Efficient Parallel RSA Decryption Algorithm for Many-core GPUs with CUDA

Yu-Shiang Lin, Chun-Yuan Lin[1], Der-Chyuan Lou
Department of Computer Science and Information Engineering
Chang Gung University
Taoyuan 333, Taiwan, ROC
coldfunction@gmail.com, {cyulin, dclou}@mail.cgu.edu.tw

Abstract

Cryptography is an important technique among various applications. In the telecommunication, cryptography is necessary when an untrusted medium is communicated in the network. RSA is a public-key cryptography algorithm to use a pair (N, E) as the public key and D as the private key. The N is the product of two large prime numbers p and q that are kept secret. It is very hard and no known polynomial time algorithms can be used to extract p and q from a large number N. There are many methods of factoring large numbers have been proposed. The advantages of computing power and memory bandwidth for modern GPUs have made porting applications on it become a very important issue. In this paper, we proposed an efficient parallel RSA decryption algorithm for many-core GPUs with CUDA. The experimental results showed that the proposed GPU-based algorithm can achieve 1197.5x average speedup compared with the CPU-based algorithm, and within a reasonable time to find out the result of factoring large numbers.

Keywords: Cryptography, Parallel Processing, RSA, CUDA, Graphics Processing Units

1. Introduction

Cryptography is an important technique among various applications, especially for Internet and business transactions. For most of communication applications, several specific security requirements are needed, including the authentication, privacy, integrity, and non-repudiation. Cryptography can protect data and be used for the user authentication. In the telecommunication, cryptography is necessary when an untrusted medium is communicated in the network. Most of cryptography algorithms can be classified into three types: secret-key (symmetric) cryptography, public-key (asymmetric) cryptography, and hash functions [21].

For the secret-key cryptography, only a single key is used for both encryption and decryption, and it is also called symmetric encryption. Some secret-key cryptography algorithms have been proposed in the past, such as Data Encryption Standard (DES) [24], Advanced Encryption Standard (AES) [25], and International Data Encryption Algorithm (IDEA) [16]. For the public-key cryptography, two keys are used; one for encryption and another for the decryption, and the public-key cryptography is also called asymmetric encryption. Some public-key cryptography algorithms have been proposed in the past, such as RSA [31], Diffie-Hellman Key Exchange [6], and Elliptic Curve Cryptography (ECC) [12, 34]. For hash functions, no key is used, but a mathematical transformation is used to irreversibly encrypt the information, and it is also called one-way encryption. Some hash function algorithms have been proposed in the past, such as Message Digest (MD) algorithms (such as MD5 [30]), Secure Hash Algorithm (SHA) [33], and Whirlpool [1].

RSA is an asymmetric cryptography algorithm and developed by Rivest, Shamir, and Adleman in 1978 [31]. RSA is still widely used in hundreds of software products and electronic data. In the RSA algorithm, a pair (N, E) and D are the public key and private key, respectively. The N is the product of two large prime numbers p and q, and the D is selected according to the formula: $E \cdot D = (1 \bmod \psi)$, where $\psi = (p-1)\times(q-1)$. To encrypt a plaintext message M with RSA algorithm, a ciphertext C is computed by the formula: $M^E \bmod N$ with public key (N, E). To decrypt the ciphertext by reversing the above operation, the message M is computed by the formula: $C^D \bmod N$ with public key N and private key D. Therefore, for encrypting a plaintext message M or decrypting a ciphertext C, it is very important to compute the arithmetic modulo N efficiently. Moreover, the security of RSA algorithm relies on the hardness of factoring the large number N without the private key D. Fortunately, it is very hard and no known polynomial time algorithms can be used to extract p and q from a large number N, such as RSA-2048 for 2048-bit integers.

In order to accelerate the speed of factoring the large number N, several efficient methods were proposed, such as Fermat's Factorization [20], Pollard's p-1 Factorization [28], Pollard's ρ Factorization [29], and The elliptic curve

[1] Corresponding author

Factorization [17]. However, these methods still are very time-consuming under the modern CPU, even for a medium-sized number *N*, such as RSA-64 and RSA-128. Therefore, these methods were re-designed with the reconfigurable hardware device in the past. For example, in 2005, Pelzl *et al.* proposed the hardware-based implementation for the elliptic curve Factorization [27] on FPGA and an embedded microcontroller. In 2006, Gaj *et al*. proposed another implementation of elliptic curve Factorization with the reconfigurable hardware method [11] on FPGA, and achieved better performance than that by Pelzl *et* al. [27]. In 2010, Chen and Schaumont proposed a scalable parallel programming scheme, pSHS, to map the Montgomery multiplication to a general multicore architecture [2]. Montgomery multiplication is an important part of modular multiplications and exponentiations in the public-key cryptography. In 2007, on the IBM Cell processor, Costigan and Scott have tried to accelerate RSA in the OpenSSL library [5], and after that Costigan and Schwabe also have implemented a fast elliptic curve Diffie-Hellman key exchange [4] on the Cell processor in 2009.

Current high-end graphics processing units (GPUs), contain up to hundreds cores per chip, are very popular in the high performance computing community. GPU is a massively multi-threaded processor and expects the thousands of concurrent threads to fully utilize its computing power. In the past, several cryptography algorithms have been ported on GPUs. Cook *et al.* studied the feasibility of implementing symmetric-key ciphers for AES in a GPU using the OpenGL API [3]. Yamanouchi also proposed a similar approach with OpenGL extension specific for AES [36]. Moss *et al*. investigated the implementation and performance of modular exponentiation using a GPU with the OpenGL Shading Language to execute operations required in the RSA algorithm [22, 23]. They focused on implementing the modular multiplication using a Residue Number System (RNS) with the large number *N*. Fleissner also implemented an accelerated Montgomery method for modular exponentiation with General-purpose computing on graphics processing units (GPGPU) [8]. Due to more layer transfer interface call of GPGPU using graphics APIs (OpenGL, DirectX, and etc.), these works cannot make effective for using computing power of the GPU.

The ease of access GPUs by using Compute Unified Device Architecture (CUDA) [26], as opposite to graphic APIs, has made the supercomputing available to the mass. CUDA uses a new computing architecture, named Single Instruction Multiple Threads (SIMT), and SIMT is different from the Flynn's classification [9]. The advantages of the computing power and memory bandwidth for modern GPUs have made porting applications on it become a very important issue. Manavski [19] used the CUDA API as the work proposed by Rosenberg [32] to implement AES. In 2008, Szerwinski *et al.* employed CUDA API to develop efficient modular exponentiation and elliptic curve scalar multiplication [35]. Harrison and Waldron also provided a GPU sliding window exponentiation implementation with CUDA API based on Montgomery exponentiation using both radix and residue number system representations [13]. Hermans *et al.* proposed GPU implementations for NTRU Encrypt in 2010 [14]. Jang *et al.* designed a GPU approach for SSL with CUDA [15]. Fan *et al.* presented a novel parallelized implementation of RSA algorithm using JCUDA and Hadoop [7].

Although several approach as shown in above have been proposed to accelerate the RSA algorithm by using a GPU with CUDA, however, all of them focused on encrypting a plaintext message *M* or decrypting a ciphertext *C*. Therefore, how to compute the arithmetic modulo *N* efficiently is the important issue. Most of them tried to improve the implementation of modular exponentiation with Montgomery method and RNS. According to our best knowledge, no work has been proposed or proven to accelerate the speed of factoring the large number *N* by using a GPU with CUDA. A near approach was proposed by Fujimoto to accelerate the computation of the greatest common divisor (GCD) for long integers with CUDA [10]. However, this work did not be applied to factor the large number *N* for RSA algorithm. Hence, in this paper, a GPU-based Pollard's *p*-1 Factorization Algorithm, *GPFA*, was proposed to accelerate the speed of factoring the large number *N* by using a GPU with CUDA. Since the computations in the Pollard's *p*-1 Factorization can be subdivided into independent iterations, *GPFA* used the inter-task parallelization [18] technique to do the computations. We implemented *GPFA* with various parameters and obtained corresponding performance. We also analyzed the relationship between parameters and performance in this paper. In the experimental tests, we compared *GPFA* with the CPU-based Pollard's *p*-1 Factorization Algorithm, *CPFA*. *GPFA* can achieve 1197.5x average speedup compared with *CPFA* among the testing data set, constituted of RSA-41 to RSA-64. RSA-64 can be factored within 40 seconds by *GPFA* in the test.

This paper is organized as follows. In Section 2, preliminary concepts for Pollard's *p*-1 Factorization and CUDA programming model were described briefly. Section 3 introduced the implementations of *CPFA* and *GPFA* with proposed custom number system. Analysis and experimental results were shown in Section 4 and Section 5, respectively.

2. Preliminary Concepts

2.1 Pollard's p-1Factorization

Pollard's *p*-1 Factorization method was developed by Pollard in 1974 [28]. The method is based on the Fermat's little theorem, which states:

If p is a prime number and a is an integer not divisible by p, then

$$a^{p-1} \equiv 1 (\mathrm{mod}\ p) \tag{1}$$

To factor a large number N is to find a prime number p if $p \mid N$, and then obtain a formula: $a^{p-1} \equiv 1 (\mathrm{mod}\ p)$, and it follows that

$$K = a^{p-1} - 1 \equiv 0 (\mathrm{mod}\ p) \tag{2}$$

If assumed that p-1 is m, and m can be increased from m = 1, 2, 3, …, until $gcd(K, N) = p$. However, using this method to find an exact m is not an efficient method, since it needs to do p-1 operations and the time complexity grows exponentially when p increases. It means that it needs a way to find an exact m quickly. The idea of Pollard's p-1Factorization is not to find the exact m directly and assume that an integer m', where it satisfied p-1 | m', then $m' = (p-1)^c$ and obtain formula:

$$a^{m'} - 1 = a^{(p-1)^c} - 1 = 1^c \equiv 0 (\mathrm{mod}\ p) \tag{3}$$

and

$$\gcd(a^{m'}, N) = p \tag{4}$$

Therefore, we only need to find an integer m' which it satisfied p-1 | m'. In order to get an exact m', the possibility to meet conditions increases when generating many prime numbers before the factorization.

2.2 CUDA programming model

CUDA is an extension of C/C++ which users can write scalable multi-threaded programs for GPUs computing [26]. The implementation of the CUDA program is divided into two parts: host and device. The host mainly is executing by CPU and the device is mainly executing by GPU. The program which is executed on the device called a *kernel.* The *kernel* can invoke as a set of concurrently executing threads, and *kernel* program will be executed by threads. These threads are in the hierarchical organization which can be combined into thread blocks and grids. A grid is a set of independent thread blocks, and a thread block contains many threads. Threads in a block can communicate and synchronize with each other. Threads within a thread block can communicate through a *per-block shared memory* (*PBSM*), whereas threads in different blocks cannot communicate or synchronize directly. In addition to *PBSM*, there are four kinds of memory type: per-thread private local memory (*LM*), global memory (*GM*) for data shared by all threads, texture memory (*TM*), and constant memory (*CM*). Among these memory types, *CM* and *TM* can be regarded as fast read only caches; the fastest memories are the registers and *PBSM*.

The basic processing unit in the NVIDIA's GPU architecture is called the *Streaming Processor* (*SP*). There are many *SP*s which actually do the computations on GPU. A group of *SPs* can be combined into a *Stream Multiprocessor* (*SM*). While the program runs the *kernel* function, the GPU device schedules thread blocks for execution on the *SM*. The threads running on the *SM* in small groups of 32, called warps, is SIMT scheme, every *SM* have a warp scheduler to execute warps. For example, NVIDIA GeForce GTX 260, there is 16KB of *PBSM* for each *SM* with 16,384 32-bit registers. The number of thread blocks assigned to the *SM* is affected by the registers and *PBSM* used in a thread block. *SM* can be assigned up to 8 thread blocks. The *GM*, *LM*, *TM*, and *CM* are all located on the GPU's memory. In addition to *PBSM* accessed by single thread block and registers only accessed by single thread, the other memory can be used by all the threads. The caches of *TM* and *CM* are limited to 8KB per *SM*. The best access strategy for *CM* is all threads read the same memory address. The texture cache is designed for threads to read between the proximity of the address would be take a better reading efficiency. In NVIDIA new architecture Fermi, there have more hardware expansion. For example, NVIDIA C2050, there is configurable 48 KB or 16KB of *PBSM*, since it add the parallel cache mechanism with the configurable L1 and L2 Cache, L1 cache for each *SM* and L2 cache shared by all *SM*. In the Fermi architecture, the number of *SPs* can be up to 512, and two warp schedulers per *SM*.

3. Methods

3.1 CPFA (CPU-based Pollard's p-1 Factorization Algorithm)

In this paper, *CPFA* is implemented according to the conditions shown in Section 2.1 and the designed *CPFA* algorithm was shown below. In *CPFA*, the goal is to factor a public key N to find p or q. Since the procedure of Pollard's p-1 Factorization may need to process the large integers, a custom integer system (CIS) was proposed to represent and do the operations for large integers. For example, for an integer N represented as a decimal number 52012173456240253_{10} or hexadecimal number 0xB8C8CBD2DAEE7D, CIS can be used to represent it and compute the following factorization. The proposed CIS was designed for *CPFA* and *GPFA*, respectively, and described in Section 3.2. Before executing *CPFA*, a *prime table* consisting of prime numbers is needed. The number of prime numbers in the *prime table* is not fixed. In the experimental tests, a *prime table* with 173,057,268 prime numbers ranged among 32-bit integers was constructed. The value of an integer B is assumed smaller than the biggest prime number in the *prime table*. All of the primer numbers smaller than B are extracted from the *prime table* to do the computations in step 3. Therefore, the value of B will affect the computation time of step 3 directly. Under a fixed number of loop iterations T_c, the value of B will be changed to twice when *CPFA* could not find p or q for an integer N.

CPU-based Pollard's p-1 Factorization Algorithm

//Object: to find p or q from an integer N
//Load the *prime table* from a disk to the main memory

for (integer i from 1 to T_c)
{
Step1. Choose an integer a_c, it could be 2 or generated randomly.
Step2. Extract a prime number p smaller than B from the *prime table*.
Step3. Compute:

$$e = \prod_{2 \le p \le B} p^{\lfloor \log_B / \log_p \rfloor}$$

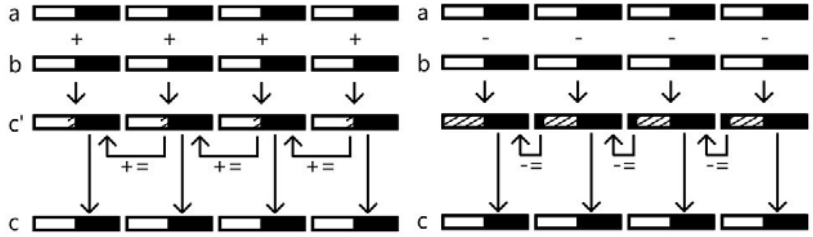
Step4. Let $b = a_c^e \bmod N$, if $1 < gcd(b\text{-}1, N) < N$, then return the value of greatest common divisor $gcd(b\text{-}1, N)$.
Step5. Follow step 4, if $gcd(b\text{-}1, N)$ equals to 1 or N, then go to step 2.
Step6. If finding a prime number p larger than B, then execute the next iteration (B=2B).
}

3.2 CIS (Custom Integer System)

In the past, many research tried to improve the implementation of modular exponentiation with Montgomery method and RNS. For a large integer in the RNS, it can be encoded into an RNS representation with a basis, a set of co-prime integers, and then this integer is stored as a vector of components (modulo the basis for each component). For the multiplication and addition of two large integers encoded by RNS with the basis, it is easy to do since the computation of each component is independent. This system is useful for encrypting a plaintext message M or decrypting a ciphertext C with a fixed pair (N, E) and D. However, the RNS may be not suitable for *CPFA* and *GPFA*. The reason is that there are an integer a_c, many prime numbers p, and an integer N should be computed in steps 3 to 5. For many prime numbers p, it may be time-consuming to select a feasible basis for each p and then translate this p to a vector of components. Therefore, it needs to design an adjustable data structure to represent the integer and do the following factorization.

Hence, CIS and its operations are designed and implemented for *CPFA* and *GPFA*, respectively. However, some extra operations or data structures are not implemented specifically in CIS, such as the operations or structures for the negative integer number, since CIS is only designed for *CPFA* and *GPFA* at present. In general, a large integer can be formed as a character array to store each digit by one byte. For example, an integer number 123 can be stored in a character array {1, 2, 3} with size of 3 bytes. This method is simple but needs more space when doing the operations of the integers. For example, when doing the sum operation for two integers 123 and 987, it needs 9 bytes to store these two integers and the possible carry for each digit. Moreover, it needs 6 addition operations, not 3 addition operations, to compute the digit addition and the carry. Therefore, a naive idea in CIS is to use the unsigned integer type to store the integer. If an integer number is larger than the scope of one unsigned integer, then use two or more unsigned integer to store this integer number. In addition, the reserve space in the unsigned integer type is used to store the possible carry. An example of integer representation in CIS is shown in Figure 1. In Figure 1, the size of unsigned integer type is 32-bit and we can use eight unsigned integer (256-bit of total) to store a 128-bit integer. The black area is the half size (16-bit) of the unsigned integer to store the integer (128-bit of total) and the white area is the half size (16-bit) of the unsigned integer to store the possible carry (128-bit of total). By using the unsigned integer type, it can reduce the space requirement for the large integer; however, the operations for large integers represented in CIS should be designed.

Figure 1: An example of integer representation in CIS.



(a) addition operation in CIS. (b) subtraction operation in CIS.

Figure 2: The addition and subtraction operations for two large integers in CIS.

For the operations of large integers represented in CIS, the procedure is to do the operation for each unsigned integer in sequential. Figure 2 shows the addition and subtraction operations for two large integers in CIS, respectively. In Figure 2(a), each integer is represented in CIS and stored in arrays *a*, *b*. The addition result without the possible carry is stored in array *c*` and then combine the carry to obtain the final result. In Figure 2(b), the procedure of subtraction operation for two large integers in CIS is similar to that of addition operation. However, the borrowing problem should be considered. In addition to addition and subtraction operations of large integers, we also implement the multiplication, shift, and modulo operations of large integers in CIS.

3.3 *GPFA* (*GPU-based Pollard's p-1 Factorization Algorithm*)

In CUDA, there are two parallelization techniques [18] to map tasks into threads or thread blocks, one is *inter-task parallelization* and another is *intra-task parallelization*. For *inter-task parallelization*, each thread exactly executes one task; for *intra-task parallelization*, each task is executed by one thread block. In this paper, the proposed *GPFA* is designed by using the *inter-task parallelization* technique.

The designed *GPFA* algorithm was shown below.

GPU-based Pollard's p-1 Factorization Algorithm

//Object: to find *p* or *q* from an integer *N*
//Load the *prime table* from a disk to the main memory of CPU and then Transfer it to the global memory of GPU.

//gridDim.x is the built-in variable represents the size of grid (number of thread blocks in one grid).
// blockDim.x is the built-in variable represents the size of block (number of threads in one thread block).
//blockIdx.x is the built-in variable represents the 1-D thread block index within the grid.
//threadIdx.x is the built-in variable represents the 1-D thread index within the thread block.

int linearID = blockDim.x×blockIdx.x+threadIdx.x;
int total_num_of_thread = gridDim.x×blockDim.x;
int a_g = 2+linearID;

for (integer *i* from blockIdx.x×blockDim.x to blockIdx.x×blockDim.x+T_g -1)
{

//*pr*(*j*) is the *j*-th prime number in the *prime table*.
for (unsigned int *j*= linearID; *pr*(*j*) < *B* ; *j*+= total_num_of_thread)
{
Compute:

$$e = \prod_{2 \le p \le B} p^{\lfloor \log_B / \log_p \rfloor}$$

Let $b = a_c^e \bmod N$, if 1 < *gcd*(b-1, *N*) < *N*, then return the value of greatest common divisor *gcd*(*b*-1, *N*) to the global memory.

Follow last step, if *gcd*(*b*-1, *N*) equals to 1 or *N*, then continue.
}

$a_g = a_g \times a_g$ %RAND_MAX+*i*+linearID;
}

In the beginning, the *GPFA* is the same with *CPFA* to load the *prime table* from a disk to the main memory of CPU, and make necessary adjustment with B and T_g. However, in *GPFA*, the data must be transferred from CPU to GPU and then execute the GPU kernel function. The first problem in *GPFA* is how to allocate the *prime table* and store the results (p or q) in the GPU's memory. Since the *prime table* will be accessed frequently by threads in thread blocks, it is worth to use a cache mechanism to access it. When the size of the *prime table* is small, it can be stored in the *CM* with 64 KB size limitation; however, the *prime table* in the experimental tests is close up to 70 MB, the *prime table* is allocated in the *TM*. Figure 3 shows that each thread loads the unique prime number from the *prime table* in the *TM*, and the registers of each thread are used to store a large integer N and an integer a_g. The integer a_g is selected as a random number a_c in *CPFA*. Three arguments, a_g, prime numbers, and a large integer N, are involved in the execution of kernel function on GPU. In *GPFA*, if one thread obtains the p or q, it returns the result to the *GM* immediately and then the result is transferred from the *GM* of GPU to the main memory of CPU.

In CUDA, there is an important characteristic that each thread in a thread block has its own unique thread ID. Hence, *GPFA* can use different IDs (threads) to deal with different data (prime numbers in the *prime table*). By this characteristic, the unique ID of each thread among a grid can be calculated, called thread linearize. In the 1-D type of thread blocks and threads, thread linearize is:

$$\text{linearID} = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x}, \quad (5)$$

where the blockDim.x, blockIdx.x, and threadIdx.x are the built-in variables in CUDA, they individually represent the size of block (number of threads in one block), block ID (thread block index within the grid), and thread ID (thread index within the thread block), respectively. For the selection of a_g, it can use the thread linearize to pick out the unique a_g for each thread, such as:

$$a_g = 2 + \text{linearID} \quad (6)$$

The execution steps of inner loop in *GPFA* are almost the same as the steps 3 to 5 in *CPFA*, except for returning the value of gcd result to the *GM*. By the SIMT, when *GPFA* could not find p or q for an integer N in the inner loop, the value of a_g will be updated by each thread with the formula:

$$a_g = a_g \times a_g \,\%\text{RAND\_MAX} + i + \text{linearID} \quad (7)$$

There are T_g times controlled by the external loop to update the value of a_g by each thread in thread blocks.

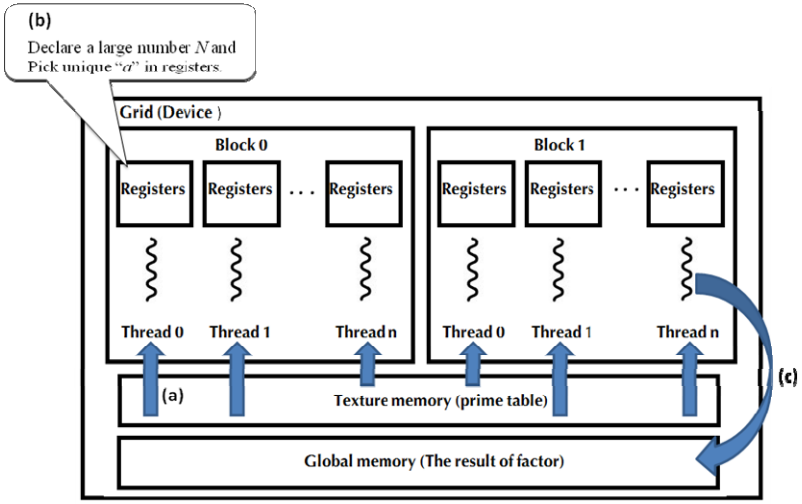


Figure 3: Memory allocation scenario of *GPFA* in CUDA.

4. Performance Analysis

In *CPFA*, the main object is to find p or q for a large integer N by generating a random value a_c and sequential searching a series of prime numbers (denoted as pr_i) from the *prime table* (inner loop in *CPFA*). Assume that there are k prime numbers $\{pr_1, pr_2, ..., pr_{k-1}, pr_k\}$ in the *prime table*. If the number of external loop (T_c) in *CPFA* is u, there are u random values of $\{a_{c,1}, a_{c,2}, ..., a_{c,u-1}, a_{c,u}\}$. The computation cost of *CPFA* is determined by the B and T_c. In the worst case, where the result of prime number is equal to B and T_c is equal to u, the time complexity of *CPFA* is O(ku). Although the values of B and T_c can be set to small, *CPFA* may not find the result. According to the asymptotic law for the distribution of prime numbers, the number of prime numbers ($\pi(x)$), less than or equal to a real number x, is close to

$$\pi(x) = \frac{x}{\ln x}. \quad (8)$$

Therefore, k equals to $\pi(B) = \frac{B}{\ln B}$. Assumed that t is a fixed value, the time complexity of *CPFA* is O($\frac{B}{\ln B}u$).

In this paper, *GPFA* is designed by using the *inter-task parallelization* technique. Assume that there are k prime numbers $\{pr_1, pr_2, ..., pr_{k-1}, pr_k\}$ in the *prime table*. It means that each thread do the computation for a pair (pr_i, a_g) in the inner loop. If there are gridDim.x×blockDim.x threads to do the tasks in the inner loop, it needs y times to do the computation for all prime numbers in the worst case, where

$$y = \frac{k}{\text{gridDim.x} \times \text{blockDim.x}} = \frac{\pi(B)}{\text{gridDim.x} \times \text{blockDim.x}} = \frac{B}{\ln B \times (\text{gridDim.x} \times \text{blockDim.x})} \quad (9)$$

Therefore, if the number of external loop (T_g) in *GPFA* is s and s is a fixed value, the time complexity of *GPFA* is $O(s \times (\frac{B}{\ln B \times (\text{gridDim.x} \times \text{blockDim.x})}))$.

Given a detail assumption for the cost of computing a prime number, t_c for *CPFA* and t_g for *GPFA*, the theoretical computation time of *CPFA* and *GPFA* in the worst case will be $(\frac{B}{\ln B}u)t_c$ and $(s \times (\frac{B}{\ln B \times (\text{gridDim.x} \times \text{blockDim.x})}))t_g$, respectively. Hence, the theoretical speedup can be calculated according to the formula:

$$speedup = \frac{Time(CPFA)}{Time(GPFA)} = \frac{(\frac{B}{\ln B}u)t_c}{(s \times (\frac{B}{\ln B \times (\text{gridDim.x} \times \text{blockDim.x})}))t_g} = (\frac{u \times t_c}{s \times t_g}) \times (\text{gridDim.x} \times \text{blockDim.x}) \quad (10)$$

According to the above formula, the speedup increases when the number of threads increases. However, the number of threads is not infinite. By the *inter-task parallelization*, the number of threads in a thread block is bounded according to the memory usage. Moreover, the number of concurrent thread blocks is bounded according to the number of *SM*s. Besides, considering the u and s, in general, s is less than u since each thread can obtain various values of a_g and then the possibility to meet conditions by *GPFA* is larger than that by *CPFA*. However, considering the t_c and t_g, t_g is larger than t_c since the clock rate of CPU is faster than that of *SP* on GPU. Overall, *GPFA* can achieve better performance than *CPFA* according to the performance analysis.

Table 1: The testing data set represented as hexadecimal numbers.

Length	*N*	*p*	*q*
RSA-41	12B1F259795	721F7	29EFD3
RSA-44	89FD383381B	120FC7	7A3D0D
RSA-46	3CF5F89ED5F5	C50069	4F37AD
RSA-47	600FF385C031	FF52D9	605119
RSA-48	878D4C7D68E9	ABB039	CA1E31
RSA-56	B8C8CBD2DAEE7D	D985797	D978D0B
RSA-64	6926C73F919FA3E7	79E6711B	DCD39125

5. Experimental Results

In this paper, *GPFA* was implemented on three various GPU architectures: GTX-260, S1070, and C2050 GPUs, and *CPFA* was implemented on Intel Core2 Quad Q8200 2.33GHz CPU with 4G RAM running the Linux system. The testing data set consisted of RSA-41 to RSA-64, shown in Table 1, was used to evaluate *CPFA* and *GPFA*. For the testing data RSA-41 to RSA-56, the value of B was set to 100,000; the value of B was set to 200,000 for RSA-64. For the C2050 GPU, the experimental results are classified into configure and nonconfig (non-configure) states to represent the configurable L1 cache of 48 KB and 16 KB in *GPFA*, respectively. In *GPFA*, the CUDA built-in variables gridDim.x (size of grid) and blockDim (size of block) were set to 1024 and 64, respectively.

Figure 4 illustrated the execution time by *CPFA* and *GPFA* under various platforms, where y-axis is the scale of logarithm to base 10. In *GPFA*, the worst case is RSA-64 factored within 40 seconds; however, the worst case in *CPFA* is RSA-56 factored within 7,350 seconds. Since the factoring a large integer by the Pollard's p-1 Factorization algorithm can be seen as a search problem under a possibility, hence, the worst case for *CPFA* and *GPFA* may be different. Moreover, the computation time is nonlinear, even is non-incremental, when the size of input data increases. From Figure 4, the experimental results showed that *GPFA* can greatly reduce the computation time by *CPFA*.

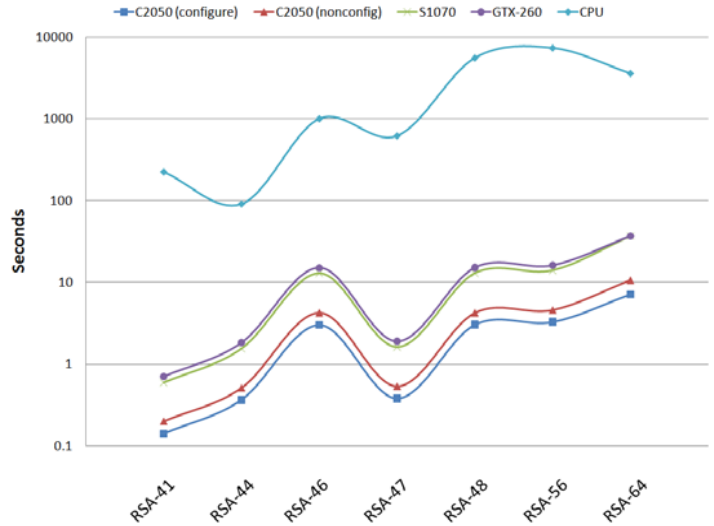


Figure 4: The execution time by *CPFA* and *GPFA* under various platforms.

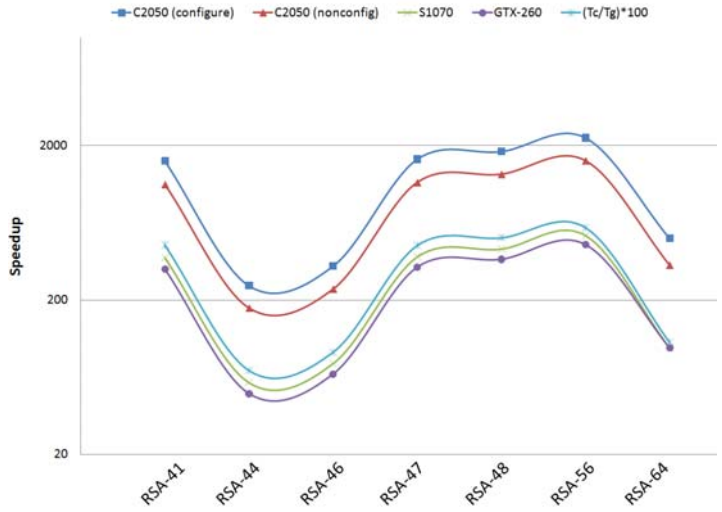


Figure 5: The speedups by comparing *GPFA* with *CPFA* under various platforms.

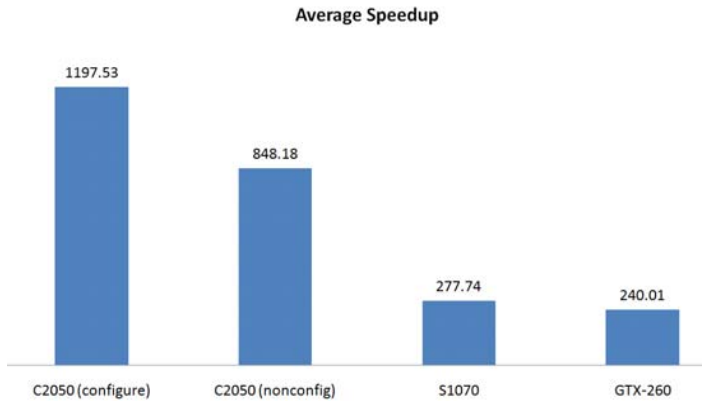


Figure 6: The average speedups achieved by *GPFA* under various platforms.

Figure 5 illustrated the speedups by comparing *GPFA* with *CPFA* under various platforms, where y-axis is the scale of logarithm to base 10. From Figure 5, *GPFA* can achieve 2248x speedup for RSA-56 under the C2050(configure) GPU. Although the speedups by comparing *GPFA* with *CPFA* among the testing data set are not linear (search problem), *GPFA* achieved 1197.5x average speedup under the C2050(configure) GPU (Figure 6). Figure 6 showed the average speedups achieved by *GPFA* under various platforms. From Figure 6, *GPFA* achieved at least 240x average speedup under the GTX-260. Moreover, the performance by *GPFA* under C2050(configure) is better (about 1.4x) than that by *GPFA* under C2050(nonconfig). This result showed that the performance increases when the cache size on GPU increases.

6. Conclusions

RSA is a public-key cryptography algorithm to use a pair (N, E) as the public key and D as the private key. The security of RSA algorithm relies on the hardness of factoring the large number N without the private key D. Recently, it is very hard and no known polynomial time algorithms can be used to extract p and q from a large number N. However, GPU is a massively multi-threaded processor and expects the thousands of concurrent threads to fully utilize its computing power. Hence, it may be a challenge for the RSA algorithm to protect the data when using GPUs with CUDA to factoring the large number N.

In this paper, an efficient parallel RSA decryption algorithm, *GPFA*, for many-core GPUs with CUDA was proposed. The experimental results showed that *GPFA* can achieve 1197.5x average speedup compared with *CPFA*, and within 40 seconds to find out the result of factoring a RSA-64 integer. Although *GPFA* is not used to factoring RSA-128 or larger integers in this paper, it may be possible to factor them by using multiple-GPUs within a reasonable time.

Acknowledgement

Part of this work was supported by the National Science Council under the grants NSC-100-2221-E-126 -007 -MY3 and Industrial Technology Research Institute under the grants SCRPD2B0081.

References

[1] Barreto, P., and Rijmen, V., "The Whirlpool hashing function", *First open NESSIE Workshop*, 2000.

[2] Chen, Z., and Schaumont, P., "pSHS: A Scalable Parallel Software Implementation of Montgomery Multiplication for Multicore Systems", *Proc. Design, Automation and Test in Europe*, 843-848 (2010).

[3] Cook, D., Ioannidis, J., Keromytis, A., and Luck, J., CryptoGraphics: Secret Key Cryptography Using Graphics Cards, 2005.

[4] Costigan, N., and Schwabe, P. "Fast elliptic-curve cryptography on the Cell Broadband Engine", *LNCS*, 368-385 (2009).

[5] Costigan, N., and Scott, M. "Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Play station 3", Cryptology ePrint Archive, 061 (2007).

[6] Diffie, W., and Hellman, M., "New directions in cryptography", *IEEE Trans. Information Theory*, vol. 22, 644-654 (1976).

[7] Fan, W., Chen, X., Li, X., "Parallelization of RSA Algorithm Based on Compute Unified Device Architecture", *Proc. Ninth International Conference on Grid and Cloud Computing*, 174-178 (2010).

[8] Fleissner, S., "GPU-Accelerated Montgomery Exponentiation", *LNCS 4487*, 213 (2007).

[9] Flynn, M., "Some Computer Organizations and Their Effectiveness", *IEEE Trans. Comput*., vol. C-21, 948 (1972).

[10] Fujimoto, N., "High Throughput Multiple-Precision GCD on the CUDA Architecture", *Proc. IEEE International Symposium on Signal Processing and Information Technology*, 507-512 (2009).

[11] Gaj, K., Kwon, S., Baier, P., Kohlbrenner, P., Le, H., Khaleeluddin, M., Bachimanchi, R., "Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware", *LNCS 4249*, 119-133 (2006).

[12] Hankerson, D., Menezes, A., and Vanstone, S.A., Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004.

[13] Harrison, O., and Waldron, J., "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware", *Proc. 2nd International Conference on Cryptology in Africa: Progress in Cryptology*, 350-367 (2009).

[14] Hermans, J., Vercauteren, F., and Preneel, B., "Speed Records for NTRU", *Proc. International conference on Topics in Cryptology*, 73-88 (2010).

[15] Jang, K., Han, S., Han, S., Moon, S., Park, K., "Accelerating SSL with GPUs", *ACM SIGCOMM*, 437-438 (2010).

[16] Lai, X., and Massey, J.L., and Murphy, S., "Markov ciphers and differential cryptanalysis", *Adv. Cryptology*, 17-38(1992).

[17] Lenstra Jr., H. W., "Factoring integers with elliptic curves", *Annals of Mathematics*, vol. 126, no. 3, 649-673 (1988).

[18] Liu, Y., Maskell, D., Schmidt, B. "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units", *BMC Research Notes*, vol. 2, 73 (2009).

[19] Manavski, S, "Cuda compatible GPU as an efficient hardware accelerator for AES cryptography". *Proc. IEEE Signal Processing and Communication*, 65-68 (2007).

[20] McKee, J., "Speeding Fermat's factoring method", *Mathematics of Computation*, vol. 68, 1729-1737 (1999).

[21] Menezes, A.J., Paul, C.V.O., and Scott, A.V, Hand-book of applied cryptography, CRC Press, Boca Raton, Florida, 1996, ISBN 0-8493-8523-7.

[22] Moss, A., Page, D., and Smart, N.P., "Toward Acceleration of RSA Using 3D Graphics Hardware", *Proc. 11th IMA international conference on Cryptography and coding*, 364-383 (2007).

[23] Moss, A., Page, D., and Smart N.P., "Executing Modular Exponentiation on a Graphics Accelerator", *IACR Cryptology ePrint Archive*, 187 (2007).

[24] National Bureau of Standards, Data Encryption Standard, FIPS-Pub.46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977.

[25] National Institute of Standards and Technology, Federal Information Processing Standard 197, The Advanced Encryption Standard (AES), 2001.

[26] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable parallel programming with CUDA", *ACM Queue*, vol. 6, 40-53 (2008).
[27] Pelzl, J., Simka, J., Kleinjung, T., Franke, J., Priplata, C., Stahlke, C., Drutarovsky, M., Fischer, V., and Paar, C. "Area-time efficient hardware architecture for factoring integers with the elliptic curve method", *Journal IEE Proc. Information Security,* vol. 152, no. 1, 67-78 (2005).
[28] Pollard, J. M., "Theorems of Factorization and Primality Testing", *Proc. Cambridge Philosophical Society*, vol. 76, no. 3, 521-528 (1974).
[29] Pollard, J. M., "A Monte Carlo method for factorization", *BIT Numerical Mathematics*, vol. 15, no. 3, 331-334 (1975).
[30] Rivest, R., MIT Laboratory for Computer Science and RSA Data Security, Inc. 1992.
[31] Rivest, R.L., Shamir, A., and Adleman, L.M., "A method for obtaining digital signatures and public-key cryptosystems", *Comm. ACM*, vol. 21, no. 2, 120-126 (1978).
[32] Rosenberg, U., Using Graphic Processing Unit in Block Cipher Calculations. Master's thesis, University of Tartu (2007).
[33] Secure Hash Standard. National Institute of Standards and Technology, Washington, 1995. Note: Federal Information Processing Standard 180-1.
[34] Standards for Efficient Cryptography Group (SECG), SEC 1: Elliptic Curve Cryptography, Version 1.0, September 20, 2000.
[35] Szerwinski, R., and Guneysu, T., "Exploiting the Power of GPUs for Asymmetric Cryptography", *LNCS 5154*, 79-99 (2008).
[36] Yamanouchi, T., AES Encryption and Decryption on the GPU. Addison Wesley Professional, 2007.

Yu-Shiang Lin received a B.S. degree in Department of Computer Science and Information Engineering from Chang Gung University in 2010. He is currently a Master student in the Department of Computer Science and Information Engineering at Chang Gung University. His research interests are in the areas of Parallel Processing and Next-Generation Sequencing.

Chun-Yuan Lin received a B.S. degree in Department of Information Engineering and Computer Science from Feng Chia University in 1999, and the M.S. and Ph.D. degrees in Department of Information Engineering and Computer Science from Feng Chia University in 2000 and 2003, respectively. He joined the Institute of Molecular and Cellular Biology and the Department of Computer Science at National Tsing Hua University as a post-doctoral fellow in 2003 and 2006, respectively. In 2007, he joined the Department of Computer Science and Information Engineering at Chang Gung University as an assistant professor. He also is a faculty Member at Research center for Emerging Viral Infections in Chang Gung University. His research interests are in the areas of Parallel and Distributed Computing, Proteomics, Genomics, Systems Biology, Next-Generation Sequencing and Computational Chemistry.

Der-Chyuan Lou was born in Chiayi, Taiwan, Republic of China, on Mar. 18th, 1961. He received the B.S. degree from Chung Cheng Institute of Technology (CCIT), National Defense University, Taiwan, R.O.C., in 1987, and the M.S. degree from National Sun Yat-Sen University, Taiwan, R.O.C., in 1991, both in electrical engineering. He received the Ph.D. degree in 1997 from the Department of Computer Science and Information Engineering at National Chung Cheng University, Taiwan, R.O.C. He was an Assistant, Lecturer, Associate Professor, and Professor with the Department of Electrical Engineering at CCIT, from 1987 to 2009. He had served as Director of Computer Center of CCIT from 2004 to 2006. Currently, he is a Professor with the Department of Computer Science and Information Engineering, Chang Gung University. His research interests include multimedia security, steganography, cryptography, computer arithmetic, and distributed system. Prof. Lou is currently an Area Editor for Security Technology of Elsevier Science's Journal of Systems and Software. He is the owner of the eleventh AceR Dragon Ph.D. Dissertation Award. He has been selected and included in the 15th and 29th edition of Who,s Who in the World which has been published in 1998 and 2012, respectively.