Implementation of public key algorithms in CUDA

Hao Wu

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2010

Abstract

In the field of cryptography, public key algorithms are widely known to be slower than symmetric key alternatives for the reason of their basis in modular arithmetic. The modular arithmetic in e.g. RSA and Diffie Hellman is computationally heavy when compared to symmetric algorithms relying on simple operations like shifting of bits and XOR. Therefore, how to make a more efficient and faster implementation of public key algorithms is publicly concerned.

With the development of the GPGPU (General-purpose computing on graphics processing units) field, more and more computing problems are solved by using the parallel property of GPU (Graphics Processing Unit). CUDA (Compute Unified Device Architecture) is a framework which makes the GPGPU more accessible and easier to learn for the general population of programmers. This is because it builds on C and hides many of the complicated details of how the GPU works from a CUDA developer. Using the unique properties of the GPU through CUDA has greatly increased the efficiency of many computational problems. Multiplication of big integers is one of the building blocks in doing modular arithmetic. Running the public key algorithms by use of the parallel properties of the GPU in modular multiplication and modular exponentiation may be a solution to this problem.

The target in this research is to study and analyse the majority of algorithms related to the modular multiplication and modular exponentiation, and then to design and make an implementation of a public key algorithm in CUDA. Finally, this project will compare the performance between the GPU implementation and the CPU implementation in order to look into the possibility of improving the performance of public key algorithms. The research questions are divided into four groups, the first one regarding modular multiplication and modular exponentiation of big integers and their parallelism, the second one about integrating parallel modular multiplication and modular exponentiation into the public key algorithm, the third one concerning optimization of the algorithm, and final one regarding performance comparison of public key algorithm between the GPU implementation and the CPU implementation.

Acknowledgements

First of all I will like to thank my supervisor Patrick Bours for valuable guidance through this master thesis. His continued support, interest and inspiration were helpful during the thesis. I will also like to thank my co-supervisor Maciej Pietka for suggestions toward the end of the thesis period. I would also like to thank them for introducing me to the interesting fields that are parallel computation and GPGPU.

My next thanks are going to my thesis opponent, Fredrik Gundersen, for fabulous feedbacks to my thesis. I would also like to thank my student friends for motivation and discussions during the master thesis.

A special thanks goes also to all my friends and family for accompanying and supporting me during the thesis period.

Hao Wu, 30th June 2010

Contents

List of Figures

List of Tables

Abbreviations

- GPU - Graphics Processing Unit
- CPU - Central Processing Unit
- CUDA - Compute Unified Device Architecture
- GPGPU - General-purpose computing on graphics processing units
- MP - Multiprocessor
- SP - Stream processor
- IO - Input and output
- SPMD - Single Program Multiple Data
- SIMD - Single Instruction Multiple Data

1 Introduction

This chapter introduces the topics covered by this project, the problem description, the justification, motivation and benefits, the research questions, and planned contributions for the master thesis.

1.1 Topic covered by the project

Public key cryptography is a fundamental and widely used technology around the world. Most public key algorithms are based on modular arithmetic including RSA, Elgamal and Diffie hellman. Public key encryption and decryption is computationally heavy because a lot of modular multiplications with very large numbers is needed to perform these tasks. Therefore public key algorithm is known to be much slower then symmetric key algorithms. Recently the field of using GPUs for general purpose computing has become more widespread. Many computational problems have gained a significant performance increase by using the highly parallel properties of the GPU. CUDA is a framework which makes these kinds of implementations more available to the general public of programmers.

In this master project, we are looking into the possibility of improving the performance of public key algorithms by using CUDA, and compare the performance between the GPU implementation and the CPU implementation.

1.2 Keywords

Public key algorithm, parallel computation, CUDA, GPU

1.3 Problem description

Implementing a public key cryptosystem is always a tradeoff between security and efficiency. The problem with the number theoretic cryptosystems (i.e. RSA) is that they require a lot of computational power for providing a high level of security and most likely a low level of efficiency. Public key algorithms are known to be slower than symmetric key alternatives because of their basis in modular arithmetic. Therefore, how to make a more efficient and faster implementation of public key algorithms is concerned.

Running the public key algorithms by use of the parallel properties of the GPU in modular multiplication and modular exponentiation may be a solution to this problem. Multiplication of big integers is one of the building blocks in doing modular arithmetic. The field of General-purpose GPU which is about solving problems other than graphics rendering using the GPU was until recently without a good solution. CUDA is a framework which makes these kinds of implementations more available to the general public of programmers. Using the unique properties of the GPU through CUDA has greatly increased the efficiency of many computational problems.

The target in this research is to study and analyse the majority of algorithms related to the modular multiplication and modular exponentiation, and then to design and make an implementation of a public key algorithm in CUDA. Finally, this project will compare the performance between the GPU implementation and the CPU implementation in order to look into the possibility of improving the performance of public key algorithms.

1.4 Justification, motivation and benefits

The necessity for information security has become more and more widespread during these days. Fast modular exponentiation algorithms are often considered of practical significance in public-key cryptosystems. Parallelization of public key algorithms could be very useful for a high level of security system and save a lot of computation time. With the combination of them, the public key cryptosystem will be more efficient and effective for those kinds of system.

Furthermore, in this research the performance of public key algorithm will be compared between the GPU implementation and the CPU implementation. It could be used to determine the direction of parallelization of public key algorithms in the future. With the development of the GPGPU field, modern graphics processing units (GPUs) have been at the leading edge of increasing chip-level parallelism. Current NVIDIA GPUs are many core processor chips with parallelism architecture. This degree of hardware parallelism reflects the fact that GPU architectures evolved not only to fit the needs of real-time computer graphics but also parallel computing. On the other hand, the GPU is easy use and cheaper compared to a computer cluster for the purpose of parallel computations. So the research in this field will have a different angle for parallel computation.

1.5 Research questions

The research questions are divided into four groups, the first one regarding modular multiplication and modular exponentiation of large integers and their parallelism, the second one about integrating parallel modular multiplication and modular exponentiation into the public key algorithm, the third one concerning optimization of the algorithm, and final one regarding performance comparison of public key algorithm between the GPU implementation and the CPU implementation.

First of all, public key encryption and decryption are computationally heavy because a lot of modular multiplication and modular exponentiation with very large numbers are needed to perform these tasks. Because the bit-length of a key needs to be larger than 1024 bits for security reasons, the computations for public key cryptosystem are time-consuming. This stage mainly concerns the majority of algorithms related to the modular multiplication and modular exponentiation, and how to effectively parallelize the modular multiplication and modular exponentiation for big integers in CUDA.

Second, how can parallel modular exponentiation be integrated into a public key algorithm? Can the whole public key algorithm only be implemented on a GPU by using CUDA?

Moreover, how to optimize this algorithm in order to achieve a more efficient implementation on a GPU?

Finally, the ordinary public key algorithm will be implemented on a CPU and parallel public key algorithm on a GPU, and their performance will be compared. Can a public key crypto algorithm be implemented fast on a CUDA-enabled GPU by using the massive parallel processing properties? Is it more efficient to implementat it on a CPU?

1.6 Planned contributions

This research will come up with results on how different the performance of public key algorithms between the GPU implementation and the CPU implementation will be. The focus is to contribute with new ideas on how parallel implementation of a public key algorithm on a CUDA-enabled GPU, how to achieve more efficient implementation in CUDA, and how to design public key algorithm from hardware aspect. If the difference of the performance is significant, it would be no problems to use this technology to realize the public key cryptosystem by using CUDA. Otherwise, we should reconsider the method of parallelization for public key algorithms from another angle.

2 State of the art

At the present time, security becomes a tremendously important issue to deal with when the Internet provides essential communication between millions of people and is being increasingly used as a tool for commerce. There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of cryptography. Cryptography is the science of writing in secret code and is an ancient art. In data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about any network, particularly the Internet.

Generally there are two types of cryptographic schemes typically used to accomplish these goals: secret key (symmetric) cryptography and public-key (asymmetric) cryptography. The operation of cryptography typically includes two processes: encryption as the process of transforming information so that it is unintelligible to an intruder, and decryption as the process of transforming the encrypted information so that it is intelligible again.The original unencrypted data is referred to as plaintext. It is encrypted into ciphertext, which will in turn be decrypted into usable plaintext.

With secret key cryptography, a single key is used for both encryption and decryption, e.g., Data Encryption Standard (DES) [5] and Advanced Encryption Standards (AES) [6]. The biggest difficulty with this approach, of course, is the distribution of the key. Public-key cryptography has been said to be the most significant development in cryptography in the last hundreds of year. In this scheme, a two-key cryptosystem is used in which two parties could engage in a secure communication over a non-secure communication channel without having to share a secret key, e.g. RSA[7], Elliptic Curve Cryptography (ECC) [8].

The following four requirements have been identified as the framework for information security [9]:

- Confidentiality: Protecting the data from all but the intended receivers.
- Authentication: Proving one's identity.
- Integrity: Ensuring no unauthorized alteration of data.
- Non-repudiation: Preventing an entity from denying previous commitments or actions.

The universal technique for providing confidentiality of transmitted data is conventional cryptography. However conventional cryptosystems do not satisfy the requirements of authentication, integrity, and non-repudiation. Public key cryptography is the first truly revolutionary advance in cryptography that satisfies these requirements [10].

2.1 Storage structure for large number

Nowadays, most compilers support 64 bits integer operation, where the integers calculated must be at most 64 bits in length, which is too short for the RSA algorithm. In practical applications, the length of a key n must be large enough in order to guarantee the security of a public-key cryptographic system. So the efficiency of a public-key cryptographic system depends on the large number calculation speed. The classic large number storage method [11] is string based, a large number is stored in a character type array, and then we can construct the corresponding function to perform add, subtract, multiply and divide operation based on the array. However the efficiency of this scheme is very low because for a 1024 bits number, the length of the decimal form is about several hundred, any numeric operation should do multiple nested loops on two long character array, besides a large extra space is needed to store the carry flag and middle results, which leads to heavy system resource occupation and low efficiency [12] [13] [14].

In [15], a dynamic implementation of big integers in C++ is presented. There are two object types used in big integer implementation, CDigit and CBigInt, as represented by in Figure 1.

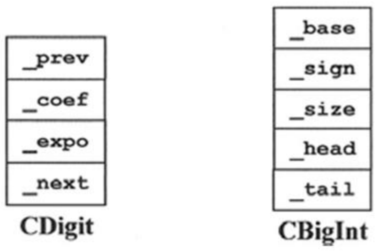


Figure 1: Two object types: CDigit and CBigInt

CDigit type of objects are used to store the digits making a big integer. It has the digit in a given base, the weight of the digit represented by the exponent of the base, the address of the digit to the left, and the address of the digit to the right. CBigInt type of objects are used to store big integers as a doubly-linked list of CDigit type of objects. It holds the base in which the number is being stored, sign of the number, size to represent the number of non-zero digits unless there is only one zero digit in the number, head to store the address of the first digit, and tail to store the address of the last digit. A big integer such as 400000020502 can be represented by the following expression where b=10 is the base of the big integer, and corresponding doubly-linked list representation is shown in Figure 2.

$$400000020502 = 4b^{15} + 2b^4 + 5b^2 + 2.$$

Generally any big integer of size n can be represented by the following expression:

$$c_{n-1}b^{n-1} + c_{n-2}b^{n-2} + c_{n-3}b^{n-3} \ldots + c_0 b^0.$$

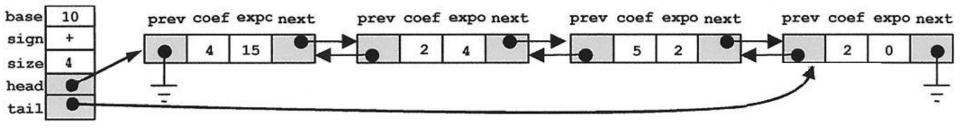


Figure 2: Doubly-linked list representation for the integer 4000000000020502

Obviously, this data structure efficiently saves the memory if the big integer contains lots of zero digits, and reduces the time that transfers the data among different devices. However, arbitrary access of any digit in CBigInt is less efficient than the classic large number storage method. Thus it is not fit for the parallel computation.

A lot of other research has been done to enhance the speed of a cryptosystem. In [16], an efficient public key encryption scheme was proposed, which is an improved and enhanced version of original RSA scheme. The proposed RSA encryption scheme is based on linear group over the ring of integer modulo a composite modulus n which is the product of two distinct prime numbers. This encryption scheme has no restriction in encryption and decryption order and is claimed to be efficient, scalable and dynamic. [17] proposes a new method to realize a unified architecture for both RSA and ECC public key cryptosystems using a Signed-Digit (SD) number system so that the carry propagation in the RSA computation can be avoided. Hence, the critical path for the computation of RSA and ECC with the same key length can be shortened compared to other methods using a full adder implementation.

All these methods more or less try to improve the structure or architecture of public-key algorithm in order to enhance the computing speed of them.

2.2 Public-key cryptography

The first revolution event in the era of the public key cryptography is coming in 1976 when Diffie and Hellman [18] published their well-known paper entitled "New directions in cryptography". This paper proposed a great concept for public key cryptography and to build a scheme without a secure communication, but able to provide a secret communication. However, Diffie and Hellman suggested such technique for distributing the private key to be employed in the classical schemes in insecure communication channel [16].

In 1978 Rivest, Shamir and Adleman (RSA) [7] introduced the first applied scheme which is the most popular public key scheme. The security of the RSA public key scheme is depended on the intractability of factoring the integer modulus which is the product of two large and distinct prime numbers. Elliptic Curve Cryptography (ECC) was first proposed for cryptographic use independently by Neal Koblitz [19] and Victor Miller [8] in 1986 and 1987. In 1979, Rabin [20] suggested a scheme which also relied on the factoring of a composite modulus, which is the product of large Blum integer numbers and the result of decryption scheme are four messages; just one from the results represent the original message. In 1992, Shimada [21] enhanced the Rabin scheme by using the extension Rabin public key encryption scheme, which employed certain assumption in a private key utilizing the Jacobi symbol. In 1998, Okamoto [22] proposed a new public key cryptosystem as secure as factoring relied on RSA and Rabin schemes. In 1999, Pointcheval [23] introduced a new public key encryption scheme based on the dependent RSA and Rabin Schemes. In 2006, Sahadeo Padhye [24] modified dependent RSA and Rabin public key cryptosystem using certain conditions to public and private keys.

While compared with secret key crypto, public-key crypto can either be used for data encryption or digital signature. However, the disadvantage of number theoretic cryptosystems is that they require a lot of computational power providing a high level of security and most likely a low level of efficiency. Public key algorithms are known to be slower than symmetric key alternatives because of their basis in modular arithmetic. The modular arithmetic in e.g. Diffie Hellman, ECC and RSA, is computationally heavy when compared to symmetric algorithms relying on simple operations like XOR and shifting of bits.

Taking RSA, which is the most widely used public key algorithm, as an example. In order to guarantee the security of an RSA system, the length of public and private keys is usually greater than 1024 bits in current commerce use. Consequently the key generation and data encryption/decryption process are all large number operations, which make the speed of an RSA algorithm about 1000 times slower than a DES algorithm [11]. The processing speed is a major drawback of the RSA algorithm either for hardware or software implementation, so how to design an effective large number operation scheme is an important question.

2.3 Modular exponentiation

2.3.1 Modular arithmetic

In general, public-key cryptographic systems consist of raising elements to large powers and reducing the result modulo some given element. Such operation is usually called modular exponentiation and is performed by using modular multiplications repeatedly. The practicality of a given cryptographic system, like DH and RSA, depends heavily on how fast modular exponentiations are performed. Consequently, it also depends on how efficiently modular multiplications are done as these are at the base of the computation. This problem has received much attention over the years.

The characteristics of the modular arithmetic like addition, subtraction, and multiplication as follows [25]:

$$\begin{aligned}(u+v) \bmod m &= ((u \bmod m) + (v \bmod m)) \bmod m \\ (u-v) \bmod m &= ((u \bmod m) - (v \bmod m)) \bmod m \\ (u \times v) \bmod m &= ((u \bmod m) \times (v \bmod m)) \bmod m\end{aligned}$$

Therefore the modular arithmetic can be applied to the any step of the computation procedure if it only includes operations of addition, subtraction and multiplication.

This section discusses methods for computing integer modular exponentiation, that is, raising an integer g to an integer power e and then reducing the result modulo some given integer m, especially when g, e, and m are rather large.

2.3.2 Naive modular exponentiation

The naive method of modular exponentiation applies modular multiplication repeatedly. For example g=4, e=13, and m=497. The calculation of $c \equiv g^e (modm)$ is presented in Table 1. The final answer for c is thus 455. It performs the modular multiplication e-1 times. This method is not efficient because e-1 modular multiplications are required.

In the field of public key algorithm, the exponent e is usually very large in order to provide a high level of security. Therefore the performance of public key cryptosystems is mainly determined by the implementation efficiency of the modular multiplication and modular exponentiation. In addition the plaintext, the cipher text, or possibly a partially ciphered text are usually large (i.e. 1024 bits or more). Thus it is essential to attempt to minimize the number of modular multiplications performed and to reduce the time required by a single modular multiplication in order to improve time requirements of the encryption and decryption operations. In fact, there are much more efficient methods.

1. e=1, c= 4 mod 497 = 4.
2. e=2, c=(4×4) mod 497 = 16 mod 497 = 16.
3. e=3, c=(16×4) mod 497 = 64 mod 497 = 64.
4. e=4, c=(64×4) mod 497 = 256 mod 497 = 256.
5. e=5, c=(256×4) mod 497 = 1024 mod 497 = 30.
6. e=6, c=(30×4) mod 497 = 120 mod 497 = 120.
7. e=7, c=(120×4) mod 497 = 480 mod 497 = 480.
8. e=8, c=(480×4) mod 497 = 1920 mod 497 = 429.
9. e=9, c=(429×4) mod 497 = 1716 mod 497 = 225.
10. e=10, c=(225×4) mod 497 = 900 mod 497 = 403.
11. e=11, c=(403×4) mod 497 = 1612 mod 497 = 121.
12. e=12, c=(121×4) mod 497 = 484 mod 497 = 484.
13. e=13, c=(484×4) mod 497 = 1936 mod 497 = 445.

Table 1: Modular exponentiation applies modular multiplication repeatedly.

Algorithm: Right-to-left binary modular exponentiation

```
Input: an element g and integer $e \geq 1$, and a modulus m.
Output: $g^e$ mod m.
1. $A = 1, S = g, E = e$.
2. While $E \neq 0$ do the following:
   2.1. If E is odd, then $A = (A \cdot S)$ mod m, $E = E - 1$.
   2.2. $E = E/2$.
   2.3. If $E \neq 0$, then $S = (S \cdot S)$ mod m.
3. Return ( A ).
```

Table 2: Right-to-left binary modular exponentiation.

2.3.3 Repeated square-and-multiply methods

The repeated square-and-multiply modular exponentiation algorithm [25] is based on the simple observation that for an even e, $g^e \bmod m = (g^{e/2} \times g^{e/2}) \bmod m$. The recursive definition of exponentiation by squaring is illustrated by Figure 3.

$$
ModExp(g, e, m) = \begin{cases} 1, & if\ e = 0 \\ (g \times ModExp(g, (e-1), m)) \bmod m, & if\ e\ is\ odd \\ ModExp(g, e/2, m)^2 \bmod m, & if\ e\ is\ even \end{cases}
$$

Figure 3: Recursive definition of modular exponentiation by squaring.

The repeated square-and-multiply algorithm reduces the amount of modular multiplications needed to at most 2t, where t is the number of bits in the binary representation of the exponent e. This method is a great improvement for a large e. The Table 2 describes the algorithm of Right-to-left binary modular exponentiation [25] to compute $g^e \bmod m$, which is base on the idea of modular exponentiation by squaring.

This algorithm is called right-to-left binary modular exponentiation, because the binary representation of the exponent is computed from right to left. The exponent e is actually broken into its binary representation. The lowest bits of e are considered first.

The Table 3 describes Left-to-right binary modular exponentiation algorithm [25]. This algorithm considers the binary representation of the exponent from left to right.

```
Algorithm: Left-to-right binary modular exponentiation
Input: an element g and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$,and a modulus m.
Output: $g^e$ mod m.
1. A = 1.
2. For i from t down to 0 do the following:
   2.1. $A = (A \cdot A) \bmod m$.
   2.2. If $e_i = 1$, then $A = (A \cdot g) \bmod m$.
3. Return ( A ).
```

Table 3: Left-to-right binary modular exponentiation.

i	e_i	A (step2.1)	A (step 2.2)
3	1	1	g mod m
2	1	g^2 mod m	g^3 mod m
1	0	g^6 mod m	g^6 mod m
0	1	g^{12} mod m	g^{13} mod m

Table 4: Left-to-right binary moduar exponentiation with the exponent 1101.

For example, the exponent e is the binary 1101. Table 4 lists the value of A in each iterative implementation of Left-to-right binary modular exponentiation algorithm. The leftmost 1 of e will be considered first. Then we have another bit, so we square. That's g^2. Now, the new bit of e is 1, so we multiply a g, that's g^3. We have another bit, so again square, that's g^6. The new bit is 0, so nothing is multiplied. And we have one more bit, so once again square, getting g^{12}, and finally multiplying a g, getting g^{13}. Indeed, 1101 is the binary representation of 13.

Obviously, the repeated square-and-multiply methods of modular exponentiation are far more efficient than the naive method of repeated multiplication. In general, the mentioned repeated square-and-multiply algorithms are almost the same speed.

The Table 5 describes Left-to-right k-ary modular exponentiation algorithm [25], which is a generalization of Left-to-right binary modular exponentiation algorithm. But this algorithm processes more than one bit of the exponent per iteration. This method is only efficient if the pre-computation is done once and used multiple times.

In this algorithm, the exponent e is broken into larger pieces since it is in base 2^k, instead of breaking the exponent into bits of its base-2 representation. In this way, it can save some computations. In a similar manner, Right-to-left binary modular exponentia-

```
Algorithm: Left-to-right k-ary modular exponentiation
Input: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$,where $b = 2^k$ for some $k \geq 1$,and a modulus m.
Output: $g^e$ mod m.
1. Precomputation.
   1.1. $g_0 = 1$.
   1.2. For i from 1 to $(2^k - 1)$ do: $g_i = (g_{i-1} \cdot g) \bmod m$. ( Thus $g_i = g^i$ mod m ).
2. A = 1.
3. For i from t down to 0 do the following:
   3.1. A = ( $A^{2^k}$ ) mod m.
   3.2. A = ( $A \cdot g_{e_i}$ ) mod m.
4. Return ( A ).
```

Table 5: Left-to-right k-ary modular exponentiation.

Algorithm: Sliding-window exponentiation

```
Input: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_2$, with $e_t = 1$, an integer $k \geq 1$,
and a modulus $m$.
Output: $g^e$ mod m.
1. Precomputation.
   1.1. $g_1 = g$, $g_2 = g^2$.
   1.2. For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} = (g_{2i-1} \cdot g_2)$ mod m.
2. $A = 1, i = t$
3. While $i \geq 0$ do the following:
   3.1. If $e_i = 0$ then do: $A = A^2$ mod $m$, $i = i - 1$.
   3.2. Otherwise ($e_i \neq 0$), find the longest bitstring $e_i e_{i-1} \cdots e_l$
        such that $i - l + 1 \leq k$ and $e_l = 1$, and do the following:
        $A = A^{2^{i-l+1}} \cdot g_{(e_i e_{i-1} \cdots e_l)_2}$, $i = l - 1$.
4. Return ( $A$ ).
```

Table 6: Sliding-window exponentiation.

i	A	Longest bitstring
13	1	101
10	g^5	101
7	$(g^5)^8 g^5 = g^{45}$	111
4	$(g^{45})^8 g^7 = g^{367}$	-
3	$(g^{367})^2 = g^{734}$	-
2	$(g^{734})^2 = g^{1468}$	101
0	$(g^{1468})^8 g^5 = g^{11749}$	-

Table 7: An example of Sliding-window exponentiation.

tion algorithm can be generalized to the k-ary case.

2.3.4 Sliding-window exponentiation

Sliding-window exponentiation algorithm [25] as shown in Table 6 reduces the amount of precomputation compared to k-ary exponentiation algorithm, and reduces the average number of multiplications performed.

The k is called the window size. For example, take $e = 11749 = (10110111100101)_2$ and $k = 3$. Table 7 illustrates the steps of Sliding-window exponentiation algorithm. When e_i is 0, i is equal to 4 and 3 in the Table 7, the result A is just squared as shown in the step 3.1 of Table 6.

2.4 Modular multiplication

The naive method of modular exponentiation applies modular multiplication repeatedly. There are two different ways to perform modular multiplication $A \times B$(mod m): multiplying, i.e. computing $P = A \times B$; then reducing, i.e. $R = P$(mod m) or interleave the multiplication and the reduction steps. The straightforward way to implement a multiplication is based on an iterative adder-accumulator for the generated partial products. But this solution is very slow since the final result is only available after n clock cycles; n is the size of the operands.

There are various algorithms that implement modular multiplication. The most prominent are Karatsuba Ofman's [26] and Booth's [27] methods for multiplying, Barrett's [28] [29] [30] method for reducing, and Montgomery's algorithms [31] for interleaving

$$\begin{aligned} X \times Y &= (x_4x_3x_2x_1x_0)_b \times (y_4y_3y_2y_1y_0)_b \\ &= (x_4x_3x_2x_1x_0)_b \times y_0 \\ &+ (x_4x_3x_2x_1x_0)_b \times y_1 \times b \\ &+ (x_4x_3x_2x_1x_0)_b \times y_2 \times b^2 \\ &+ (x_4x_3x_2x_1x_0)_b \times y_3 \times b^3 \\ &+ (x_4x_3x_2x_1x_0)_b \times y_4 \times b^4 \end{aligned}$$

Table 8: The multiplication procedure.

$$\begin{aligned} X \times Y \bmod m = &\ (\ (x_4x_3x_2x_1x_0)_b \times (y_4y_3y_2y_1y_0)_b\ ) \bmod m \\ = &\ (\ (\ (x_4x_3x_2x_1x_0)_b \times y_0\ ) \bmod m \\ + &\ \ (\ (x_4x_3x_2x_1x_0)_b \times y_1 \times b\ ) \bmod m \\ + &\ \ (\ (x_4x_3x_2x_1x_0)_b \times y_2 \times b^2\ ) \bmod m \\ + &\ \ (\ (x_4x_3x_2x_1x_0)_b \times y_3 \times b^3\ ) \bmod m \\ + &\ \ (\ (x_4x_3x_2x_1x_0)_b \times y_4 \times b^4\ ) \bmod m \\ &\ \ )\bmod m \end{aligned}$$

Table 9: Naive interleaving multiplication and reduction.

multiplication and reduction.

2.4.1 Naive interleaving multiplication and reduction

An integer $X = (x_4x_3x_2x_1x_0)_b$ multiplied by another integer $Y= (y_4y_3y_2y_1y_0)_b$ base b is described in Table 8. In this procedure, the operation that an intermediate result is multiplied by b^i $(i > 0)$ can be performed by left shifts.

According to the characteristics of the modular arithmetic, modular reduction can be applied to each intermediate result in order to avoid generating a large number when computing $X \times Y \bmod m$ as shown in Table 9. The advantage is that 2n-digit full product doesn't need to be stored before the modular reduction starts.

2.4.2 Karatsuba-Ofman Method

Karatsuba-Ofman's algorithm is considered one of the fastest ways to multiply long integers. Karatsuba-Ofman's algorithm [26] is based on a divide-and-conquer strategy. A multiplication of a 2n-digit integer is reduced to two n-digits multiplications, one (n+1)-digits multiplication, two n digits subtractions, two right-shift operations, two n-digits additions and two 2n-digits additions. This algorithm was proposed long ago but it is still as far as we know.

The basic step of Karatsuba's algorithm is shown in Table 10. It allows us to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts. Taking

$$\begin{aligned} N &= (\ x_1 \cdot b^k + x_0\ ) \cdot (\ y_1 \cdot b^k + y_0\ ) \\ &= x_1 \cdot y_1 \cdot b^{2k} + b^k(x_1 \cdot y_0 + x_0 \cdot y_1) + x_0 \cdot y_0 \end{aligned}$$

Table 10: The basic step of Karatsuba's algorithm.

$$\begin{aligned}
&1234 = 12 \times 10^2 + 34\\
&5678 = 56 \times 10^2 + 78\\
&z_2 = 12 \times 56 = 672\\
&z_0 = 34 \times 78 = 2652\\
&z_1 = (12+34)(56+78) - z_2 - z_0 = 46 \times 134 - 672 - 2652 = 2840\\
&\text{result} = z_2 \times 10^{2\times 2} + z_1 \times 10^2 + z_0\\
&\qquad = 672 \times 10000 + 2840 \times 100 + 2652 = 7006652
\end{aligned}$$

Table 11: An example of Karatsuba's algorithm.

an example shown in Table 11, compute the product of 1234 and 5678, to describe the basic method of this algorithm.

In the practical public key cryptosystem, the three multiplications in Karatsuba's basic step could involve large numbers. Therefore, those products can be computed by recursive calls of the Karatsuba's algorithm. The recursion can be applied until the numbers are so small that they can be computed directly. The Karatsuba-Ofman recursive multiplication algorithm applied modular reduction is shown in Table 12. Moreover, the "Product i" (i =1, 2, 3) can be computed in parallel and applied modular arithmetic to the result.

```
Algorithm KaratsubaOfman(X, Y, m)
    If (Size(X) = 1) Then KaratsubaOfman= OneBitMultiplier(X, Y)
    Else Product1 := KaratsubaOfman(High(X), High(Y), m);
        Product2 := KaratsubaOfman(Low(X), Low(Y), m);
        Product3 := KaratsubaOfman(High(X)+Low(X), High(Y)+Low(Y), m);
        KaratsubaOfman := ( RightShift(Product1, Size(X)) mod m
            + RightShift(Product3-Product1-Product2, Size(X)/2) mod m
            + Product2 ) mod m;
End KaratsubaOfman.
```

Table 12: Karatsuba-Ofman recursive multiplication algorithm.

2.5 Modular reduction

2.5.1 Naive modular reduction

A modular reduction is simply the computation of the remainder of an integer division. It can be presented by:

$$X \bmod m = X - \lfloor X/m \rfloor \times m$$

The naive sequential division algorithm, see the Table 13, successively subtracts the modulus until the remainder that is non-negative and smaller than the modulus is found. Note that a negative remainder may be obtained after a subtraction. In that case, we have to store the last non-negative remainder which will be the expected remainder. Nevertheless, a division is very expensive even compared with a multiplication.

2.5.2 Barrett modular reduction

Barrett modular reduction [28] [29] [30] shown in Table 14 computes $r = x \bmod m$ given x and m. The algorithm requires the precomputation of the quantity $\mu = \lfloor b^{2k}/m \rfloor$

```
Algorithm NaiveReduction(P, M)
    Int R = P;
    Do R = R - M;
    While R > 0;
    If R ≠ 0 Then R = R + M;
    Return R;
End NaiveReduction
```

Table 13: Naive reduction algorithm.

```
Algorithm: Barrett modular reduction
Input: positive integers $x = (x_{2k-1} \cdots x_1 x_0)_b$, a modulus $m$, and $\mu = \lfloor b^{2k}/m \rfloor$;
assume $b > 3$.
Output: $r = x \bmod m$.
1. $q_1 = \lfloor x/b^{k-1} \rfloor$,  $q_2 = q_1 \cdot \mu$,  $q_3 = \lfloor q_2/b^{k+1} \rfloor$.
2. $r_1 = x \bmod b^{k+1}$, $r_2 = q_3 \cdot m \bmod b^{k+1}$, $r = r_1 - r_2$.
3. If $r < 0$ then $r = r + b^{k+1}$.
4. While $r \geq m$ do: $r = r - m$.
5. Return ( r ).
```

Table 14: Barrett modular reduction.

where b is the base. The reduction then takes the form shown as below, which requires two k-bit multiplies and one k-bit subtract.

$$r = x - \left\lfloor \lfloor x/b^{k-1} \rfloor \cdot \mu / b^{k+1} \right\rfloor \cdot m \,, \qquad \mu = \lfloor b^{2k}/m \rfloor$$

It is advantageous for the modular exponentiation because many reductions are performed with a single modulus. The precomputation takes a fixed amount of work, which is negligible in comparison to modular exponentiation cost. Typically, the radix b is chosen to be close to the word-size of the processor. However, Barrett Reduction can only reduce numbers that are, at most, twice as long as the modulus.

All divisions performed in Barrett modular reduction algorithm are simple right-shifts of the base b representation. In addition, all modular reduction in this algorithm can be performed with a smart method because the modulus is b^{k+1}. For example:

$x = (234235)_b$, $b = 10$, $k = 3$
$\lfloor x/b^{k-1} \rfloor = \lfloor 234235/100 \rfloor = 2342$. (right shifts)
$x \bmod b^{k+1} = 234235 \bmod 10000 = 4235$. (last k+1 digits of x are the result)

Barrett reduction, when used to reduce a single number, is slower than a normal divide algorithm. However, by precomputing some values, one can easily far exceed the speed of normal modular reductions. Barrett reduction can be used to reduce various numbers modulo a single number many times, for example, when doing modular exponentiation. Barrett reduction is not particularly useful when used with small numbers (32 or 64 bits); its benefits occur when using numbers that are implemented by multiple precision arithmetic libraries, such as when implementing the RSA cryptosystem, which uses modular exponentiation with large numbers, to encrypt and decrypt.

2.6 Montgomery's algorithms

It is very inefficient that the computation performed in the naive modular multiplication algorithm for the reason that it may require 2n-1 subtractions, 2n comparisons and an

Algorithm: Montgomery modular multiplication

```
Input: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$
with $0 \leq x, y \leq m$, $R = b^n$ with $\gcd(m, b)$=1, and $m' = -m^{-1}$ mod $b$.
Output: $xyR^{-1}$ mod $m$.
1. $A = 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to (n-1) do the following:
   2.1. $u_i = (a_0 + x_i y_0)m'$ mod $b$.
   2.2. $A = (A + x_i y + u_i m)/b$.
3. If $A \geq m$ then $A = A - m$.
4. Return ( $A$ ).
```

Table 15: Montgomery modular multiplication.

extra addition. The Montgomery's algorithm [31] is one of the widely used algorithms for efficient modular multiplication. This algorithm computes the product of two integers modulo a third one without performing division by the modulus m.

The Montgomery modular multiplication algorithm shown in Table 15 is the most efficient modular multiplication algorithm available. The Montgomery multiplication methods constitute the core of the modular exponentiation operation which is the most popular method used in public-key cryptography for encrypting and signing digital data.

The RSA algorithm and the Diffie-Hellman key exchange scheme require the computation of modular exponentiation, which is broken into a series of modular multiplications by the application of the binary or k-ary methods.

The Montgomery modular multiplication algorithm computes

$$\text{Mont}(\ x, y, m\ ) = x \cdot y \cdot R^{-1} \bmod m.$$

given $x, y < m$ and R such that $\gcd(m, R) = 1$. Even though the algorithm works for any R which is relatively prime to m, it is more useful when R is taken to be a power of the radix. In this case, the Montgomery modular multiplication algorithm performs divisions by a power of the radix, which is an intrinsically fast operation of right shifts as mentioned early. This leads to a simpler implementation than ordinary modular multiplication.

As shown in Table 15, multiplication modulo the base b and division by b are both intrinsically fast operations as mentioned in previous section, since b is a power of base. Thus the Montgomery modular multiplication algorithm is potentially faster and simpler than ordinary computation of xy mod m, which involves division by m.

However, it is not a good idea to use the Montgomery modular multiplication algorithm when a single modular multiplication is to be performed, because it is time-consuming to convert the final output xyR^{-1} mod m into the desired result xy mod m. It is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute modular exponentiation.

Using the binary methods for computing the powers as shown in Table 16, Montgomery modular exponentiation algorithm replace the exponentiation operation by a series of square and multiplication operations modulo m.

Montgomery modular exponentiation algorithm computes $x^e mod m$. The definition of m' requires that $\gcd(m, R) = 1$. For integers u and v where $0 \leq u, v \leq m$, define $\text{Mont}(u, v, m)$ to be uvR^{-1} mod m as computed by Montgomery modular multiplication algorithm shown in Table 15.

Algorithm: Montgomery modular exponentiation

```
Input: $m = (m_{l-1} \cdots m_1 m_0)_b$, $R = b^l$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $m' = -m^{-1} \bmod b$,
with $e_t = 1$,and $1 \le x \le m$,
Output: $x^e \bmod m$.
1. $\bar{x} = \mathrm{Mont}(x, R^2 \bmod m, m)$, $A = R \bmod m$.
2. For i from t to 0 do the following:
   2.1. $A = \mathrm{Mont}(A, A, m)$.
   2.2. If $e_i = 1$ then $A = \mathrm{Mont}(A, \bar{x}, m)$.
3. $A = \mathrm{Mont}(A, 1, m)$.
4. Return ( $A$ ).
```

Table 16: Montgomery modular exponentiation.

Montgomery modular multiplication algorithm can't be directly applied to modular exponentiation due to the extra factor R, and two extra processes are needed to operate modular exponentiation. One is mapping to convert input plaintext x into $xR \bmod m$ shown in step 1 of Table 16, and the other is re-mapping to remove the extra factor R from the output of modular exponentiation shown in step 3. Finally, the last output result is in the desired form.

2.7 Cryptography in CUDA

The encryption activity is computationally intensive, and shows a significant feature of parallelism. On the other hand, cheap multicore processors are readily available on graphics hardware, and toolchains for development of general purpose programs are being released by the vendors.

With the emergence of CUDA architecture and tools, many fields are significant speedup such as creating breakthrough applications in areas such as image recognition, realtime HD video playback and encoding, and cryptography computation.

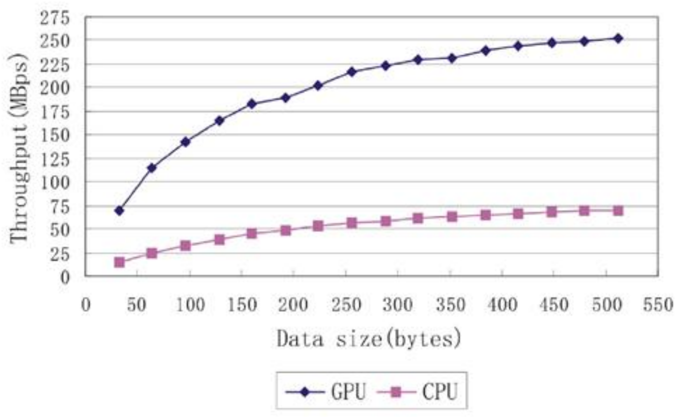
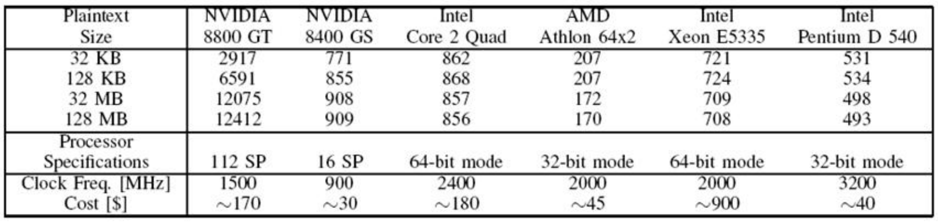


Figure 4: MD5-RC4 encryption performance comparison on different data sizes of each data object [1].

In 2009, [1] presents an efficient implementation for MD5-RC4 encryption using NVIDIA GPU with CUDA programming framework. The MD5-RC4 encryption algorithm was implemented on NVIDIA GeForce 9800GTX GPU. The performance of its solution is compared with the implementation running on an AMD Sempron Processor LE-1200 CPU. The results show that the GPU-based implementation exhibits a performance gain of about 3-5 times speedup for the MD5-RC4 encryption algorithm.

Figure 4 taken from [1] shows the encryption throughputs comparison on different data sizes of each data object. From this figure, it is obviously that the encryption throughputs increased with the increase of the data size in each data object. When the data size is 32 bytes, the throughput of GPU-based implementation is 70MBps, which is 5 times greater than the CPU-based one. Moreover, on each given input data size of the data object, the CUDA-based implementation gained a much greater throughput in comparison to the CPU-based implementation.



Plaintext Size	NVIDIA 8800 GT	NVIDIA 8400 GS	Intel Core 2 Quad	AMD Athlon 64x2	Intel Xeon E5335	Intel Pentium D 540
32 KB	2917	771	862	207	721	531
128 KB	6591	855	868	207	724	534
32 MB	12075	908	857	172	709	498
128 MB	12412	909	856	170	708	493
Processor Specifications	112 SP	16 SP	64-bit mode	32-bit mode	64-bit mode	32-bit mode
Clock Freq. [MHz]	1500	900	2400	2000	2000	3200
Cost [$]	∼170	∼30	∼180	∼45	∼900	∼40

Figure 5: AES performance comparison among the GPUs and four common CPUs: Throughput [Mbps] [2].

On the other hand, [2] investigated the possibility of using the GPU supported by CUDA as a co-processor to ease the CPU load when encrypting or decrypting data streams in web server applications. This research has shown how to effectively implement the AES block cipher using the CUDA and its programming model, extracting as much parallelism as possible from the algorithm with both coarse and fine grained approaches. It provided an extensive quantitative evaluation on a range of NVIDIA GPUs based on the G80 architecture and scaling from 16 to 112 cores. These experiments show the AES block cipher and similar algorithms are possible to efficiently use the GPU as a co-processor. In addition, this solution is cost effective when compared to the assembly level optimized CPU-based implementations of the AES built in the OpenSSL library. The Figure 5 taken from [2] shows the performance comparison among the GPUs and other four common CPUs. On the whole, [2] reports throughput improvements of up to 14 times over the CPU implementations chosen as baseline, as well as the comparison of the performance and cost that is about 73 Mbps per dollar for the NVIDIA 8800 GT against the 4 Mbps per dollar of the Intel Core 2 Duo.

All those researches do not focus on the public-key algorithm, but they make us believe that it is possible to implement public-key algorithm efficiently by using the GPU with CUDA technology.

3 CUDA

Because of the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit (GPU) has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 6 which is taken from [3].

CUDA is a general-purpose programming system for NVIDIA GPUs and was first publicly released in the end of 2007. By using CUDA, the CUDA-enabled GPU (so-called device) is exposed to the CPU (so-called host) as a co-processor. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer. To perform useful work, data must be transferred between the memory space of the host computer and CUDA device(s). For this reason, performance results must include input and output (IO) time to be informative.

At the heart of CUDA is the ability for programmers to keep thousands of threads busy. The current generation of NVIDIA GPUs can efficiently support a very large number of threads, and as a result they can deliver one to two orders of magnitude performance increase in application performance.

3.1 Kernel

A kernel [3] is a function callable from the host and executed on the CUDA device simultaneously by many threads in parallel. In fact CUDA executes a function in the Single Program Multiple Data (SPMD) model, which means that a user-configured number of threads run the same program on different data. Each thread will execute the same kernel function and will operate upon only a single data element. Each thread is distinguished from all the others by block and thread indices that can be used to determine the data element the thread will access. CUDA organizes a parallel computation using the abstractions of threads, blocks and grids, and the simple definitions [3] [32] as follows:

- Thread is just an execution of a kernel with a given index. Each thread uses its index to access data elements such that the collection of all threads cooperatively processes the entire data set.
- Block is a group of threads. Threads within a block can execute concurrently or serially and in no particular order. They can be coordinated using the synchronization function that makes a thread stop at a certain point in the kernel until all the other threads in its block reach the same point.
- Grid is a group of blocks. There's no synchronization at all between the blocks.

These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by Figure 7 [3]. On the other hand, the computation of threads, blocks and grids are distributed as follows:

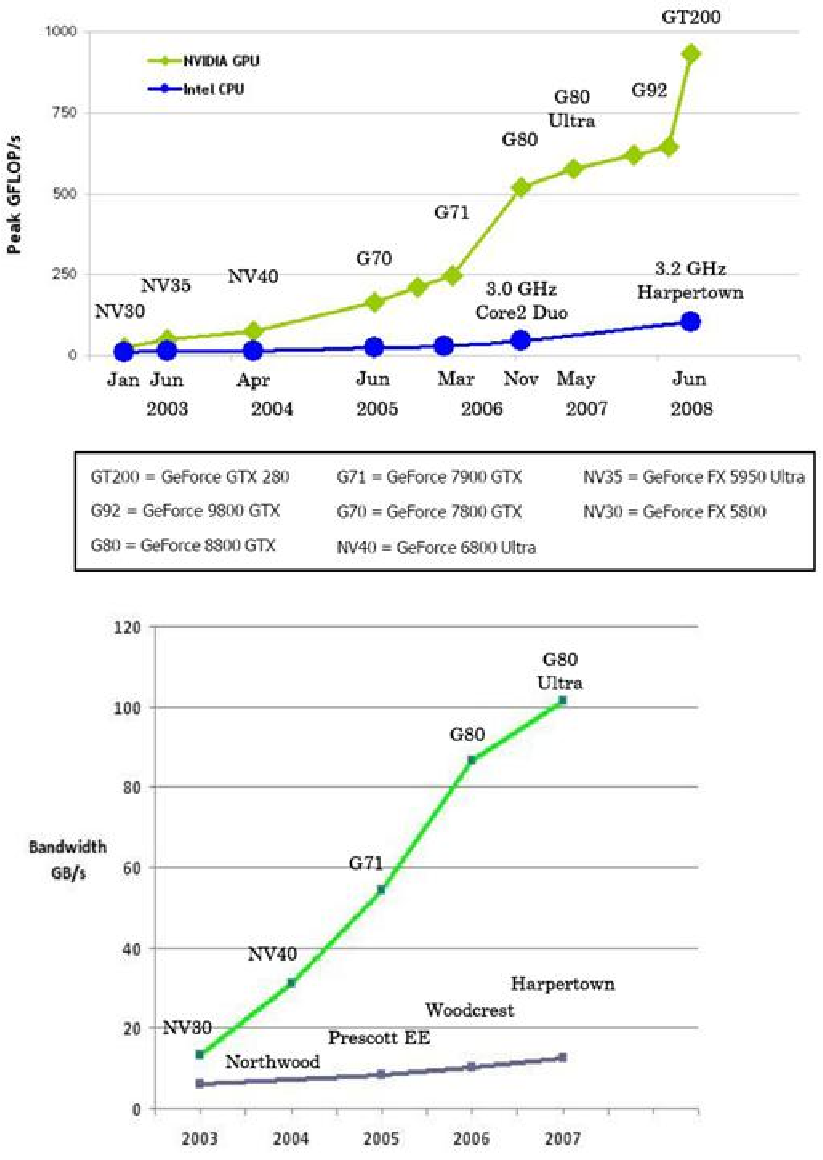


Figure 6: Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU [3].

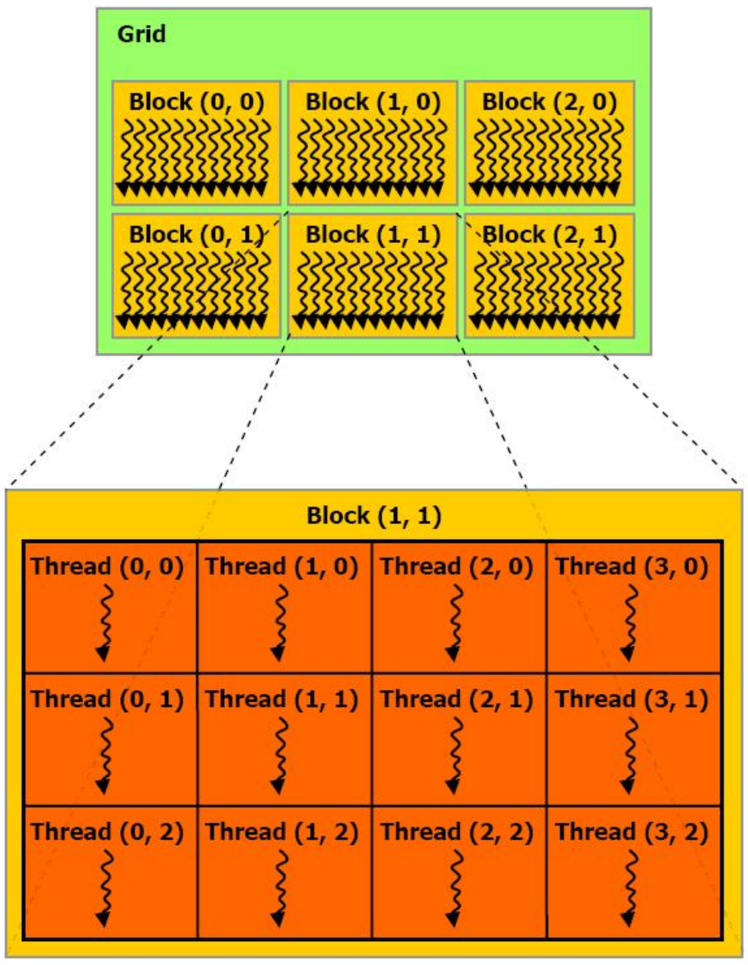


Figure 7: Grid of Thread Blocks [3].

- Grid $\rightarrow$ GPU: An entire grid is handled by a single GPU chip.
- Block $\rightarrow$ MP: The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.
- Thread $\rightarrow$ SP: Each MP is further divided into a number of stream processors (SPs), with each SP handling one or more threads in a block.

From the host's point of view, kernel invocations are asynchronous function calls. Synchronization is done explicitly by calling a synchronization function, or implicitly when the host tries to access memory on the device. In both cases, synchronization takes the form of a barrier that blocks the calling host thread until all previously called kernels have been finished.

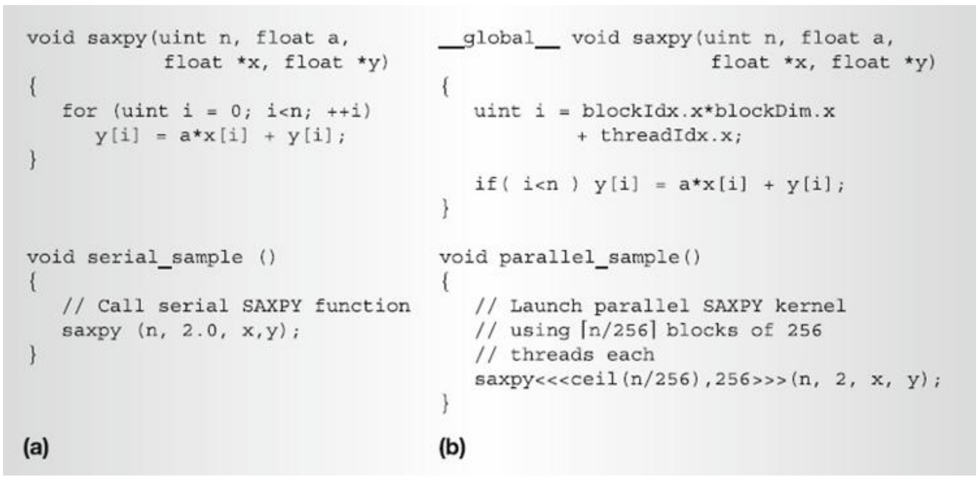
When the CUDA device is idle, the kernel immediately starts running based on the execution configuration and according to the function arguments. Meanwhile, the host continues to the next line of code after the kernel launch. At this point, both the CUDA device and host are simultaneously running their separate programs. If another kernel is called by the host immediately, it waits until all threads have finished on the device.

Each active block is split into SIMD (Single Instruction Multiple Data) groups of threads called warps. Each warp contains the same number of threads, called the warp size, which are executed by the multiprocessor in a SIMD fashion. This means each thread within a warp is broadcast the same instruction from the instruction store, which directs the thread to perform some operation and manipulation of local or global memory.

Active warps are time-sliced. The thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources. The order of execution of the warps within a block and of blocks themselves is undefined, which means they can occur in any order.

By using the GPU tens of thousands or even more threads can be processed at the same time. It uses this massive parallelism to hide the costs of memory accesses by efficient thread scheduling, i.e., threads are removed from a processor while waiting for a read from memory to complete. The memory at the device is called global memory and can be accessed by both the host and all processors of the device.

Figure 8 taken from [4] shows some basic features of parallel programming with CUDA for computing $y = a \cdot x + y$. It contains straightforward implementations, both sequential and parallel. Given vectors x and y containing n floating-point numbers, it performs the update $y = a \cdot x + y$. The serial implementation is a simple loop that computes one element of y in each iteration. The parallel kernel effectively executes each of these independent iterations in parallel, assigning a separate thread to compute each element of y. The $\_\_global\_\_$ modifier indicates that the procedure is a kernel entry point, and the extended function call syntax saxpy $<<< B, T >>>$ (...) is used to launch the kernel $saxpy()$ in parallel across B blocks of T threads each. Each thread of the kernel determines which element it should process from its integer thread block index ($blockIdx.x$), its index within its block ($threadIdx.x$), and the total number of threads per block ($blockDim.x$). This example demonstrates a common parallelization pattern, where a serial loop with independent iterations can be executed in parallel across many threads.



```
void saxpy(uint n, float a,
           float *x, float *y)
{
   for (uint i = 0; i<n; ++i)
      y[i] = a*x[i] + y[i];
}


void serial_sample ()
{
   // Call serial SAXPY function
   saxpy (n, 2.0, x,y);
}
```

(a)

```
__global__ void saxpy(uint n, float a,
                      float *x, float *y)
{
   uint i = blockIdx.x*blockDim.x
          + threadIdx.x;

   if( i<n ) y[i] = a*x[i] + y[i];
}

void parallel_sample()
{
   // Launch parallel SAXPY kernel
   // using ⌈n/256⌉ blocks of 256
   // threads each
   saxpy<<<ceil(n/256),256>>>(n, 2, x, y);
}
```

(b)

Figure 8: Parallel programming with CUDA: serial (a) and parallel (b) kernels for computing $y = a \cdot x + y$ [4].

3.2 Memory hierarchy

In combination with the hierarchy of processing units, the CUDA-enabled GPU provides a memory hierarchy [3] [32]:

- Global memory: This memory is built from a bank of SDRAM chips connected to the GPU chip. Any thread in any MP can read or write to any location in the global memory. Sometimes this is called device memory. Potentially 150x slower than register or shared memory.
- Texture cache: This is a memory within each MP that can be filled with data from the global memory so it acts like a cache. Threads running in the MP are restricted to read-only access of this memory.
- Constant cache: This is a read-only memory within each MP.
- Shared memory: This is a small memory within each MP that can be read/written by any thread in a block assigned to that MP. Can be as fast as a register when there are no bank conflicts or when reading from the same address.
- Registers: Each MP has a number of registers that are shared between its SPs. The fastest form of memory on the multi-processor.
- Local memory: It implies "local in the scope of each thread". It is a memory abstraction, not an actual hardware component of the multi-processor. In actuality, local memory gets allocated in global memory by the compiler and delivers the same performance as any other global memory region. Local memory is basically used by the compiler to keep anything the programmer considers local to the thread but does not fit in faster memory for some reason.

Figure 9 schematically illustrates a thread that executes on the device has access to global memory and the on-chip memory through the memory types [3].

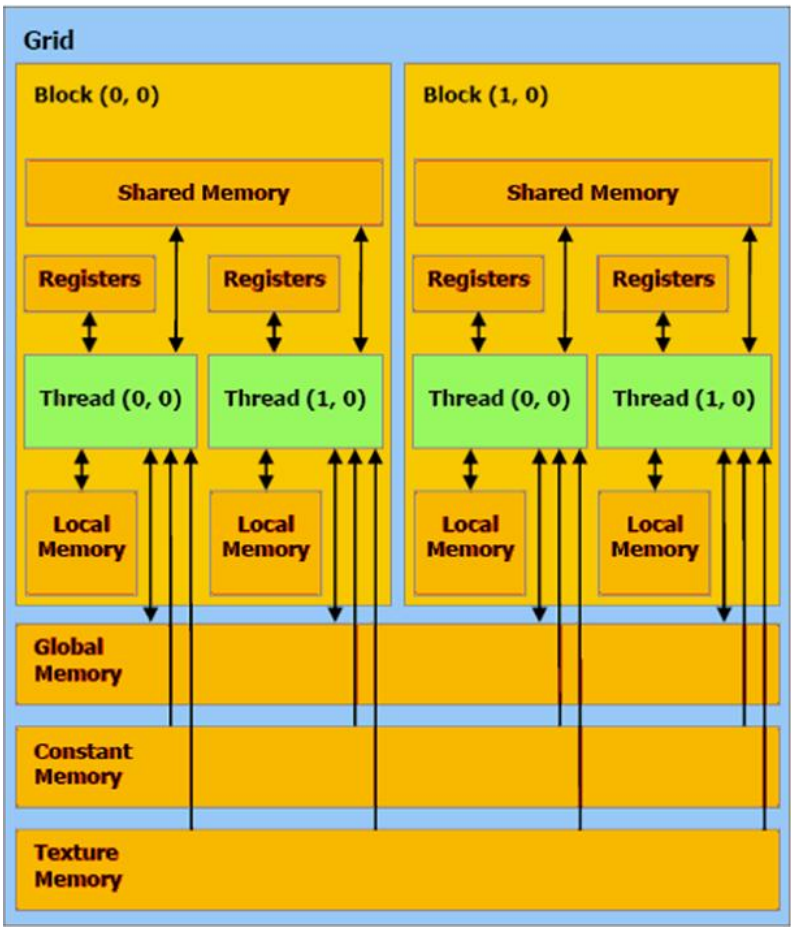


Figure 9: Memory Hierarchy [3].

Obviously, higher performance applications must reuse data in some fashion, which is the function of shared and register memory. It is important to note that threads within a block can communicate with each other through local multi-processor resources because the CUDA execution model specifies that a block can only be processed on a single multiprocessor. In other words, data written to shared memory within a block is accessible to all other threads within that block, but it is not accessible to a thread from any other block. Shared memory with these characteristics can be implemented very efficiently in hardware which translates to fast memory accesses for CUDA developers.

3.3 Program in CUDA

With the CUDA architecture and tools, developers are achieving dramatic speedups in fields such as medical imaging and natural resource exploration, and cryptography. One of the major benefits of CUDA as compared to other GPU programming systems is its use of a C dialect, such that an original C function for the CPU can often be transformed into a CUDA kernel with only slight modifications. CUDA provides to developers C libraries that expose all device functionalities needed to integrate CUDA into a C program. Furthermore CUDA enables this unprecedented performance via standard APIs such as OpenCL and DirectX Compute, and high level programming languages such as C/C++, Fortran, Java, Python, and the Microsoft .NET Framework.

The programmer, in order to write a CUDA program, normally begins from a sequential version and proceeds through the following steps [33]:

1. Identify a kernel, and package it as a separate function.
2. Specify the grid of GPU threads that executes it, and partition the kernel computation among these threads, by using blockIdx and threadIdx inside the kernel function.
3. Manage data transfer between the host memory and the GPU memories (global, constant and texture), before and after the kernel invocation. This includes redirecting variable accesses in the kernel to the corresponding copies allocated in the GPU memories.
4. Perform memory optimizations in the kernel, such as utilizing the shared memory and coalescing accesses to the global memory .
5. Perform other optimizations in the kernel in order to achieve an optimal balance between single-thread performance and the level of parallelism.

In addition a CUDA program may include multiple kernels, thus the above procedure needs to be applied to each of them.

In order to give the experience to developers who don't have a CUDA-enabled GPU board on the PC but still want to try running CUDA program, the emuDebug configuration is available. This configuration uses a software emulation of a CUDA device instead of the actual hardware found on the graphics card. This will link-in a CUDA device emulator that runs on the host. The emulator becomes the target for all the CUDA API calls and executes the kernel. The program will run just like a CUDA device is there, except slower.

3.4 Optimization in CUDA

In order to achieve an excellent performance of parallel computation, optimizations have to be considered and performed in kernel.

The warp size is the number of threads running concurrently on an MP. The homogeneity of the threads in a warp has a big effect on the computational throughput. If all the threads are executing the same instruction, then all the SPs in an MP can execute the same instruction in parallel. But if one or more threads in a warp is executing a different instruction from the others, then the warp has to be partitioned into groups of threads based on the instructions being executed, after which the groups are executed one after the other. This serialization reduces the throughput as the threads become more and more divergent and split into smaller and smaller groups. So it pays to keep the threads as homogenous as possible.

Optimizing the performance of CUDA applications most often involves optimizing data accesses which includes the appropriate use of the various CUDA memory spaces. Appropriate use of these memory spaces can have significant performance implications for CUDA applications.

On the other hand, how the threads access global memory also affects the throughput. Computations run much faster if the GPU can coalesce several global addresses into a single burst access over the wide data bus that goes to the external SDRAM. Conversely, reading/writing separated memory addresses requires multiple accesses to the SDRAM which slows the performance down. To help the GPU combine multiple accesses, the addresses generated by the threads in a warp must be sequential with respect to the thread indices.

4 Experimental methods

This project focuses on how to make a more efficient and faster implementation of public-key algorithms. Two experiments implementing a public key algorithm are performed on different hardware platforms. One is to implement the selected algorithms normally on a CPU with different data sizes, and then record the execution time and other related data. Another is to execute designed parallel algorithms on a CUDA-enabled GPU, and record related data as well. Finally the performance comparison is performed between those experiments.

The parallelization of public key algorithms is mainly performed in the part of modular multiplication and modular exponentiation. Therefore, this project implements a representative public-key algorithm RSA respectively on the CPU and the CUDA-enabled GPU, and compares their performances to find out whether the public-key algorithm could be implemented faster and more efficient on a GPU. Theoretically, the performance that RSA implemented on a GPU should be better than that on the CPU since parallelization of RSA is performed on the CUDA-enabled GPU with massive parallel processors. In addition, there are still other related issue concerned in this project, such as time consumption in data transfer between host and device.

The CUDA driver API and C runtime for CUDA are two of the programming interfaces to CUDA [32]. The C runtime for CUDA handles kernel loading and kernels' setting before they are launched. The implicit code initialization, CUDA context management, CUDA module management (cubin and function mapping), kernel configuration, and parameter passing are all performed by the C runtime for CUDA. In addition, CUDA supports C++ code and can be compiled with any C++ compiler. However, the current version of CUDA does not support all features of C++. Therefore, all functions in this project are mostly performed in C.

4.1 Experimental equipments

The machine used for this project needed to have a CUDA enabled graphics card from NVIDIA. Table 17 shows some basic specifications of the computer which is adopted in the project. Table 18 shows some basic specifications for the NVIDIA GeForce GT 130M graphic card.

Processor:	Intel Core2 Duo P8700 processor 2.53 GHz
RAM:	4GB 1100Mhz
Hard Drive:	Western Digital WD3200BEVT 320GB
Graphics card:	NVIDIA GeForce GT 130M
Operating System:	Windows Vista Home Premium

Table 17: Basic specifications of test computer.

Table 19 shows the property of CUDA-enabled GPU used in this project. It shows the limits on the sizes of blocks and grids. A block is one-, two- or three-dimensional with the maximum sizes of the x, y and z dimensions being 512, 512 and 64, respectively, and

GPU Engine and Memory Specs	**Description**
Processor Cores:	32
Gigaflops:	144
Processor Clock (MHz):	1500
Memory Clock (MHz):	800 (GDDR3)
Standard Memory Config:	512 MB
Memory Interface Width:	128-bit
Memory Bandwidth (GB/sec):	25 (GDDR3)

Table 18: Basic specifications of NVIDIA GeForce GT 130M.

CUDA property	**Description**
Number of multiprocessors	4
Number of cores	32
Total amount of global memory	512M bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	16384 bytes
Total amount of registers available per block	8192
Warp size	32
Maximum number of threads per block	512
Maximum sizes of each dimension of a block	$512 \times 512 \times 64$
Maximum sizes of each dimension of a grid	$65535 \times 65535 \times 1$
Clock rate	1.50 GHz
Concurrent copy and execution	Yes

Table 19: CUDA property of NVIDIA GeForce GT 130M.

such that $x \times y \times z \leq 512$, which is the maximum number of threads per block. Blocks are organized into one- or two-dimensional grids of up to 65,535 blocks in each dimension. The primary limitation here is the maximum of 512 threads per block, primarily imposed by the small number of registers that can be allocated across all the threads running in all the blocks assigned to an MP. The thread limit constrains the amount of cooperation between threads because only threads within the same block can synchronize with each other and exchange data through the fast shared memory in an MP. The warp size is the number of threads running concurrently on an MP.

4.2 RSA

The RSA public-key cryptosystem was developed by R.L. Rivest, A. Shamir, and L. Adleman in 1978 [7]. The RSA algorithm is simply the modular exponentiation. The modulus n is the product of two large primes: $n = pq$. Euler's totient function of n is given by

$$\phi(n) = (p-1)(q-1).$$

Now, select a number $1 < e < \phi(n)$ such that

$$gcd(e, \phi(n)) = 1,$$

and compute d with

$$d = e^{-1} mod \phi(n).$$

Where e is the public exponent and d is the private exponent. A small public exponent is usually selected as e. The modulus n and the public exponent e are published. The value of d and prime numbers p and q are kept secret.

The encryption operation is performed using the public key e, as follows:

$$C \equiv M^e (mod n)$$

Where M is the plaintext such that $0 \leq M < N$, and C is the ciphertext which can be decrypted using the secret key d, as follows:

$$M \equiv C^d (mod n)$$

The correctness of RSA algorithm follows from Euler's theorem: Let n and a be positive, relatively prime integers. Then

$$a^{\phi(n)} = 1 \text{ (mod } n).$$

Since we have $ed = 1$ mod $\phi(n)$, i.e., $ed = 1 + K\phi(n)$ for some integer K, we can write

$$\begin{aligned} C^d &= (M^e)^d \text{ (mod } n) \\ &= M^{ed} \text{ (mod } n \text{)} \\ &= M^{1+K\phi(n)} \text{ (mod } n \text{)} \\ &= M \cdot (M^{\phi(n)})^K \text{ (mod } n \text{)} \\ &= M \cdot 1 \text{ (mod } n) \end{aligned}$$

Provided that $gcd(M, n) = 1$. The exception $gcd(M, n) > 1$ can be dealt as follows. According to Carmichael's theorem

$$M^{\lambda(n)} = 1 \text{ (mod } n)$$

Where $\lambda(n)$ is Carmichael's function which takes a simple form for $n = pq$, namely,

$$\lambda(pq) = \frac{(p-1)(q-1)}{gcd(p-1,q-1)}$$

Note that $\lambda(n)$ is always a proper divisor of $\phi(n)$ when n is the product of distinct odd primes; in this case $\lambda(n)$ is smaller than $\phi(n)$. Now, the relationship between e and d is given by

$$M^{ed} = M \text{ (mod } n) \text{ if } ed = 1 \text{ (mod } \lambda(n) \text{)}$$

Provided that n is a product of distinct primes, the above holds for all M, thus dealing with the above-mentioned exception $gcd(M, n) > 1$ in Euler's theorem.

In this project, modular multiplication and modular exponentiation with large numbers are the key points of RSA. Barrett modular reduction [28] [29] [30] and Montgomery modular multiplication [31] are mainly considered. In these algorithms, the operation of addition, subtraction, right shift, multiplication and exponentiation can be implemented in parallel and described in the following sections.

4.3 Methods

4.3.1 Storage structure for large number

When the computation for large numbers is regarded, how to construct storage structure for the large number must be considered first. There are several methods mentioned in Chapter 2.

A modified large number storage method based on the classic method [11] is used in this project because each digit of a large number must be accessed directly in order to perform the parallel computation on CUDA. The storage structure of large numbers is illustrated by Table 20.

```
struct SLargeNumber
{
    int digits[DIGITS_MAX_LEN];
    int sign;
    int header;
    int length;
    int base;
}
```

Table 20: The storage structure of large numbers.

SLargeNumber type of objects are used to store large numbers. The structure SLargeNumber has five data members: "digits" to store the digits making a large number, "sign" to hold the sign of the large number, "header" to hold the address of the first digit, "length" to hold the number of digits used, and "base" hold the radix of the large number.

For example, a large number -979238938 base 10 can be represented in the SLargeNumber structure by the Table 21.

i	0	1	2	3	4	5	6	7	8
digits[i]	8	3	9	8	3	2	9	7	9

sign	-1	header	0	length	9	base	10

Table 21: SLargeNumber structure representation for the large number -979238938.

In this storage structure, each digit of a large number can be accessed directly by each thread when the parallel computation is performed. Based on this storage structure, the corresponding function can be constructed to perform operations for large numbers. In the following sections, three elements "digits", "length", and "base" are used to describe the structure of large numbers. Other elements are not so important to illustrate experimental methods.

4.3.2 Parallel addition and subtraction

Parallel addition and subtraction are performed as shown in Figure 10. When computing $Z = X \pm Y$, where $X = (X_n\ X_{n-1}\ X_{n-2} \cdots X_4\ X_3\ X_2\ X_1\ X_0\ )_b$, $Y = (Y_n\ Y_{n-1}\ Y_{n-2} \cdots Y_4\ Y_3\ Y_2\ Y_1\ Y_0\ )_b$, $Z = (Z_n\ Z_{n-1}\ Z_{n-2} \cdots Z_4\ Z_3\ Z_2\ Z_1\ Z_0\ )_b$, and b is the base of large numbers, each thread T_i $(0 \le i \le n)$ is in charge of computing a pair of digits X_i and Y_i. For all threads T_i do the following in parallel:

$$Z_i = X_i \pm Y_i \ (0 \le i \le n)$$

Z_i $(0 \le i \le n)$ are temporary variables which are large enough to store the results computed by threads. When all Z_i are available, a update function is called to compute the carry of Z. However, this update function is not implemented in parallel but in sequential, because the carry of Z_i will influence the value and carry of Z_{i+1}. The source code of parallel subtraction implemented in CUDA can be found in the Appendix A.

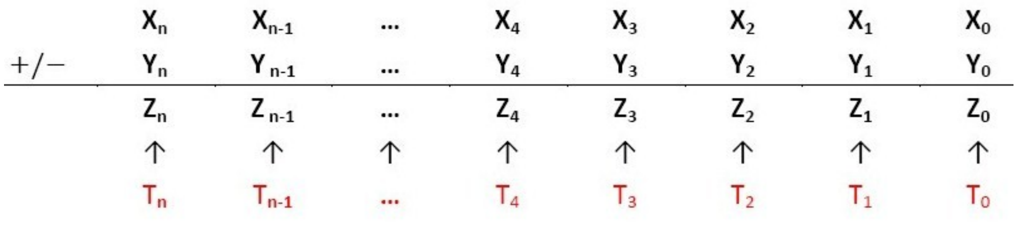

X_n X_{n-1} ... X_4 X_3 X_2 X_1 X_0
+/− Y_n Y_{n-1} ... Y_4 Y_3 Y_2 Y_1 Y_0
Z_n Z_{n-1} ... Z_4 Z_3 Z_2 Z_1 Z_0
T_n T_{n-1} ... T_4 T_3 T_2 T_1 T_0

Figure 10: Parallel addition and subtraction.

4.3.3 Parallel right shifts

When computing $Z = X/b$ where b is the base of large numbers, right shift in parallel can be performed to compute it as shown in Figure 11. Each thread copies X_i $(0 < i \le n)$ to Z_{i-1}. The source code of parallel right shifts implemented in CUDA can be found in the Appendix B.

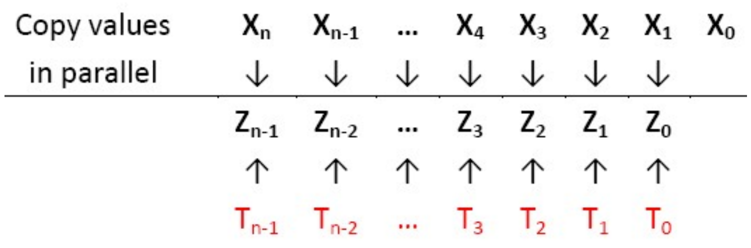

Copy values in parallel
X_n X_{n-1} ... X_4 X_3 X_2 X_1 X_0
Z_{n-1} Z_{n-2} ... Z_3 Z_2 Z_1 Z_0
T_{n-1} T_{n-2} ... T_3 T_2 T_1 T_0

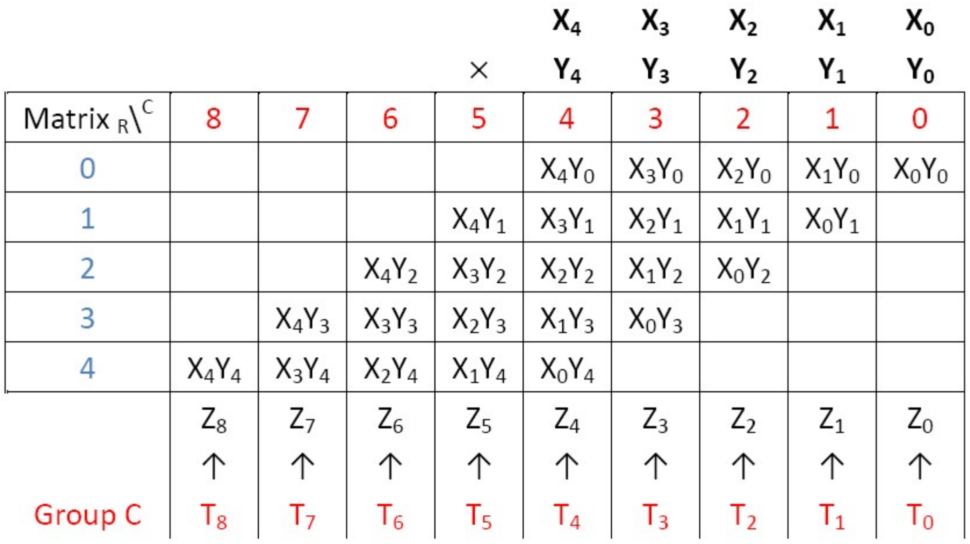
Figure 11: Parallel right shift.

4.3.4 Parallel multiplication

When computing $Z = XY$, all available threads are divided into two groups: Group R and Group C. The Figure 12 shows an example that all intermediate values are stored in a matrix. An intermediate value in a cell of the matrix is generated by a single thread in Group R. In the case of Figure 12, 25 calling threads calculate intermediate results X_iY_j $(0 \le i, j \le 4)$ in parallel. The source code of parallel multiplication implemented in CUDA can be found in the Appendix C

Then the values in each column of the matrix are added together to generate Z_i. In this process, the addition performed for each column is handled in parallel by threads in Group C. When just one column is considered, the addition can be computed as shown in Figure 13.

On the other hand, a lookup table of X as shown in Figure 14 can be used to precompute all possible intermediate values in the matrix. It is obvious that X and Y are



					X_4	X_3	X_2	X_1	X_0
				×	Y_4	Y_3	Y_2	Y_1	Y_0
Matrix $_R\backslash^C$	8	7	6	5	4	3	2	1	0
0					X_4Y_0	X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0
1				X_4Y_1	X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1	
2			X_4Y_2	X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2		
3		X_4Y_3	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3			
4	X_4Y_4	X_3Y_4	X_2Y_4	X_1Y_4	X_0Y_4				
	Z_8	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0
	↑	↑	↑	↑	↑	↑	↑	↑	↑
Group C	T_8	T_7	T_6	T_5	T_4	T_3	T_2	T_1	T_0

Figure 12: Multiplication and intermediate values stored in a matrix.

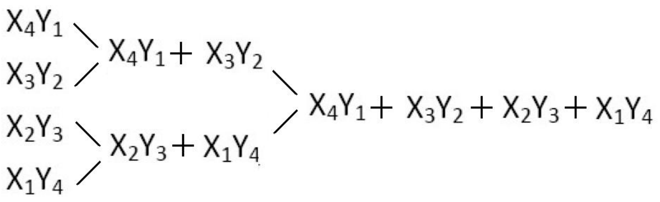
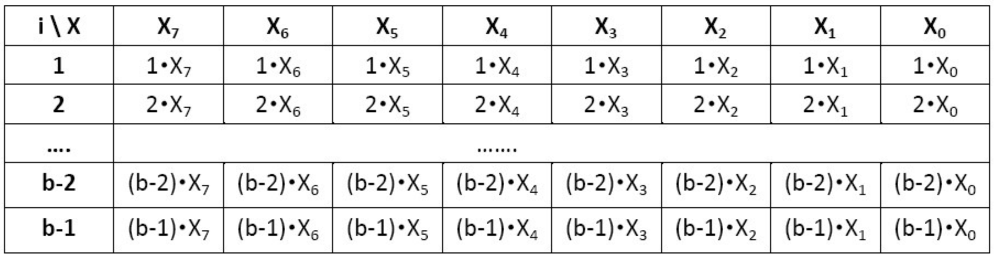


Figure 13: Add sub results in a column in parallel.

large numbers, and many intermediate values are computed repeatedly.



i \ X	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0
1	$1 \cdot X_7$	$1 \cdot X_6$	$1 \cdot X_5$	$1 \cdot X_4$	$1 \cdot X_3$	$1 \cdot X_2$	$1 \cdot X_1$	$1 \cdot X_0$
2	$2 \cdot X_7$	$2 \cdot X_6$	$2 \cdot X_5$	$2 \cdot X_4$	$2 \cdot X_3$	$2 \cdot X_2$	$2 \cdot X_1$	$2 \cdot X_0$
….	…….							
b-2	$(b-2) \cdot X_7$	$(b-2) \cdot X_6$	$(b-2) \cdot X_5$	$(b-2) \cdot X_4$	$(b-2) \cdot X_3$	$(b-2) \cdot X_2$	$(b-2) \cdot X_1$	$(b-2) \cdot X_0$
b-1	$(b-1) \cdot X_7$	$(b-1) \cdot X_6$	$(b-1) \cdot X_5$	$(b-1) \cdot X_4$	$(b-1) \cdot X_3$	$(b-1) \cdot X_2$	$(b-1) \cdot X_1$	$(b-1) \cdot X_0$

Figure 14: A lookup table of X.

In Figure 14, b is the base of large numbers. All values in the lookup table can be generated in parallel. By using the lookup table of X, the intermediate values of matrix can be located based on the value of Y_i. So it is unnecessary to compute all intermediate values when implementing parallel multiplication.

Because threads of different blocks cannot be synchronized, the Parallel Multiplication is divided into three kernels:

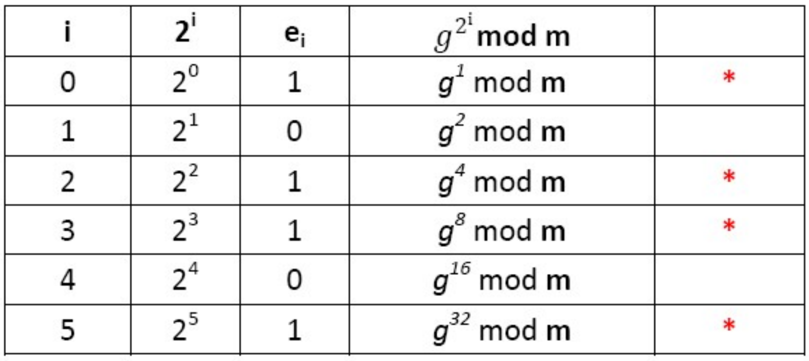
- To construct a lookup table of X;
- To calculate intermediate results, and get Z without updating the carry;
- To compute and update the carry and length of Z.

4.3.5 Parallel modular exponentiation

The parallel modular exponentiation method is based on the observation of binary modular exponentiation method. The exponent e is converted into binary representation. The computation of modular exponentiation is the product as the formula:

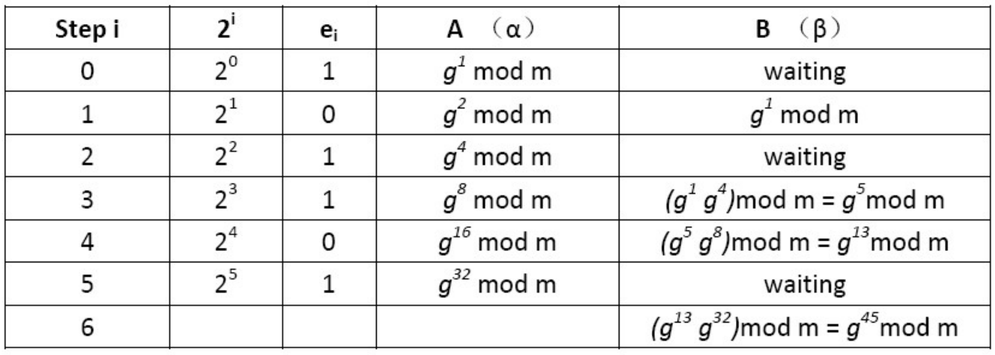
$$g^{(e_t e_{t-1} \cdots e_1 e_0)_2} \bmod m = \prod g^{2^i} \bmod m, \text{ for each } i \text{ where } e_i = 1.$$

For example, using modular exponentiation to compute $g^{45} \bmod m$, the exponent 45 is first converted into binary form $e = (45)_{10} = (101101)_2$. The exponent e is represented by 6 bits. Then, pre-computing the powers $g^{2^i} \bmod m$, $i = 0, \ldots, 5$ as shown in Figure 15.



i	2^i	e_i	g^{2^i} mod m	
0	2^0	1	g^1 mod m	*
1	2^1	0	g^2 mod m	
2	2^2	1	g^4 mod m	*
3	2^3	1	g^8 mod m	*
4	2^4	0	g^{16} mod m	
5	2^5	1	g^{32} mod m	*

Figure 15: Pre-computing the powers $g^{2^i} \bmod m$, $i = 0, \ldots, 5$.



Step i	2^i	e_i	A （α）	B （β）
0	2^0	1	g^1 mod m	waiting
1	2^1	0	g^2 mod m	g^1 mod m
2	2^2	1	g^4 mod m	waiting
3	2^3	1	g^8 mod m	$(g^1\ g^4)$mod m = g^5mod m
4	2^4	0	g^{16} mod m	$(g^5\ g^8)$mod m = g^{13}mod m
5	2^5	1	g^{32} mod m	waiting
6				$(g^{13}\ g^{32})$mod m = g^{45}mod m

Figure 16: The procedure of computing g^{45} mod m by means of the parallel modular exponentiation.

The pre-computing results labeled with $*$, whose corresponding coefficients in the binary representation of the exponent 45 are non-zero, are used for computing the final result as follow:

$$\begin{aligned} g^{45} \bmod m &= (g^1 \bmod m)(\ g^4 \bmod m)(\ g^8 \bmod m)(\ g^{32} \bmod m) \\ &= g^{(1+4+8+32)} \bmod m = g^{45} \bmod m \end{aligned}$$

In this example, 5 times of modular square and 3 modular multiplications are required if the initialization of data (when $i = 0$) is not considered. This method can be parallelized and used in the parallel computation. Two Groups of threads α and β are employed to compute the modular exponentiation in parallel. The Group α is to calculate g^{2^i} mod m for each i by using repeated square and modular reduction, another Group β is to multiple each intermediate result used for computing the final result as soon as the group α generates a new intermediate result.

The Figure 16 shows the procedure of computing g^{45} mod m by means of the parallel modular exponentiation. Still, the first initialization of data is not considered because it multiplies g by 1 without modular reduction. Group α performs 5 modular squares, and Group β executes 3 modular multiplications. However, Groups α and β implement their tasks in parallel. Thus the processing time for the whole parallel program is more or less equal to the implementing time of Group α with the addition of once modular multiplication, if the number of processors is enough for the parallel modular exponentiation.

5 Results

This chapter contains the results for the performance of modular multiplication, modular exponentiation, and RSA. The programs have been tested with different sizes of input large numbers, and execution times of GPU implementation and CPU implementation have been compared against each other. Execution times of modular multiplication recorded are in milliseconds (ms), other times are in seconds (s).

5.1 Modular multiplication

Barrett modular reduction and Montgomery modular multiplication are mainly considered in this project. However, Montgomery modular multiplication implemented in CUDA is slower than modular multiplication by using Barrett modular reduction. Because, in each iteration of Montgomery modular multiplication, the synchronization of threads must to be employed after the step 2.2 of Table 15. If the digit-length of m is n, the synchronization will be called at least n times. That makes the implementation of Montgomery modular multiplication show. Appendix E shows the execution time of modular multiplication and exponentiation by using Montgomery's algorithm.

Bit-length	Time_{GPU} (ms)	Time_{CPU} (ms)	R_1 ($\text{Time}_{\text{CPU}}/\text{Time}_{\text{GPU}}$)
1024	3.104	2.280	0.734
2048	4.812	8.953	1.861
3072	6.466	19.478	3.012
4096	9.550	34.081	3.569
8192	21.653	132.981	6.141

Table 22: Execution time of modular multiplication between CPU implementation and GPU implementation.

Thus Barrett modular reduction is adopted for GPU implementation, and Montgomery modular multiplication is used for CPU implementation (See the Appendix D). Table 22 and Figure 17 show the execution time and performance comparison of modular multiplication between GPU implementation and CPU implementation.

In Table 22, the total time of GPU implementation is the run time of modular multiplication kernel with the addition of memory copy time. The bit-length has no significant influence on memory copy time. The reason would be the memory used for large numbers are not too large compared to the capability of memory copy between host and CUDA's device.

The ratio R_1 in the Table 22 shows the ratio between run time of CPU implementation and run time of GPU implementation. It has increased with the growth of bit-length.

At 1024 bits, the GPU program is slightly slower than the CPU program. From 2048 bits to 8192 bits, the GPU run time is significantly lower than the CPU run time. The GPU implementation is 6.141 times faster than the CPU implementation when large numbers are 8192 bit-length.

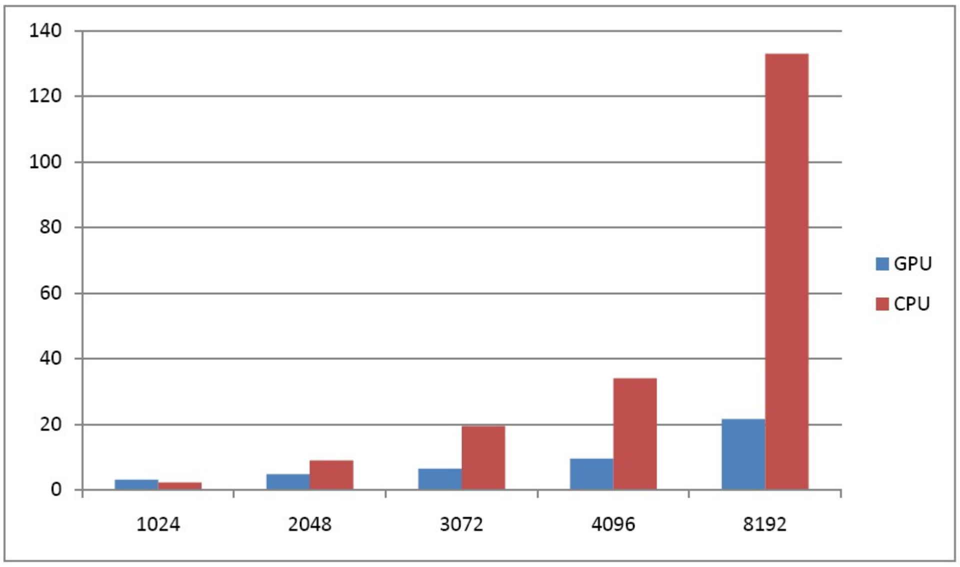


Figure 17: Performance comparison of modular multiplication between CPU implementation and GPU implementation.

5.2 Modular exponentiation

Table 23 and Table 24 show the execution time of modular exponentiation implemented on the GPU and the CPU respectively. The number of bit '1' contained in the exponent influences the execution time of modular exponentiation. In order to have a fair comparison, times shown in Table 23 and Table 24 are execution times with 100 groups input data which are generated randomly.

Bit-length	Time_{min} (s)	Time_{max} (s)	Average Time (s)
1024	3.338	5.228	4.102
2048	12.710	16.231	14.004
3072	26.515	31.473	28.538
4096	48.207	55.633	50.987
8192	237.288	275.802	252.358

Table 23: Execution time of modular exponentiation on GPU.

Bit-length	Time_{min} (s)	Time_{max} (s)	Average Time (s)
1024	3.459	4.431	3.624
2048	26.412	31.392	27.750
3072	88.652	101.004	94.429
4096	211.038	256.377	224.456
8192	1,622.980	1,914.893	1,722.923

Table 24: Execution time of modular exponentiation on CPU.

The Table 25 shows the ratios of execution time between CPU implementation and GPU implementation for minimal, maximal, and average time. Figure 18 shows average times comparison of modular exponentiation between GPU implementation and CPU implementation.

	Time$_{min}$	Time$_{max}$	Average Time
Bit-length	CPU/GPU	CPU/GPU	CPU/GPU (R_2)
1024	1.04	0.85	0.883
2048	2.08	1.93	1.982
3072	3.34	3.21	3.389
4096	4.38	4.61	4.402
8192	6.84	6.94	6.827

Table 25: The ratios of execution time of modular exponentiation between CPU implementation and GPU implementation.

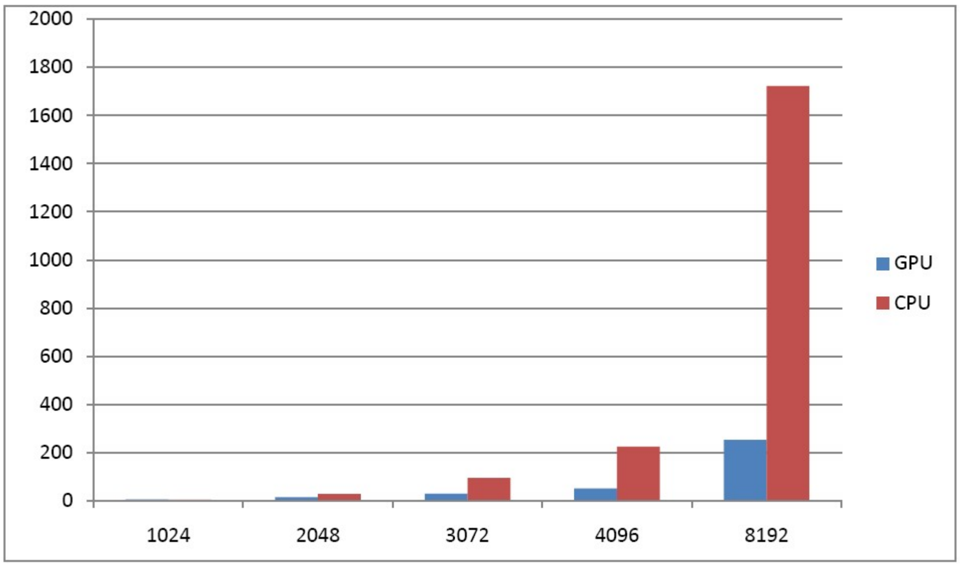


Figure 18: Performance comparison of modular exponentiation between CPU implementation and GPU implementation.

5.3 RSA

Table 26 and Table 27 describe the minimal, maximal, and average execution time of RSA implemented respectively on the GPU and the CPU with 200 groups RSA's data. In Table 26, execution time of GPU implementation includes memory copy time which is around 1.2 ms for all cases.

Bit-length	$Time_{min}$ (s)	$Time_{max}$ (s)	Average Time (s)
1024	3.458	5.931	4.338
2048	12.655	17.286	15.423
3072	26.713	31.758	29.408
4096	47.236	62.730	51.967
8192	233.466	290.189	250.962

Table 26: Execution time of RSA on GPU.

Bit-length	$Time_{min}$ (s)	$Time_{max}$ (s)	Average Time (s)
1024	3.285	5.094	3.592
2048	25.451	38.629	27.655
3072	85.409	106.617	90.199
4096	205.003	275.373	215.727
8192	1,648.426	1,947.808	1,735.480

Table 27: Execution time of RSA on CPU.

For all cases of bit-length, the average execution times shown in Table 26 and Table 27 are close to the execution times shown in Table 23 and Table 24. The Table 28 shows the ratios of execution time between CPU implementation and GPU implementation for minimal, maximal, and average time. Figure 19 shows average times comparison of RSA between GPU implementation and CPU implementation.

	$Time_{min}$	$Time_{max}$	Average Time
Bit-length	CPU/GPU	CPU/GPU	CPU/GPU (R_3)
1024	0.95	0.867	0.81
2048	2.01	2.23	1.79
3072	3.10	3.36	3.07
4096	4.34	4.39	4.15
8192	7.06	6.71	6.92

Table 28: The ratios of RSA's execution time between CPU implementation and GPU implementation.

5.4 Result summary

The experimental results show public key algorithms can be implemented fast on a CUDA-enabled GPU by using the massive parallel processing properties.

Comparing R_1 shown in Table 22 with R_2 shown in Table 25 and R_3 shown in Table 28, it is obvious that the parallel modular exponentiation has not significantly improved the speed of RSA implemented on the GPU as shown in Figure 20. The reason would be that the amount of available processors is far less than the number of processors needed for the parallel modular exponentiation. In the GPU implementation of modular

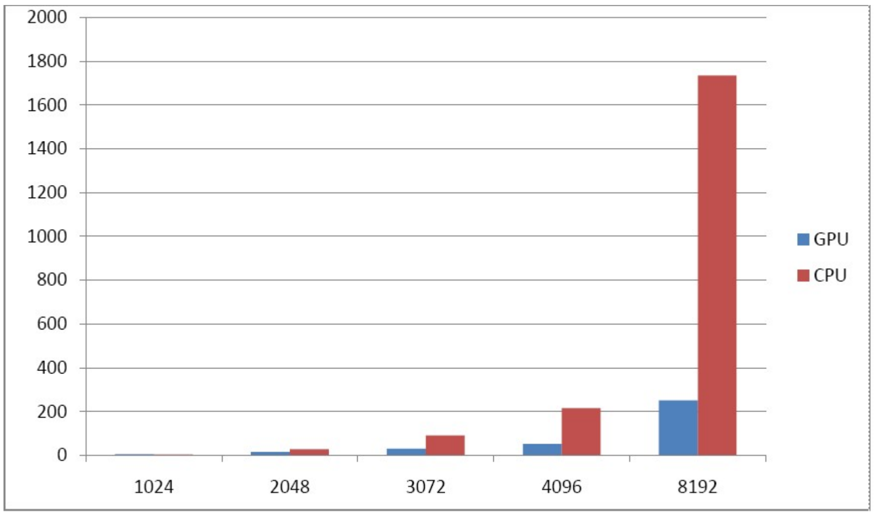


Figure 19: Performance comparison of RSA between CPU implementation and GPU implementation.

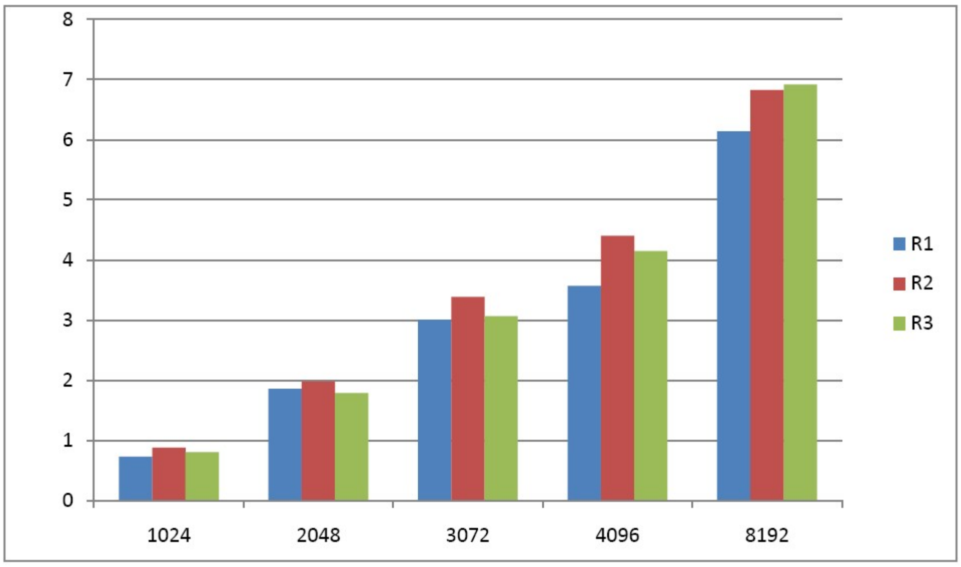


Figure 20: Ratios comparison.

exponentiation, the performances are only improved in the kernels that do not use many processors for computation.

In the CPU implementation, if the number of bits for RSA is doubled, the n-by-n multiplies will take four times as long. Further the exponent is doubled. So the execution time of RSA will take 8 times as long. On the other hand, in the GPU implementation, the execution time will take 4-5 times as long if the bit-length is doubled. For this reason, the ratios $R_i (1 \leq i \leq 3)$ have increased with the growth of bit-length.

6 Conclusion and future work

6.1 Discussion

The GPU used in this experiment has 32 processing cores working at 1.5 GHz, and the CPU working at 2.53GHz. In theory, the GPU implementation can achieve up to $32\times1.5\div2.53\approx18.97$ times faster than the CPU implementation. However, the experimental results show the GPU implementation of RSA has reached up to 6.9 times faster than the CPU version. Several factors influence the performance of CUDA's program.

First of all, some computational methods are still implemented in sequential. For example, the carry update of large numbers after parallel methods including parallel addition, parallel subtraction, and parallel multiplication.

The most important is that the access speeds of various memories are different. The global memory of GPU is very slow. In order to achieve better performance, the data stored in the global memory must be transferred to the shared memory of GPU if this data will be used more than once. In addition, the shared memory can be only shared by threads in a block. It means the temporary result calculated by those threads of a block has to be transferred back to global memory, if this temporary result will be used by threads of other blocks in the future. Compared to the performance of program by using global memory only, the program adopted shared memory has better performance. However, the procedure of data transfer between global memory and shared memory takes extra run time.

Moreover, in the GPU implementation, the number of threads needed for a kernel is far more than available processors. So threads are time-sliced on a round-robin basis, which causes extra run time as well. On the other hand, the CPU implementation does not have the problem of sliced time.

In the parallel computation, some threads would be completed a subtask before others, so the synchronization of threads is essential. In this case, the thread that finished its subtask will be idle, and waiting for all threads completed their subtask. The idle thread takes some run time.

6.2 Conclusion

This project studied and analyzed the majority of algorithms related to public key algorithms, and then designed and made an implementation of a public key algorithm RSA in CUDA. From the performance comparison between the GPU implementation and the CPU implementation, it looked into the possibility of improving the performance of public key algorithms by using CUDA-enabled GPU with the massive parallel processing properties .

The understanding of CUDA's architecture and how threads are organized in CUDA are very important for this project to have an efficient implementation of public key algorithms.

The experimental results show public key algorithms can be implemented fast on a CUDA-enabled GPU. With the massive parallel processing properties, the public key algorithm implemented on the GPU is more efficient than on the CPU.

6.3 Future work

The programs realized in this project are only implemented in a single experimental environment. They could be performed on different platforms to test and compare their performances in the following research.

Moreover, the parallelization of public key algorithms is mainly performed in the part of modular multiplication and modular exponentiation. Barrett modular reduction and Montgomery modular multiplication are mainly considered. However, some methods are not discussed in this project, such as the factor method and the power three method given by Knuth [26] for modular exponentiation, canonical recoding algorithm [34] [35] [36] for cryptographic algorithm, and fast decryption using the CRT [37] [36] [26] [38].

Also some improved Barrett modular reduction and Montgomery algorithms are valuable to be discussed in the future. The parallelization of these methods could give us to look into the possibility of improving the performance of public key algorithms by using CUDA from other angles.

Bibliography

[1] Li, C., Wu, H., Chen, S., Li, X., & Guo, D. Aug. 2009. Efficient implementation for md5-rc4 encryption using gpu with cuda. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, 167–170.

[2] Di Biagio, A., Barenghi, A., Agosta, G., & Pelosi, G. May 2009. Design of a parallel aes for graphics hardware using the cuda framework. In *Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 1–8.

[3] Cuda programming guide. *http://developer.download.nvidia.com*.

[4] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., & Volkov, V. July-Aug. 2008. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4), 13–27.

[5] Smid, M. & Branstad, D. May 1988. Data encryption standard: past and future. *Proceedings of the IEEE*, 76(5), 550–559.

[6] Xiao, Y., Guizani, S., Sun, B., Chen, H.-H., & Wang, R. 27 2006-Dec. 1 2006. Nis05-1: Performance analysis of advanced encryption standard (aes). In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, 1–5.

[7] Rivest, R. L., Shamir, A., & Adleman, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 26(1), 96–99.

[8] Miller, V. S. 1986. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, 417–426, New York, NY, USA. Springer-Verlag New York, Inc.

[9] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. 1997. *Handbook of applied cryptography*. CRC Press.

[10] Mohapatra, P. K. 2000. Public key cryptography. *Crossroads*, 7(1), 14–22.

[11] Fu, C. & Zhu, Z.-L. Oct. 2008. An efficient implementation of rsa digital signature algorithm. In *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference on*, 1–4.

[12] Cramer, R. & Shoup, V. 2000. Signature schemes based on the strong rsa assumption. *ACM Trans. Inf. Syst. Secur.*, 3(3), 161–185.

[13] Gennaro, R., Jarecki, S., Krawczyk, H., & Rabin, T. 1996. Robust and efficient sharing of rsa functions.

[14] Boneh, D. & Franklin, M. 2001. Efficient generation of shared rsa keys. *J. ACM*, 48(4), 702–722.

[15] Chandra, S. S. & Chandra, K. 2005. Cbigint class: an implementation of big integers in c++. *J. Comput. Small Coll.*, 20(4), 77–83.

[16] Aboud, S., Al-Fayoumi, M., Al-Fayoumi, M., & Jabbar, H. April 2008. An efficient rsa public key encryption scheme. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, 127–130.

[17] Wang, Y., Maskell, D., Leiwo, J., & Srikanthan, T. Dec. 2006. Unified signed-digit number adder for rsa and ecc public-key cryptosystems. In *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*, 1655–1658.

[18] Diffie, W. & Hellman, M. Nov 1976. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6), 644–654.

[19] Koblitz, N. 1987. Elliptic curve cryptosystems. In *Mathematics of Computation*, 203–209. American Mathematical Society.

[20] Rabin, M. O. Digitalized signatures and public-key functions as intractable as factorization. Technical report, Cambridge, MA, USA, 1979.

[21] Shimada, M. Nov. 1992. Another practical public-key cryptosystem. *Electronics Letters*, 28(23), 2146–2147.

[22] Okamotol, U. & Uchiyamal, S. July. 1998. A new public-key cryptosystem as secure as factoring. *Advances in Cryptology — EUROCRYPT'98*, 1403/1998, 308–318.

[23] Pointcheval, D. July. 1999. New public key cryptosystems based on the dependent-rsa problems. *Advances in Cryptology — EUROCRYPT '99*, 1403/1999, 239–254.

[24] Padhye, S. October. 2006. On drsa public key cryptosystem. *the International Arab Journal of Information Technology,*, 3, 334–336.

[25] Menezes, A. J., Vanstone, S. A., & Oorschot, P. C. V. 1996. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA.

[26] Knuth, D. 1981. The art of computer programming:seminumerical algorithms,. *seminumerical algorithms*, 2.

[27] Booth, A. 1951. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 236–240.

[28] Barrett, P. 1986. Implementating the rivest, shamir and aldham public-key encryption algorithm on standard digital signal processor. *Proceedings of CRYPTO'86, Lecture Notes in Computer Science*, 311–323.

[29] Bewick, G. 1994. Fast multiplication algorithms and implementation.

[30] Dhem, J. May 1998. Design of an efficient public-key cryptographic library for risc-based smart cards.

[31] Montgomery, P. L. Apr 1985. Modular multiplication without trial division. *Mathematics of Computation, Vol. 44, No. 170*, 519–521.

[32] Nvidia cuda c programming best practices guide. *http://developer. download. nvidia. com*.

[33] Han, T. D. & Abdelrahman, T. S. 2009. hicuda: a high-level directive-based language for gpu programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 52–61, New York, NY, USA. ACM.

[34] G.W.Reitwiesner. 1960. Binary arithmetic. *Advances in Computers*, 231–308.

[35] Gosling, J. august 1979. Book review: Computer arithmetic: Principles, architecture, and design. *Computers and Digital Techniques, IEE Journal on*, 2(4), 183.

[36] Koren, I. 1993. *Computer arithmetic algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[37] H.L.Garner. June 1959. The residue number systems. *IRE Transactions on Electronic Computers*, (8(6)), 140–147.

[38] Lipson, J. D. 1981. *Elements of Algebra and Algebraic Computing*. MA: Addison-Wesley.

A Subtraction in CUDA

This appendix contains the source code of parallel subtraction implemented in CUDA.

```
__global__ void cuda_Sub_SameLength(SLargeNum *r1,
                        SLargeNum *r2,SLargeNum *r)
{
        // r = r1 - r2
        // r1 and r2 must be greater than 0
        // and assume the length of r1 and r2 are the same

        int idx = blockIdx.x*blockDim.x + threadIdx.x;
        int length_r1 = r1->length;

        for( int i = length_r1-1; i>=0; i-- )
        {
                if(r1->digits[i] < r2->digits[i])
                {       //r1 < r2
                        SLargeNum *temp = r1;
                        r1 = r2;
                        r2 = temp;
                        r->sign = NEGATIVE;
                        break;
                }
                else if(r1->digits[i] > r2->digits[i])
                {       //r1 > r2
                        r->sign = POSITIVE;
                        break;
                }
        }

        if( idx < length_r1)
        {
                r->digits[idx] =
                        r1->digits[idx] - r2->digits[idx];
        }
        else if( idx == length_r1)
        {
                r->length = length_r1;
                r->base = BASE;
                r->header = 0;
                r->sign = POSITIVE;
        }
}
```

```
__global__ void cuda_Sub_SameLength_Update(SLargeNum *r
                                , int offset)
{
        // update r after subtraction
        int idx = blockIdx.x*blockDim.x + threadIdx.x;

        if( idx == 0 )
        {
                int len = r->length ;
                int carry = 0;
                int temp = 0;
                int i=0;
                for(; i<len; i++)
                {       //udpate
                        temp = r->digits[i] + carry;
                        if( temp >= 0 )
                        {
                                carry = 0;
                                r->digits[i] = temp % BASE;
                        }
                        else
                        {
                                carry = -1;
                                r->digits[i] = temp + 10;
                        }
                }

                if( carry == -1 )
                {
                        // set the sign of r
                        r->sign = NEGATIVE;
                }

                for(int l=len-1; l>=0; l--)
                {
                        //check the length of r
                        if(r->digits[l] != 0)
                        {
                                r->length = l+1;
                                break;
                        }
                }
        }
}
```

B Right shifts in CUDA

This appendix contains the source code of parallel right shifts with k bits implemented in CUDA.

```
__global__ void cuda_RightShifts(SLargeNum *tempResult
                        ,SLargeNum *Result,int k)
{
        // Right shifts with k bits
        int idx = blockIdx.x * blockDim.x + threadIdx.x;

        int length_tempResult = tempResult->length;
        if( idx < length_tempResult)
        {
          //The target position of Result
                int target = idx - k;
                if( target >= 0)
                {
                        Result->digits[target] =
                                tempResult->digits[idx];
                }
        }
        else if(idx == length_tempResult)
        {
                Result->header = 0;
                Result->length = length_tempResult - k;
                Result->sign = POSITIVE;
                Result->base = BASE;
        }
}
```

C Multiplication in CUDA

This appendix contains the source code of parallel multiplication implemented in CUDA.

```
__global__ void Multiplication(SLargeNum *first,
                               SLargeNum *second,
                               SLargeNum *tempResult)
{
        // Multiplication in CUDA
        // tempResult = first * second;

        int idx = blockIdx.x*blockDim.x + threadIdx.x;
        int length_first = first->length;
        int length_second = second->length;

        //copy the first and second to shared memory
        __shared__ unsigned char sh_first[DIGITS_MAX_LEN/2];
        __shared__ unsigned char sh_second[DIGITS_MAX_LEN/2];

        for( int td = threadIdx.x; td < length_first ;
                               td = td + blockDim.x )
        {
                sh_first[td] = first->digits[td];
        }

        for( int td = threadIdx.x; td < length_second ;
                               td = td + blockDim.x )
        {
                sh_second[td] = second->digits[td];
        }
        __syncthreads();

        // Calculate all values for each column
        if (idx < length_first)
        {
                int m = 0; //row of intermediate results
                int n = idx; //column of intermeidate results
                int temp = 0;

                while( n >= 0 && m < length_second       )
                {
                        temp = temp + sh_second[m]*sh_first[n];
                        m++;
                        n--;
                }
                tempResult->digits[idx] = temp;
        }
        else if ( idx < ((length_first + length_second)-1) )
        {
                int n = length_first - 1;
                int m = idx - n ;
                int temp = 0 ;

                while ( m < length_second && n>=0)
                {
                        temp = temp + sh_second[m]*sh_first[n];
                        m++;
                        n--;
                }
                tempResult->digits[idx] = temp;
        }

        if( idx == 0 )
        {
                tempResult->length =
                length_first + length_second - 1;
        }

}
```

```
__global__ void cuda_Carry_Update(SLargeNum *tempResult)
{
        int idx = blockIdx.x*blockDim.x + threadIdx.x;
        if( idx == 0 )
        {
                int len = tempResult->length ;
                int carry = 0;
                int temp = 0;
                int i=0;
                for(; i<len; i++)
                {
                        temp = tempResult->digits[i] + carry;
                        carry = temp / BASE;
                        tempResult->digits[i] = temp % BASE;
                }

                if( carry != 0 )
                {
                        tempResult->digits[i] = carry;
                        tempResult->length = i+1;
                }
        }
}
```

D Montgomery modular multiplication

This appendix contains the source code of Montgomery modular multiplication implemented on CPU.

```

void MonMul(SLargeNum *x, SLargeNum *y, SLargeNum *m, int mp,
                SLargeNum *w, SLargeNum *xiy, SLargeNum *uim,
                SLargeNum *temp1, SLargeNum *temp2)
{
        // Montgomery Multiplication
        // w = xyR^(-1) mod m wherer R=base^n
        // and n is the length of m;


        for(int i=0; i<=m->length; i++)
        {
                w->digits[i] = 0;
        }

        w->length = m->length+1;
        w->header = 0;
        w->sign = x->sign * y->sign;
        w->base = x->base;

        for(int i=0; i<m->length;i++)
        {
                int xi;
                if( i < x->length)
                {
                        xi = x->digits[i];
                }
                else
                {
                        xi = 0;
                }
                int ui = ( (w->digits[0]
                        + xi * y->digits[0]) * mp ) % BASE;

                //xiy = xi * y
                Mul_LN_Int(y, xi, xiy);

                //uim = ui * m
                Mul_LN_Int(m, ui, uim);

                //temp1 = xiy + uim
                Addition_2LargeNum(xiy, uim, temp1);

                //temp2 = temp1 + w
                Addition_2LargeNum(temp1, w, temp2);

                // w = temp2 / base using right shift once
                for(int j=1; j<temp2->length; j++)
                {
                        w->digits[j-1] = temp2->digits[j];
                }
                w->length = temp2->length - 1;
        }

        if( Compare_2LN(w,m) )
        {
                Sub_2LN(w,m,temp1);
                w = temp1;
        }

}
```

E Execution time of Montgomery's programs in CUDA

This appendix contains the execution time of modular multiplication and exponentiation by using Montgomery's algorithm in CUDA.

Bit-length	Time_{GPU} (ms)
1024	43.203
2048	81.653
3072	125.405
4096	162.710
8192	325.522

Table 29: Execution time of Montgomery modular multiplication in CUDA.

Bit-length	Time_{GPU} (s)
1024	54.625
2048	221.758
3072	491.339
4096	971.857
8192	3699.907

Table 30: Execution time of Montgomery modular exponentiation in CUDA.