Implementation of Efficient Operations over GF(232) Using Graphics Processing Units

Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai

Implementation of Efficient Operations over $GF(2^{32})$ using Graphics Processing Units

Satoshi Tanaka[1,2], Takanori Yasuda[2], and Kouichi Sakurai[1,2]

[1] Kyushu University, Fukuoka Japan
{tanasato@itslab.inf,sakurai@csce}.kyushu-u.ac.jp
[2] Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka, Japan
yasuda@isit.or.jp

Abstract. Evaluating non-linear multivariate polynomial systems over finite fields is an important subroutine, e.g., for encryption and signature verification in multivariate public-key cryptography. The security of multivariate cryptography definitely becomes lower if a larger field is used instead of $GF(2)$ given the same number of bits in the key. However, we still would like to use larger fields because multivariate cryptography tends to run faster at the same level of security if a larger field is used. In this paper, we compare the efficiency of several techniques for evaluating multivariate polynomial systems over $GF(2^{32})$ via their implementations on graphics processing units.

Key words: Efficient implementation; multivariate public-key cryptography, GPGPU

1 Introduction

1.1 Background

The security of the public-key cryptography depends on the complexity of math problems. Of course, the public-key cryptography uses well-known hard problems, however, some of them will be insecure in the future. For example, the security of RSA public-key cryptosystem[5] is based on the complexity of integer factorization. However, quantum computers can solve it in polynomial time[6]. Hence, RSA public-key cryptosystem is going to be insecure in the future.

Therefore, some researchers study post-quatum cryptosystemsas, none of the known quantum algorithms can solve the problem in polynomial time.. The multivariate public-key cryptosystem (MPKC) is expected a candidate of post-quatum cryptosystems. Starting from the seminal work on MPKCy [3], researchers have provided efficient signature scheme [2] and provably secure symmetric-key cipher [1]. The security of MPKC is based on the complexity of solving non-linear multivariate quadratic polymial equations over a finite field (MP), which is known as NP-complete.

Evaluating non-linear multivariate polynomial systems over a finite field is an important subroutine in MPKC. The core operations in evaluating a multivariate

polynomial system are additions and multiplications over a finite field. Therefore, accelerating these operations will accelerate all multivariate cryptosystems. On the other hand, the security of a MPKC depends on the numbers of unknowns and polynomials, as well as the order of the finite field.

1.2 Related works

Typically, $GF(2)$ and its extension fields are used in many multivariate cryptosystems, as additions over them only need cheap XOR operations. In the past, uses of $GF(2)$, $GF(2^4)$ and $GF(2^8)$ have been considered for multivariate cryptosystems [1]. For $GF(2^{16})$, multiplications are implemented by using intermediate fields of $GF(2^{16})$ on CPU and GPU[7]. Figure 1 shows the result of [7].

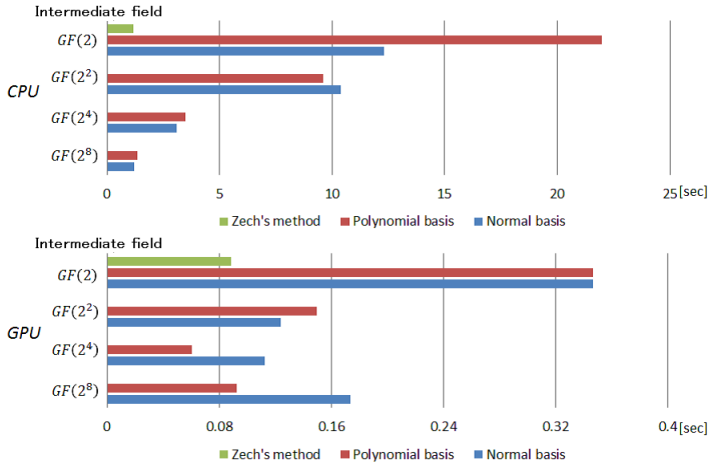

Fig. 1. The result of multiplications over $GF(2^{16})$[7]

1.3 Our contribution

In this paper, we have simply extended to $GF(2^{32})$ case from $GF(2^{16})case$[7]. We study arithmetic operations over $GF(2^{32})$ and their implementation because we expect that they will be used for MPKC in the future.

In the following, we compare four methods for multiplications over $GF(2^{32})$ with CPU and graphics processing units (GPU) implementations.

2 Operations over Extension Fields

2.1 Extension of Finite Field

Let p be a prime, and $F = GF(p^m)$, $K = GF(p^n)(m, n \geq 1)$. Then K is an extension field of F, where $m \mid n$. We asuume that $k = n/m$, $q = p^m$. Then the

Frobenius map σ_0 of K/F is defined as:

$$\sigma_0(a) = a^q.$$

σ_0 is a map from K to K. It satisfies:

1. $\sigma_0(a+b) = \sigma_0(a) + \sigma_0(b)\ \forall a, b \in K$;
2. $\sigma_0(ab) = \sigma_0(a)\sigma_0(b)$; and
3. $\sigma_0(\alpha) = \alpha\ \forall \alpha \in F$.

The Galois group $\mathrm{Gal}(K/F)$ is given by:

$$\mathrm{Gal}(K/F) = \{\sigma : K \mapsto K : automorphism | \sigma(\alpha) = \alpha\ (\forall \alpha \in F)\}.$$

The elements of a Frobenius map σ_0 are the elements of the Galois group $\mathrm{Gal}(K/F)$, in which the group operation is simply function composition. Moreover, $\mathrm{Gal}(K/F)$ is a cyclic group generated by the Frobenius map σ_0, i.e.,

$$\mathrm{Gal}(K/F) = \{\sigma_0^0, \sigma_0^1, \sigma_0^2, \ldots, \sigma_0^{k-1}\}.$$

2.2 Additions over Extension Fields

K can be represented as a set of polynomials of degrees less than k over F. Now, we choose a degree-k irreducible polynomial over F:

$$f_0(x) = x^k + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0, a_0, \ldots, a_{k-1} \in F. \tag{1}$$

Then, K can be described by following the formula:

$$K = \{c_{k-1}x^{k-1} + \cdots + c_1 x + c_0 | c_0, \ldots, c_{k-1} \in F\}.$$

Addition $e_1 + e_2$ can be represented as:

$$e_1 + e_2 := e_1(x) + e_2(x) \mod f_0(x),$$

where $e_1, e_2 \in K$. Therefore, we can compute addtions over extension fields by summations of coefficients of polynomials over the base field.

2.3 Multiplications over Extension Fields

Polynomial Basis Let K be a set of polynomials over F. Then, we can compute multiplication $e_1 * e_2$, where $e_1, e_2 \in K$, by:

$$e_1 * e_2 := e_1(x) * e_2(x) \mod f_0(x), \tag{2}$$

Zech's Method $K^* := K \setminus \{0\}$ is a cyclic group. Therefore, K^* has a generator $\gamma \in K^*$, and $K = \langle \gamma \rangle$. Then we can represent any element in K^* as γ^ℓ, where ℓ is an integer. In particular, $\gamma^\ell \neq \gamma^{\ell'}$, $0 \leq \ell \neq \ell' \leq p^n - 2$. In this way, K^* can be represented by $[0, p^n - 2]$. Then, multiplications over K^* can be computed by integer additions modulo $p^n - 1$.

Normal Basis There exists an $\alpha \in K$ for a finite Galois extension K/F such that $\{\sigma(\alpha)|\sigma \in \mathrm{Gal}(K/F)\}$ is an F-basis of K, which is called a normal basis of K/F. A normal basis of K/F can thus be denoted by:

$$\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{k-1}}\}. \tag{3}$$

Then, an element $a \in K$ can uniquely be written as:

$$a = c_0\alpha + c_1\alpha^{p^m} + \cdots + c_{k-1}\alpha^{p^{(k-1)m}},\ c_0, \ldots, c_{k-1} \in F. \tag{4}$$

Let $a = [c_0, c_1, \ldots, c_{k-1}]_n \in K$ be defined in Eq. (4). Then Frobenius map $\sigma_0(a)$ as:

$$\sigma_0(a) = a^q = [c_{k-1}, c_0, c_1, \ldots, c_{k-2}]_n. \tag{5}$$

In other words, $\sigma_0(a)$ is simply a right circular shift [4].

Furthermore, let $a = [c_0, c_1, \ldots, c_{k-1}]_n, b = [c'_0, c'_1, \ldots, c'_{k-1}]_n \in K$, and the result of the multiplication $a * b$ be $[d_0, d_1, \ldots, d_{k-1}]_n$. Then, every d_i, where $0 \leq i < k$, can be computed by evaluating the quadratic polynomials of $c_0, c_1, \ldots, c_{k-1}, c'_0, c'_1, \ldots, c'_{k-1}$ over F. Let $d_i = p_i(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1})$, $\forall 0 \leq i < k$. According to Eq. (5), we can compute $\sigma_0(a * b)$ by:

$$\begin{aligned}
\sigma_0(a * b) &= [d_{k-1}, d_0, d_1, \ldots, d_{k-2}]_n \\
&= \sigma_0(a) * \sigma_0(b) \\
&= [c_{k-1}, c_0, \ldots, c_{k-2}]_n * [c'_{k-1}, c'_0, \ldots, c'_{k-2}]_n \\
&= [p_0(c_{k-1}, c_0, \ldots, c_{k-2}, c'_{k-1}, c'_0, \ldots, c'_{k-2}), \ldots, \\
&\quad\ p_{k-1}(c_{k-1}, c_0, \ldots, c_{k-2}, c'_{k-1}, c'_0, \ldots, c'_{k-2})]_n.
\end{aligned} \tag{6}$$

By comparing coefficients, d_{k-2} can be computed by:

$$d_{k-2} = p_{k-1}(c_{k-1}, c_0, c_1, \ldots, c_{k-2}, c'_{k-1}, c'_0, c'_1, \ldots, c'_{k-2}),$$

with Eq. (6). In the same way, we can compute $\sigma_0^2(a * b), \ldots$, for all i by doing right circular shifts and computing all the d_r's by evaluating p_{k-1}.

Multiplication Tables We create a multiplication table by offline precomputing all combinations of multiplications over K. Then, we can compute multiplications by looking up the multiplication table.

2.4 Analysis of Multiplication Algorithms

Polynomial Basis Let $e_1, e_2 \in K$ be $c_{k-1}x^{k-1} + \cdots + c_1x + c_0$ and $c'_{k-1}x^{k-1} + \cdots + c'_1x + c'_0$, respectively, and an irreducible polynomial f_0 be defined as Eq. (1). Then, the addition $e_1 + e_2$ needs k additions over F. On the other hand, the multiplication $e_1 * e_2$ can be computed by:

$$e_1 * e_2 = c_{k-1}c'_{k-1}x^{2k-2} + \cdots + c_0c'_0 \mod f_0(x).$$

In this method, we need compute multiplications $c_i c'_j$ for $0 \leq i, j < k$ and summations $\sum_{i+j=t, i,j \geq 0} c_i c'_j$ for $0 \leq t \leq 2(k-1)$ over F. The summation $\sum_{i+j=t, i,j \geq 0} c_i c'_j$ needs t and $2k-t-2$ additions for $0 \leq t < k$ and $k \leq t \leq 2(k-1)$ respectively. Therefore, it needs $(k-1)^2$ additions and k^2 multiplications over F if schoolbook multiplication is used. Moreover, $e_1 * e_2$ takes $k\lceil \log_2 p^m \rceil \simeq n\lceil \log_2 p \rceil$ bits of memory.

Zech's Method In this method, a multiplication over K needs one integer addition modulo $k-1$. On the other hand, addition is not simple. Therefore, we convert it to the polynomial basis for additions and convert it back to the cyclic group representation for multiplications. Therefore, a multiplication needs three such conversions. One is for converting from polynomial to cyclic group representation, while the other is the opposite. Therefore, an addition takes k additions over F, similar to the polynomial basis representation, and a multiplication needs one integer addition modulo $k-1$ plus three conversions between polynomial and cyclic group representations. Moreover, since the tables represent maps from K to itself, Zech's method needs $2p^n\lceil \log_2 p^n \rceil$ bits of memory.

Normal Basis Let $a, b \in K$ be $[c_0, \ldots, c_{k-1}]_n$ and $[c'_0, \ldots, c'_{k-1}]_n$, respectively. An addition over K takes k additions over F, similar to the polynomial basis method. On the other hand, a multiplication $a * b$ takes $2(k-1)$ right circular shift operations and k evaluations of a fixed (quadratic) polynomial $p_{k-1}(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1})$. An evaluation of a quadratic polynomial takes $k^2 - 1$ additions and $2k^2$ multiplications over F. We can further speed up such an evaluation by precomputing common multiplications $c_i c_j$ over F, where $0 \leq i, j \leq k-1$. Moreover, we can modify formula for c_i, c_j, c'_i, c'_j to :

$$\begin{aligned} &p_{k-1}(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1}) \\ &= c_0 c'_0 + \textstyle\sum_{0 \leq i < j < k} s_{i,j}(c_i + c_j)(c'_i + c'_j) \; \forall (i,j), s_{i,j} \in F, \end{aligned}$$

where $i \neq j$. Therefore, a multiplication over K needs $k(k-1)(k+2)/2$ additions and $k(k^2+1)/2$ multiplications over F plus $2(k-1)$ right circular shift operations. Moreover, the normal basis method needs $(k^2-k+2)\lceil \log_2 p^m \rceil/2$ bits of memory.

Multiplication Tables An addition over K can be computed using k additions over F. On the other hand, a multiplication over K needs only one table look-up. Since the entire multiplication table needs to store every possible combination of multiplications over K, this method requires $p^{2n}\lceil \log_2 p^n \rceil$ bits of memory.

3 GPGPU via CUDA

A graphics processing unit (GPU) is a special-purpose processor for accelerating computer graphics computations. Due to the nature of its computational tasks, GPUs can handle many operations in parallel in a high speed.

General-purpose GPU (GPGPU) computing is a technique that uses GPUs for general-purpose computation. Since GPUs are designed for single instruction multiple data (SIMD) operations, they are quite efficient for parallel processing. On the other hand, they are not so efficient when there is limited amount of parallelism. Therefore, the most important task in GPGPU is to identify or manufacture parallelism in the algorithms to be implement.

3.1 CUDA API

CUDA is a development environment for NVIDIA's GPUs [8]. Before CUDA, GPGPU must have been done via hacking OpenGL or DirectX. These tools are not easy to use for non-experts of graphics programming, which was changed by the introduction of CUDA.

In CUDA, hosts correspond to computers, whereas devices correspond to GPUs. In CUDA, a host controls one or more devices attached to it. A kernel is a function that the host uses to control the device(s). In earlier CUDA, only one kernel can run at a time, and a program launches a kernel whenever parallel processes is needed. A kernel handles several number of blocks in parallel. A block also handles multiple threads in parallel. Therefore, a kernel can handle many threads simultaneously.

3.2 Parallelization for CUDA implementations

In CUDA API, we should consider how parallelize algorithms on GPUs. Especially, the number of threads in each block is important. This number is defined by GPUs. For example, NVIDIA GeForce 580 GTX can use 1,024 threads in each block registers. On the other hand, this number is alsor confined by the number of registers in blocks. Every thread use different registers for variables in kernels. When the total number of registers in every thread is greater than the number of registers in blocks, GPUs shows unexpected behavior (e.g. GPUs are halted). Therefore, we should parallelize algorithms for threads lest numbers of threads is greater than these GPU limitations.

4 Multiplications over $GF(2^{32})$

We can use XOR operations for computing additions over $GF(2)$. Moreover, multiplications over $GF(2)$ can be computed by the logical conjunctions AND.

4.1 Costs of multiplications over $GF(2^{32})$

Table 1 shows the costs of multiplications over $GF(2^{32})$.

The polynomial basis method and the normal basis method need a lot more computational cost. On the other hand, Zech's method and using multiplication table are impractical, as it needs 32 GBytes and 64 EBytes of memory space, respectively.

Table 1. Costs of multiplications over $GF(2^{32})$.

Methods	Computational cost	Memory space
Polynomial basis method	961 XOR + 1,024 AND + 1 MOD	4 Byte
Zech's method	1 ADD + 1 MOD + 3 LOOKUP	32 GByte
Normal basis method	16,864 XOR + 16,400 AND + 2 SHIFT	125 Byte
Multiplication table	1 LOOKUP	64 EByte

4.2 Using Intermediate Fields

Although the multiplication table method is impractical for $GF(2^{32})$, it is possible for $GF(2^8)$, as the table there requires only 256 KByte. Also, Zech's method over $GF(2^{32})$ needs just 256 KByte. Here, we consider a method using an intermediate field $GF(2^l)$ for $GF(2^{32})/GF(2)$, where $l = 2, 4, 8, 16$. In this method, we can compute multiplications over $GF(2^{32})$ by considering it as an extension field over $GF(2^l)$ and by using the polynomial basis method or the normal basis method. For example, since the extension degree $k = 4$ for $GF(2^{32})/GF(2^8)$, we can compute multiplication over $GF(2^{32})$ by 9 additions over $GF(2^8)$(72 XORs), 16 table look-ups, and one modulo over $GF(2^8)$ with the polynomial basis method, or 288 XORs and 34 table look-ups with the normal basis method.

Similarly, we estimate the computational costs of multiplications over $GF(2^{32})$ using $GF(2^2)$, $GF(2^4)$ or $GF(2^{16})$. We show the computational costs of multiplications over $GF(2^{32})$ using these intermediate fields in Table 2.

Table 2. Costs of multiplications over $GF(2^{32})$ using intermediate fields.

Intermediate field $GF(2^l)$	Computation method		Computational cost				Memory space
	$GF(2^l)/GF(2)$	$GF(2^{32})/GF(2^l)$	XOR	LOOKUP	MOD	ADD	
$GF(2^2)$	Multiplication table	Polynomial basis	450	512	1	-	6B
		Normal basis	4,320	4,112	-	-	35B
$GF(2^4)$		Polynomial basis	196	256	1	-	132B
		Normal basis	1,120	1,040	-	-	143B
$GF(2^8)$		Polynomial basis	72	128	1	-	64kB+4B
		Normal basis	288	272	-	-	64kB+4B
$GF(2^{16})$	Zech's method	Polynomial basis	16	12	4	4 + 1	256kB+4B
		Normal Basis	64	15	5	5	256kB+4B

5 Experimentation

We implement the three basic multiplication methods, namely, polynomial basis, Zech's method, and normal basis, over $GF(2^{32})$ on CPU and GPU. We evaluate and compare the running time of 67,108,864 multiplications with random elements over $GF(2^{32})$ for each methods. Similarly, we also implement and perform the same experiment using intermediate fields as follows:

1. Multiplication table + polynomial basis method:
   $GF(2^{32})/GF(2^k)/GF(2)$ $(k = 1, 2, 4, 8)$
2. Multiplication table + normal basis method:
   $GF(2^{32})/GF(2^k)/GF(2)$ $(k = 1, 2, 4, 8)$
3. Zech's method + polynomial basis method:
   $GF(2^{32})/GF(2^{16})/GF(2)$
4. Zech's method + normal basis method:
   $GF(2^{32})/GF(2^{16})/GF(2)$

Moreover, we describe primitive polynomials for each field extensions.

1. $GF(2^{32})/GF(2)$:
   $Y^{32} + Y^{22} + Y^2 + Y + 1 = 0$
2. $GF(2^{32})/GF(2^2)/GF(2)$:
   $Y^{16} + Y^3 + Y + X = 0$
3. $GF(2^{32})/GF(2^4)/GF(2)$:
   $Y^8 + Y^3 + Y + X = 0$
4. $GF(2^{32})/GF(2^8)/GF(2)$:
   $X^4 + Y^2 + (X + 1)Y + (X^3 + 1) = 0$
5. $GF(2^{32})/GF(2^{16})/GF(2)$:
   $Y^2 + Y + X^{13} = 0$

5.1 Environment of Implementation

All the experiments are performed on Ubuntu 10.04 LTS 64bit, Intel Core i7 875K and NVIDIA GeForce 580 GTX with 8 GBytes of DDR3 memory.

Table 3 shows constructions of parallelizations of 67,108,864 multiplications on NVIDIA GeForce 580 GTX for each multiplication method.

Table 3. Constructions of parallelizations on NVIDIA GeForce 580 GTX.

Intermediate field $GF(2^l)$	Polynomial basis			Normal basis		
	Block	Thread	Iteration	Block	Thread	Iteration
$GF(2)$	32,768	32	64	32,768	32	64
$GF(2^2)$	32,768	128	16	32,768	128	64
$GF(2^4)$	32,768	512	4	32,768	512	4
$GF(2^8)$	32,768	512	4	32,768	512	4
$GF(2^{16})$	32,768	512	4	32,768	512	4

5.2 Experiment Results

We show the result of implementations in Figure 2.

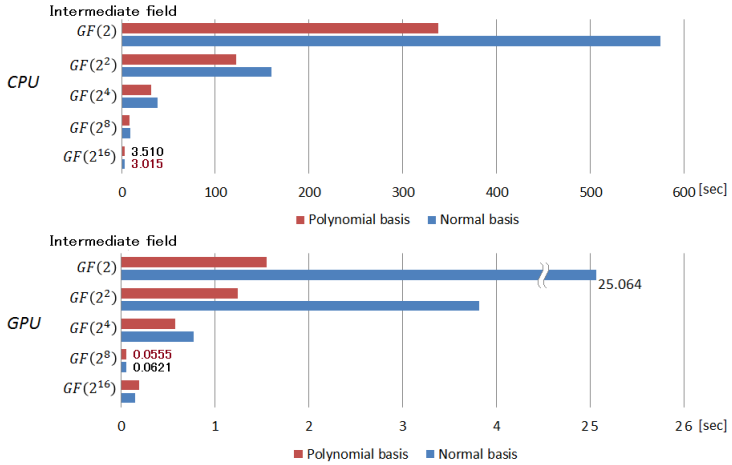

Fig. 2. Experimental result of multiplications over $GF(2^{32})$

In CPU implementations, the normal basis method using $GF(2^{16})$ is the fastest, possibly because it needs the fewest computations among every method.

On the other hand, in GPU implementations, the polynomial basis method using $GF(2^8)$ for the intermediate field is the fastest. We believe that the GPU cannot efficiently access the tables in Zech's method over $GF(2^8)$, as these tables are too large to fit into the fast memory on GPU.

6 Conclusions

In this work, we have implemented and compare of several multiplication methods over $GF(2^{32})$. In CPU implementations, the normal basis method using $GF(2^{16})$ is the fastest. The second fastest is the polynomial method over $GF(2^{16})$. On the other hand, for GPU implementations, it seems that $GF(2^8)$ is a very efficient intermediate field for building extension fields over it. Comparing CPU and GPU implementations, the fastest GPU implementation is about 49 times faster than the fastest one on the CPU. However, our GPU implementations are just parallelized CPU ones. We expect our research makes enables to enhance the security of MPKC by converting small finite fields (like $GF(2^8)$) to $GF(2^{32})$.

In future work, we would like to discuss optimizing multiplication methods for both CPU and GPU. Moreover, we would like to evaluate MPKCs using our implementaions.

Acknowledgement

This work is partly supported by "Study on Secure Cryptosystem using Multivariate Polynomial System," no. 0159-0172, Strategic Information and Communications R&D Promotion Programme, the Ministry of Internal Affairs and Communications, Japan, Grants for Excellent Graduate Schools, the Ministry of Education Culture, Sports, Science and Technology, Japan and "The Constitutive Theory of Non-Interactive Zero-Knowledge Proof based on Probability Certification and Practical Enhancement to the Cryptography," no. 25540004, Grant-in-Aid for Challenging Exploratory Research, Japan Society for the Promotion of Science. The authors are grateful to Xavier Dahan for his valuable comments on our proposal.

References

1. Berbain, B., Gilbert, H., Pataring, J.: QUAD: a Pratical Stream Cipher with Provable Security. In EUROCRYPTO'06, LNCS, vol.4004, pp.109-128, Springer, 2006.
2. Jintai, D., Dieter, S.: Rainbow, a new multivariable polynomial signature scheme. In Applied Cryptography and Network Security - ACNS 2005, LNCS, vol. 3531, pp.164-175, Springer, 2005.
3. Matsumoto, T., Imai, H.: Public quadratic polynomial-tuples for efficient signature-varification and message-encryption. LNCS, vol.330, Proceedings of EUROCRYPT'88, pp. 419-453, Springer, 1988.
4. Menezes, A.J., van Oorschot, Vanstone, S.A.: Handboook of Applied Cryptography, CRC Press 1997.
5. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. In Communications of the ACM, Vol. 21, No. 2, pp. 120?126, 1978.
6. Shor, P. W.: Algorithms for quantum computation. In discrete logarithms and factoring, 1994.
7. Tanaka, S., Yasuda, T., Yang, B.-Y., Cheng. C.-H., Sakurai, K.: Efficient Computing over $GF(2^{16})$ using Graphics Processing Unit. In the Seventh International Workshop on Advances in Information Security, 2013.
8. NVidia Developer Zone
   `https://developer.nvidia.com/category/zone/cuda-zone`