Performance Evaluation of RSA and NTRU over GPU with Maxwell and Pascal Architecture

Xian-Fu Wong¹, Bok-Min Goi¹, Wai-Kong Lee²,
and Raphael C.-W. Phan³

¹*Lee Kong Chian Faculty of Engineering and Science, Universiti Tunku Abdul Rahman, Sungai Long, Malaysia*

²*Faculty of Information and Communication Technology, Universiti Tunku Abdul Rahman, Kampar, Malaysia*

³*Faculty of Engineering, Multimedia University, Cyberjaya, Malaysia*
E-mail: wongxf92@utar.my; {goibm; wklee}@utar.edu.my;
raphael@mmu.edu.my

Received 2 September 2017; Accepted 22 October 2017;
Publication 20 November 2017

Abstract

Public key cryptography important in protecting the key exchange between two parties for secure mobile and wireless communication. RSA is one of the most widely used public key cryptographic algorithms, but the Modular exponentiation involved in RSA is very time-consuming when the bit-size is large, usually in the range of 1024-bit to 4096-bit. The speed performance of RSA comes to concerns when thousands or millions of authentication requests are needed to handle by the server at a time, through a massive number of connected mobile and wireless devices. On the other hand, NTRU is another public key cryptographic algorithm that becomes popular recently due to the ability to resist attack from quantum computer. In this paper, we exploit the massively parallel architecture in GPU to perform RSA and NTRU computations. Various optimization techniques were proposed in this paper to achieve higher throughput in RSA and NTRU computation in two GPU platforms. To allow a fair comparison with existing RSA implementation techniques, we proposed to evaluate the speed performance in the best case

Journal of Software Networking, 201–220.

doi: 10.13052/jsn2445-9739.2017.010

This is an Open Access publication. © 2017 the Author(s). All rights reserved.

(least '0' in exponent bits), average case (random exponent bits) and worse case (all '1' in exponent bits). The overall throughput achieved by our RSA implementation is about 12% higher in random exponent bits and 50% higher in all 1's exponent bits compared to the implementation without signed-digit recoding technique. Our implementation is able to achieve 17713 and 89043 2048-bit modular exponentiation per second on random exponent bits in GTX 960M and GTX 1080, which represent the two state of the art GPU architecture. We also presented the implementation of NTRU in this paper, which is 62.5 and 38.1 times faster than 2048-bit RSA in GTX 960M and GTX 1080 respectively.

Keywords: RSA, NTRU, GPU, Signed-digit recoding, Montgomery exponentiation.

1 Introduction

Mobile and wireless communication technologies are growing by leaps and bounds in the past decade, which foster the emergence of Cloud Computing. One of the key aspects of these technologies is the security features offered to protect the user's privacy during communication. RSA is a Public Key Cryptosystem (PKC) widely used for encrypting messages or generating digital signatures to provide authentication feature in secure communication. The core operation in RSA is modular exponentiation, which can be represented by the equation $C = M^e \bmod N$, where M is the plaintext, C is the ciphertext, e is the exponent and N is a large prime number. Computing modular exponentiation is non-trivial when the integer size is large. For example, RSA is consider insecure if the bit-size is smaller than 1024-bit; this implies that the implementation of modular exponentiation in RSA needs to handle integers of at least 1024-bit.

A straightforward implementation of 1024-bit modular exponentiation with the exponent as large as 1024-bit will require a lot of modular multiplication to be performed. Moreover, each modular multiplication involves a 2048-bit (2049-bit if the carry present) product followed by an expensive division. To simplify the computation, Montgomery Multiplication [7] was introduced to avoid the expensive division by replacing it with cheaper shift operations. Binary method [8] is also widely used to reduce the number of modular multiplication needed to compute a modular exponentiation.

The emergence of Quantum Computer (QC) posed a serious challenge on many current PKC that builds upon factorization (RSA) and elliptic curve (ECC) hard problems. Nth-degree truncated polynomial ring (NTRU) [9] is an emerging PKC that is resistant to QC attack and widely known as one of the strong candidates for post quantum cryptography. NTRU is built upon the shortest vector problem in a lattice, which is not known to be susceptible to QC attack. The main operation in NTRU encryption and decryption is polynomial multiplication, which is known to be faster than RSA. NTRU can be computed even faster if coefficients of the polynomial used are of small values (binary or ternary) and sparse. It is also being standardized under IEEE [10].

Graphics Processing Unit (GPU) is massively parallel processors capable of computing thousands of threads in parallel. GPU has been used in various applications to accelerate cryptographic algorithms [1–4]. This motivates us to explore the possibility to compute RSA in parallel using GPU, which is very useful for server applications that need to handle millions of authentications from the clients simultaneously, such as assessing personal and business information through mobile devices.

GPU was used in computing cryptographic algorithms including RSA. Neves et al. [1] analyzed different methods to interleave Montgomery multiplication on GPU with Tesla architecture and found FIOS and FIPS method is better than the most commonly used CIOS method. They were able to achieve throughput of 41426 512-bit modular exponentiations per second. On the other hand, Leboeuf et al. [2] mentioned that CIOS method is more suitable for GPU with Fermi architecture; their implementation was able to achieve throughput of 1.24 to 1.72 times greater than the fastest implementation on the same GPU. Recently, Emmart and Weems [3] introduced four methods to perform multiply-accumulate instruction across multiple generations of GPU (Tesla, Fermi, Kepler, and Maxwell) and compared their performance. They found that every GPU architecture achieved its best performance with different methods, as the clock per operation for the multiply-accumulate instruction are different for some GPU. In another work, Emmart et al. [4] discovered the optimal use of the instructions, memory, registers, and threads on the GPU with Maxwell architecture. In addition, they claimed that their implementation is much faster than the state of the art by using the row-oriented multiply and reduce. However, the work from Emmart et al. [3, 4] do not consider the signed digit recoding technique, which can further improve the performance of Montgomery exponentiation. Wu et al. [5] proposed a technique CMM-SDR

which improve the conventional signed-digit recoding method to accelerate modular exponentiation. However, no experimental result was presented in their paper.

Based on our observation of previous works [1–4], we found that the experiments were only performed based on random bit exponent. Since the number of bit “1” occurs in the exponent can greatly reduce the performance of modular exponentiation, it is not fair to evaluate the performance solely based on random bit exponent that is not reproducible. Hence, we proposed to evaluate the performance by including the best case (least ‘0’ in exponent bits) and worst case (all ‘1’ in exponent bits). This gives a better understanding of the actual performance of an implementation as well as providing a fair and reproducible comparison.

In this paper, we implemented the CMM-SDR method proposed by Wu et al. [5] in GPU. We evaluated the performance in two GPUs, GTX 960M (Maxwell) and GTX 1080 (Pascal). GTX 960M is similar to the GPU used by Emmart et al. [3, 4], while GTX 1080 represents the state of the art GPU architecture available in the market as of 2017. Our implementation can achieve 19528 (best case), 17713 (random exponent) and 17822 (worst case) 1024-bit modular exponentiations per second in GTX 960M. On GTX 1080, the performance is 102379 (best case), 89043 (random exponent) and 90562 (worst case) 1024-bit modular exponentiations per second. We also implemented NTRU in the same GPU platforms to compare its performance with RSA.

The layout of this paper is organized as follows. In Section 2, we introduce the background of Montgomery multiplication, binary exponentiation, signed digit recoding algorithm [5] and the CMM-SDR Montgomery algorithm [5] and followed by our proposed GPU implementation. In Section 3, the experimental setup and result, then Section 4, analysis and discussion. Lastly, the conclusion of our work in Section 5.

2 Background

To compute arithmetic involving large integer size, a common method is to represent the large integer in radix form. The coefficients of the radix form are stored in an array; the arithmetic computations are then performed on these coefficients. If one of these coefficients overflows, we need to perform carry propagation to avoid error. For example, the number 12345 can be represented in radix form of base 10:

$$12345 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0 \quad (1)$$

We store 1, 2, 3, 4, 5 in an array in this case. In this paper, the large integer is represented in radix 2^{32} as the GPU is a 32-bit processor. Hence, a 1024-bit integer can be represented in radix form with 32 coefficients (which is also referred to as *limbs* in the literature). Similarly, 2048-bit can be represented using 64 *limbs*, each *limb* is 32-bit.

2.1 Montgomery Multiplication

The conventional way to perform modular multiplication requires expensive division operation. Instead of using expensive division, Montgomery multiplication is able to perform the reduction by using addition and bit shifting with a base to the power of two, which is optimized for majority hardware architectures. Notice that, this requires conversion from radix form to Montgomery form at the beginning of computation; it also requires another conversion back to radix form at the end of the computation. These two conversions are expensive, but it is still beneficial to use Montgomery multiplication for modular exponentiation because most of the computations can be done in Montgomery form. Algorithm 1 shows the operations involved.

Note that if the b is selected as a power of two, the modular reduction in line 6 can be replaced by bitwise shifting, which is very fast in most of the computer hardware.

Algorithm 1: Montgomery Multiplication.

Input: $x = (x_{n-1} \dots x_1 x_0)_b$, $y = (y_{n-1} \dots y_1 y_0)_b$, $m = (m_{n-1} \dots m_1 m_0)_b$ with $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$
 // m^{-1} is the inverse modular of b
 Output: $A = xyR^{-1} \bmod m$ or $\text{Mont}(x, y)$
 // R^{-1} is the inverse modular of m

1. *begin*
2. $A = 0$; // $A = (a_n a_{n-1} \dots a_1 a_0)_b$
3. for $i = 0$ to $n-1$ *do*
4. *begin*
5. $u_i = (a_0 + x_i y_0) m' \bmod b$;
6. $A = (A + x_i y + u_i m) / b$;
7. *end*;
8. if $A \geq m$ then $A = A - m$;
9. Return(A);
10. *end*.

Algorithm 2: Binary Montgomery Exponentiation.

```

Pre-compute:  $\tilde{x} = \text{Mont}(x, R^2 \bmod m)$ ,  $A = R \bmod m$ 
//  $x$ ,  $m$ , and  $R$  are the inputs of Algorithm 1
Input:  $\tilde{x}$ ,  $m$ , and  $e = (e_t \dots e_1 e_0)_2$  with  $e_t = 1$ 
//  $(e_t \dots e_1 e_0)_2$  is the binary representation of  $e$ 
Output:  $x^e \bmod m$ 
1. begin
2.   for  $i = t$  to 0 do                                // Scan exponent bits from
3.     begin                                           // right to left
4.        $A = \text{Mont}(A, A)$ ;                          // Squaring
5.       if  $e_i = 1$  then  $A = \text{Mont}(A, \tilde{x})$ ;          // Multiplication
6.     end;
7.    $A = \text{Mont}(A, 1)$ ;
8.   Return( $A$ );
9. end.

```

2.2 Binary Montgomery Exponentiation

Binary method (Algorithm 2) can be used in conjunction with Montgomery multiplication to perform modular exponentiation. The algorithm begins by scanning the exponent bits from right to left; if the bit is ‘0’, only squaring is performed; if the bit is ‘1’, an additional Montgomery multiplication is performed.

2.3 Signed Digit Recoding

In binary method for exponentiation, the speed performance is determined by the number of ‘1’ bit in the exponent, as additional multiplication is required for every ‘1’ bit. Signed-digit recoding (Algorithm 3) reduce the number of ‘1’ bit in the exponent. The output of this recoding method always has extra one digit than the binary representation. For example, decimal 31 is represented as [1, 1, 1, 1, 1] (5 digits) in binary, but represented as [1, 0, 0, 0, 0, -1] after signed digit recoding (6 digits). The number of zero has been increased compared to the binary representation, but the number of one is reduced.

2.4 CMM-SDR Montgomery Algorithm

Wu et al. [5] proposed an improvement to the conventional signed-digit recoding technique, named CMM-SDR (Algorithm 3). For each iteration, if the scanned bit is “1” or “-1”, then it is a multiply and square, if “0”, then it is only a squaring.

Algorithm 3: Signed Digit Recoding.Input: $E = (r_{m-1}r_{m-2} \dots r_0)_2$ where r_i is the binary representation of E Output: $E_{SD} = (e_n e_{n-1} \dots e_0)_{SD}$ // e_i is the signed digit representation of E , where $n \geq i \geq 0$ and $e_i \in \{0, 1, -1\}$

1. *begin*
2. $c_0 = 0; r_{m+1} = 0; r_m = 0;$
3. *for* $i = 0$ *to* m *do*
4. *begin*
5. $c_{i+1} = \lfloor (c_i + r_i + r_{i+1}) / 2 \rfloor;$ // $\lfloor x \rfloor$: the largest integer $\leq x$
6. $e_i = c_i + r_i - 2(c_{i+1});$
7. *end;*
8. *end.*

Algorithm 4: CMM-SDR Montgomery Algorithm.Input: M, N, E_{SD}, R where M is the message, N is the modular
and E_{SD} is the exponent in signed digit form// R is the input of Algorithm 1 and E_{SD} is the output of Algorithm 3Output: $C = M^{E_{SD}} \bmod N$

1. *begin*
2. $S \equiv MR \bmod N, C \equiv R \bmod N, D \equiv R \bmod N$
3. *for* $i = 0$ *to* m *do*
4. *begin*
5. *if* $(e_i = 1)$ *then* $C = \text{Mont}(S, C);$ // Multiplication
6. *if* $(e_i = -1)$ *then* $D = \text{Mont}(S, D);$ // Multiplication
7. $S = \text{Mont}(S, S);$ // Squaring
8. *end;*
9. $C = \text{Mont}(C, 1); D = \text{Mont}(D, 1);$
10. $C \equiv C \times D^{-1} \bmod N$ // $C \equiv M^{E_{SD}} \bmod N$
11. *end.*

Based on the theoretical complexity analysis by Wu et al. [5], as the probability of computing $\text{REDC}(S, C)$, $\text{REDC}(S, D)$ and $\text{REDC}(S, S)$ are same with the occurrence of the signed digit “1”, “-1” and “0”, then together with the respective n number of single precision multiplications. Thus, in averagely:

- $\text{REDC}(S, C)$ requires $\frac{1}{6}(2n^2 + n)$ single-precision multiplications;
- $\text{REDC}(S, D)$ requires $\frac{1}{6}(2n^2 + n)$ single-precision multiplications;
- $\text{REDC}(S, S)$ requires $\frac{2}{3}(n^2 + 2n + 2)$ single-precision multiplications.

With CMM-SDR, the occurrence of “0” digit is higher and able to save the number of multiplication compared to the original method. For example, to compute the exponent of decimal 31 (scan from right to left):

- In binary, $[1, 1, 1, 1, 1]$, requires $5(2n^2+n)+5(n^2+2n+2) = 15n^2+15n+10$ single-precision multiplications;
- In signed digit, $[1, 0, 0, 0, -1]$, requires $2(2n^2+n)+6(n^2+2n+2) = 10n^2+14n+12$ single-precision multiplications single-precision multiplications.

Notice that, there is an operation involved modular inverse at the end of this algorithm. In order to reduce the extra cost of computing expensive in the modular inverse, we can perform the inverse modular multiplication with the technique introduced by Koc et al. [6] which still utilize the usage of cheap division in reduction.

3 NTRU Encrypt

NTRU is a lattice-based PKC devised by Hoffstein et al. [9] in 1996. Besides the outstanding security level (quantum computer resistant), NTRU is also attracting many attentions for its performance on embedded platforms with low resources, due to its low power consumption and fast encryption speed.

3.1 NTRU Key Generation

The degree of truncated polynomial ring (N) is set to be a prime integer. In this paper, we only implement $N = 401$, which has a security level of 112-bit [11], similar to 2048-bit RSA. Vectors f and g are randomly chosen from the truncated polynomial ring (usually the coefficients is small in size for better performance), whereas p and q are a co-prime integer pair for coefficient modular operations. In practical implementations, the more accepted value for $p = 3$ and $q = 2048$ (power of 2). The key generation is performed through the equation below:

$$H = pf_q * g \bmod q \quad (2)$$

f_q is the multiplicative inverses of f . Note that each coefficient in the polynomial requires a modular operation for all the calculations. The symbol “*” denotes convolution production of polynomial (polynomial multiplication). H is used as the public key, while f is kept as the private key.

3.2 NTRU Encryption

Before encryption, the plaintext is first translated into the polynomial form M . NTRU encryption is expressed as the follows:

$$E = r * H + M \bmod q \quad (3)$$

r is a random polynomial used in encryption process to obfuscate the correlation between plaintext and ciphertext. To achieve fast encryption speed, r is usually a polynomial with binary (0 and 1) or ternary (−1, 0 and 1) coefficients. Note that the polynomial multiplication (*) between r and H is the most time-consuming operation in NTRU encryption. However, since r is sparse, various techniques can be applied to speed up the polynomial multiplication. The plaintext M in polynomial form is added to the result of convolution to complete the encryption process.

3.3 NTRU Decryption

Polynomial f and its multiplicative inverse f_q are kept as private key. Decryption is performed through the three following steps:

$$D1 = E * f \bmod q \quad (4)$$

$$D2 = D1 \bmod p \quad (5)$$

$$D3 = f_p * D2 \bmod p \quad (6)$$

To date, NTRU is still considered secure from various attacks. Moreover, there is no known attack from Shor's algorithm, which makes it a quantum resistant PKC. Moreover, NTRU is more efficient compared to RSA, which makes it an attractive alternative to commonly used PKC.

4 Proposed GPU Implementation

4.1 RSA Implementation

GPU has deep memory architecture with various memory types; each of them has their own strength and limitation. We implemented CMM-SDR Montgomery multiplication based on coarse-grained parallelism, whereby each thread is assigned to compute one modular exponentiation. Since each thread is independent of each other, there is no intense communication between threads, so shared memory does not provide significant benefits to our implementation. At the same time, the computations within one thread are somehow more intensive compared to fine-grained implementation. Thus, we do not limit the number of registers used per thread and let the compiler allocates as much as it could.

First, we precompute the values of R , C , D and S , then copy these pre-computed values, together with M' (required to compute Montgomery multiplication), M and E_{SD} to global memory in GPU. Notice that all the

Figure 1 Fine-grained Parallelism vs. Coarse-grained parallelism.

values are represented in multi-limbs (32-bit each) and store in the form of arrays, except M' which is store in register.

Next, 32000 threads are launched to perform 32000 modular exponentiations; the threads are organized as 125 blocks per grid, and 256 threads per block. Each thread has to load the values of R , M , E_{SD} , C , D and S into local memory and M' into register. During the computations, C , D and S will be used to store the intermediate values. The results of Montgomery exponentiation are stored in global memory and copied to the CPU memory after the computations are completed.

4.2 NTRU Implementation

According to the key management recommendation of NIST [12], 2048-bit RSA has a security level of around 112-bit. We implemented NTRU with $N = 401$ and $q = 2048$, which has the same security level as 2048-bit RSA. Following the new parameters proposed by Hoffstein et al. [11], each of the polynomial r (random values), H (public key) and M (plaintext) in NTRU encryption process has 401 coefficients. In this paper, r is represented as a ternary polynomial with hamming weight $d_m = 101$; M is also represented as ternary polynomial; H is a dense polynomial. However, all polynomial multiplications in this paper are treated as dense polynomial and sparse polynomial multiplication. Further performance improvement can be obtained by using the product form polynomial, which is out of the scope of this paper.

GPU is responsible for performing the polynomial multiplications (between r and H) and polynomial addition with M . 51328 threads are launched to perform 128 polynomial multiplication and additions; the threads are organized as 128 blocks per grid, and 401 threads per block. In another

word, NTRU is implemented with fine grain parallelism, whereby each block is responsible to perform one NTRU encryption (including polynomial multiplication and additions).

We have implemented two versions of NTRU encryption:

- NTRU-Naïve: the polynomial multiplication between r and H is performed as conventional convolution process with complexity $O(N^2)$. The random polynomial r is pre-computed in CPU and copied to the shared memory of GPU. Public key H is also stored in shared memory for frequent access;
- NTRU-Sparse: The random polynomial r is pre-computed in CPU and the location of nonzeros are copied to the shared memory of GPU. Public key H is also stored in shared memory for frequent access. The polynomial multiplication between r and H only needs to be performed for nonzero coefficients in r , which greatly improved the encryption efficiency.

Similar to RSA computation, the results of polynomial multiplication and addition are stored in global memory and copied to the CPU memory after the computations are completed.

5 Experimental Setup and Result

Most of the available works evaluate the performance based on random bit patterns on the exponents. However, these random bit patterns are difficult to reproduce by others as no information is provided regarding the random seed and algorithm used for generating random numbers. In order to perform a fair comparison with other available works, we proposed to evaluate the performance based on three different bit patterns. The first bit pattern is the smallest exponent (prime number with least number of '0' in the exponent), the second bit pattern is random exponent and third bit pattern is the largest exponent (prime number with the most number of '1' in the exponent). This corresponds to the best case, average case and worst case respectively.

We evaluated the performance of 1024-bit and 2048-bit modular exponentiation on GTX 960M (Maxwell) and GTX 1080 (Pascal). We design and setup three different scenarios to compute the modular exponentiations, with the largest, smallest and random exponent bits. Each scenario is performed for 20 times and the average result is reported. Besides, we only record the time taken for memory transaction within GPU and the computation of modular exponentiation. The time for pre-computation, copy data in between CPU and GPU and result verification are not recorded.

The throughput is calculated as the number of modular exponentiation computed per second.

Figure 2 and 3 show the results of our experiment on GTX 960M and GTX 1080. The results are compared with conventional Montgomery Multiplication without CMM-SDR technique.

Figure 4 shows the results of NTRU-Naïve and NTRU-sparse implementation on GTX 960M and GTX 1080 with $N = 401$, $q = 2048$, and $dm = 101$.

Figure 2 Average throughput for 1024-bit Montgomery Exponentiation.

Figure 3 Average throughput for 2048-bit Montgomery Exponentiation.

Figure 4 NTRU Naïve vs. Sparse Encryption.

6 Analysis and Discussion

6.1 The Smallest Exponent Bits (Best Case)

From Figure 2 and 3, we can see that the throughputs for the conventional method are always higher than the CMM-SDR method in this particular case. Referring to Table 1, the number of non-zero remains the same as in CMM-SDR; instead, CMM-SDR method has an extra computation of zero in both 1024 bits and 2048 bits. In fact, CMM-SDR requires extra one additional Montgomery Multiplication (compare **line 7, Algorithm 2** and **line 9, Algorithm 4**) and the extra computation of modular inverse and modular multiplication (**line 10, Algorithm 4**). CMM-SDR suffers the computation overhead in this case. As a result, the conventional method is more efficient than CMM-SDR for the case of smallest exponent bits.

6.2 The Largest Exponent Bits (Worst Case)

The CMM-SDR method started to shine in this scenario as its throughput is around 50% higher than the conventional method. From the Table 2, we

Table 1 Numbers of non-zero and zero in smallest exponent bits for conventional and CMM-SDR (1024-bit and 2048-bit modular exponentiation)

	Conventional	CMM-SDR
Non-zero	1	1
Zero	1023/2047	1024/2048

Table 2 Numbers of non-zero and zero in largest exponent bits for conventional and CMM-SDR (1024-bit and 2048-bit modular exponentiation)

	Conventional	CMM-SDR
Non-zero	1024/2048	2
Zero	0	1024/2048

can see that the number of non-zero is greatly reduced in CMM-SDR. In this scenario, the conventional method needed to compute 1024 (1024 bits) and 2048 (2048 bits) times of squaring and multiplication, whereas CMM-SDR method only needed to compute 2 times (1024 bits and 2048 bits). Thus, the CMM-SDR method is more efficient in this case.

6.3 Random Exponent Bits (Average Case)

In random exponent bits, the CMM-SDR method still able to outperform the conventional method, the overall throughput is about 12% higher than the conventional method. From the Table 3, we can see that the number of non-zero is still greatly reduced in CMM-SDR, from 497 to 7 in 1024-bit modular exponentiation and 1015 to 16 in 2048-bit modular exponentiation. However, the computation overhead as mentioned in Section 5.1 which limits the maximum achievable throughput.

6.4 Performance Comparison of Our Work with Recent Work

The work from Emmart and Weems [3] is the fastest among all coarse grain modular exponentiation in GPU. They are using the GTX 750Ti (640 cores) from Maxwell architecture, which is the same with GTX 960M (640 cores) we used. We are able to achieve 17.17k modular exponentiation per second (random exponent bits) which is slower than the achievement by Emmart and Weems [3] (22.72k). However, our implementation is not fully optimized compared to them. Firstly, they used fixed window exponentiation which scans multiple bits per iteration, but our method only scans one bit per iteration.

Table 3 Numbers of non-zero and zero in random exponent bits for conventional and CMM-SDR (1024-bit and 2048-bit modular exponentiation)

	Conventional	CMM-SDR
Non-zero	497/1015	7/16
Zero	527/1033	1017/2032

Secondly, we have not fully optimized the operation to store and load the message, modulus, and exponent in GPU, which involves optimized usage of local memory and registers. Thirdly, they also used CUDA PTX assembly code to fully optimized the implementation. In fact, our work can be integrated with the techniques proposed by Emmart and Weems [3] to further improve the throughput of modular exponentiation in GPU.

On the other hand, we also evaluated the same implementation in GTX 1080 with the latest GPU architecture, Pascal. GTX 1080 consists of 2560 cores, which is four times more than GTX 960M (640 cores). From our experiments, the throughput achieved by GTX 1080 is 3.5–4.2 times more than GTX 960M, which is coherent with the hardware capability of both hardware platforms. GTX 960M is more widely used for low-end mobile computing system like laptops; we selected this platform to perform a direct comparison with Emmart and Weems [3]. Conversely, GTX 1080 can be used in a server environment to handle massive digital signatures (RSA) in parallel.

6.5 Comparing NTRU and RSA

The results presented in Section 5 shows that NTRU has better speed performance compared to 2048-bit RSA for same security level (112-bit). The best throughput of 2048-bit RSA in GTX 960M and GTX 1080 are summarized in Table 4, together with the throughput achieved by NTRU-Naïve and NTRU-Sparse.

NTRU-Naïve is 62.5 (GTX 960M) and 38.1 (GTX 1080) times faster than the best throughput achieved by 2048-bit RSA (where the majority of the exponent bits are 0). This shows that NTRU is a strong candidate to replace RSA as it is quantum resistant and very efficient. By skipping the computations in zero coefficients, NTRU-Sparse is 2.0–3.8 times faster than NTRU-Naïve in two different GPU platforms, which further improve the speed performance of NTRU implementation.

Table 4 Best throughput (Operations per second) for NTRU and 2048-bit RSA on GTX 960M and GTX 1080

	GTX 960M	GTX 1080
2048-bit RSA (smallest exponent)	2563	12274
NTRU-Naïve	160296	467208
NTRU-Sparse	610760	951238

7 Conclusion

In this paper, we have shown that our GPU implementation is able to achieve high throughput by incorporating the CMM-SDR method by Wu *et al.* [5]. Although our proposed implementation does not show good result in smallest exponent bits, it eventually shows good result in random exponent bits which is more closely related to the real world scenarios. By integrating the technique proposed in this paper to the work from Emmart and Weems [3], the modular exponentiation can achieve higher throughput in GPU platforms. On the other hand, we have also shown that NTRU encryption is eventually much faster than RSA at the similar security level. By employing sparse polynomial multiplication technique, the speed of NTRU encryption can be greatly improved.

Acknowledgements

This research is partially supported by the Malaysia Ministry of Science, Technology & Innovation (MOSTI) eScience fund 01-02-11-SF0201 and 01-02-11-SF0202.

References

- [1] Neves, S., and Araujo, F. (2011). "On the performance of GPU public-key cryptography," in *22nd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP – 2011)*, Santa Monica, CA, USA.
- [2] Leboeuf, K., Muscedere, R., and Ahmadi, M. (2013). "A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, Beijing, China.
- [3] Emmart, N., and Weems, C. (2015). "Pushing the Performance Envelope of Modular Exponentiation Across Multiple Generations of GPUs," in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)*, Hyderabad, India.
- [4] Emmart, N., Luitjens, J., Weems, C., and Woolley, C. (2016). "Optimizing Modular Multiplication for NVIDIA's Maxwell GPUs," in *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH 2016)*, Silicon Valley, CA, USA.

- [5] Chia-Long, W., Der-Chyuan, L., and Te-Jen, C. (2008). “An efficient Montgomery exponentiation algorithm for public-key cryptosystems,” in *2008 IEEE International Conference on Intelligence and Security Informatics*, Taipei, Taiwan.
- [6] Savas, E., and Koc, C. (2000). The Montgomery modular inverse-revisited. *IEEE Trans. Comput.* 49, 763–766.
- [7] Montgomery, P. (1985). Modular Multiplication Without Trial Division. *Math. Comput.* 44, 519.
- [8] Kaya Koc, C., Acar, T., and Kaliski, B. (1996). Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro.* 16, 26–33.
- [9] Hoffstein, J., Pipher, J., and Silverman, J. H. (1998). NTRU: A Ring Based Public Key Cryptosystem. In *Algorithmic Number Theory (ANTS III)*, Lecture Notes in Computer Science 1423, 267–288.
- [10] IEEE P1363.1. *Public-Key Cryptographic Techniques Based on Hard Problems over Lattices*. Available at: <http://grouper.ieee.org/groups/1363/lattPK/index.html> [Accessed at 31 July 2017].
- [11] Hoffstein, J., Pipher, J., Schanck, J. M., Silverman, J. H., Whyte, W., and Zhang, Z. (2017). *Choosing Parameters for NTRUEncrypt*, CT-RSA 2017, pp. 3–18.
- [12] Recommendation for Key Management, Special Publication 800-57, Part 1, Rev. 4, NIST, 01/2016.

Biographies

Xian-Fu Wong received his B.CS degree from Universiti Tunku Abdul Rahman (UTAR), Malaysia in year 2015. He is currently completing a masters in Engineering Science at UTAR, and his research interests are in the areas of cryptology, algorithms and GPU computing.

Bok-Min Goi received his B.Eng. degree from University of Malaya (UM) in 1998, and the M.Eng.Sc and Ph.D. degrees from Multimedia University (MMU), Malaysia in 2002 and 2006, respectively. He is now the Dean and a professor in the Lee Kong Chian Faculty of Engineering and Science, Universiti Tunku Abdul Rahman (UTAR), Malaysia. Ir. Prof. Goi was also the General Chair for ProvSec 2010 and CANS 2010, Programme Chair for IEEE-STUDENT 2012 and Cryptology 2014, and the TPC members for many crypto/security conferences. His research interests include cryptology, security protocols, information security, digital watermarking, computer networking and embedded systems design. He is a senior member of the IEEE and corporate member of the IEM, Malaysia.

Wai-Kong Lee was born in Malaysia in 1982. He received the B.Eng. in Electronics and M.Sc. degree from Multimedia University in 2006 and 2009 respectively. He is now a Ph.D. candidate with the Faculty of Engineering and Science, University Tunku Abdul Rahman, Malaysia. His research interests are in the areas of GPGPU, cryptography and energy harvesting.

Raphael C.-W. Phan received the Ph.D. (Eng.) degree in security from Multimedia University, Cyberjaya, Malaysia. He held academic positions with Australian, Swiss, and British universities before taking up his current Chair position. His current research interests include diverse areas of security and privacy with a focus on privacy preservation and processing of data in the encrypted domain. Dr. Phan was/is the General Chair of Mycrypt 05 and Asiacrypt 07, and the Program Chair of ISH 05 and Mycrypt 16. He has served on the technical program committees of international conferences since 2005. He is a Co-Designer of BLAKE, one of the five hash function finalists of the NIST SHA-3 competition. He has an Erdős number of 2.