From: (b) (6)
To: Liu, Yi-Kai (Fed)
Subject: Review Request
Date: Tuesday, November 28, 2017 3:46:13 PM
Attachments: 0049.pdf

Hi, Yi-Kai,

Would you be available to review the following attached paper? "Asymptotically faster quantum algorithms to solve multivariate quadratic equations"

It looks pretty interesting: it combines XL with Grover search. I will also pay close attention to this paper, but I trust your judgement more than my own as to the correctness and significance.

If you were already assigned this paper, or if you have too much to do, I'll probably ask Stephen so that I can be confident in the analysis.

The deadline for reviews is Dec. 17, and I'm intending to add some of my own comments on the subreviewers comments if it is necessary... would you be able to do it by Dec. 15th?

Either way, thanks. Feel free to ask me for a return favor if you need help in the review process.

Cheers,
Daniel

Asymptotically faster quantum algorithms to solve multivariate quadratic equations

Daniel J. Bernstein[1] and Bo-Yin Yang[2]

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Institute of Information Science
Academia Sinica, 128 Section 2 Academia Road, Taipei 115-29, Taiwan
by@crypto.tw

Abstract. This paper designs and analyzes a quantum algorithm to solve a system of m quadratic equations in n variables over a finite field $\mathbf{F}_q$. In the case $m = n$ and $q = 2$, under standard assumptions, the algorithm takes time $2^{(t+o(1))n}$ on a mesh-connected computer of area $2^{(a+o(1))n}$, where $t \approx 0.45743$ and $a \approx 0.01467$. The area-time product has asymptotic exponent $t + a \approx 0.47210$.

For comparison, the area-time product of Grover's algorithm has asymptotic exponent 0.50000. Parallelizing Grover's algorithm to reach asymptotic time exponent 0.45743 requires asymptotic area exponent 0.08514, much larger than 0.01467.

Keywords: FXL, Grover, reversibility, Bennett–Tompa, parallelization, asymptotics

1 Introduction

By definition, a NAND gate reads two bits $a, b \in \mathbf{F}_2$ as input and produces a bit $c = 1 - ab$ as output. It is well known that any function from ℓ-bit strings to ℓ'-bit strings, for any ℓ and any ℓ', can be viewed as being computed by a circuit built from NANDs.

For example, one can compute the 2-bit-to-2-bit function $(a, b) \mapsto (ab, a + b)$ by computing $c = 1 - ab$, $d = 1 - ac$, $e = 1 - bc$, $f = 1 - cc$, $g = 1 - de$;

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported by the European Commission under Contract ICT-645622 PQCRYPTO; by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005; and by the U.S. National Science Foundation under grant 1314919. This work also was supported by Taiwan Ministry of Science and Technology (MoST) grant 105-2923-E-001-003-MY3 and an Academia Sinica Investigator Award. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: c77423932ceeda61ddf009049efc0749daadd023. Date: 2017.11.25.

note that $f = ab$ and $g = a + b$. By further composition one can build, e.g., an integer-addition circuit producing a 32-bit output from two 32-bit inputs; a circuit computing a 256-bit SHA-256 output from a fixed-length input; and a circuit computing a 2048-bit RSA public key as a product of two secret 1024-bit primes.

Circuits built from NANDs are one of the standard models of computation. They are also, fundamentally, how computation is carried out today.[3] The problem of inverting one of today's computations—for example, finding a preimage for a hash output, or finding a secret key given a public key, or finding a plaintext given a public key and a ciphertext—can thus be viewed as the problem of solving a system of multivariate quadratic ("MQ") equations. **Quadratic** here means "degree at most 2", so quadratic equations include linear equations.

Specifically, each NAND gate can be expressed as a quadratic equation in at most three variables, such as the equation $c = 1 - ab$ in variables a, b, c, or the equation $f = 1 - cc$ in two variables c, f. Note that the second equation can be simplified to the linear equation $f = 1 - c$, using the fact that $c^2 = c$. Each known output bit for the computation can be expressed as a linear equation such as $g = 0$.

1.1. Random systems of MQ equations. Formally, a quadratic equation $\sum_{j \geq k} \alpha_{j,k} x_j x_k = \beta$ in n variables $x_1, x_2, \ldots, x_n$ over $\mathbf{F}_2$ is specified by a sequence of $n(n+1)/2+1$ coefficients $\alpha_{1,1}, \alpha_{2,1}, \alpha_{2,2}, \ldots, \alpha_{n,n}, \beta$. A system of m quadratic equations in n variables is thus specified by $m(n(n+1)/2+1)$ coefficients. The equation-solving problem is to determine, given these coefficients, whether there exists a solution $(x_1, x_2, \ldots, x_n) \in \mathbf{F}_2^n$ to all m equations, and if so to find *some* solution. Note that a reliable method to determine existence of a solution can be used recursively to find a solution.

There is a vast literature on fast equation-solving techniques that rely on special structure of the coefficients. For example, systems of linear equations ($\alpha_{j,k} = 0$ if $j \neq k$) are easy to solve. More generally, any nonzero linear equation can be eliminated, along with one of the variables used in the equation, producing a system of $m - 1$ quadratic equations in $n - 1$ variables. As another example, some systems have a "triangular" structure that makes them easy to solve: one can first solve for one variable without regard to the rest, then solve for another variable, etc. As yet another example, the problem of factoring a 256-bit integer into two 128-bit factors has a tremendous amount of mathematical structure, and this structure is exploited by factorization algorithms that run in mere minutes on a laptop today. But none of these examples have a noticeable chance of applying to a uniform random system with $m = n = 256$: a system of 256

[3] One can object to the circuit model of computation as being too restrictive: (1) in the algorithms literature it is common to treat random access to an arbitrarily large array as a single operation taking a single unit of "time"; (2) the algorithms literature also allows "branches". However, (1) for any particular size of array, random access can be implemented as a series of NANDs—which is essentially how physical RAM devices work; (2) branches are equivalent to—and physically implemented as—random access to an array of instructions.

equations in 256 variables in which each coefficient is chosen uniformly and independently at random from $\mathbf{F}_2$.

What is the fastest way to attack a uniform random system with large m and large n? This question has an important application in post-quantum cryptography: solving such a system, in particular with m slightly smaller than n, conjecturally breaks a typical MQ signature system such as Patarin's classic "HFE$^{\mathrm{v}-}$". Specifically, in these systems, the public key is a list of coefficients $\alpha_{i,j,k}$, conjectured to be difficult to distinguish from uniform random; the hash of a message is a list of coefficients β_i, which in the "random-oracle model" are uniform random by definition; and a solution to the corresponding system of equations is a signature on that message.

One of the central reasons for interest in these signature systems is that they allow very short signatures: it seems that a secure post-quantum MQ signature can be even shorter than a pre-quantum ECC signature. But the security evaluation here relies critically on the difficulty of solving a uniform random system of MQ equations.

There are also various proposals for MQ encryption systems where security analysis relies on a slight variant of the same question: m is taken somewhat larger than n, and one wants to know the fastest way to attack a uniform random *solvable* system. Algorithms designed to solve random systems of MQ equations have also had some applications beyond MQ cryptography, as illustrated by the attack in [8] against some small-key code-based encryption systems.

1.2. Performance of various algorithms for random systems. We consider asymptotic attack cost as $m \to \infty$ and $n \to \infty$ with an essentially constant ratio m/n. Specifically, let μ be a real number with $\mu \geq 1$, and assume that m is a function of n satisfying $m/n \in \mu + o(1)$ as $n \to \infty$.

All of the algorithmic issues that we analyze are visible for the frequently used case $\mu = 1$, and specifically $m = n$; the reader should feel free to focus on this case. Standard HFE$^{\mathrm{v}-}$ parameters actually take m slightly smaller than n but still have $m/n \in 1 + o(1)$ as $n \to \infty$. Beware, however, that FXL and GroverXL for $\mu = 1$ use XL for $\mu > 1$.

Brute-force search uses at most $N^{1+o(1)}$ operations where $N = 2^n$: there are N possibilities for $(x_1, x_2, \ldots, x_n)$, and checking one possibility uses $N^{o(1)}$ operations. For $m < n$ (the "underdetermined" case) one can reasonably expect a solution to appear within just 2^m possibilities, but the assumption $\mu \geq 1$ means that 2^m does not beat $N^{1+o(1)}$.

Brute-force search is asymptotically beaten by Gröbner-basis techniques. In particular:

- "Extended linearization" (XL) uses just $N^{0.87280\ldots+o(1)}$ operations in the case $\mu = 1$, under plausible assumptions that have been checked in various experiments.
- Even better, combining brute-force search with XL produces "fixing followed by extended linearization" (FXL), which uses just $N^{0.79106\ldots+o(1)}$ operations in the case $\mu = 1$ under the same assumptions.

The exponents $0.87280\ldots$ and $0.79106\ldots$ here, modulo a calculation error (0.785 instead of 0.79106), were published by Yang, Chen, and Courtois in 2004 [23]. See Section 2 for further history and an explanation of how XL works.

Brute-force search is also asymptotically beaten by the recent Lokshtanov–Paturi–Tamaki–Williams–Yu algorithm [15], which uses at most $N^{0.8765+o(1)}$ operations. This algorithm is randomized but, for each input, is proven to produce the correct result with negligible chance of error. The exponent $0.8765+o(1)$ is above $0.79106\ldots+o(1)$, and worst-case provability is outside the scope of our paper. We do not know whether the ideas in [15] can save time in FXL.

1.3. Quantum algorithms for random systems. Quantum computers beat brute-force search in a different way: namely, Grover's algorithm uses $N^{0.5+o(1)}$ operations. These operations are serial, but simply running A parallel copies of Grover's algorithm reduces time by a factor $A^{1/2}$. For example, parallel Grover takes time $N^{0.46+o(1)}$ on a quantum computer of total area $N^{0.08+o(1)}$, or time $N^{0.35+o(1)}$ on a quantum computer of area $N^{0.3+o(1)}$.

The main question considered in this paper is whether Grover's method can be usefully combined with XL. We answer this question in the affirmative. Our main contributions are the design and analysis of an algorithm "GroverXL" that, under the same assumptions used to analyze FXL, has exponent below $0.5+o(1)$.

We analyze this algorithm first in a simplified operation-count metric, and then in realistic area and time metrics for a parallel two-dimensional mesh-connected architecture. For example, for $m = n$, GroverXL takes time $N^{t+o(1)}$ on a mesh-connected quantum computer of area $N^{a+o(1)}$, where the user can choose either of the following parameter sets (t, a):

$$(t, a, t+a, t+a/2) = (0.45742\ldots, 0.01467\ldots, 0.47210\ldots, 0.46476\ldots) \quad \text{or}$$
$$(t, a, t+a, t+a/2) = (0.44962\ldots, 0.02557\ldots, 0.47519\ldots, 0.46240\ldots).$$

The area-time product is $N^{t+a+o(1)}$, and parameter set 1 is designed to optimize this exponent $t+a$. GroverXL can be further parallelized, taking time $N^{t-p+o(1)}$ on a mesh-connected quantum computer of area $N^{a+2p+o(1)}$; parameter set 2 is designed to optimize this area-time tradeoff. For example, the time exponent drops to 0.35 with area exponent $0.22481\ldots$, whereas reaching time exponent 0.35 with parallel Grover needs area exponent 0.30000 as noted above.

We state our results more generally for systems of m quadratic equations in n variables over $\mathbf{F}_q$. The generalization from $\mathbf{F}_2$ to $\mathbf{F}_q$ appears in many MQ systems and in further applications. Of course, guessing elements of $\mathbf{F}_q$ becomes slower as q increases; for sufficiently large q, one should simply use XL.

2 XL and FXL

This section reviews the XL and FXL algorithms to solve m equations in n variables over a finite field $\mathbf{F}_q$. For simplicity we consider solely quadratic equations, although the ideas can easily be extended to cubic systems and higher.

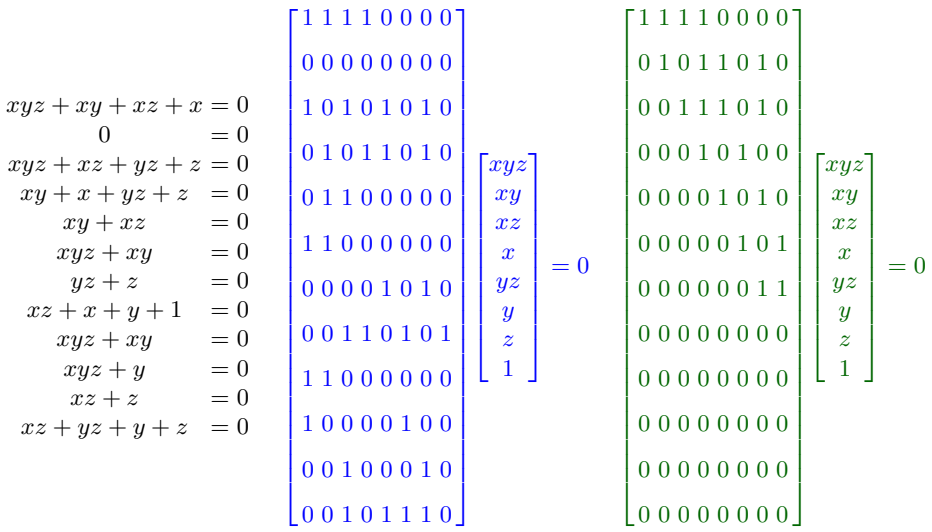

Fig. 2.2. Small example of XL. The goal is to find a solution to the following system of three equations in three variables x, y, z over $\mathbf{F}_2$: $xy+x+yz+z = 0$; $xz+x+y+1 = 0$; $xz + yz + y + z = 0$. Left column (black): Twelve equations obtained as $x, y, z, 1$ times each of the original equations; note that x^2, y^2, z^2 are replaced with x, y, z respectively. Middle column (blue): Same twelve equations expressed in matrix form; the matrix is a Macaulay matrix. Right column (green): Equations obtained by applying Gaussian elimination to the Macaulay matrix. Three of the resulting equations are $x + y = 0$, $y + 1 = 0$, and $z + 1 = 0$, implying $(x, y, z) = (1, 1, 1)$. This is a solution, and therefore the only solution, to the original system.

This section also analyzes the asymptotic performance of XL and FXL, assuming that m, the XL degree parameter d, and the FXL fixing parameter f grow linearly with n. In particular, this section reviews the asymptotic number of monomials and the asymptotic cost of linear algebra. See Section 3 for quantum speedups, and Section 4 for analysis of the overall costs for random systems when d/n and f/n are optimized.

2.1. XL: extended linearization. XL was introduced by Lazard [13]. It was rediscovered and given the name "XL" in [6].

XL begins by computing a **degree-d Macaulay matrix** as follows. Multiply each of the original m quadratic equations by each monomial of degree at most $d - 2$; each product is called a "relation". Each relation is a linear combination of monomials of degree at most d. The Macaulay matrix is, by definition, the matrix of coefficients in these linear combinations. See Figure 2.2 for a small example with $d = 3$.

If the relations have a linear combination of the form $1 = 0$ then the original system of equations has no solution. XL recognizes this situation by linear alge-

bra on the Macaulay matrix: it checks whether the vector $(0, 0, \ldots, 1)$, with 1 at the position of monomial 1, is a linear combination of the rows of the matrix.

More generally, XL checks whether the relations have a linear combination involving only monomials of degree at most 1; i.e., whether the relations imply a linear equation among the variables. This linear equation reduces the original system to a smaller system that can be solved recursively (and further independent equations reduce the system even more). Recognizing this situation is again linear algebra on the Macaulay matrix.[4]

An alternative is to check whether the relations have a linear combination involving only powers of a single variable. The resulting univariate equation is easily solved by fast root-finding algorithms, and each root produces a smaller system that can be solved recursively. There can be "fake" roots that do not correspond to solutions of the original system, but experiments suggest that for random systems these "fake" roots rapidly produce contradictions in subsequent levels of recursion.

There is no guarantee that XL will produce any of this information. Increasing d can produce more information, but increasing d also produces many more monomials, as discussed below. A common way to use XL is to try $d = 2$, then $d = 3$, and so on, until the system is solved. As d increases, there appears to be a sharp transition from (1) XL solving very few systems to (2) XL solving almost all systems; the transition point is quantified in Section 4.

2.3. The number of monomials, and the field equation. A basic combinatorics exercise states that the number of monomials of degree $\leq d$ in n variables $v_1, v_2, \ldots, v_n$ is exactly the binomial coefficient $\binom{n+d}{d}$: the monomial $v_1^{e_1} v_2^{e_2} \cdots v_n^{e_n}$ with $e_1 + e_2 + \cdots + e_n \leq d$ corresponds to the d-element subset $\{e_1 + 1, (e_1 + 1) + (e_2 + 1), \ldots, (e_1 + 1) + \cdots + (e_n + 1)\}$ of $\{1, 2, \ldots, n + d\}$.

If q is small then one can save time in XL and FXL by using the field equation $v^q = v$ to eliminate monomials with exponents larger than $q - 1$. For example, if $q = 2$ then one uses only squarefree monomials; if v^2 appears then one immediately replaces it with v. There are only $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{d}$ squarefree monomials of degree $\leq d$. The example in Figure 2.2 uses this speedup.

More generally, define $\varphi_q \in \mathbf{Z}[z]$ as the polynomial $(1 - z^q)/(1 - z) = 1 + z + \cdots + z^{q-1}$. The number of monomials of degree d in n variables with exponent at most $q - 1$ is the coefficient of z^d in φ_q^n, which we abbreviate $[z^d]\varphi_q^n$. The number of monomials of degree $\leq d$ is $\sum_{k \leq d} [z^k]\varphi_q^n$, or equivalently $[z^d]((1 + z + z^2 + \cdots)\varphi_q^n)$.

[4] Part of the literature suggests, incorrectly, that this requires computing echelon form. In fact, it simply requires solving linear equations. Specifically, finding x such that Mx is zero outside $n+1$ positions is the same as finding x such that $M'x = 0$, where M' removes those positions from M. To find a uniform random r such that $M'r = 0$, one can take a uniform random v, compute $M'v$, use any method to find a solution x to $M'x = M'v$, and compute $r = x - v$. Then Mr is sampled uniformly at random from the space of vectors Mx that are zero outside the specified positions. If the space has positive dimension then each r has at least a 50% chance of discovering this.

The asymptotic behavior of the binomial coefficient $\binom{n}{d}$ is singly exponential in n when d is linear in n. The same is true more generally for $[z^d]\varphi_q^n$.

Specifically, assume that $d/n \in \delta + o(1)$ as $n \to \infty$, where $0 < \delta < q-1$. Then the number of monomials of degree d in n variables with exponents at most $q-1$ is $2^{(\mathrm{mon}_q(\delta)+o(1))n}$, where mon_q is defined below. The number of monomials of degree $\leq d$ in n variables with exponents at most $q-1$ is also $2^{(\mathrm{mon}_q(\delta)+o(1))n}$ if $\delta \leq (q-1)/2$, and $2^{(\lg q+o(1))n} = (q+o(1))^n$ for all $\delta \geq (q-1)/2$, where lg means $\log_2$.

The definition of $\mathrm{mon}_q(\delta)$ for $0 < \delta < q-1$ is as follows: $\mathrm{mon}_q(\delta) = \lg(\varphi_q(\rho)/\rho^\delta)$, where $\rho \in \mathbf{R}$ is the unique positive root of the polynomial

$$\left(\frac{z}{1-z} - \frac{qz^q}{1-z^q} - \delta\right)\varphi_q = -\delta + (1-\delta)z + (2-\delta)z^2 + \cdots + (q-1-\delta)z^{q-1} \in \mathbf{R}[z].$$

To see that this polynomial has a positive root, observe that the constant coefficient $-\delta$ is negative while the top coefficient $q-1-\delta$ is positive. To see that the root is unique, observe that $z/(1-z) - qz^q/(1-z^q) - \delta$ is an increasing function of z when z is positive (its derivative is $1/(1-z)^2 + q^2z^{q-1}/(1-z^q)^2 > 0$) and that φ_q is positive when z is positive.

It is sometimes convenient to also define $\mathrm{mon}_q(0) = 0$ and $\mathrm{mon}_q(q-1) = 0$. Then mon_q is a continuous function on the interval $[0, q-1]$.

For example, mon_2 is exactly the binary entropy function: $\mathrm{mon}_2(\delta) = -\delta \lg \delta - (1-\delta)\lg(1-\delta)$. As $q \to \infty$, the values $\mathrm{mon}_q(\delta)$ converge up to what one might call $\mathrm{mon}_\infty(\delta)$, namely $(1+\delta)\,\mathrm{mon}_2(\delta/(1+\delta)) = (1+\delta)\lg(1+\delta) - \delta \log \delta$.

As a more complicated example, $\mathrm{mon}_3(\delta) = \lg(1+\rho+\rho^2) - \delta \lg \rho$, where ρ is the unique positive root of the polynomial $-\delta + (1-\delta)z + (2-\delta)z^2$; i.e., $\rho = (\delta - 1 + \sqrt{1+6\delta-3\delta^2})/(2(2-\delta))$. If $d/n \in \delta + o(1)$ as $n \to \infty$ then the number of monomials of degree d in n variables with exponents at most 2 is $((1+\rho+\rho^2)/\rho^\delta + o(1))^n$.

2.4. Understanding mon_q: the saddle-point method. The fact that mon_q is the asymptotic exponent for the number of monomials follows from a standard trick in analytic combinatorics called the "saddle-point method". For mon_q it is enough to apply a simple case of this method, making assumptions that we quote from [9, Section VIII.8.1]:

- B and C are power series with nonnegative coefficients. Our mon_q application takes $B = \varphi_q$ and $C = 1$.
- The constant coefficient of B is nonzero.
- The nonzero coefficients of B are at indices whose greatest common divisor is 1.
- B has a positive radius R of convergence in the complex plane. For us $R = \infty$.
- C has radius of convergence $\geq R$.
- T is the limit of $zB'(z)/B(z)$ as z approaches R from below. For us $T = q-1$.

The saddle-point method then states the following. Fix δ with $0 < \delta < T$. If δn is an integer, then the coefficient $[z^{\delta n}]C(z)B(z)^n$ is $(c+o(1))C(\rho)B(\rho)^n/\rho^{\delta n+1}\sqrt{n}$,

where ρ is the unique positive root of $\rho B'(\rho)/B(\rho) = \delta$, and c is an explicit nonzero constant. See [9, Proposition VIII.8].

In particular, for any δ strictly between 0 and $q-1$, the coefficient $[z^{\delta n}]\varphi_q^n$ is $(c+o(1))/\rho\sqrt{n}$ times the nth power of $\varphi_q(\rho)/\rho^\delta$, where ρ is the unique positive root of $\rho\varphi_q'(\rho)/\varphi(\rho) = \delta$, i.e., $\rho/(1-\rho) - q\rho^q/(1-\rho^q) = \delta$.

This is called the "saddle-point method" as a reference to the name "saddle points" for roots of the derivative of an analytic function. The connection to saddle points arises as follows. Cauchy's integration formula states that

$$[z^m]C(z)B(z)^n = \frac{1}{2\pi i}\oint \frac{C(z)B(z)^n\,dz}{z^{m+1}} = \frac{1}{2\pi i}\oint \frac{C(z)F(z)^n\,dz}{z},$$

where $\oint$ integrates on any countour circling once (counterclockwise) around the origin in the complex plane, and where $F(z) = B(z)/z^{m/n}$. The saddle-point method chooses a contour that passes through one or more saddle points of $\log F(z)$, i.e., through roots of $B'(z)/B(z) - m/nz$: in the simple case mentioned above, this contour is a circle of radius ρ around the origin, passing through the unique positive root of $zB'(z)/B(z) = \delta$. One can show, under reasonable assumptions, that the integral is asymptotically dominated by the portion of the integral around the saddle points. For a full exposition see, e.g., [21] or [9, Chapter 8].

2.5. Fast linear algebra. Write A for the number of monomials analyzed above: the number of monomials of degree $\le d$ in n variables with exponents at most $q-1$. Each of the m equations provided as input to XL produces at most A relations, namely one relation for each monomial of degree at most $d-2$. The total number of relations is at most mA. In other words, the Macaulay matrix has A columns and at most mA rows. We focus on the situation that A is exponential in n while m is linear in n; the matrix then has $A^{1+o(1)}$ rows and columns.

Each of the original equations is assumed to be quadratic, and therefore has $O(n^2)$ terms. Consequently each relation also has $O(n^2)$ terms: i.e., there are only $O(n^2)$ nonzero entries in each row of the Macaulay matrix. The Macaulay matrix is thus extremely sparse.

Sparsity saves time in linear algebra. The fastest methods known to solve an $A \times A$ *dense* system of linear equations use $A^{\omega+o(1)}$ operations where $\omega \approx 2.37$, while sufficient *sparsity* reduces the number of operations to $A^{2+o(1)}$. The idea of applying sparse linear-algebra techniques to speed up XL was mentioned by Yang and Chen in 2004 [22], analyzed in more detail by Yang, Chen, and Courtois later the same year [23], and demonstrated in various XL implementations starting in 2006.

We focus exclusively on Wiedemann's algorithm [20] for sparse linear algebra over finite fields. The algorithm is described in [20, page 59] as an "$O(n_0(\omega + n_1 \log n_1)\log n_0)$ expected time method of producing a solution to any linear system [over $\mathbf{F}_q$], providing a solution exists". Here "ω" is the total number of nonzero entries in the matrix; "n_0" and "n_1" are the minimum and maximum of the number of rows and the number of columns; and "time" counts operations

in a sequential RAM model, with each addition and multiplication in $\mathbf{F}_q$ taking time 1.

In the XL situation mentioned above (m linear in n, and A exponential in n), "n_0" and "n_1" and "ω" are all bounded by $A^{1+o(1)}$, so the number of operations in Wiedemann's algorithm is $A^{2+o(1)}$. A closer look shows that the algorithm is bottlenecked by a series of $A^{1+o(1)}$ matrix-vector multiplications, using vector length $A^{1+o(1)}$. The matrix and all other intermediate quantities also fit into $A^{1+o(1)}$ field elements.

2.6. Communication costs and parallelization. Each of the sparse matrix-vector multiplications described above consists of $A^{1+o(1)}$ random accesses to an array of $A^{1+o(1)}$ field elements.[5] A simplified operation-count metric states that each random access has cost 1, independent of A.

This operation-count metric is a poor predictor of the time spent on computation, for two basic reasons. First, if one is spending $A^{1+o(1)}$ dollars on computer hardware, then one can afford as many as $A^{1+o(1)}$ small processing cores that operate in parallel; this could reduce the time by a factor as large as $A^{1+o(1)}$ if there is enough work to do in parallel. Second, the distance between array elements is forced to grow as a positive power of A, correspondingly increasing the time necessary for communication.

There are many previous papers designing algorithms for a two-dimensional $A^{0.5+o(1)} \times A^{0.5+o(1)}$ mesh of small parallel processing cores, with each core connected locally to its neighbors. For example, Brent and Kung showed in [3] how to multiply A-bit integers in time $A^{0.5+o(1)}$, and there are several papers showing how to sort A small items in time $A^{0.5+o(1)}$.

Sorting can in turn be used to implement a batch of random accesses, and in particular to parallelize Wiedemann's algorithm in this model, as Bernstein [2] pointed out in the context of integer factorization. This reduces the time for Wiedemann's algorithm, and the time for XL, to $A^{1.5+o(1)}$.

2.7. FXL: fixing followed by extended linearization. FXL was proposed by Courtois, Klimov, Patarin, and Shamir in 2000 [6].

FXL solves a system of m quadratic equations in n variables $v_1, v_2, \ldots, v_n$ over $\mathbf{F}_q$ as follows. There are q^f possibilities for the last f variables $v_{n-f+1}, \ldots, v_n$. For each possibility, use XL to solve the resulting system of m quadratic equations in $n - f$ variables. In other words, guess (fix) f variables before running XL.

Increasing f by 1 costs a factor q in the number of guesses. However, it also increases the ratio $m/(n-f)$, and this often has a benefit of decreasing the degree d needed for XL to succeed. Optimized FXL exponents for random systems are presented in Section 4.

Note that FXL can be trivially parallelized, reducing the time by any desired factor up through q^f at the expense of increasing area by a similar factor.

[5] As q grows, one has to account for the growing cost of reading, writing, and arithmetic on field elements. For simplicity we focus on asymptotic statements as $n \to \infty$ with q fixed.

3 ReversibleXL and GroverXL

It is conceptually straightforward to replace the brute-force search in FXL with Grover's quantum search method, reducing the number of search iterations to its square root. However, Grover's method requires the underlying function—the function that is evaluated on an input to see whether the input is a solution to the search—to be computed *reversibly*, with no data ever erased.

The XL computation described in the previous section does not fit this model. The computation is constantly erasing data: it repeatedly overwrites a vector with a matrix-vector product. This section analyzes the costs of fitting XL into Grover's method.

3.1. Reversible computation. There would be no difficulty if our goal were merely to count operations: simply keep a journal of all intermediate results, and then run the computation again in reverse order, as in [1, Lemma 1].

Formally, any sequence of NANDs is converted into a reversible computation as follows. Say there are input bits $b_1, b_2, \ldots, b_i$, followed by NANDs $b_{i+1} = 1 - b_{f(i+1)}b_{g(i+1)}$, $b_{i+2} = 1 - b_{f(i+2)}b_{g(i+2)}$, and so on through $b_T = 1 - b_{f(T)}b_{g(T)}$, where $f(j) < j$ and $g(j) < j$. Some of the bits are specified to be output bits; for simplicity assume that these are not the input bits and are not used for further computations.

Consider the following reversible circuit applied to T bits $b_1, b_2, \ldots, b_T$. First apply the following "NOT-Toffoli" gates: $b_{i+1} \leftarrow b_{i+1} + 1 - b_{f(i+1)}b_{g(i+1)}$; $b_{i+2} \leftarrow b_{i+2} + 1 - b_{f(i+2)}b_{g(i+2)}$; and so on through $b_T \leftarrow b_T + 1 - b_{f(T)}b_{g(T)}$. Then apply the same gates again in reverse order to the **ancilla** bits, i.e., the bits that are *not* output bits.

If bits $b_{i+1}, \ldots, b_T$ all start as 0 then the reversible circuit first computes exactly what the original NANDs did, and then it sets the ancilla bits back to 0. More generally, if the ancilla bits all start as 0 then the reversible circuit adds the output of the original function into the output bits, while leaving the input bits untouched and setting the ancilla bits back to 0.

In short, this reversible computation produces $(x, 0, F(x) + y)$ if the input is $(x, 0, y)$. This is what it means to compute a function F reversibly.

3.2. Saving space: the Bennett–Tompa conversion. The circuit described above uses T bits of storage (or T qubits in the context of Grover's method). This amount of hardware is usually vastly larger than the amount of hardware needed for the original computation: in particular, the storage for XL expands quadratically. This begs the question of whether one can afford this amount of hardware, and the question of whether the same amount of hardware can be more productively used in other ways.

Bennett proved in [1, Theorem 1] that any computation using time T and space S can be converted into a reversible computation using time $O(T^{\log_2 3})$ and space $O(S \log T)$. Bennett also proved, with credit to Tompa, that $\log_2 3$ can be replaced by $1 + \epsilon$ for any $\epsilon > 0$. Bennett's theorem is stated for multitape Turing machines; we return below to issues of parallelization and communication costs.

Bennett's $\log_2 3$ conversion works as follows. Decompose a computation into two halves: specifically, say the output is $C_2(C_1(x))$, where x is the input. Starting from $(x, 0, y)$, reversibly compute C_1 by the same construction recursively, obtaining $(x, C_1(x), y)$; reversibly compute C_2 by the same construction recursively, obtaining $(x, C_1(x), C_2(C_1(x))+y)$; and then reversibly compute C_1 again, obtaining $(x, 0, C_2(C_1(x)) + y)$ as desired. This takes three half-size computations. The required space is proportional to the number of levels of recursion; the point here is that ancillas used for C_1 are reused for C_2.

More generally, the Bennett–Tompa conversion splits a computation into k parts, each taking time (approximately) T/k. Starting from $(x, 0, \ldots, 0, y)$, compute C_1, then C_2, and so on through C_k, obtaining

$$(x, C_1(x), C_2(C_1(x)), \ldots, C_{k-1}(\cdots C_1(x)\cdots), C_k(\cdots C_1(x)\cdots) + y).$$

Then compute C_{k-1}, then C_{k-2}, and so on through C_1, obtaining

$$(x, 0, \ldots, 0, C_k(\cdots (x)\cdots) + y).$$

This is the same strategy used above for the extreme case that each C_i is a single NAND. Allowing larger computations C_i produces time exponent $\log_k(2k-1)$, while increasing the space by a factor that depends on k.

Taking $k \in 2^{\Theta(\sqrt{\log T})}$ produces time and space within factors $2^{O(\sqrt{\log T})}$ of the original computation, as pointed out by Knill in [12, Theorem 2.12]. In the context of our XL analyses, these factors are $2^{o(n)}$ and therefore do not affect our asymptotic exponents. Knill also pointed out some smaller optimizations that are not visible in our exponents.

3.3. Parallelizing the Bennett–Tompa conversion. We point out that the idea of the Bennett–Tompa conversion is compatible with massively parallel computation and local communication, in particular communication on a realistic two-dimensional mesh architecture.

Assume that the original computation is a sequence of T time steps, where each step is carried out in parallel by A small processing cores. The cores are arranged in a $\sqrt{A} \times \sqrt{A}$ mesh, with edges between adjacent cores. Formally, core (i, j) has state $s[i, j, t]$ at time t consisting of a small number of bits; $s[i, j, t+1]$ is the output of a small computation applied to $s[i, j, t]$, $s[i-1, j, t]$, $s[i, j-1, t]$, $s[i+1, j, t]$, $s[i, j+1, t]$, with states past the edge defined to be empty. For our XL application, "small" can be defined as subexponential in n, while T and A grow exponentially with n.

We convert this into a reversible computation on a $\sqrt{A} \times \sqrt{A}$ mesh of small cores as follows. Divide the original computation into k parts $C_1, C_2, \ldots, C_k$, each taking time approximately T/k. Core (i, j) starts with $(s[i, j, 0], 0, \ldots, y_{i,j})$. Recursively apply C_1 reversibly, recursively apply C_2 reversibly, and so on through C_k, obtaining

$$(s[i, j, 0], s[i, j, t_1], \ldots, s[i, j, t_k] + y_{i,j}).$$

Then apply C_{k-1} reversibly, apply C_{k-2} reversibly, and so on through C_1, obtaining

$$(s[i, j, 0], 0, \ldots, s[i, j, t_k] + y_{i,j})$$

as desired. The base case of the recursion is a time-1 parallel computation consisting of small local computations, each of which is applied reversibly with small overhead.

This conversion visibly expands the state in each core. The final state is accompanied by a journal of k earlier states; and each level of recursion needs its own journal, overall multiplying the state size by $k(\log T)/\log k$. Each level of recursion also multiplies the time by $(2k-1)/k$. As in [12], we take $k \in 2^{\Theta(\sqrt{n})}$, so that the overall area and time overheads are subexponential in n. The expanded area also implies a slowdown in communication, but this is again subexponential in n.

3.4. ReversibleXL and GroverXL. ReversibleXL is, by definition, the result of applying the above parallel conversion to the XL computation of whether a system has a solution. As noted in Section 2, XL can fail to determine whether a system has a solution, but we assume that d is chosen large enough that XL works for all systems provided to ReversibleXL; see Section 4.

GroverXL solves a system of quadratic equations in n variables as follows. Use Grover's method to search through the q^f possibilities for the last f variables, applying ReversibleXL to each possibility. If the system has a solution then Grover's method returns a random choice of solution; otherwise it returns a uniform random choice of the last f variables. Either way, substitute this choice into the system, and use XL to see whether there is a solution for the remaining variables.

GroverXL takes the time for $q^{f/2+o(1)}$ quantum computations of ReversibleXL, plus a final computation of XL. Like other applications of Grover's method, GroverXL can be parallelized across many separate computations, increasing the area by a corresponding factor while dividing the time by the square root of the same factor. The limit of "many" is q^f, at which point one should simply use FXL.

4 Analysis for random systems

This section presents asymptotic cost exponents for solving random systems of $(\mu + o(1))n$ quadratic equations in n variables over $\mathbf{F}_q$, assuming $\mu \geq 1$. Exponents are shown for various small choices of q and for various choices of μ ranging from 1.0 up through 2.0.

Exponent e means that the cost is $2^{(e+o(1))n}$ as $n \to \infty$, or equivalently $(2^e + o(1))^n$. Simple brute-force search has exponent $\lg q$, and Grover's algorithm has exponent $0.5 \lg q$, where as before $\lg = \log_2$. In all cases GroverXL has better exponents.

4.1. A script for computing cost exponents. To simplify verification, and to let the reader easily compute cost exponents for further pairs (q, μ), we include a script to compute exponents. See Figures 4.2 and 4.3. This script uses the free Sage computer-algebra system, version 8.0.

The script covers both GroverXL and FXL. In each case it covers two different metrics: (1) the exponent of a simplified operation count, and (2) the exponent

```
import collections

# generic caching mechanism
class memoized(object):
  def __init__(self,func):
    self.func = func
    self.cache = {}
    self.__name__ = 'memoized:' + func.__name__
  def __call__(self,*args):
    if not isinstance(args,collections.Hashable):
      return self.func(*args)
    if not args in self.cache:
      self.cache[args] = self.func(*args)
    return self.cache[args]

@memoized
def lg(x):
  return log(x*1.0)/log(2.0)

Zx.<x> = ZZ[]
Ry.<y> = RR[]
Zxz.<z> = Zx[]
Zxza.<a> = Zxz[]

@memoized
def phi(q):
  return Zx((1-x^q)/(1-x))

@memoized
def monpoly(q):
  h = x/(1-x) - q*x^q/(1-x^q)
  return Zx(phi(q)*h)

@memoized
def mon(q,kappa):
  if not q in ZZ: raise Exception('q must be integer')
  if q < 2: raise Exception('q must be at least 2')
  if kappa < 0: return -Infinity
  if kappa == 0: return 0
  if kappa == q-1: return 0
  if kappa > q-1: return -Infinity
  g = Ry(monpoly(q)) - kappa*Ry(phi(q))
  roots = g.roots(RR)
  rho = max(r for r,e in roots)
  return lg(phi(q)(rho)/rho^kappa)
```

Fig. 4.2. Script for computing cost exponents, part 1: caching and `mon` computation.

of the area-time product AT on a two-dimensional mesh-connected computer. In the context of parallelizing Grover's method, another metric of interest is the exponent of the $\sqrt{AT}$ product; but for parallelized GroverXL this turns out to be identical to the first metric.

The script tries five values of q, namely $2, 3, 4, 5, 16$; this is specified by `doit(2)` through `doit(16)` at the end of the script. The script takes a few hours to run, almost entirely for $q = 16$.

There are three nested loops for each q: `search` is either 0.5 for GroverXL or 1 for FXL; `linalg` is either 2 for a simplified operation-count metric or 2.5 for area-time product on a two-dimensional mesh; and `k` tries each μ between 1 and 2 in steps of 0.01. For each choice of $(q, \texttt{search}, \texttt{linalg}, \mu)$, the script prints

```
@memoized
def deltapoly(q):
  h = (-x/z - q*z^(q-1)/(1-z^q) + 1/(1-z)
       - 2*z*a/(1-z^2) + 2*q*z^(2*q-1)*a/(1-z^(2*q)))
  h *= z*(1-z^(2*q))/(1-z)
  return Zxza(h)

@memoized
def delta(q,mn):
  hmn = deltapoly(q)(QQ(mn)).discriminant()
  roots = hmn.roots(RR)
  if not roots: return -1
  return min(r for r,e in roots if r>0)

@memoized
def alpha(q,mn):
  return mon(q,delta(q,mn))

def doit(q):
  for search in [0.5,1]:
    searchlgq = search * lg(q)
    for linalg in [2,2.5]:
      def f(mn): return (searchlgq - linalg*alpha(q,mn))/mn
      bestvalue,bestmn = find_local_maximum(f,1,10)
      bestmn = RR(bestmn)
      for k in range(100,201):
        mn = k*0.01
        x = max(mn,bestmn)
        context = '%d %.1f %.1f' % (q,search,linalg)
        cost = searchlgq-mn*f(x)
        alphax = alpha(q,x)
        print context,'%.3f' % mn,cost,x,alphax,alphax*mn/x
        sys.stdout.flush()

doit(2)
doit(3)
doit(4)
doit(5)
doit(16)
```

Fig. 4.3. Script for computing cost exponents, part 2: δ optimization and f optimization.

one line showing the exponent for solving $m = (\mu + o(1))n$ random quadratic equations in n variables over $\mathbf{F}_q$.

4.4. Understanding the XL exponent. Guessing variables does not save time if the system is sufficiently overdetermined: i.e., if μ is larger than a particular cutoff μ_0 then FXL and GroverXL both boil down to XL. The script computes the cost exponent for XL in three steps:

- Compute δ as explained in Section 4.5. The XL degree d is $(\delta + o(1))n$.
- Compute $\alpha = \mathrm{mon}_q(\delta)$. The number of monomials in XL is $2^{(\alpha+o(1))n}$.
- The exponent is $\texttt{linalg} \cdot \alpha$.

Specifically, XL takes time $T = 2^{(1.5\alpha+o(1))n}$ on a mesh of area $A = 2^{(\alpha+o(1))n}$, so AT is $2^{(2.5\alpha+o(1))n}$. In a simplified operation-count metric the exponent is only 2α. Note that the $\sqrt{A}T$ exponent is also 2α; as mentioned above, the $\sqrt{A}T$ metric is identical to the simplified operation-count metric for these algorithms.

4.5. Understanding δ. The script uses XL degree $d \in (\delta + o(1))n$, where δ is computed as follows. Define h as the polynomial

$$z\frac{1-z^{2q}}{1-z}\left(\frac{-x}{z} - \frac{qz^{q-1}}{1-z^q} + \frac{1}{1-z} - \frac{2\mu z}{1-z^2} + \frac{2\mu q z^{2q-1}}{1-z^{2q}}\right)$$

in the polynomial ring $\mathbf{R}[x,z]$. Define $\Delta \in \mathbf{R}[x]$ as the discriminant of h with respect to z. Then Δ has a unique positive real root, namely δ.

This is a concise statement of a calculation explained in the previous XL literature. For example, for $q = 2$ and $m = n$, the XL exponent $0.87280\ldots$ and the FXL exponent $0.79106\ldots$ were calculated this way in [23]. The rest of this subsection reviews the main steps in the argument that this is the correct asymptotic degree for XL.

Recall that A, the number of monomials in XL, is the coefficient of z^d in $\varphi_q(z)^n/(1-z)$, where $\varphi_q(z) = (1-z^q)/(1-z)$; in short, $[z^d](\varphi_q(z)^n/(1-z))$. The number of relations is at most $m[z^{d-2}](\varphi_q(z)^n/(1-z))$. A more careful analysis shows that the linear span of the relations has codimension (i.e., A minus the dimension) at least $[z^d](\varphi_q(z)^n/(1-z)\varphi_q(z^2)^m)$. See [22, Theorem 2]; see also [7].

As d increases, there is a sharp transition in the behavior of the coefficient $[z^d](\varphi_q(z)^n/(1-z)\varphi_q(z^2)^m)$, and in the experimentally observed behavior of XL for random systems. If d is noticeably below a cutoff analyzed below, then the coefficient is a huge positive integer, and XL almost always fails for random systems (although it does succeed for some special systems of interest): there are not enough relations to provide interesting information about any small subset of the monomials. As d grows past the cutoff, the coefficient crosses below 0 and rapidly becomes quite negative, and XL almost always succeeds for random systems. If the coefficient happens to be extremely close to 0 then XL will often succeed and often fail (there are often some accidental extra dependencies between relations), but adding $o(n)$ to d eliminates this vacillation.

We analyze the asymptotics of this coefficient as in Section 2.4. First use Cauchy's integration formula

$$[z^d]\left(\frac{\varphi_q(z)^n}{(1-z)\varphi_q(z^2)^m}\right) = \frac{1}{2\pi i}\oint\left(\frac{\varphi_q(z)^n\,dz}{z^{d+1}(1-z)\varphi_q(z^2)^m}\right) = \frac{1}{2\pi i}\oint\frac{F(z)^n\,dz}{z(1-z)}$$

where $F(z) = \varphi_q(z)/z^{d/n}\varphi_q(z^2)^{m/n}$. Then substitute $d = \delta n$ and $m = \mu n$, and apply the saddle-point method to compute an asymptotic formula for the integral as $n \to \infty$. This asymptotic formula involves powers of the form $F(\rho)^n$, where ρ runs through the complex roots of the logarithmic derivative

$$\frac{F'(z)}{F(z)} = \frac{-\delta}{z} - \frac{qz^{q-1}}{1-z^q} + \frac{1}{1-z} - \frac{2\mu z}{1-z^2} + \frac{2\mu q z^{2q-1}}{1-z^{2q}}.$$

Multiplying this logarithmic derivative by $z(1-z^{2q})/(1-z)$ produces the polynomial h defined earlier, with δ substituted for x. The roots of h are essentially the roots of F'/F; a closer look shows that h has an extra root -1 if q is odd, but this does not affect the calculation of the cutoff.

Finally, with some work one can see that the phase transition from positive coefficients to negative coefficients occurs exactly when h has a double root, i.e., exactly when the discriminant Δ is zero. See generally [5] for the theory of double saddle points, and [23] for applications to XL.

4.6. Understanding the FXL and GroverXL exponents. More generally, for any $\mu \geq 1$, the script computes the cost exponents for FXL and GroverXL as follows:

- Choose $\lambda \geq \mu$ as explained below. Assume that $(1-\mu/\lambda+o(1))n$ variables are fixed, i.e., that XL is given $(\mu+o(1))n$ equations in $(\mu/\lambda+o(1))n$ variables.
- Compute α as in Section 4.4, starting from λ rather than μ. Then XL and ReversibleXL take time $T = 2^{(1.5\alpha+o(1))(\mu/\lambda+o(1))n}$ using area $T = 2^{(\alpha+o(1))(\mu/\lambda+o(1))n}$; i.e., in base 2^n, they have time exponent $1.5\alpha\mu/\lambda$ and area exponent $\alpha\mu/\lambda$ (and operation-count exponent $2\alpha\mu/\lambda$).
- For FXL, add $(1-\mu/\lambda)\lg q$ to the time exponent (and the operation-count exponent) to account for the cost of brute-force search. For GroverXL, add $0.5(1-\mu/\lambda)\lg q$. In other words, add $\texttt{search}(1-\mu/\lambda)\lg q$.

To summarize, the exponent is $\texttt{linalg} \cdot \alpha\mu/\lambda + \texttt{search}(1-\mu/\lambda)\lg q$, where α is implicitly a function of λ.

This formula shows that λ is best chosen to maximize $\texttt{search}(\lg q)/\lambda - \texttt{linalg} \cdot \alpha/\lambda$. The script uses Sage's `find_local_maximum` to find the maximum of this function on $[1, 10]$; larger inputs did not help for the range of `linalg` etc. that we use. The position of this maximum is `bestmn`, exactly the cutoff μ_0 mentioned above. The script then defines $\lambda = \max\{\mu, \mu_0\}$.

4.7. Example: GroverXL for $q = 2$. The polynomial h defined earlier is $(1-2\mu-\delta)z^3+(-2\mu-\delta)z^2+(1-\delta)z-\delta$. The discriminant Δ of h with respect to z is a quartic polynomial in δ, so the equation $\Delta = 0$ can be solved explicitly by radicals, and it is easy to see the unique positive root:

$$\delta = F(\mu) = -\mu + \frac{1}{2} + \frac{1}{2}\sqrt{2\,\mu^2 - 10\,\mu - 1 + 2\,\sqrt{\mu^4 + 6\,\mu^3 + 12\,\mu^2 + 8\,\mu}} \quad (1)$$

For example, if $(q, \mu) = (2, 1)$, then $\delta = 0.0899798\ldots$; and $\alpha = \text{mon}_2(\delta) = -\delta\lg\delta - (1-\delta)\lg(1-\delta) = 0.436402\ldots$. This means that XL uses degree $2^{(0.08997\ldots+o(1))n}$, and $2^{(0.43640\ldots+o(1))n}$ monomials. The operation-count exponent is $2 \cdot 0.43640\ldots = 0.87280\ldots$.

As another example, $\mu_0 = 1.81626\ldots$ maximizes $(1-2\,\text{mon}_2(F(\mu_0)))/\mu_0$. For $\mu = \mu_0$, XL has $\delta = F(\mu_0) = 0.05573\ldots$ and $\alpha = \text{mon}_2(\delta) = 0.31026\ldots$, for operation-count exponent $0.62052\ldots$.

FXL, when optimized for operation count, fixes enough variables to reach m/μ_0 remaining variables. For example, again for $(q, \mu) = (2, 1)$, FXL runs XL with $m = (1+o(1))n$ equations and $(1/\mu_0+o(1))n = (0.55058\ldots+o(1))n$ variables. XL's operation-count exponent in base 2^n is then $0.55058\ldots \cdot 0.62052\ldots = 0.34164\ldots$. The remaining $(1-1/\mu_0+o(1))n = (0.44941\ldots+o(1))n$ variables are found by brute-force search, so the final exponent for FXL is $0.79106\ldots$.

	$\mu = 1.0$	$\mu = 1.1$	$\mu = 1.2$	$\mu = 1.3$	$\mu = 1.4$	$\mu = 1.5$	$\mu = 1.6$	$\mu = 1.7$	$\mu = 2.0$
$q = 2$	0.46240	0.45864	0.45488	0.45112	0.44737	0.44361	0.43985	0.43609	0.42481
$q = 3$	0.70425	0.69542	0.68660	0.67778	0.66895	0.66013	0.65131	0.64248	0.61601
$q = 4$	0.85848	0.84433	0.83018	0.81602	0.80187	0.78772	0.77357	0.75942	0.71696
$q = 5$	0.96843	0.94918	0.92993	0.91068	0.89142	0.87217	0.85292	0.83367	0.77591
$q = 16$	1.42604	1.36865	1.31125	1.25386	1.19646	1.13907	1.08167	1.02428	0.86575

Table 4.10. GroverXL operation-count exponent for $q \in \{2, 3, 4, 5, 16\}$ and various μ. Operation count ignores communication costs. Each exponent is rounded down to multiple of 0.00001. For comparison, Grover's algorithm without XL has exponents $0.50000, 0.79248, 1.00000, 1.16096, 2.00000$, independently of μ.

	$\mu = 1.0$	$\mu = 1.1$	$\mu = 1.2$	$\mu = 1.3$	$\mu = 1.4$	$\mu = 1.5$	$\mu = 1.6$	$\mu = 1.7$	$\mu = 2.0$
$q = 2$	0.47210	0.46931	0.46652	0.46373	0.46094	0.45815	0.45536	0.45257	0.44420
$q = 3$	0.72468	0.71790	0.71112	0.70434	0.69756	0.69078	0.68400	0.67722	0.65688
$q = 4$	0.88987	0.87886	0.86785	0.85683	0.84582	0.83481	0.82380	0.81278	0.77975
$q = 5$	1.01016	0.99508	0.98000	0.96492	0.94984	0.93476	0.91968	0.90460	0.85937
$q = 16$	1.53753	1.49128	1.44503	1.39879	1.35254	1.30629	1.26005	1.21380	1.07506

Table 4.11. GroverXL cost exponent for $q \in \{2, 3, 4, 5, 16\}$ and various μ. Cost is area-time product on two-dimensional mesh-connected computer. Each exponent is rounded down to multiple of 0.00001. For comparison, Grover's algorithm without XL has exponents $0.50000, 0.79248, 1.00000, 1.16096, 2.00000$, independently of μ.

We emphasize that this calculation so far is not new: FXL was analyzed this way in [23]. Our main contributions are the design and analysis of GroverXL. Our script is also new (and possibly the first public software to automate these analyses), as are the area-time analyses.

For GroverXL, we find minimum exponent $0.47210\ldots$ for the area-time product by taking $\mu_0 = 7.74234\ldots$ to maximize $(0.5 - 2.5 \operatorname{mon}_2(F(\mu_0)))/\mu_0$. We also find minimum operation-count exponent $0.46240\ldots$ by taking $\mu_0 = 5.63489\ldots$ to maximize $(0.5 - 2 \operatorname{mon}_2(F(\mu_0)))/\mu_0$.

4.8. Example: GroverXL for $q = 3$. The polynomial h is now $(-\delta + 2 - 4\mu)z^4 - z^3 + (-\delta + 1 - 2\mu)z^2 + z - \delta = 0$. The discriminant Δ is a degree-6 equation that is again solvable in radicals for δ as a function of μ. This solution is quite complex, presumably less efficient than the more general root-finding techniques used by our script.

Numerical computations proceed as in Section 4.7. For example, for $(q, \mu) = (3, 1)$ we find minimum area-time exponent $0.72468\ldots$ (compared to $1.27507\ldots$ for FXL) by taking $\mu_0 = 5.36509\ldots$, and minimum operation-count exponent $0.70425\ldots$ (compared to $1.17521\ldots$ for FXL) by taking $\mu_0 = 4.11429\ldots$.

4.9. Tables of results. Tables 4.10 and 4.11 show the GroverXL operation-count exponent and cost exponent respectively, as computed by the script from Section 4.1. Tables 4.12 and 4.13 show the exponents for the amount of hardware.

	$\mu = 1.0$	$\mu = 1.1$	$\mu = 1.2$	$\mu = 1.3$	$\mu = 1.4$	$\mu = 1.5$	$\mu = 1.6$	$\mu = 1.7$	$\mu = 2.0$
$q = 2$	0.02557	0.02812	0.03068	0.03324	0.03579	0.03835	0.04091	0.04346	0.05114
$q = 3$	0.05219	0.05741	0.06263	0.06785	0.07306	0.07828	0.08350	0.08872	0.10438
$q = 4$	0.07882	0.08670	0.09458	0.10247	0.11035	0.11823	0.12611	0.13400	0.15764
$q = 5$	0.10377	0.11415	0.12453	0.13490	0.14528	0.15566	0.16604	0.17641	0.20755
$q = 16$	0.26759	0.29434	0.32110	0.34786	0.37462	0.40138	0.42814	0.45490	0.43287

Table 4.12. GroverXL space exponent when parameters are optimized for operation count, for $q \in \{2, 3, 4, 5, 16\}$ and various μ. Operation count ignores communication costs. Each exponent is rounded down to multiple of 0.00001.

	$\mu = 1.0$	$\mu = 1.1$	$\mu = 1.2$	$\mu = 1.3$	$\mu = 1.4$	$\mu = 1.5$	$\mu = 1.6$	$\mu = 1.7$	$\mu = 2.0$
$q = 2$	0.01467	0.01614	0.01760	0.01907	0.02054	0.02200	0.02347	0.02494	0.02934
$q = 3$	0.03196	0.03516	0.03835	0.04155	0.04475	0.04794	0.05114	0.05434	0.06393
$q = 4$	0.04992	0.05491	0.05990	0.06489	0.06988	0.07488	0.07987	0.08486	0.09984
$q = 5$	0.06696	0.07365	0.08035	0.08704	0.09374	0.10044	0.10713	0.11383	0.13392
$q = 16$	0.18512	0.20363	0.22215	0.24066	0.25917	0.27768	0.29620	0.31471	0.37025

Table 4.13. GroverXL area exponent when parameters are optimized for cost, for $q \in \{2, 3, 4, 5, 16\}$ and various μ. Cost is area-time product on two-dimensional mesh-connected computer. Each exponent is rounded down to multiple of 0.00001.

For example, the top-left entries in these tables are for $q = 2$ and $\mu = 1.0$. The entries are, respectively, 0.46240, 0.47210, 0.02557, and 0.01467. The first and third numbers indicate that GroverXL uses $2^{(0.46240\ldots+o(1))n}$ operations in space $2^{(0.02557\ldots+o(1))n}$, when GroverXL parameters are optimized for operation count. The second and fourth numbers indicate that GroverXL has area-time product $2^{(0.47210\ldots+o(1))n}$ using area $2^{(0.01467\ldots+o(1))n}$, when GroverXL parameters are optimized for area-time product.

For comparison, Tables 4.14 and 4.15 show the FXL operation-count exponent and cost exponent. Note that the case $q = 16$ and $\mu = 2.0$ has the same operation-count exponent for FXL as for GroverXL; in this case μ is above $\mu_0 \approx 1.80$, and guessing is not helpful (although it does help in area-time product, since then $\mu_0 \approx 2.16$). For smaller values of q and μ, GroverXL has better exponents than FXL, which in turn has better exponents than XL.

References

[1] Charles H. Bennett, *Time/space trade-offs for reversible computation*, SIAM Journal on Computing **18** (1989), 766–776. Citations in this document: §3.1, §3.2.

[2] Daniel J. Bernstein, *Circuits for integer factorization: a proposal* (2001). URL: https://cr.yp.to/papers.html#nfscircuit. Citations in this document: §2.6.

[3] Richard P. Brent, H. T. Kung, *The area-time complexity of binary multiplication*, Journal of the ACM **28** (1981), 521–534. URL: http://wwwmaths.anu.edu.au/~brent/pub/pub055.html. Citations in this document: §2.6.

	$\mu = 1.0$	$\mu = 1.1$	$\mu = 1.2$	$\mu = 1.3$	$\mu = 1.4$	$\mu = 1.5$	$\mu = 1.6$	$\mu = 1.7$	$\mu = 2.0$
$q = 2$	0.79106	0.77017	0.74928	0.72838	0.70749	0.68660	0.66570	0.64481	0.58466
$q = 3$	1.17521	1.13423	1.09325	1.05228	1.01130	0.97033	0.92935	0.88839	0.78134
$q = 4$	1.39851	1.33836	1.27821	1.21807	1.15792	1.09777	1.03763	0.98102	0.84342
$q = 5$	1.54347	1.46563	1.38778	1.30993	1.23209	1.15424	1.07963	1.01421	0.86056
$q = 16$	2.00814	1.80896	1.60977	1.42959	1.29463	1.18842	1.10168	1.02899	0.86575

Table 4.14. FXL operation-count exponent for $q \in \{2, 3, 4, 5, 16\}$ and various μ. Operation count ignores communication costs. Each exponent is rounded down to multiple of 0.00001. For comparison, brute-force search without XL has exponents $1.00000, 1.58496, 2.00000, 2.32192, 4.00000$, independently of μ.

	$\mu = 1.0$	$\mu = 1.1$	$\mu = 1.2$	$\mu = 1.3$	$\mu = 1.4$	$\mu = 1.5$	$\mu = 1.6$	$\mu = 1.7$	$\mu = 2.0$
$q = 2$	0.85284	0.83813	0.82341	0.80870	0.79398	0.77926	0.76455	0.74983	0.70569
$q = 3$	1.27507	1.24409	1.21310	1.18211	1.15112	1.12013	1.08914	1.05816	0.96519
$q = 4$	1.52698	1.47968	1.43237	1.38507	1.33777	1.29047	1.24317	1.19587	1.05396
$q = 5$	1.69553	1.63289	1.57025	1.50761	1.44497	1.38233	1.31969	1.25705	1.07570
$q = 16$	2.29409	2.12350	1.95291	1.78232	1.61828	1.48552	1.37710	1.28624	1.08219

Table 4.15. FXL cost exponent for $q \in \{2, 3, 4, 5, 16\}$ and various μ. Cost is area-time product on two-dimensional mesh-connected computer. Each exponent is rounded down to multiple of 0.00001. For comparison, brute-force search without XL has exponents $1.00000, 1.58496, 2.00000, 2.32192, 4.00000$, independently of μ.

[4] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Bo-Yin Yang, *Solving quadratic equations with XL on parallel architectures*, in CHES 2012 [**18**] (2012), 356–373. URL: https://eprint.iacr.org/2016/412.

[5] Clive R. Chester, Bernard Friedman, Fritz Ursell, *An extension of the method of steepest descents*, Proceedings of the Cambridge Philosophical Society **53** (1957), 599–611. Citations in this document: §4.5.

[6] Nicolas Courtois, Alexander Klimov, Jacques Patarin, Adi Shamir, *Efficient algorithms for solving overdefined systems of multivariate polynomial equations*, in Eurocrypt 2000 [**17**] (2000), 392–407. URL: http://minrank.org/xlfull.pdf. Citations in this document: §2.1, §2.7.

[7] Claus Diem, *The XL-algorithm and a conjecture from commutative algebra*, in Asiacrypt 2004 [**14**] (2004), 323–337. Citations in this document: §4.5.

[8] Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, Jean-Pierre Tillich, *Algebraic cryptanalysis of McEliece variants with compact keys*, in Eurocrypt 2010 [**10**] (2010), 279–298. URL: https://www.iacr.org/archive/eurocrypt2010/66320290/66320290.pdf. Citations in this document: §1.1.

[9] Philippe Flajolet, Robert Sedgewick, *Analytic combinatorics*, Cambridge University Press, 2009. ISBN 978-0-521-89806-5. URL: http://ac.cs.princeton.edu/home/. Citations in this document: §2.4, §2.4, §2.4.

[10] Henri Gilbert (editor), *Advances in cryptology—EUROCRYPT 2010, 29th annual international conference on the theory and applications of cryptographic techniques, French Riviera, May 30–June 3, 2010, proceedings*, Lecture Notes in Computer Science, 6110, Springer, 2010. ISBN 978-3-642-13189-9. See [8].

[11] Philip N. Klein (editor), *Proceedings of the twenty-eighth annual ACM-SIAM symposium on discrete algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16–19*, SIAM, 2017. See [15].

[12] Emanuel Knill, *An analysis of Bennett's pebble game* (1995). URL: http://arxiv.org/abs/math/9508218. Citations in this document: §3.2, §3.3.

[13] Daniel Lazard, *Résolution des systèmes d'équations algébriques*, Theoretical Computer Science **15** (1981), 77–110. URL: https://www.sciencedirect.com/science/article/pii/0304397581900645. Citations in this document: §2.1.

[14] Pil Joong Lee (editor), *Advances in cryptology—ASIACRYPT 2004, 10th international conference on the theory and application of cryptology and information security, Jeju Island, Korea, December 5–9, 2004, proceedings*, Lecture Notes in Computer Science, 3329, Springer, 2004. See [7].

[15] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, Huacheng Yu, *Beating brute force for systems of polynomial equations over finite fields*, in SODA 2017 [**11**] (2017), 2190–2202. URL: https://www.iacr.org/archive/eurocrypt2010/66320290/66320290.pdf. Citations in this document: §1.2, §1.2.

[16] Javier Lopez, Sihan Qing, Eiji Okamoto (editors), *Information and Communications Security, 6th international conference, ICICS 2004, Malaga, Spain, October 27–29, 2004, proceedings*, Lecture Notes in Computer Science, 3269, Springer, 2004. ISBN 3-540-23563-9. See [23].

[17] Bart Preneel (editor), *Advances in cryptology—EUROCRYPT 2000, international conference on the theory and application of cryptographic techniques, Bruges, Belgium, May 14–18, 2000, proceeding*, Lecture Notes in Computer Science, 1807, Springer, 2000. See [6].

[18] Emmanuel Prouff, Patrick Schaumont (editors), *Cryptographic hardware and embedded systems—CHES 2012—14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*, Lecture Notes in Computer Science, 7428, Springer, 2012. ISBN 978-3-642-33026-1. See [4].

[19] Huaxiong Wang, Josef Pieprzyk, Vijay Varadharajan (editors), *Information security and privacy: 9th Australasian conference, ACISP 2004, Sydney, Australia, July 13–15, 2004, proceedings*, Lecture Notes in Computer Science, 3108, Springer, 2004. ISBN 3-540-22379-7. See [22].

[20] Douglas H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory **32** (1986), 54–62. MR 87g:11166. Citations in this document: §2.5, §2.5.

[21] Roderick Wong, *Asymptotic approximations of integrals*, Academic Press, 1989. ISBN 0-12-762535-6. Citations in this document: §2.4.

[22] Bo-Yin Yang, Jiun-Ming Chen, *Theoretical analysis of XL over small fields*, in ACISP 2004 [**19**] (2004), 277–288. URL: http://precision.moscito.org/by-publ/recent/xxl2-update.pdf. Citations in this document: §2.5, §4.5.

[23] Bo-Yin Yang, Jiun-Ming Chen, Nicolas Courtois, *On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis*, in ICICS 2004 [**16**] (2004), 401–413. URL: http://www.iis.sinica.edu.tw/papers/byyang/2384-F.pdf. Citations in this document: §1.2, §2.5, §4.5, §4.5, §4.7.