ takes at most $3 \cdot (3 + 1) + 1$ xor instructions and another $4 \cdot 2$ instructions for computing the indices of the coordinates that changed during enumeration. In general, for a function f of degree D whose output is a linear function in k variables, evaluating f requires $\mathcal{O}(D \cdot k)$ machine instructions.

To correctly enumerate with Gray Code encodings, during initialization the function f and its partial derivatives must be evaluated at certain initial points. The proof as well as the pseudocode for the recursive initialization procedure was given in the paper that first proposed the algorithm [20]. This thesis omits the proof however provides an iterative version of the initialization procedure (see Algorithm 7).

Since a machine instruction operates on machine words, which for example have size 64 for 64-bit architectures, multiple functions can be evaluated with Gray code enumeration simultaneously. Therefore, the algorithm described above can be applied to fix $n - k$ variables in an extracted sub-system $\mathcal{S}$ of m equations in n variables, as long as m is no larger than the machine word size.

Nevertheless, to fix $n - k$ variables in $\mathcal{S}$ with all $(n - k)$-tuples in $\{0, 1\}^{n-k}$, the sub-system must be evaluated 2^{n-k} times. Since Gray code enumeration is highly parallelizable, one method for reducing the size of the search space is to divide the search space into N smaller subspaces of size $\frac{2^{n-k}}{N}$ and launch N threads to perform enumeration. In this manner, each thread evaluates the same sub-system with a different starting point. However, an alternative approach is to first *fix t variables* in the sub-system $\mathcal{S}$ with all t-tuples in $\{0, 1\}^t$ to create 2^t smaller sub-systems in $n - t$ variables. With this alternative approach, the starting point of each thread can be the same, and although the sub-systems are distinct from each other, their last order partial derivatives with respect to the $n - t - k$ variables that must be fixed remain identical. These two properties of the alternative approach may be desirable in certain situation, e.g. when implementing the algorithm on the GPU architecture (see Section 4.2).

Algorithm 6 Gray Code Enumeration for Linear Systems

```
procedure GrayCodeUpdate
    Input:
        the result of evaluating a degree-$D$ function $f(x_{k+1}, x_{k+2}, \dots, x_n)$ on the
            current point whose output is a linear function in $x_1, x_2, \dots, x_k$ that is
            represented by $k+1$ bits: $(r_1, r_2, \dots, r_k, r_c)$
        coordinates of the next point, which is represented by an integer whose last bit is
            the last coordinate $x_n$ and so on: $c$
        the results of evaluating the degree $i$ partial derivatives of $f$, $1 \le i \le D$
            at previous points, which use the same representation as $(r_1, r_2, \dots, r_k, r_c)$: eval

    $\mathrm{idx}_1 \leftarrow$ count_trailing_zero($c$)    ▷ Compute the indices of the coordinates that changed
    tmp $\leftarrow$ Flip the bit at index $\mathrm{idx}_1$ in $c$
    for $i = 2$ to $D$ do
        $\mathrm{idx}_i \leftarrow$ count_trailing_zero(tmp)

        if $\mathrm{idx}_i$ is valid then    ▷ May not be valid as $c$ may not have $D$ bits set to 1
            tmp $\leftarrow$ Flip the bit at index $\mathrm{idx}_i$ in tmp
        end if
    end for

    for $i = D$ to 2 do    ▷ Update the evaluation of partial derivatives
        if $\mathrm{idx}_i$ is valid then
                ▷ Evaluation of $\frac{\partial^{i-1} f}{\partial x_{(n-\mathrm{idx}_{i-1})} \partial x_{(n-\mathrm{idx}_{i-2})} \cdots \partial x_{(n-\mathrm{idx}_1)}}$
            $(b_1, b_2, \dots, b_k, b_c) \leftarrow \mathrm{eval}[\mathrm{idx}_{i-1}, \mathrm{idx}_{i-2}, \dots, \mathrm{idx}_1]$
                ▷ Evaluation of $\frac{\partial^{i} f}{\partial x_{(n-\mathrm{idx}_{i})} \partial x_{(n-\mathrm{idx}_{i-1})} \cdots \partial x_{(n-\mathrm{idx}_1)}}$
            $(b'_1, b'_2, \dots, b'_k, b'_c) \leftarrow \mathrm{eval}[\mathrm{idx}_i, \mathrm{idx}_{i-1}, \mathrm{idx}_{i-2}, \dots, \mathrm{idx}_1]$

            for $j = 1$ to $k$ do
                $b_j \leftarrow b_j \oplus b'_j$
            end for
            $b_c \leftarrow b_c \oplus b'_c$

            $\mathrm{eval}[\mathrm{idx}_{i-1}, \mathrm{idx}_{i-2}, \dots, \mathrm{idx}_1] \leftarrow (b_1, b_2, \dots, b_k, b_c)$
        end if
    end for

    $(b_1, b_2, \dots, b_k, b_c) \leftarrow \mathrm{eval}[\mathrm{idx}_1]$    ▷ Evaluate $f$ on the next point
    for $j = 1$ to $k$ do
        $r_j \leftarrow r_j \oplus b_j$
    end for
    $r_c \leftarrow r_c \oplus b_c$

    Output $(r_1, r_2, \dots, r_k, r_c)$
end procedure
```

Algorithm 7 Iterative Initialization of Data Structures for Gray Code Enumeration

```
procedure GrayCodeInit
    Input:
        a degree-$D$ function $f(x_{k+1}, x_{k+2}, \ldots, x_n)$ whose output is a linear function
            in $x_1, x_2, \ldots, x_k$
        container for results of evaluating the degree-$i$ partial derivatives of $f$, $1 \leq i \leq D$: eval

    for $\mathrm{idx}_1 = 0$ to $n - k - 1$ do
        if $\mathrm{idx}_1 = 0$ then
            $p_1 \leftarrow$ all 0's, in total $n - k$ digits
        else
            $p_1 \leftarrow$ all 0's except the bit at index $\mathrm{idx}_1 - 1$, e.g. for $\mathrm{idx}_1 = 1$, $p_1 = 00\ldots01$
        end if

        $\mathrm{eval}[\mathrm{idx}_1] \leftarrow \frac{\partial f}{\partial x_{(n-\mathrm{idx}_1)}}(p_1)$

        for $\mathrm{idx}_2 = \mathrm{idx}_1 + 1$ to $n - k - 1$ do
            if $\mathrm{idx}_2 = \mathrm{idx}_1 + 1$ then
                $p_2 \leftarrow p_1$
            else
                $p_2 \leftarrow$ Flip the bit at index $\mathrm{idx}_2 - 1$ of $p_1$
            end if

            $\mathrm{eval}[\mathrm{idx}_2, \mathrm{idx}_1] \leftarrow \frac{\partial^2 f}{\partial x_{(n-\mathrm{idx}_1)} \partial x_{(n-\mathrm{idx}_2)}}(p_2)$

                        $\ddots$ ▷ $D - 3$ level of loops, where one is wrapped by its previous level

                            for $\mathrm{idx}_D = \mathrm{idx}_{D-1} + 1$ to $n - k - 1$ do        ▷ Reduce to constant
                                $\mathrm{eval}[\mathrm{idx}_D, \ldots, \mathrm{idx}_2, \mathrm{idx}_1] \leftarrow \frac{\partial^D f}{\partial x_{(n-\mathrm{idx}_1)} \partial x_{(n-\mathrm{idx}_2)} \cdots \partial x_{(n-\mathrm{idx}_D)}}$
                            end for
                        $\iddots$


        end for
    end for
end procedure
```

Algorithm 8 Perform Gray Code Enumeration in Parallel

```
procedure ParallelGrayCodeEnumerate
    Input:
        an extracted sub-system of $m$ degree-$D$ equations in $n$ variables $x_1, x_2, \dots, x_n$
            which can be turned into a linear system in $x_1, x_2, \dots, x_k$ by fixing the
            rest of the $n - k$ variables: $\mathcal{S} = \{f_1, f_2, \dots, f_m\}$
        number of threads to launch: $2^t$

    for each $v_i = (x_{n-t+1}, \dots, x_n)$ in $\{0,1\}^t$ do     ▷ Initialize data structure for enumeration
        $\mathcal{S}_i \leftarrow$ Fix $\mathcal{S}$ according to $v_i$ to obtain $\{f'_{i1}, f'_{i2}, \dots, f'_{im}\}$

                        ▷ Can be done simultaneously by packing $m$ bits into a machine word
        for $j = 1$ to $m$ do
            ▷ eval$_i[j]$ is the result of evaluating the partial derivatives of $f'_{ij}$ (see Algorithm 6)
            Call GrayCodeInit($f'_{ij}$, eval$_i[j]$)
        end for
    end for

    for $i = 1$ to $2^t$ do                                    ▷ Execute the loop body in parallel
        $c \leftarrow 0$
        $\mathcal{R} \leftarrow$ Evaluate $\mathcal{S}_i$ with $(x_{k+1}, x_{k+2}, \dots, x_{n-t}) = (0, 0, \dots, 0)$
                and store the result in column-wise format as described in Algorithm 4

        while $c < 2^{n-k-t}$ do
            $\mathcal{R}' \leftarrow \mathcal{R}$                  ▷ Make a copy because TestSolvable will modify $\mathcal{R}$
            solvable $\leftarrow$ TestSolvable ($\mathcal{R}', k$)
            if solvable = true then
                solution $\leftarrow$ Call ExtractSolution ($\mathcal{R}', k$) and collect the values for $x_1, x_2, \dots, x_k$
                Output solution
            end if

            $c \leftarrow c + 1$
                        ▷ Can be done simultaneously by packing $m$ bits into a machine word
            for $j = 1$ to $m$ do
                $(r_1, r_2, \dots, r_k, r_c) \leftarrow$ Extract the evaluation of $f'_{ij}$ at the current point from $\mathcal{R}$
                                    ▷ This extraction is only for explaining the algorithm
                                            ▷ In practice there is no need to do so
                Call GrayCodeUpdate($(r_1, r_2, \dots, r_k, r_c), c$, eval$_i[j]$)
            end for
        end while
    end for
end procedure
```

3.2.4 Fixing Variables in the Sub-system before Exhaustive Search

In addition to fixing variables in the initial $\mathcal{MQ}$ system as described in the original Crossbred algorithm (see Section 3.1.5), one can further fix some variables in the extracted sub-system *before* entering the exhaustive search stage. This might be necessary as the hardware might not have enough resources, e.g. memory and registers, for performing exhaustive search on the sub-system. By fixing b variables the sub-system beforehand, one can divide the workload evenly into 2^b smaller sub-systems which require less resources for applying exhaustive search. Clearly, since the main purpose of fixing these b variables in the sub-system is to fine-tune the resource requirement, the choice of b should be adjusted based on the hardware.

To summarize, the proposed Parallel Crossbred algorithm first applies $\mathcal{MQ}$ preprocessing to fix p variables in the initial $\mathcal{MQ}$ system $\mathcal{F}$, which divides the workload into 2^p smaller $\mathcal{MQ}$ systems $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_{2^p}$. Then based on each $\mathcal{MQ}$ system $\mathcal{F}_i$ and the number of variables to keep in the final linear system k, the degree-D Macaulay matrix $\mathrm{Mac}_D^k(\mathcal{F}_i)$ is computed. Thereafter, the ReduceMacDim algorithm for reducing the dimension of $\mathrm{Mac}_D^k(\mathcal{F}_i)$ (see Algorithm 3) is applied to compute the reduced Macaulay matrix, from which a sub-system $\mathcal{S}$ can be extract with Gaussian elimination. Based on the parameter b, which is adjusted according to the amount of resources that the hardware can provide, the Parallel Crossbred algorithm fixes f variables in the extracted sub-system $\mathcal{S}$ to split the workload into 2^b smaller sub-systems $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{2^b}$. Subsequently for each smaller sub-system $\mathcal{S}_j$, the ParallelGrayCodeEnumerate algorithm (see Algorithm 8) is applied to extract solution candidates, which are then verified with the initial $\mathcal{MQ}$ system.

Algorithm 9 The Adapted Crossbred Algorithm

```
procedure ParallelCrossbred
    Input:
        an $\mathcal{MQ}$ system of $m$ equations in $n$ variables $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$
        Macaulay degree: $D$
        number of variables to keep: $k$
        number of variables to fix during $\mathcal{MQ}$ preprocessing: $p$
        number of variables to fix before entering exhaustive search stage: $b$
        number of variables to keep as the sub-system candidates: $a \cdot r$
        number of threads to launch: $2^t$

    for each $u = (x_{n-p+1}, \ldots, x_n)$ in $\{0,1\}^p$ do
        1. Fix the last $p$ variables in $\mathcal{F}$ according to $u$ to obtain an $\mathcal{MQ}$ system $\mathcal{F}'$.
        2. Compute the degree-$D$ Macaulay matrix $\mathrm{Mac}_D^k$ based on $\mathcal{F}'$.
        3. Call ReduceMacDim($\mathrm{Mac}_D^k, k, n-p, a \cdot r$) to compute the reduced Macaulay
            matrix $\mathcal{RM}$.
        4. Apply Gaussian elimination on $\mathcal{RM}$ to extract a system $\mathcal{SC}$ of
            $a \cdot r$ sub-system candidates.
        5. Extract $r$ linearly independent equations from $\mathcal{SC}$ to create the sub-system $\mathcal{S}$.

        for each $t = (x_{n-p-b+1}, \ldots, x_{n-p})$ in $\{0,1\}^b$ do
            6. Fix the last $b$ variables in $\mathcal{S}$ according to $t$ to obtain a smaller sub-system $\mathcal{S}'$.

            Call ParallelGrayCodeEnumerate($\mathcal{S}', 2^t$) and
            for each output solution candidate $s = (x_1, x_2, \ldots, x_{n-p-b})$ in $\{0,1\}^{n-p-b}$ do
                7. Combine $s$ with $t$ and $u$ then verify it with $\mathcal{F}$. Output the solution
                    and subsequently terminate if it is correct, otherwise continue.
            end for
        end for
    end for
end procedure
```

Chapter 4

Implementation

To test the performance of the proposed algorithm (see Section 3.2), a program called *MQsolver* is implemented in pure C language, i.e. not C++. This chapter documents the design choices and implementation details of MQsolver as well as the result of profiling the implementation.

4.1 Extracting a Sub-system from the Macaulay Matrix

To compute the reduced Macaulay matrix $\mathcal{RM}$, Algorithm 3 is implemented on the CPU architecture because $\mathcal{RM}$ might be too large for the off-chip memory of a GPU. Sparse matrix representation where each row only stores the indices of the non-zero columns is used for the initial Macaulay matrix because of its high sparsity. On the other hand, dense matrix representation is used for the reduced Macaulay matrix as it can be quite dense both before and after applying Gaussian elimination (see Figure 3.3).

After $\mathcal{RM}$ is computed, it can be copied to the GPU if the off-chip memory can accommodate it. Subsequently a sub-system can be extracted with Gaussian elimination on the GPU and copied back to the system main memory. On the other hand, if the size of $\mathcal{RM}$ is too large then Gaussian elimination is simply performed with the CPU. To speed up the CPU, the implementation is parallelized with the *POSIX Thread API* to distribute the workload to all CPU cores. It has been observed during experiments that the GPU implementation outperforms the CPU version by a factor of 9 in most cases. Nevertheless, a more thorough analysis is necessary in order to figure out the exact factor.

Since the size of registers on a GPU is 32 bits, and Algorithm 4 as well as Algorithm 5 requires the input linear system to be stored in column-wise format, only $2 \cdot 32$ sub-system candidates are extracted from the reduced Macaulay matrix which turned out to be sufficient to provide 32 linearly independent equations for the sub-system.

4.2 Fixing Variables in the Sub-system

Algorithm 8 for fixing $n - k$ variables in the degree-D sub-system $\mathcal{S}$ to enumerate linear systems in k variables is implemented on the GPU architecture. The data structures used by Gray code enumeration are allocated from the off-chip global memory. However, as noted in Section 3.2.3, the last partial derivatives are constants and remain the same for these 2^t smaller sub-systems $\mathcal{S}_i, 1 \leq i \leq 2^t$ obtained by fixing t variables in $\mathcal{S}$ so they can be stored in the read-only constant memory (see Section 2.2.2). The GPU threads in a warp begin enumeration with the same starting point and consequently they will access partial derivatives with respect to the same variables in each iteration. In this manner, only one load-instruction is required for the whole warp. In addition, because of the cyclic nature of Gray code enumeration (see Section 3.1.3), the data stored in the constant cache can be reused again, which amortizes the cost of the load-instruction.

Since the last order partial derivatives of $\mathcal{S}_i$ with respect to a set of variables is stored in column-wise format as described in Algorithm 4, one single 32-bit integer is sufficient. Therefore, in total $\binom{n-k-t}{D}$ 32-bit integers are required for the sub-system $\mathcal{S}$.

On the other hand, the result of evaluating non-constant partial derivatives of $\mathcal{S}_i$ with respect to a set of variables is a linear system in k variables and is stored as $k+1$ 32-bit integers. Since a warp consists of 32 threads, $32 \cdot (k+1)$ integers are required for these partial derivatives. As mentioned above, threads in a warp access partial derivatives with respect to the same set of variables in each iteration. Hence, to minimize the amount of memory to access and to avoid non-coalesced memory access (see Section 2.2.2), these $32 \cdot (k+1)$ integers in the global memory are interleaved as shown in Figure 4.1.

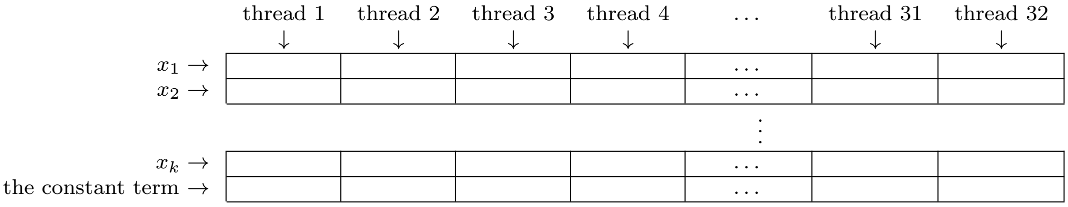


Figure 4.1: Interleaved Data Structures for Gray Code Enumeration

Those $32 \cdot (k+1)$ interleaved integers form the basic unit of the data structures for $\mathcal{S}_i$. Since $n-k-t$ variables have to be fixed during enumeration, storing results of evaluating the non-constant partial derivatives of $\mathcal{S}_i$ requires $\sum_{j=1}^{D-1} \binom{n-k-t}{j}$ such basic units for a warp. Together with the result of evaluating $\mathcal{S}_i$ at the current point, which requires one basic unit as well, a warp requires $\sum_{j=0}^{D-1} \binom{n-k-t}{j}$ basic units. Therefore, in total Algorithm 8 needs $(2^{t-5} \cdot \sum_{j=0}^{D-1} \binom{n-k-t}{j}) \cdot 32 \cdot (k+1)$ 32-bit integers in global memory and $\binom{n-k-t}{D}$ 32-bit integers in the constant memory to operate.

4.3 Testing the Solvability of a Linear System

Algorithm 4 and Algorithm 5 are also implemented on the GPU architecture because they are sub-routines of Algorithm 8. One caveat of implementing the algorithms is to avoid using an array of local variables to store the input linear system. Normally, when the size of the array is unknown during compilation, the NVCC compiler has no choice but to use local memory (see Section 2.2.2) as the storage for the array. This results in larger memory access latency as the data would then locate on the off-chip memory.

One seemly plausible method for reducing the access latency is to use shared memory (see Section 2.2.2) for the linear system. However, in practice since a GPU kernel can utilize at most 48KB of shared memory, it becomes the primary factor that limits the degree of parallelism. For example, since a linear system in 15 variables is represented by 16 32-bit integers (see Algorithm 4), a warp requires 2KB shared memory. Therefore, for an SM there can be at most 24 warps in all the active blocks. The situation becomes even worse for larger linear systems. Consequently using shared memory to store the linear system is not recommended.

A better approach is to use a macro to represent the size of the array and define the macro at compilation time. In this manner, a linear system can be stored in registers, which is even faster than shared memory. It has been observed by the author of this thesis that NVCC bundled with CUDA version 8.0 is able to optimize away the local memory usage with the help of the macro. However, with older versions, e.g. CUDA version 7.5, NVCC is less competent and fails to do so. To work around this issue, one can manually define as many array elements as the size of the array and unroll the loops used to access those array elements. For this thesis a Python script is used to achieve this, which takes less than 0.1 second. However, the consequence of generating C code at compilation is that MQsolver has to be re-compiled whenever the size of the array changes. Since the size of the array is one plus the number of variables to keep k, another Python script is created to generate C code and re-compile MQsolver according to the parameter k automatically. The compilation takes roughly 6 seconds on a 4GHz CPU with one single thread (AMD FX-8350), which is negligible.

4.4 Verification of Solution Candidates

When a solution candidate is found during Gray code enumeration, it is reported and subsequently verified with the original $\mathcal{MQ}$ system, as shown in Algorithm 8 and Algorithm 9. Ideally one would copy the solution candidate from the GPU off-chip memory back to the main memory and verify it with the CPU immediately. In practice this is not efficient as copying the solution candidate interrupts the workflow of the GPU. Therefore, an alternative approach is to store the solution candidates in a buffer and only copy them back to the main memory after the enumeration finishes.

One caveat of this approach is that a buffer must be allocated on the off-chip memory, which may have little capacity left after allocating memory blocks for the data structures used in Gray code enumeration. If the number of solution candidates is larger than the size of the buffer, some candidates must therefore be dropped. To avoid this pitfall, one can copy some equations from the original $\mathcal{MQ}$ system to the GPU which serve as a filter. Since an equation is expected to filter out half of the solution candidates, using e equations would reduce the number of candidates by a factor of 2^e. With these filtering equations the number of solution candidates can be reduced considerably. Only solution candidates that passed the filter will then be verified with the rest of the equations in the original $\mathcal{MQ}$ system by the CPU.

The number of equations to copy is subject to the implementation. For this thesis 32 equations are selected and stored in column-wise format for efficiency reasons. In this manner, to apply the filter one only needs to evaluate these $\frac{n(n-1)}{2}+n+1$ monomials in the equations with the solution candidate, where n is the number of variables in the original $\mathcal{MQ}$ system. Therefore, it takes at most $\mathcal{O}(n^2)$ machine instructions, whose execution time is observed to be completely hidden by the execution model of GPUs (see Section 2.2.2) during experiments.

4.5 Pipelining

When $\mathcal{MQ}$ preprocessing is applied, i.e fixing p variables in the original $\mathcal{MQ}$ system, one has to extracted a sub-system and subsequently perform Gray code enumeration at most 2^p times. Since Gray code enumeration is executed on the GPU, which operates independently from the CPU, one can pipeline the two stages. In other words, when performing Gray code enumeration on the GPU, a sub-system for launching the next Gray code enumeration can be computed simultaneously with the CPU. In this manner, as long as Gray code enumeration takes more computation time than extracting a sub-system, which is usually the case for large $\mathcal{MQ}$ systems, only the runtime of extracting the first sub-system will manifest. In MQsolver the pipelining of these two stages is implemented with POSIX Thread API and will be activated automatically when the CPU is used for extracting sub-systems.

4.6 Profiling the GPU Kernel

4.6.1 Metrics

In order to evaluate the quality of a GPU kernel implementation, some metrics can be used.

- Occupancy: $\frac{\text{actual number of active warps}}{\text{maximal number of active warps}}$
- GPU utilization: $\frac{\text{actual number of instructions issued}}{\text{maximal number of instructions that can be issued}}$

The first metric provides a coarse indicator of how well latencies, e.g. memory access latency and instruction fetch latency, can be hidden by computing the ratio of actual active warps on the GPU versus the maximal number of warps that the GPU can accommodate. As explained in Section 2.2.2, a warp becomes active when the block to which it belongs acquires execution resources from an SM. Nevertheless, a warp is only executed when a warp scheduler picks the warp from the pool of active warps and dispatches instructions for it. Clearly, the chance of having no eligible warps to dispatch instructions for may decrease as the size of the pool grows. Therefore, occupancy measures how close the actual number of active warps is to the theoretic maximum, which is computed based on the amount of resources required by a warp and the amount of them available on the GPU. If occupancy is lower than the theoretic maximal value, the kernel launch was not properly configured and therefore, some execution resources were not distributed, which could have been used to accommodate more warps on the GPU.

On the other hand, the second metric shows how the CUDA cores are actually utilized. At each cycle, a CUDA core is able to execute one instruction. When it fails to do so due to, e.g. memory access latency, the capacity of the hardware is not fully utilized. GPU utilization is the ratio of the number of instructions that were executed during the kernel execution versus the maximal number of instructions that could have been executed.

4.6.2 Result

MQsolver has been profiled with the Nvidia Visual Profiler and the following settings

- GPU: Nvidia Quadro M1000M, 4GB off-chip memory, 4SMs, 512 CUDA cores in total
- CUDA version: 8.0
- Input: an $\mathcal{MQ}$ system of 82 equations in 41 variables
- Macaulay degree: $D = 3$
- Number of variables to keep: $k = 15$
- Number of GPU threads: 2^{14}, i.e. $t = 14$
- $\mathcal{MQ}$ preprocessing: $p = 0$
- Number of variables to fix before enumeration: $b = 0$

The result shows that with the resources available on Quadro M1000M, the theoretical maximal occupancy is 65.62% while in practice 61.8% is achieved. The factor that decides the maximal occupancy is the number of registers required for each thread. MQsolver requires 48 register for a thread when keeping 15 variables in the linear system so a warp requires 1536 registers. Since each block may use at most 65536 registers, there can be at most 42 active warps at a time. The maximal occupancy is therefore $\frac{42}{64} = 65.62\%$ because the maximal number of warps for a streaming multiprocessor on Quadro M1000M is 64. Note that as mentioned above, occupancy is simply an estimation of how likely the latencies can be hidden. Therefore, an occupancy of 100% is not necessarily better than 61.8% as long as the number of active warps is high enough to hide various latencies (see Section 5.1.1 for an example). As the actual occupancy is very close to the theoretical maximum, the kernel launch can be considered to be properly configured.

In contrast, MQsolver achieves only 37.06% GPU utilization. Warps were stalled due to

- The warp was eligible but not selected by the warp scheduler: 3.3%
- Instruction fetch latency: 3.6%
- Data dependency: 5.5%
- Memory access latency: 73.7%
- Other: 13.9%

Therefore in theory, MQsolver can perform 2.13 times faster if the memory access latency is completely hidden and the data dependency issue is resolved. The latter issue can be achieved by enforcing instruction-level parallelism (see Section 2.2.1) while the former one can be alleviated by exploiting the cyclic nature of Gray code enumeration. In particular, as mentioned in Section 3.1.3, some of the partial derivatives are accessed more frequently during enumeration, therefore one can explicitly store these partial derivatives in faster memory, e.g. registers and shared memory. As shown in [20], in theory these partial derivatives will never be evicted under optimal cache replacement policy hence there would be no need to load or store these partial derivatives back and forth from the off-chip memory.

Chapter 5

Experiment and Analysis

In this chaper, proper choices of the parameters for the proposed algorithm are discussed and subsequently the experimental results of solving overdetermined and determined $\mathcal{MQ}$ systems with MQsolver are presented.

5.1 Choice of Parameters

5.1.1 Number of GPU Threads

As discussed in Section 4.6, occupancy provides a rough estimation of the degree of parallelism and higher occupancy is not necessarily better as long as a certain threashold value has been achieved. Therefore, one would expect that the number of threads 2^t to launch on a GPU does not have influence on the performance of MQsolver after this threshold value. To verify this conjecture, a random solvable $\mathcal{MQ}$ system of 92 equations in 46 variables was generated and used as the test bench. For this series of experiments, MQsolver was launched on a Nvidia Quadro M1000M GPU (same as the one in Section 4.6) with the following settings

- GPU: Nvidia Quadro M1000M, 4GB off-chip memory, 4SMs, 512 CUDA cores in total
- CUDA version: 8.0
- Macaulay degree: $D = 3$
- $\mathcal{MQ}$ preprocessing: $p = 0$
- Number of variables to fix before enumeration: $b = 0$
- Number of variables to keep: $k = 16$

and the result is given in Table 5.1. As shown in the table, indeed the runtime basically remains constant for $t \geq 12$. On the contrary, for $t < 12$, the degree of parallelism is not sufficient therefore the latencies manifest. In particular, when $t = 7$, there are only $2^7 = 128$ GPU threads deployed onto the GPU, which has 512 CUDA cores in total. Therefore 75% of the CUDA cores sit idle while the remaining CUDA cores have to perform all the computation. In this case, there is very little parallelism thus the performance suffers. As the number of GPU threads increases, the workload is distributed to more CUDA cores but the degree of parallelism remains insufficient.

When $t = 9$, there are $2^9 = 512$ GPU threads deployed, which is exactly the number of CUDA cores available on this particular GPU. In this case the workload is finally distributed to all the CUDA cores. Nevertheless, the degree of parallelism is still far from enough because executing a single thread on a CUDA core cannot hide latencies at all. For example, when the thread needs to load data from the global memory, which requires hundreds of cycles to access, there is no other thread that can take over the execution resources hence the CUDA core has no choice but to stall.

Starting from $t = 10$, the degree of parallelism becomes better which in turn decreases the amount of latencies that manifest. The performance gradually improves but not exponentially as when $t = 7 \sim 9$. Finally when $t = 12$, the degree of parallelism reaches a point where deploying more threads does not improve the ability of the GPU to hide latencies anymore. With those experimental settings, the threshold is therefore 2^{12}, where the optimal performance of MQsolver based on the current implementation is achieved.

Number of Threads	Memory Needed per Thread (KB)	Total Memory Needed (MB)	Constant Memory Needed (B)	Size of Search Space per Thread	Runtime for Exhaustive Search (second)
2^7	18.39	2.30	7084	2^{23}	81.40
2^8	16.87	4.22	6160	2^{22}	40.98
2^9	15.41	7.70	5320	2^{21}	20.97
2^{10}	14.01	14.01	4560	2^{20}	11.93
2^{11}	12.68	25.37	3876	2^{19}	7.16
2^{12}	11.42	45.69	3264	2^{18}	4.89
2^{13}	10.22	81.81	2720	2^{17}	5.15
2^{14}	9.10	145.56	2240	2^{16}	5.15
2^{15}	8.04	257.12	1820	2^{15}	5.04
2^{16}	7.04	450.50	1456	2^{14}	4.95
2^{17}	6.11	782.00	1144	2^{13}	4.94
2^{18}	5.25	1343.00	880	2^{12}	4.79
2^{19}	4.45	2278.00	660	2^{11}	4.86
2^{20}	3.72	3808.00	480	2^{10}	4.86

Table 5.1: Effect of Changing the Number of GPU Threads

Note that as shown in Section 4.2, the amount of global memory required for a GPU thread reduces slightly when t increases by one but the total amount of memory requirement for the GPU kernel soars up nearly twofold because the number of GPU threads becomes two times higher. As for constant memory requirement, it decreases as t increases, as shown in the same section.

5.1.2 Number of Variables to Keep

False Positives and Underdetermined Linear Systems

As mentioned in Section 4.4, once a solution is found on the GPU, it is first checked with the 32 filtering equations before reported to the CPU. The solution candidates that do not pass the filter become *false positives* and are subsequently dropped.

In addition to false positives, which are linear systems that yield a unique solution, *underdetermined linear systems* might appear during Gray code enumeration because those 32 linearly independent equations in the sub-system might become dependent after fixing variables. For example, both of the equations

$$\begin{cases} x_1x_2 + x_2 + 1 = 0 \\ x_1 + x_2 + 1 = 0 \end{cases}$$

become $x_2 + 1 = 0$ after fixing $x_1 = 0$. Consequently, when the number of remaining linearly independent equations drops below the number of variables in the linear system, the system becomes underdetermined and has multiple solutions.

Clearly, the number of false positives is related to the total number of solutions to the linear systems that are enumerated. Since the number of equations in the linear system is constant for MQsolver (which is 32, see Section 4.1), one remaining major factor is the number of variables k in a linear system. As for underdetermined linear systems, they are obviously related to k. Therefore, to investigate the effect of changing the parameter k, a solvable $\mathcal{MQ}$ system is generated as in Section 5.1.1 and subsequently tested with the following settings

- GPU: Nvidia Quadro M1000M, 4GB off-chip memory, 4SMs, 512 CUDA cores in total
- CUDA version: 8.0
- Macaulay degree: $D = 3$ and 4
- $\mathcal{MQ}$ preprocessing: $p = 0$
- Number of variables to fix before enumeration: $b = 0$

while the number of variables in a linear system k and the number of GPU threads 2^t vary. Clearly t does not affect the correctness of MQsolver and as shown in Section 5.1.1, as long as a sufficient degree of parallelism has been achieved, increasing t has no effect. Therefore changing t does not invalidate the experiments. The experimental results are given in Table 5.2. Note that the ratio of the missing pivot monomials as described in Section 3.2.1 is listed as the 2^{nd} column.

Parameters (D, k)	Ratio of Missing Pivots (%)	Initial Sparsity of the Extracted Sub-system	Number of False Positives	Number of Underdetermined Linear Systems	Number of Solution Candidates
(4, 18)	30.72	0.50	16518	0	1
(4, 17)	24.88	0.49	16142	0	1
(4, 16)	17.91	0.49	16245	0	1
(4, 15)	9.48	0.49	16371	0	1
(4, 14)	0.86	0.61	3069025	1	1
(4, 13)	0.00	0.90	2227401339	11718508	2
(4, 12)	0.00	0.93	2944024458	19912529	3
(4, 11)	0.00	0.93	4619656	35340419	2
(4, 10)	0.00	0.93	146244062	176238738	4
(4, 9)	0.00	0.96	1854266662	1359835490	48
(3, 16)	20.58	0.50	16334	0	1
(3, 15)	10.90	0.50	16489	0	1
(3, 14)	0.98	0.54	184126	0	1
(3, 13)	0.00	0.66	2396731	1	1
(3, 12)	0.00	0.74	127602703	6952	1
(3, 11)	0.00	0.79	188889712	21777	1
(3, 10)	0.00	0.84	148940573	15286	1
(3, 9)	0.00	0.84	364066881	104953	3

Table 5.2: Effect of Changing the Number of Variables to Keep

As the table shows, the number of false positives remains almost constant as long as the extracted sub-system behaves like a random system, i.e. has sparsity ≈ 0.5. The experiment shows that a sub-system may fail to do so when the ratio of the missing pivots drops below a certain threshold $\approx 1\%$, in which case the reduced Macaulay matrix is almost the same before and after applying Gaussian elimination. The number of row reduction operations performed during Gaussian elimination is therefore very low, which fails to combine the information contained in the pivot rows with the sub-system candidates that are to be extracted. Consequently, the extracted sub-system does not behave like a random system. Since the ratio of the missing pivots drops as the parameter k decreases, as shown in the table, and being able to keep one more variables in the linear system reduces the number of iterations during Gray code enumeration by a factor of two (see Algorithm 8), one should choose k as high as possible.

As for underdetermined linear systems, for all the experiments they do not appear when the sub-system behaves randomly. Therefore, another benefit of choosing k as high as possible is to reduce the chance of their appearance. Judging from the experimental results, the number of underdetermined linear systems that appear depends on the equations in the extracted sub-system, which vary from sub-system to sub-system. The analysis of the number of underdetermined linear systems is therefore not trivial and requires further investigation.

False Positive Rate

As shown in Table 5.2, when the extracted sub-system does not behave like a random system, the number of false positives is irregular and depends on what equations are in the sub-system. It is therefore not easy to analyze the number of false positives in this case. On the contrary, when the sub-system does behave semi-regularly, the ratio of false positives seems to be mathematically predictable. It has been observed in the experiments for this thesis that whenever the maximal value of k is increased by one, the false positive rate increases roughly twofold. As a rule of thumb, the following formula can be used to estimate the false positive rate based on the maximum of k

$$7.55 \cdot 10^{-6} \cdot 2^{(k_{\max}-15)}.$$

Nevertheless, the exact formula requires a more thorough analysis.

Maximal Number of Variables that Can be Kept

As discussed above, the parameter k should be chosen as high as possible. Therefore, one would be interested in the maximal value of k. As described in the original Crossbred algorithm [40], the maximum of k depends on the degree of the Macaulay matrix D as well as the number of variables n and the number of equations m in the $\mathcal{MQ}$ system, In particular, the number of linearly independent equations that can be extracted from the Macaulay matrix, which can be computed as the difference of the number of independent rows $N_{\text{indep\_row}}$ in the Macaulay matrix and the number of non-linear$_k$ monomials N_{nl} (see Section 3.1.5), must be no less than k. One can therefore compute the maximal value of k as follows

$$\begin{aligned}
N_{\text{indep\_row}} &= \begin{cases} m \cdot (n+1), \text{ when } D = 3 \\ m \cdot (\binom{n}{2} + n + 1) - (\binom{m}{2} + m), \text{ when } D = 4 \end{cases} \\
N_{\text{nl}} &= \sum_{i=2}^{D} \sum_{j=2}^{i} \binom{k}{j} \cdot \binom{n-k}{i-j} \\
N_{\text{indep\_row}} &- N_{\text{nl}} \geq k
\end{aligned}$$

With this formula, one can compute the maximal values of k for $\mathcal{MQ}$ systems where $n = m$ and $m = 2n$, based on Macaulay degree $D = 3$ and 4 respectively. The results for $n = 1 \sim 200$ are shown in Figure 5.1.

Clearly, with degree-4 Macaulay matrices, one can keep more variables than with degree-3 Macaulay matrices for most of the overdetermined and determined systems. However, for some determined systems, e.g. $n = 140$, using a degree-4 Macaulay matrix does not allow one to keep more variables than using a degree-3 matrix. In addition, one can also observe that the gap between the two curves for overdetermined systems also becomes narrower as n grows. Therefore, similar to the degree-3 Macaulay matrices, the effectiveness of degree-4 matrices is expected to become marginal, at which point degree-5 Macaulay matrices would be necessary if one wishes to keep considerably more variables than using degree-3 matrices.

Note that k grows linearly in the beginning of each curve, where the degree of regularity (see Section 3.1.1) of the $\mathcal{MQ}$ system is smaller than or equal to the Macaulay degree. In this case, a Gröbner basis can be extracted directly from the Macaulay matrix, which immediately yields a solution to the system. Therefore, all the variables can be kept.

5.1.3 Number of Variables to Fix during $\mathcal{MQ}$ Preprocessing

In Section 5.1.2 it is shown that one should choose the parameter k as high as possible and the formula for computing the maximal value of k is also provided. Since the parameter n in the formula is the number of variables in the $\mathcal{MQ}$ system, one might be tempted to fix some p variables with $\mathcal{MQ}$ preprocessing. In this manner, the number of variables in the system drops

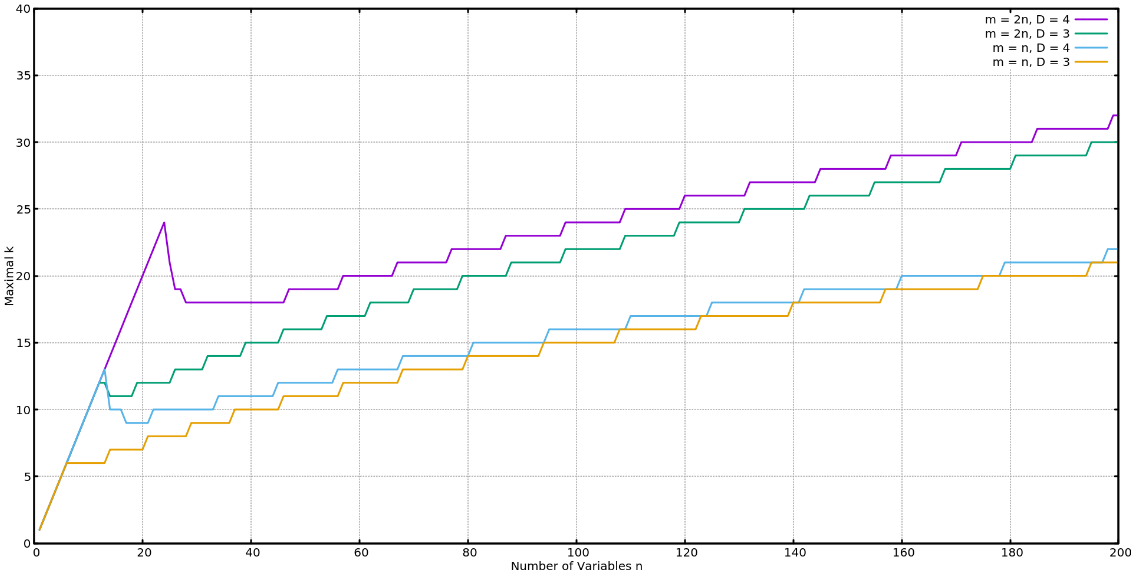


Figure 5.1: Maximal Number of Variables that Can be Kept

by p but the number of equations remains the same. Therefore, the number of variables that can be kept for this smaller $\mathcal{MQ}$ system may be higher. Obviously, this technique is more effective for an $\mathcal{MQ}$ system which has more equations.

Indeed, as shown in Figure 5.2, by fixing four variables, an $\mathcal{MQ}$ system of 148 equations in 74 variables becomes a system in 70 variables, with which one can keep one more variable (now 22 variables) by computing a degree-4 Macaulay matrix. In this manner the search space of Gray code enumeration is split into $2^4 \times 2^{74-4-22}$ instead of $1 \times 2^{74-21}$, which reduces the number of iterations for Gray code enumeration by half.

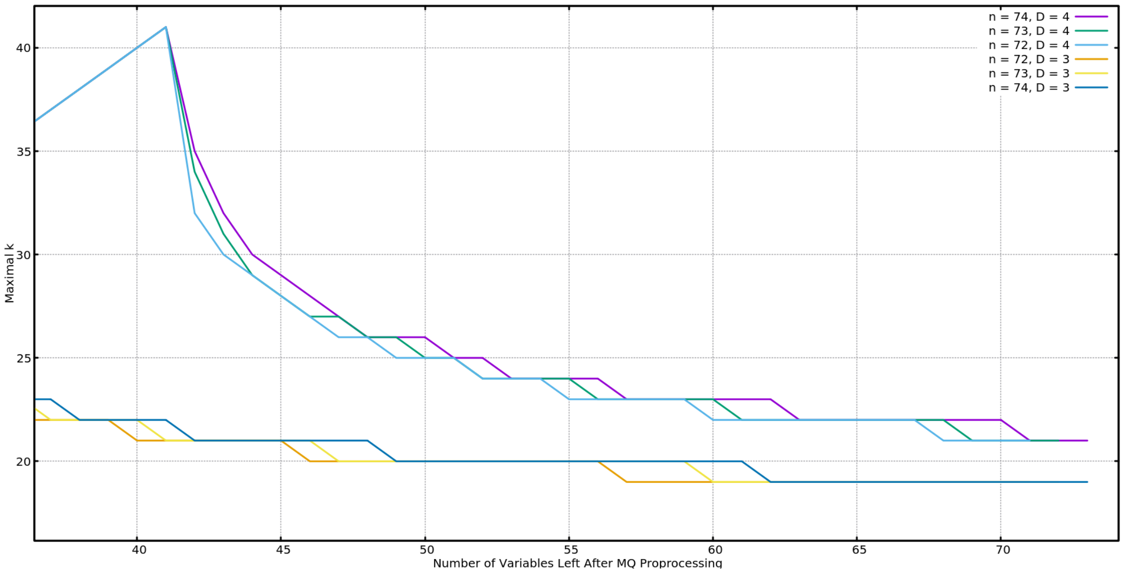


Figure 5.2: Effectiveness of $\mathcal{MQ}$ Preprocessing for $\mathcal{MQ}$ Systems where $m = 2n$

Note that in practice, one might have to extract a sub-system and subsequently launch the GPU kernel for Gray code enumeration at most 2^p times. The extra overhead for computing sub-systems and launching GPU kernels should therefore be taken into consideration. One only benefits from $\mathcal{MQ}$ preprocessing when the computation time that can be saved is more than the extra overhead.

Also note that for degree-3 Macaulay matrix, $\mathcal{MQ}$ preprocessing can be quite ineffective. As shown in Figure 5.2, for an $\mathcal{MQ}$ system where $n = 74, m = 2n$, more than 10 variables have to be fixed in order to increase the maximum of k by one. Therefore this technique is not recommended for degree-3 Macaulay matrices.

5.1.4 Number of Variables to Fix before Exhaustive Search

As mentioned in Section 3.2.4, the sole purpose of fixing variables before entering the exhaustive search stage is to reduce the amount of resources required. Therefore, one should decide other parameters first and adjust this parameter accordingly.

5.1.5 Macaulay Degree

As discussed in the original Crossbred algorithm [40], since the Macaulay matrix is used to induce cancellation of the monomials where any of the variables $x_1, x_2, \ldots, x_k$ has a degree larger than one, the degree of the Macaulay matrix must be no less than the degree of regularity of a random system of m equations in k variables. In addition, the Macaulay degree is a key factor that determines the maximal value of k, as shown in Section 5.1.2. One should therefore choose a Macaulay degree that is larger than the degree of regularity requirement and can provide sufficient linearly independent equations for the intended value of k.

One caveat of choosing the Macaulay degree is that the memory requirement must be smaller than the amount of memory available. Since both the number of rows and columns of a Macaulay matrix grow considerably when the degree increases, one might have to choose a smaller Macaulay degree and subsequently a smaller k in case the amount of memory is insufficient.

5.2 Chance of Failure

The proposed algorithm might fail to yield a solution to the input $\mathcal{MQ}$ system in two cases

1. When the number of linearly independent equations extracted from the Macaulay matrix is less than k.
2. When the number of linearly independent equations after fixing variables in the sub-system becomes less than k.

When the first case occurs, each resulting linear system appeared in Gray code enumeration would be an underdetermined linear system, which guarantees the occurrence of the second case. In the current implementation, MQsolver only extracts 64 sub-system candidates from which a sub-system is subsequently extracted (see Section 4.1). 64 sub-system candidates turned out to be sufficient since during all the experiments MQsolver did not fail to extract enough equations for the sub-system. However, for a tighter upperbound of the number of sub-system candidates needed, a more through analysis is required.

As for the second case, when solving an underdetermined linear system with the algorithms proposed in Section 3.2.2, only one single solution can be extracted, which might not be the correct one to the input $\mathcal{MQ}$ system. Consequently, Parallel Crossbred might fail to yield a solution. However, as discussed in Section 5.1.2, as long as the extracted sub-system behaves like a random system, the chance of obtaining underdetermined linear systems is too small to be observable. In all the experiments that have been conducted for this thesis where the sub-system behaved as desired, the second case never happened and MQsolver did not fail to yield a solution. Note that one method to avoid this potential failure is to check all the solutions to an underdetermined linear system with, for example, the Fast Exhaustive Search algorithm.

Also note that the original Crossbred algorithm restarts from the beginning should the first case occurs. As for the second case, the Crossbred algorithm does not mention how to tackle it.

5.3 Environment

To evaluate the performance of MQsolver, it is deployed onto the *Saber clusters* [16]. Saber is located at Technische Universiteit Eindhoven and Saber2 at University of Illinois at Chicago. Those clusters are mostly homogeneous and consist of hundreds of personal computers. Out of all the nodes in these two clusters, 27 nodes, each of which has two Nvidia graphics cards, were chosen for the experiments. 12 out of those 27 nodes have two GTX 780 graphics cards while the remaining 15 nodes have two GTX 980 cards. Each node has 32GB DDR3 ECC memory at 1600MHz (Kingston KVR16E11K4/32) and one AMD FX-8350 4GHz processor [1], which has 4 physical CPU *modules* (similar to a physical core in an Intel CPU) shared by 8 logical threads with *Cluster MultiThreading* (see Section 2.2.1), 16KB L1 data cache per thread, 2MB L2 cache per module, and 8MB L3 cache shared by the whole CPU. All nodes in the clusters have CUDA version 7.5 installed except for one, which has version 8.0. MQsolver is compiled with the back-end compiler bundled with CUDA, which is GCC version 4.8 for CUDA 7.5 and GCC version 5.3 for the newer version.

5.4 Fukuoka $\mathcal{MQ}$ Challenge

Fukuoka $\mathcal{MQ}$ challenges [60] are used as the test benches. They were created in 2015 in order to help determine appropriate parameters for public key cryptographic schemes based on $\mathcal{MQ}$ systems. In particular, type I challenges generated with seed 4 were chosen because they consist of $\mathcal{MQ}$ systems of m equations in n variables, where $m = 2n$, over $\mathbb{F}_2$. The challenges were generated in a manner that guarantees their solvability without revealing the solution(s) to the creators.

Previous records of solving type I challenges were held by the Fast Exhaustive Search and the Crossbred algorithms, which were discussed in Section 3.1.3 and Section 3.1.5, respectively. With 128 Spartan 6 FPGAs, the Fast Exhaustive Search algorithm is able to perform a full enumeration over the search space for an $\mathcal{MQ}$ system of 134 equations in 67 variables in $\frac{2^{(67-10)}}{200\text{MHz}\cdot 128\cdot 86400} = 65$ days [19]. On the other hand, the original Crossbred algorithm can enumerate the same search space in one day with a heterogeneous computer cluster consisting of 256 CPU cores [5]. To solve an $\mathcal{MQ}$ system of 148 equations in 74 variables, the Fast Exhaustive Search algorithm however requires 2924 FPGA-years while the Crossbred algorithm needs 41 CPU-years [40].

The experimental results of solving type I challenges, $n = 55 \sim 67$, with *one single* GTX 980 graphics card on a node in the Saber2 cluster are given in Table 5.3, where time is measured in seconds and all Macaulay matrices are of degree $D = 4$. The workflow of the algorithm, i.e. how the search space is split and enumerated, is listed as the 3rd column in the table. Since given parameters p and b (see Algorithm 9), $\mathcal{MQ}$ preprocessing and ParallelGrayCodeEnumerate might have to be repeated at most 2^p and 2^b times respectively, the numbers inside the parentheses in the 4th and 5th columns represent how many times they were repeated during the experiments. For all these small experiments the GPU instead of CPU is used to extract sub-systems except for the last two experiments marked with an asterisk, where the reduced Macaulay matrix becomes too large to fit into the 4GB off-chip memory of GTX 980. As shown in Table 5.3, solving an $\mathcal{MQ}$ system of 134 equations in 67 variables requires at most 354231.11 GPU-seconds = 98.39 GPU-hours, including the computation time of extracting sub-systems.

As for larger type I challenges, $n = 68 \sim 74$, the workload is evenly shared by the aforementioned 27 computers in the Saber and Saber 2 clusters by distributing the 2^p smaller $\mathcal{MQ}$ systems obtained from $\mathcal{MQ}$ preprocessing. The results are given in Table 5.4, which basically has the same format and notation as Table 5.3. In these larger experiments, sub-systems were extracted from degree-4 Macaulay matrices with the CPU and pipelined to the GPU (see Section 4.5) because the GPU off-chip memory cannot accommodate the reduced Macaulay matrix anymore. Nevertheless, the extraction of sub-system and the exhaustive search stage were pipelined therefore the computation time of the former can be completely hidden except for the first run. Also note that since GTX 780 only has 3GB of off-chip memory while GTX 980 has 4GB, the parameter t and b have to be

Number of Variables n	Parameters (D, p, k, b, t)	Size of Search Space to Enumerate $2^p \times 2^b \times 2^{n-p-k-b}$	Runtime for Extracting Sub-systems	Runtime for Exhaustive Search	Total Runtime	Maximal Runtime
55	(4, 0, 19, 0, 14)	$1 \times 1 \times 2^{36}$	387.80	318.25	706.20	706.20
56	(4, 1, 20, 0, 14)	$2^1 \times 1 \times 2^{35}$	491.60 (1)	169.94 (1)	658.67	1317.34
57	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{37}$	606.75	650.90	1258.73	1258.73
58	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{38}$	670.26	1311.97	1982.74	1982.74
59	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{39}$	741.62	2619.00	3361.77	3361.77
60	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{40}$	782.12	5211.05	5994.41	5994.41
61	(4, 0, 20, 1, 14)	$1 \times 2^1 \times 2^{40}$	872.34	5204.18 (1)	6077.13	11280.34
62	(4, 0, 20, 2, 14)	$1 \times 2^2 \times 2^{40}$	920.24	10485.95 (2)	11407.64	21892.14
63	(4, 4, 21, 0, 14)	$2^4 \times 1 \times 2^{38}$	9406.21 (11)	14827.94 (11)	24234.15	35250.72
64	(4, 3, 21, 1, 13)	$2^3 \times 2^1 \times 2^{39}$	1991.48 (2)	10469.58 (4)	12456.97	49844.24
65	(4, 3, 21, 2, 14)	$2^3 \times 2^2 \times 2^{39}$	1046.62 (1)	10517.21 (4)	11565.10	92510.64
66*	(4, 1, 21, 5, 13)	$2^1 \times 2^5 \times 2^{39}$	16268.10 (2)	133896.93 (51)	151867.70	184295.62
67*	(4, 0, 21, 7, 13)	$1 \times 2^7 \times 2^{39}$	10298.95	198835.78 (74)	209172.34	354231.11

Table 5.3: Solving Overdetermined Systems with a Single GTX 980 Graphics Card

adjusted according to GTX 780 hence in some experiments the off-chip memory of the GTX 980 graphics cards was not fully utilized.

Number of Variables n	Parameters (D, p, k, b, t)	Size of Search Space to Enumerate $2^p \times 2^b \times 2^{n-p-k-b}$	Runtime for Extracting Sub-systems	Total Runtime	Maximal Runtime (GPU-hours)
68	(4, 6, 21, 2, 13)	$2^6 \times 2^2 \times 2^{39}$	9799.15	12802.11	214.45
69	(4, 8, 22, 0, 13)	$2^8 \times 1 \times 2^{39}$	11238.49	56697.70	229.10
70	(4, 7, 22, 2, 13)	$2^7 \times 2^2 \times 2^{39}$	14367.71	44223.81	452.65
71	(4, 8, 22, 2, 13)	$2^8 \times 2^2 \times 2^{39}$	14392.00	87415.91	947.20
72	(4, 9, 22, 2, 13)	$2^9 \times 2^2 \times 2^{39}$	13912.39	144145.58	1867.44
73	(4, 8, 22, 4, 13)	$2^8 \times 2^4 \times 2^{39}$	18055.07	159585.32	3700.87
74	(4, 10, 22, 3, 13)	$2^{10} \times 2^3 \times 2^{39}$	15163.72	118323.38	8236.05

Table 5.4: Solving Overdetermined Systems with the Saber Cluster

As shown in Table 5.4, solving the largest Fukuoka type I $\mathcal{MQ}$ challenge where $n = 74$ with MQsolver takes at most 8236.05 GPU-hours, which is much less than the original Crossbred algorithm. In addition, based on the experimental result, one can estimate the security strength of this particular $\mathcal{MQ}$ system, which is the number of operations required to solve the system [29], as follows

$$\begin{aligned} \text{Num}_{\text{operations}} &= 8236.05 \cdot 2048 \cdot 10^9 \cdot 3600 \cdot 0.3706 \\ &= 22503850942464000000.00 \\ &\approx 2^{64.28} \end{aligned}$$

since a GTX 980 graphics card consists of 2048 CUDA cores operating at 1GHz, and the current implementation of MQsolver achieves only 37.06% GPU utilization (see Section 4.6). Therefore, an $\mathcal{MQ}$ system where $n = 74, m = 2n$, only provides 64.28 bits of security [29].

In order to provide 80 bits of security with an overdetermined $\mathcal{MQ}$ system where $m = 2n$, one must set $n = 92$ since when n increases by one the number of operations required to solve the $\mathcal{MQ}$ system grows by a factor of two. Nevertheless, the computation time required for solving an $\mathcal{MQ}$ system of 184 equations in 92 variables is only

$$\begin{aligned} 8236.05 \cdot 2^{(92-74-2)} &= 539757772.80 \text{ (GPU-hours)} \\ &= 61616.18 \text{ (GPU-years)}. \end{aligned}$$

since with $\mathcal{MQ}$ preprocessing $p = 8$, the maximal value of the parameter k for the system is 24 (see Figure 5.1). As the result shows, solving such $\mathcal{MQ}$ systems is feasible for an adversary supported by a nation or by a multinational conglomerate. Therefore one should not consider an overdetermined $\mathcal{MQ}$ system which has a security strength of 80 bits secure anymore.

One can also observe that whenever the maximal value of the parameter k increases, either with or without $\mathcal{MQ}$ preprocessing, the runtime almost stays the same despite the number of variables increasing by one. For example, for two overdetermined $\mathcal{MQ}$ system $\mathcal{F}_1, n = 68$ and $\mathcal{F}_2, n = 69$, by keeping 21 variables, there are 47 variables left in $\mathcal{F}_1$ to enumerate. On the other hand, for $\mathcal{F}_2$ the maximal value of k can be increased by one with $\mathcal{MQ}$ preprocessing so 22 variables can be kept. Therefore, there are also 47 variables to enumerate for $\mathcal{F}_2$. Hence for both systems the total number of iterations that have to be performed during Gray code enumeration is the same, no matter what the remaining parameters are. However, since for the $\mathcal{F}_2$ linear systems in 22 variables instead of 21 have to be computed, the overhead of each iteration of Gray code enumeration for $\mathcal{F}_2$ is slightly larger than $\mathcal{F}_1$. Hence, as shown in Table 5.4, the maximal runtime for $n = 69$ is slightly larger than $n = 68$.

As explained in Algorithm 3, extracting a sub-system from a Macaulay matrix consists of four major steps:

(I). Compute and permute the Macaulay matrix Mac_D^k.

(II). Perform row-swapping on the Macaulay matrix Mac_D^k.

(III). Reduce the dimension of the Macaulay matrix Mac_D^k.

(IV). Perform Gaussian elimination on the reduced Macaulay matrix $\mathcal{RM}$.

The computation time of these four steps in the experiments listed in Table 5.3 and Table 5.4 is given in Table 5.5 while the statistics of the original and the reduced Macaulay matrices are given in Table 5.6. Similar to Table 5.3, experiments marked with an asterisk utilized the CPU instead of GPU to perform Gaussian elimination on the reduced Macaulay matrix and the computation time is measured in seconds.

Number of Variables n	Parameters $(p, k, n-p)$	Runtime of step I	Runtime of step II	Runtime of step III	Runtime of step IV
55	(0, 19, 55)	65.72	1.77	77.30	241.41
56	(1, 20, 55)	66.78	1.80	94.69	326.71
57	(0, 20, 57)	84.22	2.34	134.10	384.94
58	(0, 20, 58)	95.22	2.59	158.10	412.62
59	(0, 20, 59)	107.29	2.88	183.40	447.49
60	(0, 20, 60)	121.03	3.17	173.09	483.68
61	(0, 20, 61)	135.76	3.54	217.07	514.08
62	(0, 20, 62)	152.05	3.92	221.63	541.31
63	(4, 21, 59)	114.34	3.07	174.42	562.16
64	(3, 21, 61)	142.09	3.76	190.60	657.33
65	(3, 21, 62)	159.10	4.18	181.50	700.71
66*	(1, 21, 65)	214.90	5.56	397.88	7514.57
67*	(0, 21, 67)	262.15	6.73	336.09	9692.84
68*	(6, 21, 62)	171.79	4.73	427.14	9146.14
69*	(8, 22, 61)	157.96	4.61	438.51	10565.94
70*	(7, 22, 63)	194.43	5.48	518.99	13629.33
71*	(8, 22, 63)	197.37	5.51	537.72	13567.85
72*	(9, 22, 63)	200.01	5.64	488.54	13095.87
73*	(8, 22, 65)	244.72	6.87	702.67	16940.85
74*	(10, 22, 64)	226.16	6.10	561.33	14271.25

Table 5.5: Runtime of Different Steps for Extracting a Sub-system

As shown in Table 5.5, most of the computation time for extracting a sub-system is spent on Gaussian elimination. Row-swapping can be implemented by simply modifying the pointers which

Number of Variables n	Ratio of Missing Pivot Elements (%)	Dimension of Mac_D^k	Dimension of $\mathcal{RM}$	Size of Mac_D^k (MB)	Size of $\mathcal{RM}$ (MB)
55	28.38	169510×368831	43720×258701	999.00	1348.90
56	32.62	172592×368831	54147×257139	1017.00	1660.28
57	32.05	188556×425924	58381×302316	1192.00	2104.57
58	31.28	198592×456838	59585×326074	1299.00	2316.62
59	30.49	208978×489406	60693×351200	1415.00	2541.68
60	29.69	219720×523686	61699×377746	1538.00	2779.16
61	28.90	230824×559737	62642×405810	1669.00	3030.97
62	28.14	242296×597619	63565×435495	1809.00	3300.65
63	31.89	223146×489406	68214×343881	1510.00	2797.31
64	31.38	242176×559737	73161×399909	1751.00	3488.58
65	30.68	254020×597619	74589×429279	1897.00	3817.88
66	30.68	283272×722866	84122×532979	2323.28	5345.55
67	29.64	305386×816664	87735×608620	2659.00	6366.32
68	28.13	265744×597619	68374×423064	1984.89	3449.16
69	32.30	261096×559737	80601×390948	1888.42	3757.26
70	31.53	282380×637393	85545×451757	2177.01	4607.76
71	30.76	286414×637393	83473×449685	2208.11	4475.77
72	30.00	290448×637393	81401×447613	2239.21	4344.18
73	29.53	313316×722866	86834×515804	2569.69	5340.34
74	28.62	307988×679121	80875×477573	2449.63	4605.49

Table 5.6: Effectiveness of Reducing the Dimension of Macaulay Matrices

store the addresses of the rows. In this manner, it can be done very efficiently, which takes a few seconds to complete. In addition, Table 5.6 shows that in practice, Algorithm 3 can reduce the size of the Macaulay matrix considerably because only $\approx 30\%$ of the pivot elements are missing when the maximum of the parameter k is chosen.

5.5 Determined Systems

To evaluate the difficulty of solving determined systems, which are not included in the Fukuoka $\mathcal{MQ}$ challenges, random solvable determined system where $n = m$ are generated and subsequently solved with one single GTX 980 graphics card on a node in the Saber2 cluster. The experimental results are given in Table 5.7, whose format and notation is the same as Table 5.3. All Macaulay matrices are of degree $D = 4$.

Number of Variables n	Parameters (D, p, k, b, t)	Size of Search Space to Enumerate $2^p \times 2^b \times 2^{n-p-k-b}$	Runtime for Extracting Sub-systems	Runtime for Exhaustive Search	Total Runtime	Maximal Runtime
46	(4, 0, 12, 0, 16)	$1 \times 1 \times 2^{34}$	33.90	47.63	82.12	82.12
47	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{35}$	36.31	96.12	132.92	132.92
48	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{36}$	39.76	190.59	230.88	230.88
49	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{37}$	42.98	380.91	424.48	424.48
50	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{38}$	46.86	754.86	802.34	802.34
51	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{39}$	50.74	1542.07	1593.46	1593.46
52	(4, 0, 12, 0, 14)	$1 \times 1 \times 2^{40}$	53.59	3049.07	3103.21	3103.21
53	(4, 0, 12, 1, 14)	$1 \times 2^1 \times 2^{40}$	57.05	6249.61 (2)	6307.22	6307.22
54	(4, 0, 12, 2, 14)	$1 \times 2^2 \times 2^{40}$	60.86	3141.67 (1)	3205.11	12635.54
55	(4, 1, 13, 1, 14)	$2^1 \times 2^1 \times 2^{40}$	95.30 (1)	3322.54 (1)	3418.48	13480.76
56	(4, 0, 13, 3, 14)	$1 \times 2^3 \times 2^{40}$	118.85	6600.55 (2)	6720.12	26521.05
57	(4, 0, 13, 4, 14)	$1 \times 2^4 \times 2^{40}$	121.54	46053.43 (14)	46175.72	52754.03
58	(4, 0, 13, 5, 14)	$1 \times 2^5 \times 2^{40}$	133.97	105432.90 (32)	105567.66	105567.66
59	(4, 0, 13, 6, 14)	$1 \times 2^6 \times 2^{40}$	144.13	197303.32 (60)	197448.24	210601.00

Table 5.7: Solving Determined Systems with a Single GTX 980 Graphics Card

For determined systems, the number of variables that can be kept is much smaller due to that fact that fewer equations are available. However, the linear systems that are enumerated during Gray code enumeration consist of fewer variables therefore the cost of each iteration is also lower. As Table 5.7 shows, solving a determined $\mathcal{MQ}$ systems in n variables is roughly as difficult as solving an overdetermined $\mathcal{MQ}$ system where $m' = 2n', n' = n + 7 \sim n + 8$. Nevertheless, Figure 5.1 shows that the gap between the number of variables that can be kept for determined and overdetermined systems gradually becomes larger as n grows. Therefore, this observation only applies to the systems in Table 5.7 but not to larger determined systems, e.g. $n = 172$.

Table 5.7 also shows that with the current implementation of MQsolver and by applying $\mathcal{MQ}$ preprocessing with $p = 5$, solving the underlying $\mathcal{MQ}$ system of a cryptographic scheme [54] that has been proposed for post-quantum cryptography, where $n = 84, m = 80$, requires at most

$$2^5 \cdot \frac{210601.00 \cdot 2^{79-59-2}}{86400 \cdot 365} = 56020.07 \text{ (GPU-years)}$$

since for an $\mathcal{MQ}$ system $\mathcal{F}$ where $n = 79, m = 80$, the maximal value of k is 15 with degree-4 Macaulay matrices (see Section 5.1.2). Therefore the $\mathcal{MQ}$ system provides at most

$$\begin{aligned} 56020.07 \cdot 365 \cdot 24 \cdot 2048 \cdot 10^9 \cdot 3600 \cdot 0.3706 &= 1340866749519691776000000.00 \\ &\approx 2^{80.15} \end{aligned}$$

80.15 bits of security. However, the computation time is expected to be only

$$\begin{aligned} \frac{56020.07}{2^4} \cdot \left(\sum_{i=0}^{2^4-1} 0.37^i \cdot 0.63 + 0.37^{2^4} \right) &= 3501.25 \cdot \left(\frac{1 - 0.37^{16}}{1 - 0.37} \cdot 0.63 + 0.37^{16} \right) \\ &= 3501.25 \text{ (GPU-years)} \end{aligned}$$

since a solution will appear with probability ≈ 0.63 whenever the search spaces of two systems like $\mathcal{F}$ have been enumerated (see the beginning of Chapter 3). Hence the expected security strength is merely

$$\begin{aligned} 3501.25 \cdot 365 \cdot 24 \cdot 2048 \cdot 10^9 \cdot 3600 \cdot 0.3706 &= 83804067127296000000000.00 \\ &\approx 2^{76.15} \end{aligned}$$

76.15 bits, which is lower than claimed [54].

As the result shows, solving the underlying $\mathcal{MQ}$ systems of this particular cryptographic scheme is clearly feasible for adversaries that are supported by nations or multi-national conglomerates. It is therefore recommended not to use the proposed parameters for this cryptographic scheme [54]. In addition, together with the experimental result shown in Section 5.4, in general $\mathcal{MQ}$ systems that only provide 80 bits of security should not be considered secure anymore.

Bibliography

[1] AMD Black Edition AMD FX 8350 / 4 GHz processor Series Specs - CNET. URL: `https://www.cnet.com/products/amd-black-edition-amd-fx-8350-4-ghz-processor-series/specs/`.

[2] Bulldozer (microarchitecture). URL: `https://en.wikipedia.org/wiki/Bulldozer_(microarchitecture)#Bulldozer_core`.

[3] Cache hierarchy. URL: `https://en.wikipedia.org/wiki/Cache_hierarchy`.

[4] GPU-accelerated Libraries for Computing. URL: `https://developer.nvidia.com/gpu-accelerated-libraries`.

[5] Fukuoka MQ Challenge. URL: `https://www.mqchallenge.org`.

[6] Convert to and from Graycode. URL: `https://en.wikipedia.org/wiki/Gray_code#Converting_to_and_from_Gray_code`.

[7] Hyper-threading. URL: `https://en.wikipedia.org/wiki/Hyper-threading`.

[8] libFES. URL: `http://www.lifl.fr/~bouillag/fes/`.

[9] Texture Memory in CUDA. URL: `https://cuda-programming.blogspot.nl/2013/02/texture-memory-in-cuda-what-is-texture.html`.

[10] Ars G., Faugère JC., Imai H., Kawazoe M., Sugita M. Comparison Between XL and Gröbner Basis Algorithms. In Lee, P.J., editor, *Advances in Cryptology - ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2004.

[11] Bardet M., Faugère JC., Salvy B. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *International Conference on Polynomial System Solving*, pages 71–74, 2004.

[12] Bardet M., Faugère JC., Salvy B., Spaenlehauer PJ. On the complexity of solving quadratic Boolean systems. *Journal of Complexity*, 29:53–75, 2013.

[13] Bardet, M., Faugère, JC., Salvy, B., Yang BY. Asymptotic Expansion of the Degree of Regularity for Semi-Regular Systems of Equations. In P. Gianni, editor, *MEGA 2005*, 2005.

[14] Berbain C., Gilbert H., Patarin J. QUAD: A Practical Stream Cipher with Provable Security. In Vaudenay S., editor, *Advances in Cryptology - EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2006.

[15] Berlekamp, E. Factoring Polynomials Over Finite Fields. *Bell System Technical Journal*, pages 1853–1859, 1967.

[16] Bernstein D.J. The Saber Cluster. URL: `https://blog.cr.yp.to/20140602-saber.html`.

[17] Beullens W., Preneel B. Field lifting for smaller UOV public keys. *IACR Cryptology ePrint Archive*, 2017. URL: `https://eprint.iacr.org/2017/776.pdf`.

[18] Billet O., Robshaw M.J.B., Peyrin T. On Building Hash Functions from Multivariate Quadratic Equations. In Pieprzyk J., Ghodosi H., Dawson E. , editor, *Information Security and Privacy. ACISP 2007*, volume 4586 of *Lecture Notes in Computer Science*, pages 82–95. Springer, 2007.

[19] Bouillaguet C., Cheng CM., Chou T., Niederhagen R., Yang BY. Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs. In Lange T., Lauter K., Lisonk P., editor, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2014.

[20] Bouillaguet C. et al. Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$. In Mangard S., Standaert FX., editor, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[21] Buchberger B. *An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal.* PhD thesis, Universität Innsbruck, 1965.

[22] Chen MS., Hülsing A., Rijneveld J., Samardjiska S., Schwabe P. SOFIA: MQ-based signatures in the QROM. *IACR Cryptology ePrint Archive*, 2017. URL: `https://eprint.iacr.org/2017/680.pdf`.

[23] Cheng CM., Chou T., Niederhagen R., Yang BY. Solving Quadratic Equations with XL on Parallel Architectures. In Prouff E., Schaumont P., editor, *Cryptographic Hardware and Embedded Systems CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2012.

[24] Clough C., Baena J., Ding J., Yang BY., Chen M. Square, a New Multivariate Encryption Scheme. In Fischlin M., editor, *Topics in Cryptology CT-RSA 2009*, volume 5473 of *Lecture Notes in Computer Science*, pages 252–264. Springer, 2009.

[25] Courtois N., Goubin, L., Patarin, J. SFLASH, a fast asymmetric signature scheme for low-cost smartcards: Primitive specification and supporting documentation, 2003.

[26] Courtois N., Klimov A., Patarin J., Shamir A. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In Preneel B., editor, *Advances in Cryptology EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[27] Courtois N.T., Pieprzyk J. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In Zheng Y., editor, *Advances in Cryptology ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[28] Ding J., Schmidt D. Rainbow, a New Multivariable Polynomial Signature Scheme. In Ioannidis J., Keromytis A., Yung M., editor, *Applied Cryptography and Network Security. ACNS 2005*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2005.

[29] Elaine Barker. *Special Publication 800-57 Rev. 4 Recommendation for Key Management, Part 1: General.* NIST, 2016.

[30] Faugère JC. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139:61–88, 1999.

[31] Faugère JC. ”A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation ISSAC 2002*, pages 75–83. ACM Press, 2002.

[32] Faugère JC., Joux A. Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases. In Boneh D., editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2003.

[33] Faugère JC., Perre L. On the security of UOV. *IACR Cryptology ePrint Archive*, 2009. URL: `https://eprint.iacr.org/2009/483.pdf`.

[34] Fraenkel A., Yesha Y. Complexity of problems in games, graphs and algebraic equations. In *Discrete Applied Mathematics*, volume 1, pages 15–30, 1979.

[35] Fusco G., Bach E. Phase transition of multivariate polynomial systems. *Mathematical Structures in Computer Science*, 19:9–23, 2009.

[36] Galbraith S.D., Smart N.P. A Cryptographic Application of Weil Descent. In Walker M., editor, *Cryptography and Coding*, volume 1746 of *Lecture Notes in Computer Science*, pages 191–200. Springer, 1999.

[37] Giesbrecht M., Lobo A., Saunders B. D. Certifying inconsistency of sparse linear systems. In *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 113–119. ACM Press, 1998.

[38] Huang YJ., Hong WC., Cheng CM., Chen JM., Yang BY. A Memory Efficient Variant of an Implementation of the F_4 Algorithm for Computing Gröbner Bases. In Yung M., Zhu L., Yang Y., editor, *Trusted Systems*, volume 9473 of *Lecture Notes in Computer Science*, pages 374–393. Springer, Cham, 2016.

[39] Patarin J. Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms. In Maurer U., editor, *Advances in Cryptology EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 1996.

[40] Joux A., Vitse V. A crossbred algorithm for solving Boolean polynomial systems. *IACR Cryptology ePrint Archive*, 2017. URL: `https://eprint.iacr.org/2017/372.pdf`.

[41] Shamir A. Kipnis A. Cryptanalysis of the oil and vinegar signature scheme. In Krawczyk H., editor, *Advances in Cryptology CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 257–266. Springer, 1998.

[42] Kipnis A., Patarin J., Goubin L. Unbalanced Oil and Vinegar Signature Schemes. In Stern J., editor, *Advances in Cryptology EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999.

[43] Kipnis A., Shamir A. Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization. In Wiener M., editor, *Advances in Cryptology CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 1999.

[44] Lazard D. Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations. In van Hulzen J.A., editor, *Computer Algebra. EUROCAL 1983*, volume 162 of *Lecture Notes in Computer Science*, pages 146–156. Springer, 1983.

[45] Mei X., Chu X. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28:72–86, 2017.

[46] Montgomery P.L. A Block Lanczos Algorithm for Finding Dependencies over GF(2). In Guillou L.C., Quisquater JJ., editor, *Advances in Cryptology EUROCRYPT 1995*, volume 921 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 1995.

[47] Murphy S., Robshaw M.J. Essential Algebraic Structure within the AES. In Yung M., editor, *Advances in Cryptology CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[48] *NVIDIA Kepler GK110 Architecture Whitepaper.* Nvidia Corporation, 2012. URL: `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[49] *CUDA C Programming Guide*. Nvidia Corporation, January 2017. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[50] *CUDA C Best Practices Guide*. Nvidia Corporation, January 2017. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`.

[51] *CUDA Compiler Driver NVCC*. Nvidia Corporation, January 2017. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf`.

[52] Patarin J., Courtois N., Goubin L. QUARTZ, 128-Bit Long Digital Signatures. In Naccache D., editor, *Topics in Cryptology CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2001.

[53] Porras J., Baena J., Ding J. ZHFE, a New Multivariate Public Key Encryption Scheme. In Mosca M., editor, *Post-Quantum Cryptography. PQCrypto 2014*, volume 8772 of *Lecture Notes in Computer Science*, pages 229–245. Springer, Cham, 2014.

[54] Sakumoto K., Shirai T., Hiwatari H. Public-Key Identification Schemes Based on Multivariate Quadratic Polynomials. In Rogaway P., editor, *Advances in Cryptology CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 706–723. Springer, 2011.

[55] Szepieniec A., Ding J., Preneel B. Extension Field Cancellation: A New Central Trapdoor for Multivariate Quadratic Systems. In Takagi T., editor, *Post-Quantum Cryptography*, volume 9606 of *Lecture Notes in Computer Science*, pages 182–196. Springer, Cham, 2016.

[56] Thomae E., Wolf C. Solving Underdetermined Systems of Multivariate Quadratic Equations Revisited. In Fischlin M., Buchmann J., Manulis M., editor, *Public Key Cryptography PKC 2012*, volume 7293 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2012.

[57] Yang BY., Chen JM. All in the XL Family: Theory and Practice. In Park C., Chee S., editor, *Information Security and Cryptology ICISC 2004*, volume 3506 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2005.

[58] Yang BY., Chen JM., Courtois N.T. On Asymptotic Security Estimates in XL and Grbner Bases-Related Algebraic Cryptanalysis. In Lopez J., Qing S., Okamoto E., editor, *Information and Communications Security. ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2004.

[59] Yang BY., Chen O.CH., Bernstein D.J., Chen JM. Analysis of QUAD. In Biryukov A., editor, *Fast Software Encryption. FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 290–308. Springer, 2007.

[60] Yasuda T., Dahan X., Huang, YJ., Takagi T., Sakurai K. A Multivariate Quadratic Challenge Toward Post-quantum Generation Cryptography. *ACM Commun. Comput. Algebra*, 49(3): 105–107, 2015.

Appendix A

Appendix

A.1 Pseudocode for Multivariate Division

Algorithm 10 Multivariate Division

```
procedure MultiDiv
    Input:
        $f_1, f_2, \cdots, f_m, f$

    Initialize:
        $q_1 \leftarrow 0, q_2 \leftarrow 0, \cdots q_m \leftarrow 0$
        $r \leftarrow 0$
        $p \leftarrow f$

    while $p \neq 0$ do
        $i \leftarrow 1$
        div_occur ← false
        while $i \leq m$ and div_occur = false do
            if LT($f_i$) divides LT($p$) then
                $q_i \leftarrow q_i + \frac{\mathrm{LT}(p)}{\mathrm{LT}(f_i)}$
                $p \leftarrow p - \frac{\mathrm{LT}(p)}{\mathrm{LT}(f_i)} \cdot f_i$
                div_occur ← true
            else
                $i \leftarrow i + 1$
            end if
        end while
        if div_occur = false then
            $r \leftarrow r + \mathrm{LT}(p)$
            $p \leftarrow p - \mathrm{LT}(p)$
        end if
    end while

    Output $q_1, q_2, \cdots, q_m, r$
end procedure
```