Efficient Parallel Evaluation of Multivariate Quadratic Polynomials on GPUs

Satoshi Tanaka[1], Tung Chou[2], Bo-Yin Yang[2], Chen-Mou Cheng[3], and Kouichi Sakurai[1]

[1] Kyushu University, Fukuoka, Japan
{tanasato@itslab.inf,sakurai@csce}.kyushu-u.ac.jp
[2] Academia Sinica, Taipei, Taiwan
{blueprint,by}@crypto.tw
[3] National Taiwan University, Taipei, Taiwan
ccheng@cc.ee.ntu.edu.tw

Abstract. QUAD is a provably secure stream cipher, whose security is based on the hardness assumption of solving multivariate quadratic polynomial systems over a finite field, which is known to be NP-complete. However, such provable security comes at a price, and QUAD is slower than most other stream ciphers that do not have security proofs.

In this paper, we discuss two efficient parallelization techniques for evaluating multivariate quadratic polynomial systems on GPU, which can effectively accelerate the QUAD stream cipher. The first approach focuses on formula of summations in quadratics, while the second approach uses parallel reduction to summations. Our approaches can be easily generalized and applied to other multivariate cryptosystems.

Keywords: Stream cipher, efficient implementation, multivariate cryptography, GPGPU.

1 Introduction

1.1 Background

Multivariate cryptography uses multivariate polynomial systems as public keys. The security of multivariate cryptography is based on the hardness of solving non-linear multivariate polynomial systems over a finite field [1]. Multivariate cryptography is considered to be a promising tool for fast digital signature because it often involves arithmetic operations in smaller algebraic structures compared with traditional public-key cryptosystems like RSA.

Non-linear multivariate polynomials can also be used to construct symmetric-key cryptosystems, e.g., the QUAD stream cipher [3]. The security of QUAD depends on the hardness of the multivariate quadratic (MQ) problem. QUAD has a provable security, but it is slow compared with other symmetric ciphers. It would be nice if we could accelerate QUAD while having a security proof, combining the strengths from the two worlds.

D.H. Lee and M. Yung (Eds.): WISA 2012, LNCS 7690, pp. 28–42, 2012.

1.2 Related Works

Berbain et al. proposed several efficient implementation techniques for multivariate cryptography [2]. GPU implementation result of the QUAD stream cipher [5]. In this paper, we reconsider GPU implementations of the QUAD stream cipher. We note that a preliminary version of this paper, "Fast Implementation and Experimentation of Multivariate Cryptography," was presented at the 6th Joint Workshop on Information Security, 2011. This version includes the results of further investigation after the preliminary version was presented at the workshop.

1.3 Contributions

Our main contribution is to accelerate the evaluation of quadratic polynomials in the QUAD stream cipher. The bottleneck computation in QUAD's encryption is to evaluate multivariate quadratic polynomial systems. In particular, we accelerate the computation of the summation in evaluating quadratic polynomials.

We investigated two parallelization strategies to evaluate summations in multivariate quadratic polynomials. The first approach focuses on formula of summations in quadratics, while the second uses parallel reduction to summations. Our techniques apply to multivariate public-key cryptography as well.

Finally, even if QUAD cannot be accelerated a lot by the proposed techniques, having a GPU implementation is still useful because we will be able to offload the computation from CPU to GPU on busy servers.

2 Multivariate Cryptography

Multivariate cryptography is a candidate of post-quantum cryptography. In multivariate cryptography, we can encrypt plaintext to ciphertext by evaluating appropriate multivariate polynomial systems over $\mathbb{GF}_q$. Or we can use it to generate keystream bits as in QUAD. In the rest of this section, we will give an overview of the multivariate stream cipher QUAD and the problems it faces.

2.1 Multivariate Polynomial Systems

Multivariate Polynomials. A term is a primitive unit which can be denoted only by using multiplications. A term consists of one constant and some unknowns. A monomial consists of a single term, and a polynomial consists a finite sum of terms, as shown in Equation 1, where x_i variables, and $\alpha, \beta_i, \gamma_{i,j}$, constants.

$$\alpha + \sum_{i=1}^{n} \beta_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} \gamma_{i,j} x_i x_j + \dots \tag{1}$$

Equation 2 gives a general description of a multivariate quadratic polynomial in n unknowns.

$$\sum_{1 \le i \le j \le n} \alpha_{i,j} x_i x_j + \sum_{1 \le i \le n} \beta_i x_i + \gamma \tag{2}$$

Multivariate Polynomials Systems and MP Problem. A multivariate polynomial system which is constructed with n unknowns and m polynomials is given in Equation 3.

$$MP(x_1, \ldots, x_n) = \{f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)\} \tag{3}$$

Solving such a system is called the MP problem. It is known as a NP-hard problem over a finite field [10] even for quadratics.

2.2 QUAD Stream Cipher

QUAD is a stream cipher proposed by Berbain et al. [3] and its security is based on the MQ assumption.

Keystream Generation. QUAD uses a random multivariate quadratic system as a pseudorandom number generator. We can construct QUAD using n unknowns and a system S consisting of m multivariate quadratic equations over $\mathbb{GF}(q)$. We denote such an instance as QUAD(q, n, r), where $r = m - n$ is the number of output keystream bits. The sketch illustrating the keystream generation algorithm is shown in Figure 1. The generator can then output an essentially endless stream of bits by repeating the above steps.

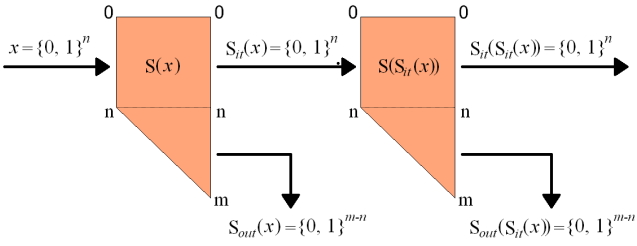

Fig. 1. Generating keystream

Computational Cost of QUAD. The computational cost of multivariate quadratics depends heavily on computing quadratic terms. The summation of quadratic terms requires $n(n+1)/2$ multiplications and additions. Therefore, the computational costs of evaluating one multivariate quadratic is $O(n^2)$. QUAD(q, n, r) requires to compute m multivariate quadratics. Since $m = kn$, the computational cost of generating key stream is $O(n^3)$.

Security Level of QUAD. The security level of QUAD is based on the MQ assumption. Berbain et al. [3] proofs that solving QUAD can be reduced to solving MQ problem. However, according to the analysis of QUAD using the XL-Wiedemann algorithm proposed by Yang et al. [11], QUAD$(256, 20, 20)$ has 45-bit security, QUAD$(16, 40, 40)$ has 71-bit security, and QUAD$(2, 160, 160)$ has less than 140-bit security.

Actually, secure QUAD requires larger constructions such as $\mathrm{QUAD}(2, 256, 256)$ or $\mathrm{QUAD}(2, 320, 320)$.

3 CUDA Computing

3.1 GPGPU

Originally, Graphics Processing Units (GPUs) are processing units for accelerating computer graphics. Recently, some online network games and simulators require very high amount of computer graphics computation. The GPU performance is growing to satisfy such requirements. As a result, GPU has a large amount of power for computation.

GPGPU is a technique for performing general-purpose computation using GPUs. In cryptography, it has been used to accelerate the encryption and decryption processes of various cryptosystems. For example, Manavski proposed an implementation of AES on GPU, which is 15 times faster than on CPU in 2007 [7]. Moreover, Osvik et al. presented a result of an over 30 Gbps GPU implementation of AES in 2010 [9]. On the other hand, GPGPU techniques have also been used for cryptanalysis. Bonenberger et al. used a GPU to accelerate polynomial generations in the General Number Field Sieve [4].

GPUs have a SIMD architecture, so it is better to handle several simple tasks simultaneously. On the other hand, the performance of a GPU core is not higher than CPU. Therefore, if we use GPU for sequential processing, it is not effective. In the GPGPU techniques, how to parallelize algorithms is an important issue.

3.2 CUDA

CUDA is NVIDIA's development environment for GPU based on C language. Various tools for using GPU existed before CUDA, but they often require hacking OpenGL or DirectX and hence are not easy to users. CUDA allows more efficient development of GPGPU by allowing the developers to work in the familiar C language.

In CUDA's terminology, "hosts" correspond to computers, while "devices" correspond to GPUs. In CUDA, the host controls the device. A kernel represents a unit of computation that a host asks a device to perform. In order to fully exploit the computational power of a GPU, a program needs to parallelize its computation in a kernel. A kernel handles some blocks in parallel. A block also handles many threads in parallel. Therefore, a kernel can handle many threads simultaneous.

NVIDIA GeForce GTX 580. In this paper, we use NVIDIA GeForce GTX 580 graphics card to perform our experiments. It is a high-end GPU in the GeForce 500 series, which was released in November, 2010. GTX 580 belongs to the Fermi architecture family, which is the successor to Tesla, the first-generation

CUDA-capable architecture. GTX 580 has 16 streaming multiprocessors (SMs), each of which consists of 32 CUDA cores, as opposed to 8 CUDA cores in Tesla.

4 Parallelization Strategies

4.1 Previous Works by Berbain et al.

Berbain et al. proposed several efficient implementation techniques of computing multivariate polynomial systems for multivariate cryptography [2]. In this paper, we use the following strategies from their work.

- Variables are treated as vectors. For example, C language defines `int` as a 32-bit integer variable. Therefore, we can use `int` as a 32-vector of boolean variables. This technique is often referred to as "bitslicing" in the literature.
- We precompute each quadratic term. Because in multivariate quadratic systems, we must compute the same $x_i x_j$ for every polynomial, so precomputing helps to save some computations.
- We compute only non-zero terms in $\mathbb{GF}(2)$. The probability of $x_i = 0$ is $1/2$, and the probability of $x_i x_j = 0$ is $3/4$. Therefore, we can reduce computational cost to about $1/4$.

4.2 Parallelizing on the GPU

In GPGPU, the most important point is the parallelization of algorithms. Because the performance of a single GPU core is worse than that of CPU, serial implementations with GPU are expected to be slower than CPU implementations.

Since the polynomials of a multivariate quadratic system are independent of each other, parallelization of a system is straightforward. Moreover, we parallelize the evaluation of each polynomial in a multivariate quadratic system. We propose two parallelization techniques, as shown in Figure 4.

The Basic Strategy of Parallelization. Let $t_{i,j} = \alpha_{i,j} x_i x_j$. Summation of quadratic terms can be considered as summation of every element of a triangular matrix, as shown on the left side of Figure 2. We assume that other elements from the matrix are zero. Therefore, we can compute summation of quadratic terms as summation of a rectangular matrix, as shown on the right side of Figure 2. Then, we can compute the summation as $\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{i,j} x_i x_j = \sum_{i=1}^{n} \sum_{j=1}^{n} t_{i,j}$ as follows.

1. We compute $S_k(x) = \sum_{i=1}^{n} t_{k,i}$ for all k in parallel.
2. We compute $\sum_{k=1}^{n} S_k(x)$.

However, such a strategy introduces some overhead caused by the extra unnecessary computations.

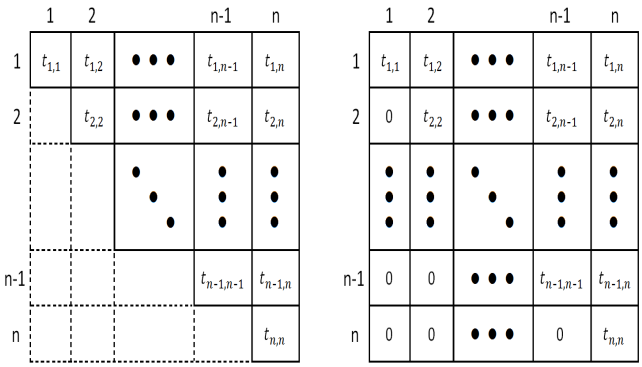

Fig. 2. Left: Evaluating quadratics on a triangular matrix. Right: Evaluating quadratics on a padded rectangle matrix.

Parallelization Method 1. Next we introduce the first strategy to reduce unnecessary computations. We reshape a triangular matrix to a rectangular matrix as shown in Figure 3, in which method of matrix reshaping is depicted. By this reshaping, we can efficiently reduce about 25% of the cost for evaluating a multivariate quadratic polynomial system.

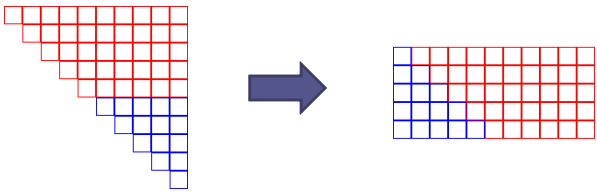

Fig. 3. Reshaping triangular to rectangular matrix

Parallelization Method 2. In the second strategy, we treat a polynomial as a vector as opposed to a matrix. Assuming that n_c is the number of GPU cores, we separate a vector into n_c-subvectors.

Moreover, we use the parallel reduction technique to compute all subvectors in parallel. The parallel reduction technique works as follows.

1. We substitute the length of subvectors for n_c.
2. We add $n_c/2 + i$-th elements to i-th elements.
3. We compute $n_c = n_c/2$.
4. While n_c is larger than 1, we iterate step 2 and 3.

The entire parallel reduction technique consists of $\log n_c$ iterations of the above steps. Therefore, we can evaluate polynomials efficiently.

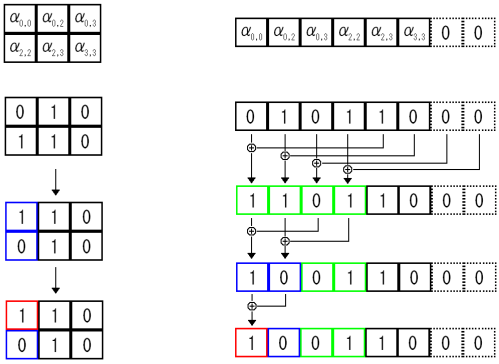

Fig. 4. Pallalelization strategies. Left: Strategy 1; right: Strategy 2

4.3 Optimization on GPU Architectures

On GPU implementations, we must consider its characteristics. Together, the cores on a GPU provide a tremendous amount of computing power, but each single GPU core is much slower than a CPU core. Therefore, we need to minimize the number of inactive GPU cores.

Optimization of Matrix Calculation. An NVIDIA GeForce GTX 580 GPU has 16 SMs, each of which has 32 CUDA cores. Since each SM handles 32 threads at a time, the number of threads should be an integral multiple of 32. In the same way, we should make sure that the algorithm can be handled by 16 SMs in parallel. Together, the total number of threads should be an integral multiple of $32 \times 16 = 512$.

In parallelization method 1, we can compute an summation in a polynomial as multiple co-summations of rows of a matrix. An n-unknown quadratic polynomial has $n(n+1)/2$ monomials. Then the long side of a rectangular matrix that is reshaped from an n-dimensional triangular matrix has n or $n+1$ elements. Although a number of a long side's elements can be counted in a process, counting incurs extra cost in the computation. Therefore, we assume that $n = 31k$, where k is a natural number. By handling a summation in a polynomial as a triangular matrix which elements are k-dimensional square submatrices, we can handle a summation as a 16×31 rectangle matrix, as shown in Figure 5. Thus we can parallelize the calculation of a matrix for 16 SMs with 32 CUDA cores per SM.

In parallelization method 2, we can parallelize a summation by the number of cores that can efficiently share data. In CUDA, we can share data in a block. Then we can parallelize a summation by 32 monomials on NVIDIA GeForce GTX 580. Therefore, we assume that $n = 32k$, where k is a natural number. Iterating time of parallelize reduction in a summation is $k(32k+1)/2$.

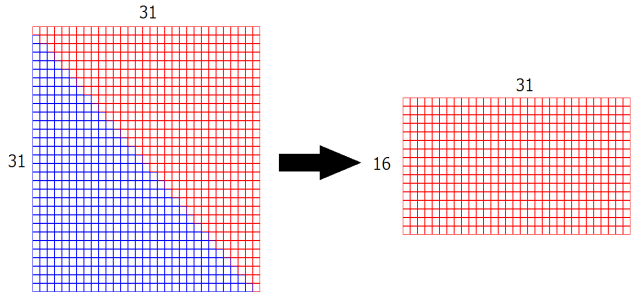

Fig. 5. Handling as a 16×31 matrix

Further Optimizations. In order to improve the efficiency, we need to break down the computation into small chunks of similar computations for parallel processing. Moreover, GPUs can't handle conditional branches efficiently, so we need to handle conditional branches differently than we do on CPU. In this case, we use a different kernel for each different number of non-zero terms. However, using a kernel for each possible number of non-zero terms would incur an extremely large amount of overhead. Therefore, we make kernels just every number of k. For example, for QUAD$(2, 512, 512)$, the maximum k is 17, so we need only 17 kernels.

4.4 Analysis of Potential Speedup

Parallelization Speedup. Originally, each polynomial in QUAD(q, n, n) requires $(n+1)\times(n+2)/2$ additions and multiplications. Moreover, QUAD(q, n, n) requires evaluation of $2n$ polynomials. Using the strategies proposed by Berbain et al. [3], we can compute each polynomial in QUAD(q, n, n) with $(n+1) \times (n+2)/8$ additions and multiplications. Therefore, we can compute QUAD(q, n, n) with $n/16$ times the cost of evaluating a single polynomial using 32-bit vectors.

Such techniques can be used by CPU implementation as well as GPU implementation. By parallelization on GPU, we can compute QUAD(q, n, n) in parallel. We can compute multiplications of a polynomial before additions. We can compute $\alpha_{i,j}x_ix_j$ in n multiplications time by we parallelize multiplications in each i and compute by every j. When we use a multivariate polynomial system over $\mathbb{GF}(2)$, we can compute multiplications by reducing monomials with a strategy of Berbain et al.

Parallelization method 1 with optimizations computes a summation in a polynomial by as a rectangle matrix, which elements are k-dimensional square submatrices. Since NVIDIA GeForce GTX 580 has 16×32 cores, each submatrices can be computed on each CUDA cores in parallel. Then, computational time of summations k-dimensional matrices is k^2 additions. Moreover, we should compute submatrices of every polynomials, then it takes mk^2; m is the number

of polynomials divided by 32. After that, we compute row co-summations in matrices. So we can compute row co-summations at one time, computational cost of row co-summations is 31 additions. When $m \leq 32$ (the number of polynomials ≤ 1024), we can compute row co-summations of all polynomials in once time. Finally, we compute a summation of row co-summations' result in 15 additions. Then, the computational costs of summations of a multivariate quadratic polynomial system can be denoted by $mk^2 + 46$ additions.

In parallelization method 2, we compute summations by parallel reductions. Parallel reductions can be computed co-summations of 32 elements on NVIDIA GeForce GTX 580 at once. Then, co-summations can be computed in 5 additions. Assuming $n = 32k$, we can compute co-summations of a polynomial by $k(32k + 1)/2$ times. Since we can compute 16 co-summations at once, actually, we can compute co- summations by $\lceil k(32k+1)/32 \rceil$ times. When $n \leq 512 = 32 \times 16$, we can compute co-summations at most $n/2$ times. Finally, we compute summations of co-summations' result of a polynomial as parallelization method 1. Then, the computational costs of summations of a multivariate quadratic polynomial system can be denoted by $(5m + 1)n/2$ additions.

5 Experiments

In this section, we present and discuss experiment results. We used NVIDIA GeForce GTX 580 GPU, as well as Intel Core $i7$ $875K$ CPU with 8 GB of memory.

5.1 Experiment Setup

We implement the evaluation of systems of $2n$-polynomials in n-unknowns for $n = 32, 64, 96, \ldots, 512$ on CPU and GPU. Finally, we compare the results of GPU and CPU implementations.

CPU Implementation. We implement evaluation of multivariate quadratic polynomial systems on the CPU by C language. We apply strategies of Berbain et al. [2] to CPU implementations.

GPU Implementation. We also apply them to GPU implementations. Moreover, we implement evaluation of multivariate quadratic polynomial systems with the parallelization strategies 1 and 2 as mentioned previously.

5.2 Experiment Results

We present the results of evaluation time of multivariate quadratic systems in Table 1. Evaluation time with the parallelization strategy 1 increase in the number of unknowns n rapidly. On the other hand, the parallelization strategy 2 increase in n slowly. Therefore, the strategy 2 is more efficient than the strategy 1.

Table 1. Evaluation time for multivariate quadratic polynomial systems

Unknowns n	Polynomials $2n$	Evaluation time (μs)		
		CPU	Strategy 1	Strategy 2
32	64	2.7	21.758	15.927
64	128	16.9	23.483	15.849
96	192	52.7	24.110	16.071
128	256	118.8	24.325	16.537
160	320	236.2	25.058	17.166
192	384	417.8	29.845	17.184
224	448	656.5	34.549	18.125
256	512	992.5	41.864	18.651
288	576	1505.4	52.442	19.408
320	640	2322.2	71.663	19.841
352	704	3409.2	90.264	20.236
384	768	4906.2	111.951	20.710
416	832	6666.4	146.331	21.420
448	896	8453.5	193.567	21.892
480	960	10545.1	256.538	22.259
512	1024	12902.0	336.299	22.785

Table 2. Encryption thoroughput of QUAD

		Throughput[Mbps]	
		QUAD$(2, 160, 160)$	QUAD$(2, 320, 320)$
CPU		0.646	0.131
GPU	Strategy 1	5.086	3.768
	Strategy 2	11.693	14.567
Berbain et al. [2]		8.45	—
Chen et al. [5]	CPU	—	6.1
	GPU	—	2.6

Futhermore, we compare result of QUAD implementations with Berbain et al. [2] and Chen et al. [5] on QUAD$(2, 160, 160)$ and QUAD$(2, 320, 320)$ in Table2. Unfortunately, QUAD$(2, 160, 160)$ with the parallelization strategy 2 is not so fast, compared with the results of Berbain et al. [2] However, QUAD$(2, 320, 320)$ with the parallelization strategy 2 is 2.3 times faster than Chen et al. [5] Moreover, it is faster than QUAD$(2, 160, 160)$. Therefore, we think that strategy 2 is suited to QUAD$(2, n, n)$, which n is a large number.

6 Conclusions

We presented two parallelization strategies for accelerating the evaluation of multivariate quadratic polynomial systems. A GPU implementation with parallelization

strategy 2 is the fastest implementation compared with previous works. Moreover, it might be suited to large finite fields. The security of QUAD depends on the scale of multivariate quadratic polynomial systems. We expect QUADs with the strategy 2 will become efficient and secure stream ciphers. Our approaches can be applied not only to the QUAD stream cipher but potentially also to other multivariate cryptosystems.

Acknowledgment. This work is partially supported by Japan Science and Technology agency (JST), Strategic Japanese-Indian Cooperative Programme on Multidisciplinary Research Field, which combines Information and Communications Technology with Other Fields, entitled "Analysis of Cryptographic Algorithms and Evaluation on Enhancing Network Security Based on Mathematical Science." The authors are grateful to Takashi Nishide for his valuable comments on our proposal.

References

1. Bard, G.V.: Algebraic Cryptranalysis. Springer (2009)
2. Berbain, C., Billet, O., Gilbert, H.: Efficient Implementations of Multivariate Quadratic Systems. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 174–187. Springer, Heidelberg (2007)
3. Berbain, C., Gilbert, H., Patarin, J.: QUAD: A Practical Stream Cipher with Provable Security. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 109–128. Springer, Heidelberg (2006)
4. Bonenberger, D., Krone, M.: Factorization of RSA-170, `http://public.rz.fh-wolfenbuettel.de/~kronema/pdf/rsa170.pdf`
5. Chen, M.-S., Chen, T.-R., Cheng, C.-M., Hsiao, C.-H., Niederhagen, R., Yang, B.-Y.: What price a provably secure stream cipher? In: Fast Software Encryption, 2010, Rump session (2010)
6. Liu, F.-H., Lu, C.-J., Yang, B.-Y.: Secure PRNGs from Specialized Polynomial Maps over Any $\mathbb{F}_q$. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 181–202. Springer, Heidelberg (2008)
7. Manavski, S.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: 2007 IEEE International Conference on Signal Processing and Communications, pp. 65–68 (2007)
8. NVIDIA CUDA, `http://developer.NVIDIA.com/object/CUDA.html`
9. Osvik, D.A., Bos, J.W., Stefan, D., Canright, D.: Fast Software AES Encryption. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 75–93. Springer, Heidelberg (2010)
10. Patarin, J., Goubin, L.: Asymmetric cryptography with s-boxes. In: First International Conference on Information and Communication Security, pp. 369–380 (1997)
11. Yang, B.-Y., Chen, O.C.-H., Bernstein, D.J., Chen, J.-M.: Analysis of `QUAD`. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 290–308. Springer, Heidelberg (2007)

A Program Sources of Evaluating a 320-Unknowns 640-Polynomials System

A.1 CPU Implementaions

```
#include <stdio.h>
#include <stdlib.h>

/* Extract non-zero unknowns */
int checkXX(int *x, int *x1) {
  int i, k;
  for (i = 0; i < 321; i++)   if (x[i]) { x1[k] = i; k++; }
  if (k & 0x01) { x1[k] = 321; k++; }
  return k;
}

/* Evaluate a system. */
int evaluateSystem(int *x1, int *S, int ***A, int k) {
  int i, j, l, tx1, tx2, tx3;
  for (i = 0; i < k / 2; i++) {
    tx1 = x1[i]; tx2 = x1[k-i-1];
    for (j = 0; j < k; j++) {
      tx3 = xx[j];
      for (l = 0; l < 20; l++) S[l] = S[l] ^ (A[l][tx1][tx3] | A[l][tx2][tx3]);
    }
  }
  return k;
}

int main(void) {
  int ***A, *x;     /* A: coefficients: 20x321x321, x: unknowns */
  int *x1, Nx1;     /* x1: non-zero unknowns. Nx1: the number of x1s */
  Nx1 = checkXX(x, x1);     /* Extract non-zero variables */
  (void)evaluateSystem(x1, KS, A, Nx1);     /* Evaluate a system */
  return 0;
}
```

A.2 GPU Implementations

Overview

```
#include <stdio.h>
#include <stdlib.h>
#include <cutil.h>
#include <cuda_runtime.h>

int main(int argc, char** argv) {
```

```
  int *x, *x1, *Gx1, Nx1, *S, *T1, *T2, *T3, *A;
  Nx1 = checkXX(x, x1);      /* Extract non-zero variables */
    (*) Evaluating a multivariate quadratic polynomial system
  return 0;
}
```

Evaluating a System with Parallelization Strategy 1

```
/* bx: blockIdx.x, by: blockIdx.y, tx: threadIdx.x, ty: threadIdx.y */
/* Reshape from a triangular matrixt to a rectangle matrix */
__global__ void SetArray6(int *xx, int *Axx) {
  __shared__ int Pxx1[192];
  int i, j, k;
  i = (by << 4) + bx;   j = (ty << 5) + tx;   k = i * 192 + j;
  Pxx1[j] = xx[j]; __syncthreads();
  if (i < j) {
    Axx[k] = (Pxx1[i] * 322) + Pxx1[j-1];
  } else {
    Axx[k] = (Pxx1[191-i] * 322) + Pxx1[191-j];
  }
}

/* Compute a row summation of a submatrix n=161-192 */
__global__ void ComputeSRow6(int *Axx, int *T, int *A) {
  int i, j, k, t1, t2, t3, t4, t5, t6, *p1, *p2;
  i = ((ty << 4) + bx) << 5) + tx;   j = 3072 * by;   k = i + j;
  p1 = Axx + (6 * i);   p2 = A + (103684 * by);
  t1 = *p1++; t2 = *p1++; t3 = *p1++; t4 = *p1++; t5 = *p1++; t6 = *p1;
 T[k] = *(p2+t1) ^ *(p2+t2) ^ *(p2+t3) ^ *(p2+t4) ^ *(p2+t5) ^ *(p2+t6));
}

/* Compute a culumn summation of a submatrix n=161-192 */
__global__ void ComputeSCul6(int *S, int *T) {
  int i, t1, t2, t3, t4, t5, t6, *p;
  i = (ty << 9) + (bx << 5) + tx;   p = S + (6 * i);
  t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++; t5 = *p++; t6 = *p;
  T[i] = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6;
}

/* Compute a row summation of a 32x16 matrix */
__global__ void ComputeRow(int *S, int *T) {
  int i, k1, k2, k3, k4, t1, t2, t3, t4, t5, t6, t7, t8, *p;
  i = (tx << 4) + bx;   p = S + (i << 5);
    t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++;
    t5 = *p++; t6 = *p++; t7 = *p++; t8 = *p++;
  k1 = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7 ^ t8;
```

```
    t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++;
    t5 = *p++; t6 = *p++; t7 = *p++; t8 = *p++;
  k2 = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7 ^ t8;
    t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++;
    t5 = *p++; t6 = *p++; t7 = *p++; t8 = *p++;
  k3 = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7 ^ t8;
    t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++;
    t5 = *p++; t6 = *p++; t7 = *p++; t8 = *p++;
  k4 = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7 ^ t8;
  T[i] = k1 ^ k2 ^ k3 ^ k4;
}

/* Compute a culumn summation of a 32x16 matrix */
__global__ void ComputeSCul(int *S, int *T) {
  int k1, k2, t1, t2, t3, t4, t5, t6, t7, t8, *p;
  p = S + (tx << 4);
    t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++;
    t5 = *p++; t6 = *p++; t7 = *p++; t8 = *p++;
  k1 = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7 ^ t8;
    t1 = *p++; t2 = *p++; t3 = *p++; t4 = *p++;
    t5 = *p++; t6 = *p++; t7 = *p++; t8 = *p++;
  k2 = t1 ^ t2 ^ t3 ^ t4 ^ t5 ^ t6 ^ t7 ^ t8;
  T[threadIdx.x] = k1 ^ k2;
}

(*) Evaluating a multivariate quadratic polynomial system in main function
  cudaMemcpy(Gx1, x1, 352 * sizeof(int), cudaMemcpyHostToDevice);
  switch (Nx1) {
    case 6: /* Nx1 = 161-192 */
      /* Bl[i]: dim3(16, i, 1), Th[i]: dim3(32, i, 1) */
      SetArray6<<<Bl[6], Th[6]>>>(Gx1, GAxx);
      ComputeSRow6<<<Bl[20], Th[6]>>>(GAxx, T1, A);
      ComputeSCul6<<<Bl[1], Th[20]>>>(T1, T2);
      break;
    case 5: /* Nx1 = 129-160 */
      ...
  }
  ComputeRow<<<16, 20>>>(T2, T3);
  ComputeCul<<<1, 20>>>(T3, S);
```

Evaluating a System with Parallelization Strategy 2

```
/* bx: blockIdx.x, by: blockIdx.y, tx: threadIdx.x */
/* Divide a quadratic polynomial as sub-vector */
__global__ void setArray20(int *x1, int *Axx, int *A) {
```

```
    __shared__ int X[320];
    int NH, x0, i, j, Ai;
    X[tx] = x1[tx+1]; __syncthreads();
    x0 = x1[0];   Ai = ((X[bx] * (321-X[bx])) >> 1) + X[tx] - X[bx] - 1;
    i = (bx * xx0) - (((bx-1) * bx) >> 1) + tx -bx - 1;   NH = ((x0-1)*x0) >> 1;
    if (tx < bx)
       for (j = 0; j < 20; j++) { Axx[i] = A[Ai]; i += NH; Ai += 103684; }
   }
}

/* Compute summations of sub-vectors */
__global__ void compLog(int *Axx, int *T) {
  __shared__ int sm[512];
    int i = (by*gridDim.x + bx) << 9;
    smem[tx]=Axx[i+tx]; __syncthreads();
    if (tx < 256) sm[tx] ^= sm[tx+256]; __syncthreads();
    if (tx < 128) sm[tx] ^= sm[tx+128]; __syncthreads();
    if (tx < 64) sm[tx] ^= sm[tx+64]; __syncthreads();
    if (tx < 32) {
       sm[tx] ^= sm[tx+32]; sm[tx] ^= sm[tx+16]; sm[tx] ^= sm[tx+8];
       sm[tx] ^= sm[tx+4]; sm[tx] ^= sm[tx+2]; sm[tx] ^= sm[tx+1];
    }
    if (tx == 0) T[i+bx] = sm[0];
}

/* Compute summations of a system */
__global__ void compLastLog(int *T1, int *S) {
  __shared__ int sm[256];
    int i = bx << 8;
    sm[tx] = T1[threadIdx.x+i];   __syncthreads();
    if (tx < 128) sm[tx] ^= sm[tx+128]; __syncthreads();
    if (tx < 64) sm[tx] ^= sm[tx+64]; __syncthreads();
    if (tx < 32) {
       sm[tx] ^= sm[tx+32]; sm[tx] ^= sm[tx+16]; sm[tx] ^= sm[tx+8];
       sm[tx] ^= sm[tx+4]; sm[tx] ^= sm[tx+2]; sm[tx] ^= sm[tx+1];
    }
    if (tx == 0) S[bx]=sm[0];
}

(*) Evaluating a multivariate quadratic polynomial system in main function
      cudaMemcpy(Gx1, x1, (x1[0]+1) * sizeof(int), cudaMemcpyHostToDevice);
      setArray20<<<x1[0], x1[0]>>>(Gx1, GAxx, A);
      compLog<<<Bl[Nx1], 512>>>(GAx1, T1); /* Bl[i]: dim3(i,20,1) */
      compLastLog<<<20, 256>>>(T1, S);
```