GPU-based parallel optimization strategy for Barrett's algorithm

Yufan Wang*[1],Chengzhao Yang[1],Shiju Wang[1],Zhengyang Han[1]
[1]School of Computer Science and Engineering,Beihang University,Beijing,China 100191

ABSTRACT

With the development of computer science, hardware devices have emerged, which greatly enhance the efficiency of computing and data processing. GPU, as a highly parallelized computing chip, can be used in areas such as personal needs and scientific computing.Additionally, advances in cryptography enable data to have more effective security; and there are some computational bottlenecks in cryptographic algorithms during practical use. The use of GPUs to accelerate cryptographic algorithms is one of the hot topics of research in the industry and academia today.

RSA algorithm is the most widely used asymmetric cryptography public-private key system, which guarantees the security of the public-private key generated by the algorithm through the difficulty of large number decomposition. In the actual encryption and decryption process of RSA, substantial large number power modulo operations are involved, which seriously affects the computing efficiency of RSA and a set of optimization strategies are urgently needed to shorten the computing time of RSA.

In RSA algorithm, there is parallel optimization space for modulo operation. In this paper, the structural optimization of the algorithm is realized based on Montgomery algorithm, and the algorithm's arithmetic optimization is realized by using Barrett reduction algorithm to achieve the algorithm-level tuning.And the optimization algorithm is deployed on the GPU side through CUDA architecture.

By comparing with the large number library GMP running on the CPU, the algorithm optimization and parallelization deployment in this paper achieves significant speedup in the RSA encryption and decryption operations.

Keywords: RSA Algorithm, Montgomery Algorithm, Barrett Reduction Algorithm, CUDA, GPU

1. INTRODUCTION

With the rapid development of electronic communication and information technology, the importance of information security in various fields such as personal life, economic construction and national security has become more and more prominent. 1976 was a milestone in the field of cryptography when Diffie and Hellman proposed the asymmetric cryptosystem, which greatly improved the security of encryption compared with the traditional symmetric cryptosystem. Among many cryptographic algorithms and protocols, RSA is one of the most widely used asymmetric cryptographic algorithms, which is widely used in information encryption, digital signatures, and authentication[1].The theoretical basis of RSA cryptographic algorithm is the power mode operation, and its security is based on the difficulty of prime factorization of large integers in number theory[2]. However, the number of bits of the public and private keys in current RSA cryptosystems is generally 1024 or 2048 bits, and may reach even larger for security, so the key generation as well as the encryption and decryption process is a large number of large power-mode calculations. Compared to other

* 21373293@buaa.edu.cn; phone 1 881 108-6981

encryption algorithms of other security levels, speed has always been a drawback of RSA algorithm, both in software and hardware implementation[3].

By studying the characteristics of RSA algorithm and analyzing many power mode and mode multiplication algorithms, this paper finds that there are many operations in the design of RSA cryptosystem that have great room for parallelization and optimization. Thinking from the hardware perspective, GPU provides a good hardware basis for the optimization direction of this project. With the rapid development of the microelectronics industry in recent years, GPUs have developed into massively parallel, multi-threaded, multicore processor systems with huge computing power[4-5]. In this paper, we can parallelize the modulo multiplication operation and deploy it on GPUs, thus using parallel computing to greatly accelerate the speed of RSA algorithm encryption and decryption.

Based on the above research considerations, this paper explores and implements the optimization of the modulo multiplication algorithm in RSA and the deployment and operation of the modulo multiplication algorithm on GPU. For the RSA encryption and decryption algorithm, this project implements the structural optimization of the algorithm based on Montgomery's algorithm on the one hand, and the computational optimization of the algorithm based on Barrett's approximate subtraction algorithm on the other hand, which greatly improves the parallelism of the algorithm. For the parallelization deployment of the algorithm, this project mainly improves the parallelization of Barrett's approximate subtraction algorithm, and then deploys Barrett's algorithm on the GPU side through CUDA (Compute Unified Device Architecture) architecture, which gives full play to the advantages of GPU parallel computing and greatly improves the speed of large integer modulo multiplication. The speedup and optimization of RSA algorithm is achieved by giving full play to the advantages of GPU parallel computing, which greatly improves the speedup of large integer modulo multiplication, compared with the serial computing of traditional algorithms on CPU.

The paper is structured as follows: first, the RSA algorithm, the object of optimization operations, and Montgomery's algorithm and Barrett's approximate subtraction algorithm, the basis of optimization of modulo multiplication operations, are introduced. Next, the scheme for implementing the optimization and parallelization of the algorithm is described. Then, the paper uses data samples to show the optimization effect compared with traditional algorithms on CPU, and analyzes and considers the advantages of this project and the areas for improvement. Finally, the work done is summarized and the outlook is expressed.

2. RELATED TECHNICAL ANALYSIS

2.1 RSA Algorithm

The RSA[6] algorithm, the most widely used asymmetric key algorithm, was proposed in 1977 by Rivest, Shamir and Adleman at MIT, U.S.A. The theoretical basis of the RSA algorithm is a number-theoretic fact: it is easy to multiply two large prime numbers, but extremely difficult to factorize their product[1].

2.1.1 Mathematical Foundations

(1)Prime number: Among the natural numbers greater than 1, a number that cannot be divided by any other natural number except 1 and itself is a prime number.

(2)Factor:An integer b is said to be a factor of a if the quotient is an integer and the remainder is 0, if the integer a is divided by an integer b that is not 0.

(3)The greatest common factor: The greatest factor that two or more integers have is called the greatest common factor of these numbers. $gcd(a, b)$ is generally used to denote the greatest common factor of a and b.

(4) Congruence: For a positive integer m, if the difference between two integers a, b is divisible by m, i.e., $(a - b) \, mod \, m = 0$, then a,b is said to be congruent to modulus m, denoted as $a \equiv b \, (mod \, m)$.

(5) Euler's theorem

Euler function is used to calculate for a given positive integer n, to meet less than or equal to n and with n there are two conditions of the number of positive integers, usually noted as x, Euler function is expressed as follows:

$$\varphi(n) = n \left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \ldots\ldots \left(1 - \frac{1}{p_{k-1}}\right)\left(1 - \frac{1}{p_k}\right) \quad (1)$$

The Euler function has the following properties:

If n is a prime number, then

$$\varphi(n) = n - 1 \quad (2)$$

If two positive integers satisfy $gcd(a, n) = 1$, then we have

$$a^{\varphi(n)} \equiv 1 \, (mod \, n) \quad (3)$$

(6) Single trapdoor function

One-way function is a one-shot function, that is, each number y in the value domain has and has only one number x in the definition domain, so that $f(x) = y$. One-way function is irreversible, that is, the value of the output of the function obtained by random input, the reverse calculation of the output is very difficult.

A single trapdoor function is a one-way function, which means that there is an extra information z such that knowing z is necessary to calculate the input in the reverse direction. The extra information z is also called a trapdoor, and the basic principle of asymmetric key algorithms is to set mathematical puzzles as traps in the single trapdoor function.

2.1.2 RSA algorithm description

RSA algorithm is the most widely used asymmetric key algorithm, which divides the key into public key and private key, and ensures the security of the algorithm by decomposing large numbers as trapdoors. In the encryption and decryption process, the sender encrypts the plaintext by one-way trapdoor function to get the ciphertext, and the receiver gets the ciphertext and trapdoor information (i.e., private key) to find out the plaintext, but it is very difficult for the receiver to find out the plaintext without the trapdoor information.The steps of RSA algorithm are divided into key generation, message encryption and decryption.

RSA key generation process:

(1) Pick any two large prime numbers p and q (confidential).

(2) Compute the overt modulus $n = p \times q$ (overt) and compute the Euler function $\varphi(n) = (p-1) \times (q-1)$ for n (confidential).

(3) Pick a random decryption key d , d is to be satisfied $0 < d < \varphi(n)$, and $gcd\left(d, \varphi(n)\right) = 1$ (confidential).

(4) Compute the encryption key e ,to satisfy $0 < e < \varphi(n)$, and $de \equiv 1\ (mod\ \varphi(n))$ (overt).

(5) Obtain the public public key pair (e, n) and the private non-public key pair (d, n) and destroy p,q and $\varphi(n)$.

Information encryption and decryption process:

The plaintext message is usually a string consisting of numbers and characters, and the first step of encryption with RSA is to digitize and block the plaintext to ensure that the length of the plaintext block is less than $log2n$ bits. Similarly, when decrypting the long ciphertext, we need to restore the plaintext by segmenting it according to this rule.

Encryption of plaintext: The sender gets the public key pair (e, n) and encrypts the plaintext m by the following equation (2.3), and the encrypted ciphertext c will be transmitted to the receiver through the unsecured channel.

$$c \equiv m^e\ (mod\ n) \tag{4}$$

Decrypt the ciphertext: after receiving the ciphertext c through the unsecured channel, the receiver decrypts the private key pair (d, n) and the following equation (2.4) to get the plaintext m.

$$m \equiv d^d\ (mod\ n) \tag{5}$$

The overall flow of the RSA algorithm is as follows

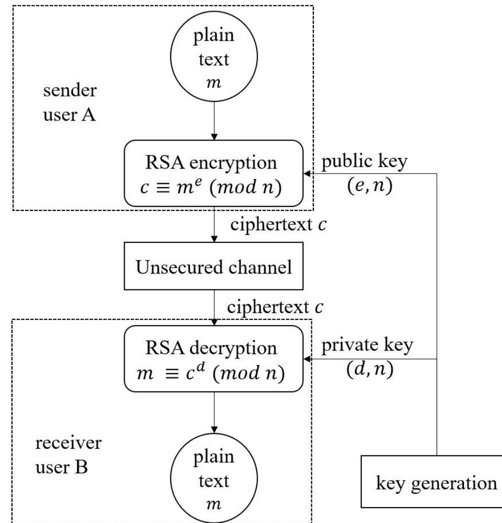

Figure 1. RSA algorithm flowchart

2.2 Montgomery modulo multiplication algorithm

The Montgomery algorithm, proposed by American mathematician Peter L. Montgomery in 1985, consists of the power modulus operation, the modulus multiplication operation, and the approximate subtraction algorithm. The structure of the Montgomery modulo operation is as follows:

Algorithm $\mathbb{Z}$-MontExp

Input: A base- b,unsigned integer $0 \leq x < N$, and a base-2, unsigned integer $0 \leq y < N$

Output: A base $-$ b, unsigned integer $r = x^y (\mathrm{mod} N)$

1. $\hat{t} \leftarrow \mathbb{Z} -$ MontMul$(1, \rho^2(\mathrm{mod\ N}))$

2. $\hat{x} \leftarrow \mathbb{Z} -$ MontMul$(\mathrm{x}, \rho^2(\mathrm{mod\ N}))$

3. for $i = |y| - 1$ **downto** 0 **step** -1 **do**

4. $\hat{t}$

$\leftarrow$ $\mathbb{Z}$-MontMul $(\hat{t}, \hat{t})$

5. if $\mathrm{y_i} = 1$ **then**

6. $\hat{t} \leftarrow$ $\mathbb{Z}$-MontMul $(\hat{t}, \hat{x})$

7. end

8. end

9. return $\mathbb{Z}$-MontMul $(\hat{t}, 1)$

Figure 2. Montgomery algorithm

2.3 Barrett reduction algorithm

Barrett reduction algorithm is a highly efficient way to compute $r = z\, mod\, p$. It is an optimization of the modulo equation for human computation, $a\, mod\, b = a - \lfloor a/p \rfloor * p$. The normal computation of $\lfloor a/p \rfloor$ involves a large division overhead. Barrett algorithm works by properly selecting the base b, so that a low-cost quotient operation yields an approximate estimate $\hat{q}$ for $q = \lfloor z/p \rfloor$, such that z minus $\hat{q} * p$ can be subtracted a small number of times to obtain the remainder r. Many of the intermediate variables in the algorithm's computation are related to the modulus only, so it is suitable for calculations that take the same modulus multiple times. The algorithm flow is as follows.

Algorithm Barrett reduction

Input: $p, b \geq 3, k = \lfloor \log_b p \rfloor + 1, 0 \leq z < b^{2k}$, and $\mu = \lfloor b^{2k}/p \rfloor$.

Output: z mod p.

1.$\hat{q} \leftarrow \left\lfloor \lfloor z/b^{k-1} \rfloor \cdot \mu / b^{k+1} \right\rfloor$

2.$r \leftarrow (\mathrm{z\ mod\ b^{k+1}}) - (\hat{q} \cdot \mathrm{p\ mod\ b^{k+1}})$

3.if $r < 0$ **then**

4.$r \leftarrow r + b^{k+1}$

5.end

6.while $r \geq p$ **do**:

7. $r \leftarrow r - p$

8.end

9.return **r**

Figure 3. Barrett reduction algorithm

For a given input z and p, the first step is to choose an appropriate base b, which is usually a power of 2. The choice of b also depends on the modulus p. In addition, an appropriate value of k is chosen so that $z < b^{2k}$, and preprocessing yields $\mu = \lfloor b^{2k}/p \rfloor$.

The most careful step in Barrett algorithm is to find the approximate estimate $\hat{q}$ of the quotient q at a lower cost and as accurately as possible. Let $\hat{q} = \lfloor \lfloor z/b^{k-1} \rfloor * \mu/b^{k+1} \rfloor$.

By
$$\left\lfloor \left\lfloor \frac{z}{b^{k-1}} \right\rfloor * \frac{\mu}{b^{k+1}} \right\rfloor = \left\lfloor \left\lfloor \frac{z}{b^{k-1}} \right\rfloor * \left\lfloor \frac{b^{2k}}{p} \right\rfloor * \frac{1}{b^{k+1}} \right\rfloor$$
$$\leq \left\lfloor \frac{z}{b^{k-1}} * \frac{b^{2k}}{p} * \frac{1}{b^{k+1}} \right\rfloor = \left\lfloor \frac{z}{p} \right\rfloor \quad \#(6)$$

We get $\hat{q} \leq q$

Further assumptions

$$\alpha = \frac{z}{b^{k-1}} - \left\lfloor \frac{z}{b^{k-1}} \right\rfloor \quad \#(7)$$
$$\beta = \frac{b^{2k}}{p} - \left\lfloor \frac{b^{2k}}{p} \right\rfloor \quad \#(8)$$

Obviously there is $0 \leq \alpha,\ \beta \leq 1$

This leads to

$$q = \left\lfloor \frac{z}{b^{k-1}} * \frac{b^{2k}}{p} * \frac{1}{b^{k+1}} \right\rfloor = \left\lfloor \frac{\left(\left\lfloor \frac{z}{b^{k-1}} \right\rfloor + \alpha \right) \left(\left\lfloor \frac{b^{2k}}{p} \right\rfloor + \beta \right)}{b^{k+1}} \right\rfloor \leq \left\lfloor \hat{q} + \left\lfloor \frac{z}{b^{k-1}} \right\rfloor + \left\lfloor \frac{b^{2k}}{p} \right\rfloor + \frac{1}{b^{k+1}} \right\rfloor \quad \#(9)$$

Since $z < b^{2k}$, $\lfloor z/b^{k-1} \rfloor \leq b^{k+1} - 1$; and since $k = \lfloor \log_b p \rfloor + 1$, $p \geq b^{k-1}$, that is, $\lfloor b^{2k}/q \rfloor \leq b^{k+1}$.

Combining the above two points, the inequality can be transformed into

$$q \leq \left\lfloor \frac{\hat{q} + b^{k+1} - 1 + b^{k+1} + 1}{b^{k+1}} \right\rfloor = \lfloor \hat{q} + 2 \rfloor \quad \#(10)$$

In summary, it can be shown that the estimated $\hat{q}$ calculated by making $\hat{q} = \lfloor \lfloor z/b^{k-1} \rfloor * \mu/b^{k+1} \rfloor$ in the first step of the algorithm is a good approximation of q, which satisfies the sufficient condition

$$q - 2 \leq \hat{q} \leq q \quad \#(11)$$

After obtaining $\hat{q}$, $z - \hat{q} * p$ is computed, and then r can be obtained by subtracting at most twice (see equation (2.10)).

Through analysis, we found that the division and modulo involved in Barrett algorithm can all be reduced by bitwise operations .For example, the division of b^{k-1} and b^{k+1} in $\hat{q}$ can be optimized as a right-shift operation in b decimal, and the modulo of b^{k+1} in the second step of Barrett algorithm can be optimized as the lower k bits in b decimal. Finding the quotient and remainder of two large integers is often a costly operation. Using the simplest trial division method to calculate the quotient and remainder of large integers, the time complexity will reach $O(n^2)$, which is less efficient, while after converting the division into a right shift in b-binary by bitwise operations, the time complexity can be reduced to $O(n)$, which reduces the cost of division and modulo taking in the algorithm and thus improves the computational efficiency. In addition, after the optimization of bitwise operations, the division and die-taking methods have good computational independence, which provides the basis for the parallelization optimization later on.

2.4 GPU and Parallel Computing

A computer graphics processing unit (GPU) is a single-chip processor that integrates lighting, geometric transformations, triangle construction, and drawing engines and has the processing power of at least 10 million polygons per second[9].GPUs have natural parallel characteristics that have greatly contributed to the rapid development of other areas of computing.

2.4.1 GPU Architecture

In terms of hardware architecture, the GPU consists of multiple stream processor clusters (SM), each equipped with multiple stream processors (SP) that can perform computations in parallel. the general architecture of the GPU is shown in Figure 4.

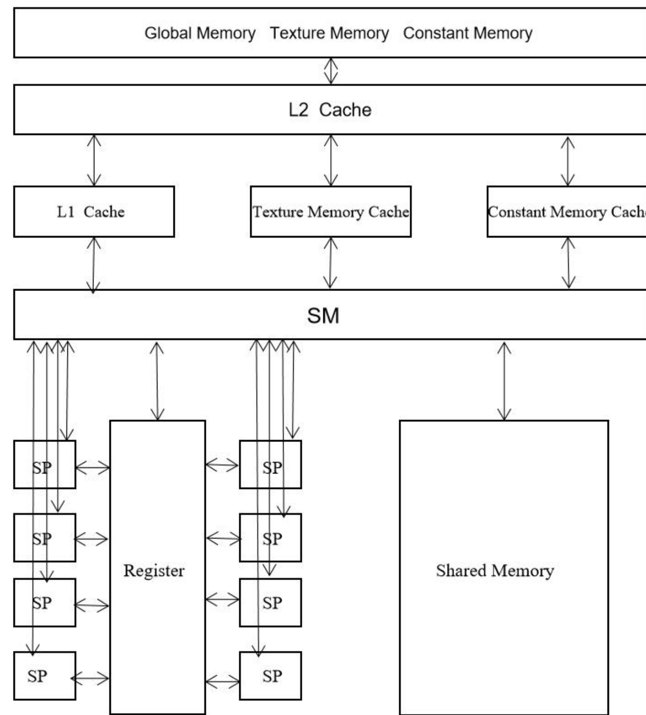

Figure 4. GPU Architecture

GPU-based parallel computing can be divided into three levels10. The most microscopic level is Instruction-Level Parallelism (ILP) on a single core, which relies on the microscopic parallelism of different operators within the processor to execute multiple instructions simultaneously; secondly, Multi-Core Parallelism, which is the integration of multiple processor cores on a single chip, where multiple processes or threads run simultaneously on these processor cores to achieve Thread-Level Parallelism / Process-Level Parallelism (TLP / PLP); again, multiprocessor parallelism, where multiple processors are placed on a printed circuit board to achieve multiprocessor-level thread or process parallelism; and finally, multiple independent computers are connected by a network to achieve cluster distributed parallelism at the independent computer level.

2.4.2 GPU hardware foundation

According to the laboratory conditions, NVIDIA GeForce RTX 3090 was selected as the GPU hardware for this project, and its performance parameters are shown in the following table:

Table 1. GPU hardware parameters.

multi-display	video memory capacity (GB)	video memory type	video memory bit width (bit)	video memory bandwidth (GB/s)	core frequency (MHz)	number of SP	number of ROPs	3D API	max resolut-ion
4	24	GDDR 6X	384	935.8	1400	10496	96	DirectX 12 Ultimate	7680*4320

2.5 CUDA Programming Model

2.5.1 Introduction to the CUDA model

CUDA (Compute Unified Device Architecture) is a parallel computing platform introduced by NVIDIA in 2007, which provides a series of APIs that allow developers to fully exploit the power of GPUs in a user-friendly programming environment. CUDA provides interfaces to C, C++, and Fortan languages, and this project can use CUDA to deploy simple and intensive operations in programs to the GPU to parallelize computation.

The CUDA programming model is a heterogeneous computing architecture that employs the mechanism of CPU and GPU working together, in which the CPU is considered as the host and the GPU as the device, as shown in Figure 5, and the CPU and GPU are connected through the PCI-E bus[7]. The CPU has rich control logic and is responsible for logical transaction processing and serial computing; the GPU has simple control logic but rich computational resources on the GPU and is responsible for performing highly threaded parallel processing tasks[8].

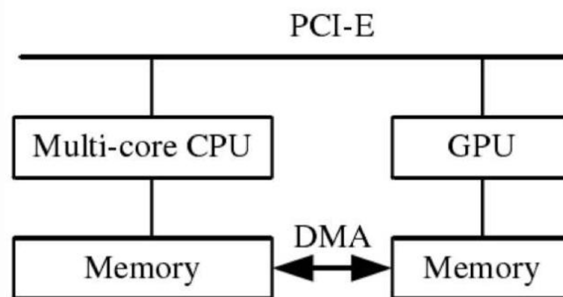

Figure 5. CPU and GPU connection schematic

CUDA has direct access to the GPU virtual instruction set and parallel computing elements. When programming with CUDA, code in traditional C or C++ languages is written on the host side (CPU) first, and for the part that needs to be accelerated by the GPU in parallel, a kernel function is written and the number of threads needed to start the kernel function is specified at the appropriate location in the host side. The kernel function is executed on the Device side, and a certain number of threads are assigned to execute the same code to achieve a simple but intensive parallel computation. In the CUDA programming model, the kernel function kernel has three levels of structure, as shown in Figure 6, which are grid,block and thread.The code in the kernel will be executed n times by n threads. The code in the kernel will be executed n times by n threads, and the n threads are placed into a number of blocks, each of which shares the data, and a

number of the same blocks form a grid. $blockIdx.x * blockDim.x + threadIdx.x$ to perform the relevant operations. During actual execution, the GPU will execute threads in batches in a thread bundle (wrap). Currently, a CUDA wrap is 32 threads, i.e. 32 threads execute an instruction at the same time at runtime.

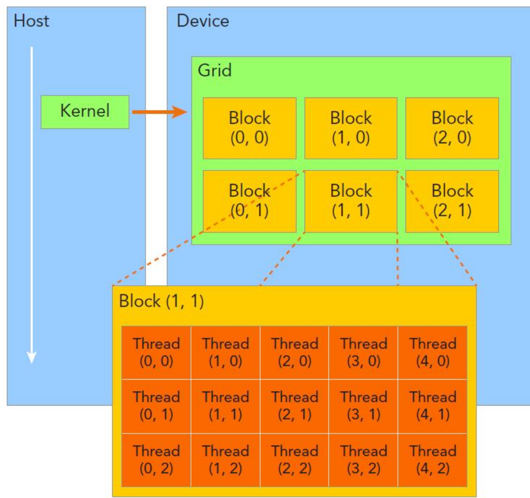

Figure 6. Schematic diagram of CUDA kernel functions

2.5.2 CUDA Software Foundation

At the CUDA software level, the C++ compiler used in this project is g++, the CUDA compiler is nvcc.exe, and the core is the NVIDIA compiler nvcc.exe.

The project also uses CUDA's built-in cuFFT library (CUDA® Fast Fourier Transform) to accomplish the large number multiplication operation FFT deployment to GPU. cuFFT library provides a simple interface to compute Fast Fourier Transforms and their inverse transforms. cuFFT library is based on the FFTW port, which is able to achieve a high speedup compared to the CPU. The cuFFT library is based on the FFTW port and has been able to achieve a very high speedup ratio compared to the CPU[7].

2.6 CPU-based large number calculation library GMP

GMP (GNU Multiple Precision Arithmetic Library) is a publicly available C/C++ library for arbitrary precision arithmetic. It defines GMP's built-in large number type mpz_class and overloads a series of common operators such as +,-,*,/,% and other operators.The main target applications of GMP are cryptographic applications and research, Internet security applications, and computer algebra systems. Compared with other C libraries of arbitrary precision, GMP library has faster and more stable performance in large number calculation.GMP has highly optimized, processor-specific assembly language code for the most important internal loops, using the whole word as the basic arithmetic type. More importantly, in response to the slowness of large number algorithms for smaller bit-width numbers, GMP uses different algorithm structures for different bit-width data, making it more powerful for both small and large bit-width numbers.

In this paper, the results of parallelization on GPU are compared with the large number reloading operators implemented in the GMP library to better reflect the significant optimization effect of algorithm optimization and parallelization deployment in RSA algorithm in this project.

2.7 Chapter Summary

This chapter introduces the RSA asymmetric key algorithm, Montgomery modulo multiplication algorithm and Barrett reduction algorithm, and lays the foundation for the subsequent work. In addition, this chapter introduces the CUDA programming framework and GPU architecture, and analyzes its parallelism characteristics.

3. ALGORITHM OPTIMIZATION AND PARALLELIZED DEPLOYMENT

3.1 Structural optimization of the algorithm based on Montgomery algorithm

RSA encryption and decryption operations are all modulo powers of large numbers. In this paper, we refer to the structure of Montgomery modulo power operation and make some adjustments. Considering the calculation of $E(m) = c = m^e \; mod \; N$ (e is the encryption key), the e in RSA encryption algorithm is often an integer of order 10^4 or even higher, if we simply multiply and then take the modulus one by one, it will take at least 10^4 times of large integer multiplication and modulo operation to perform one encryption and decryption operation, which is very costly. When the amount of data to be communicated is large, it is often necessary to perform encryption and decryption several times at the same time, and such speed will seriously affect the communication throughput and communication rate. The structure of Montgomery modulo power algorithm can be referred to, the exponent b is split by binary, the original b times of multiplication and modulo operation is optimized to $log\, b$ times of modulo operation, which greatly accelerates the operation speed. The Montgomery modulo power operation uses the Montgomery modulo multiplication algorithm, which is not as parallelizable and faster than the Barrett modulo algorithm, as the main structure of our modulo power operation algorithm, as we replace the Montgomery modulo multiplication algorithm with the parallelized Barrett algorithm.

3.2 Algorithm optimization and parallelized deployment based on Barrett's algorithm

3.2.1 Parallelization of shift operations on GPU

The first step in Barrett algorithm calculates $\hat{q} = \lfloor \lfloor z/b^{k-1} \rfloor * \mu / b^{k+1} \rfloor$, where the division of b^{k-1}, b^{k+1} can be optimized as a right-shift operation in b-binary as in Figure 7. In this operation, after the number of right-shift bits is given, the starting bit src and the target bit des of each bit are determined without any dependency on each other, which is computationally independent and can be deployed on the GPU in parallel. The subscript is obtained by $blockIdx.x * blockDim.x + threadIdx.x$, which is directly assigned to the offset digit.

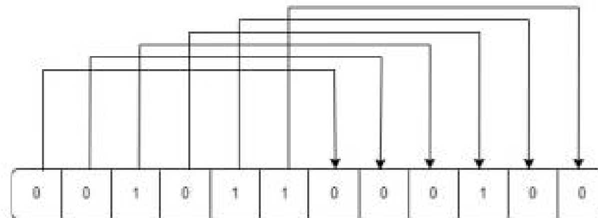

Figure 7. Schematic diagram of shift operation

It is important to note that direct parallel shifts, although computationally independent, have read-write conflicts on the same data segment, causing thread-unsafe consequences. In this regard, the solution in this paper is to use two data

segments, read from the source data segment and write to the other data segment, thus separating read and write, excluding data correlation and thread safety.

3.2.2 Parallelization of mode-taking operations on GPUs

The second step in Barrett's algorithm calculates $r \leftarrow (z\ mod\ b^{k+1}) - (\hat{q} \cdot p\ mod\ b^{k+1})$, where the modulo operation of b^{k+1} can also be optimized as a bitwise operation in b-binary, i.e., taking out the lower $(k+1)$ bits in b-binary. Taking out the lower $(k+1)$ bits of the b binary, each bit has no dependency on each other and is computationally independent, so it can also be deployed on the GPU in parallel with the CUDA architecture to improve performance.

As shown in Figure 8, the solution of this paper is to get the initial value by $blockIdx.x * blockDim.x + threadIdx.x$ for the large integer src and modulus k to be modulo, and to enumerate the subscripts by accumulating the steps of $gridDim.x * blockDim.x$ each time, setting the digits higher than $k+1$ to 0 directly, and keeping the digits lower than or equal to $k+1$ to achieve the modulo effect. After parallelized deployment, the time of the modulo computation is almost only determined by the communication efficiency between the host side and the device side, and the placement cost of the parallelized kernel function can be regarded as $O(1)$.

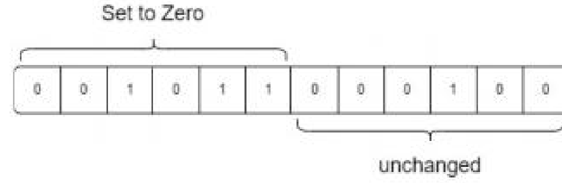

Figure 8. Schematic diagram of partial position 0

3.2.3 FFT optimization where multiplication

In this paper, we use the FFT algorithm to optimize the large multiplication part of the traditional Barrett algorithm, and also parallelize it on the GPU through the CUDA architecture.

3.2.3.1 Brief description of FFT principle

FFT is a divide-and-conquer algorithm for the efficient computation of the discrete Fourier transform of complex or real-valued data sets. It is one of the most important and widely used numerical algorithms in computational physics and general signal processing. It has inherently good partitioning properties, converting polynomial multiplication into point-valued representation by DFT, and reducing point values back to polynomial multiplication by IDFT, where it can be optimized with butterfly computational techniques to optimize the time complexity $O(n^2)$ for large number multiplication to $O(n \cdot \log_2 n)$. The following is the DFT formulation of the one-dimensional N-point

Fourier variation, where the rotation factor $W_N^{nk} = e^{-\frac{2\pi nk}{N}}(n = 0,1,2, \ldots, N-1)$.

$$X(k) = \sum_{n=0}^{N-1} x(n) * W_N^{nk} \qquad k = 0,1,2, \ldots, N-1$$

$$x(n) = \frac{1}{N} * \sum_{k=0}^{N-1} X(k) W_N^{-nk} \qquad k = 0,1,2, \ldots, N-1$$

3.2.3.2 Parallelization of multiplication operations on GPU based on FFT

Barrett's algorithm uses two large integer multiplications, $\hat{q} = \left\lfloor \left\lfloor \frac{z}{b^{k-1}} \right\rfloor * \frac{\mu}{b^{k+1}} \right\rfloor$ in the first step and $(\hat{q} * p \; mod \; b^{k+1})$ in the second step, for which the multiplication can be accelerated by FFT. In this project, we use the CUDA platform's native library cuFFT to assist in the deployment of parallelized FFT algorithms. cuFFT is modeled after FFTW and provides an interface similar to the FFTW library on the CPU, enabling users to call the API functions of the cuFFT library to complete FFT transforms and conveniently exploit the potential of GPU parallel computing.

Figure 9 shows the core code to implement parallelized multiplication with the API of cuFFT. The acceleration part of the multiplication in the algorithm is completed by first implementing the polynomial-to-point-value transformation by calling fft, then calling the kernel function to implement the parallel multiplication of point values on the GPU, and finally calling ifft to reduce the polynomial. For the FFT part encapsulate it into a function, the acceleration can be completed by directly calling the encapsulated function for $\lfloor z/b^{k-1} \rfloor$, μ/b^{k+1} calculated in the first step, and $\hat{q}$, p in the second step, and then just return the result to the whole Barrett algorithm flow again.

```
void fft(cufftComplex *in, cufftComplex *out, size_t size)
{
    cufftHandle plan;
    cufftPlan1d(&plan, size, CUFFT_C2C, 1);
    cufftExecC2C(plan, in, out, CUFFT_FORWARD);
    cufftDestroy(plan);
}
void ifft(cufftComplex *in, cufftComplex *out, size_t size)
{
    cufftHandle plan;
    cufftPlan1d(&plan, size, CUFFT_C2C, 1);
    cufftExecC2C(plan, in, out, CUFFT_INVERSE);
    cufftDestroy(plan);
}
__global__ void gpu_mult(cufftComplex *a, cufftComplex *b,
size_t size)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i >= size)  { a[i] = {0,0}; return; }
    a[i] = {a[i].x * b[i].x - a[i].y * b[i].y, a[i].x *
b[i].y + a[i].y * b[i].x};
}
```

Figure 9. Parallelized multiplication

3.3 Algorithm deployment on GPU via CUDA

The CPU-GPU collaborative architecture is shown in Figure 10, and the overall system operation flow is as follows.

(1) Read the plaintext to be encrypted from the disk, either one or more, and copy it to the allocated memory (Instances Generating).

(2) Using the CPU to convert the plaintext to ciphertext by a reversible padding scheme.

(3) Allocate the corresponding memory storage space in the video memory (cudaMalloc).

(4) Transfer the plaintext to the allocated area of the video memory using the PCI-E bus.

(5) Call the kernel function (kernel_mod) to parallelize the computation and write the result to the video memory.

(6) transfer the result back to the memory via PCI-E bus (Copying Back).

(7) Call the mature GMP large number library to compare the calculation results (Verifying Results).

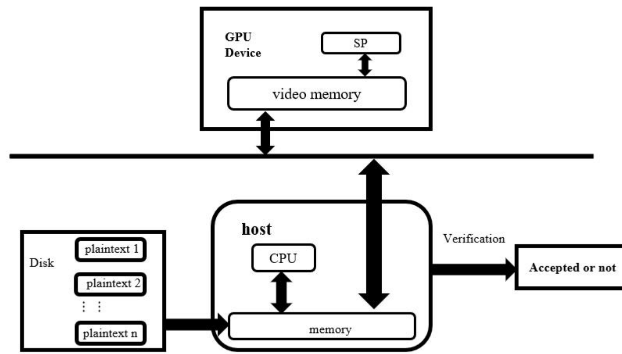

Figure 10. Schematic diagram of heterogeneous structure

In our RSA-1024 encryption experiments.

(1) the conversion of simple plaintext into ciphertext by reversible padding is left to the CPU (its processing is simpler and parallelization is not obvious).

(2) In order to reduce the time spent on data copy transfer (cost of cudaMalloc and Data Transfer), we request enough storage space in the video memory at once and copy all the ciphertext data to the video memory at once.

(3) The parallelization of the experiment is reflected in two major parts:

(i) As in Figure 11, the host side calls the kernel_mod kernel function to allocate 5120 threads ($gridDim.x = 32$, $blockDim.x = 160$) for simultaneous computation and decryption of idempotent mode operations, each thread is responsible for the computation, giving full play to the GPU parallelization computation capability;

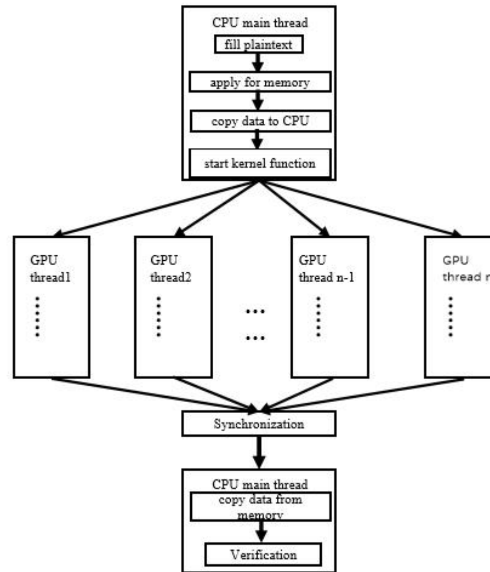

Figure 11 Schematic diagram of parallelization structure

(ii) As shown in Figure 12, the power modulus algorithm uses the parallelized Barrett algorithm. The multiplication involved in the Barrett algorithm uses the cuFFT library function to directly allocate threads on the GPU for parallelized computation, and the shift involved and the base power modulus are computed using our designed kernel function.

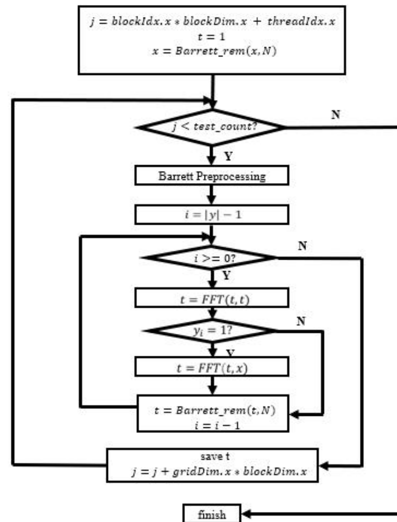

Figure 12 Flow chart of multi-thread oriented parallel algorithm

(4) Use the cudaDeviceSynchronize method to wait for all threads to finish computing and copy the data from the device side to the host side at once using cudaMemcpy.
(5) Calculate the standard answer on the CPU using the GMP serial large number library and compare it with the result calculated on the GPU to determine if it is correct.
(6) The number of digits and the number of operation samples are changed several times and compared with the serial results of the same algorithm structure in terms of correctness and speed to reflect the advantages of parallelization.
Based on the above algorithm optimization and parallelization scheme, we deploy the program to the GPU, and the following is the core code of the program, as shown in Figure 13.

```
// the actual kernel
__global__ void kernel_mod(cgbn_error_report_t *report, instance_t *instances, uint32_t count) {
  int32_t instance;

  // decode an instance number from the blockIdx and threadIdx
  instance=(blockIdx.x*blockDim.x + threadIdx.x)/TPI;
  if(instance>=count)
    return;

  context_t      bn_context(cgbn_report_monitor, report, instance);    // construct a context
  env_t          bn_env(bn_context.env<env_t>());                      // construct an environment for 1024-bit m
  env_t::cgbn_t  a, b, r,approx;                                               // define a, b, r as 1024-bit bignu
  uint32_t clz_count;
  cgbn_load(bn_env, a, &(instances[instance].a));      // load my instance's a value
  cgbn_load(bn_env, b, &(instances[instance].b));      // load my instance's b value
  int i=15;
  clz_count=cgbn_barrett_approximation(bn_env, approx, b);
   bn_env.barrett_rem( r, a, b, approx, clz_count); //r=a mod b
  while(i>=1){
  bn_env.mul(r,r,r);
  bn_env.barrett_rem( r, r, b, approx, clz_count); //r=a mod b
  i--;
  }
  bn_env.mul(r,r,r);
  bn_env.mul(r,r,a);
  bn_env.barrett_rem(r,r,b,approx,clz_count);
  cgbn_store(bn_env, &(instances[instance].mod), r);   // store r into sum
}
```

Figure 13 Core code demonstration

4. OPTIMIZATION RESULTS DISPLAY AND ANALYSIS

4.1 Optimization results comparison display

In this project, the above algorithm optimization and parallelization scheme is implemented and deployed to GPU platform to generate data by means of randomly generated large numbers to test the performance of the current program. Meanwhile, this project calls the modulo operation in the famous large number library GMP on CPU to calculate the same data with GPU to test the optimization effect of the scheme by comparing the running time of both.

First, we set the bit width of random numbers to 1024 bits and the number of test data sets to 100, 1000, 5000, 10000, 20000, 50000, 100000 to compare the running time of GPU and CPU, and the test results are shown in Figure 14.

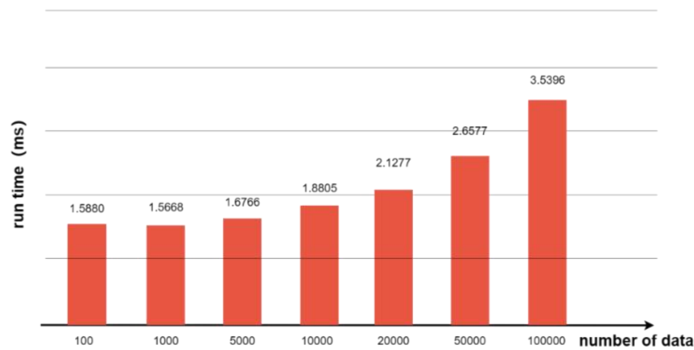

Figure 14 Comparison of GPU and CPU runtime for different data sets of 1024 bits

Next, we set the number of test data sets to 1000 and the random number bit widths to 256, 512, 1024, 2048, 4096 bits to compare the runtime of GPU and CPU, and the test results are shown in Figure 15.

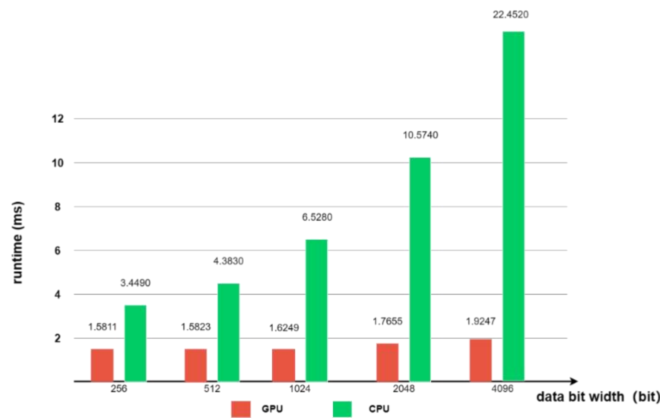

Figure 15 1000 groups of different bit width GPU and CPU runtime comparison

In addition, this project continues to verify the correctness of the GPU computing results. The evaluation program will compare the GPU computing results with the CPU computing results, and the correctness information will be output once every 500 sets of test data, and the correctness display is shown in Figure 16.

```
Genereating instances ...
Copying instances to the GPU ...
BITS: 4096
Number of cases: 5000
Running GPU kernel ...
Time for the Concurrent Barrett Algorithm:2.199040 ms
Copying results back to CPU ...
The 1st batch of 500 test points is accepted!
The 2nd batch of 500 test points is accepted!
The 3rd batch of 500 test points is accepted!
The 4th batch of 500 test points is accepted!
The 5th batch of 500 test points is accepted!
The 6th batch of 500 test points is accepted!
The 7th batch of 500 test points is accepted!
The 8th batch of 500 test points is accepted!
The 9th batch of 500 test points is accepted!
All results matched!
```

Figure 16 Correctness demonstration

Through the above data demonstration, it can be found that the GPU computing time remains low whether computing data of different bit widths or different groups of data. As the data volume and bit width gradually increase, the effect of algorithm optimization and parallelization deployment implemented in this project becomes more prominent, and the speedup compared to the GMP library on CPU becomes more and more significant. In addition, it can be seen that when computing 100000 sets of data, the computing time on GPU is only 1/327 of the computing time on GMP on CPU, which is more than 300 times faster than GMP on CPU, indicating that the algorithm optimization and parallelization deployment of this paper is very successful and achieves very good results in speeding up the encryption and decryption operations of RSA algorithm.

4.2 Performance Analysis

The basic process of starting a CUDA program for GPU parallel computation on a CPU host using a GPU device generally involves first allocating storage space for the program in the video memory (cudaMalloc), as shown in Figure 17, and then copying the data from the main memory to the video memory, with the video memory allocating threads to execute the core functions for computation. During this time, the CPU must wait for all threads in the GPU to finish executing before it can perform the subsequent computation (cudaDeviceSynchronize).

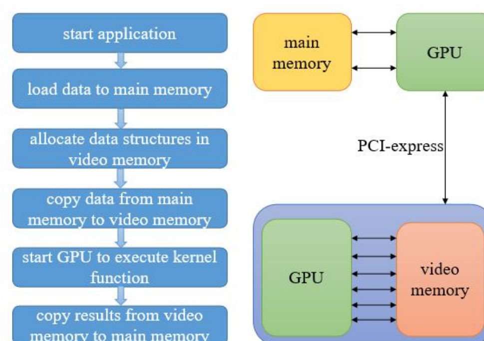

Figure 17 GPU data copy schematic

This project uses the nsys tool to analyze the performance of the program and the time spent on each CUDA API is as follows:

Table 2. Table of time overhead of nuclear functions

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
100.0	308723379	1	308728879.0	308723379	308723379	kenel_mod

Table 3. Time overhead of each CUDA API.

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
53.6	418178344	1	418178344.0	418178344	418178344	cudaMalloc
39.8	310375231	1	310375231.0	310375231	310375231	cudaDevice Synchronize
2.8	21678152	1	21678152.0	21678152	21678152	cudaMallocManaged
2.6	19880277	2	9940138.5	9230504	10649773	cudaMemcpy
1.1	8857986	1	8857986.0	8857986	8857986	cudaLaunchKernel
0.1	635183	2	317591.5	92221	542962	cudaFree

As you can see by the overhead schedule, the time to perform memory allocation (cudaMalloc) is even longer than the actual computation of the GPU core function, accounting for the main part of the total program run time. According to official NVIDIA documentation, the peak bandwidth between device memory and GPU (e.g., 2050 GB/s on the NVIDIA Tesla C144) is much higher than the peak bandwidth between host memory and device memory (2 GB/s on PCIe x8 Gen16). This difference means that data transfers between the host and GPU devices can be a bottleneck to overall application performance gains[7], and Figure 18 illustrates the performance constraints.

PERFORMANCE CONSTRAINTS

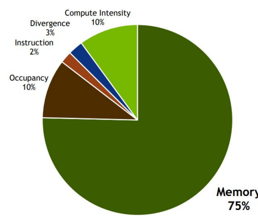

Figure 18. Schematic representation of performance constraints

4.3 Optimization direction

4.3.1 Optimizing program structure

According to the performance analysis, starting a GPU for parallelization is an operation with a huge time overhead, so the program must be properly structured to minimize the amount of data transfer between the host and the device, use fixed memory to increase the bandwidth between the host and the device, consolidate as many small data transfers as possible into one relatively large amount of data transfers, and perform as many GPU parallelization calculations as possible during one as many computations as possible during a single GPU parallelization[9].

In the program of this project, the ciphertexts to be encrypted are all transferred to the GPU at one time, and the GPU will perform multiple sets of RSA decryption and encryption operations at the same time. In real-life communication peaks and large communication traffic, it is often necessary to perform a large number of encryption and decryption tasks using the same set of public keys and keys for different plaintexts, and then the GPU parallelized computing capability can be fully utilized to perform the RSA encryption and decryption tasks.

4.3.2 Using fixed memory

Host memory is actually divided into Pageable Memory and Pinned Memory. Pageable Memory, i.e. paged memory, is memory space allocated through the OS API (malloc(), new()) and cudaMalloc will use this memory for allocation; while Pinned Memory, i.e. fixed memory, which always exists in physical memory and will not be allocated to low-speed virtual memory, can be accelerated by DMA to communicate with the device side, using APIs such as cudaHostAlloc(), cudaFreeHost(), etc. to allocate and release this block of memory.

The host (CPU) data allocation memory is pagable by default, and the GPU cannot access the pagable host memory directly[10], so when transferring data from the pagable memory to the device memory, the CUDA driver must first allocate a temporary non-pagable or fixed host array, then copy the host data to the fixed array, and finally transfer the data from the fixed array to the device memory[11]. As shown in the following figure:

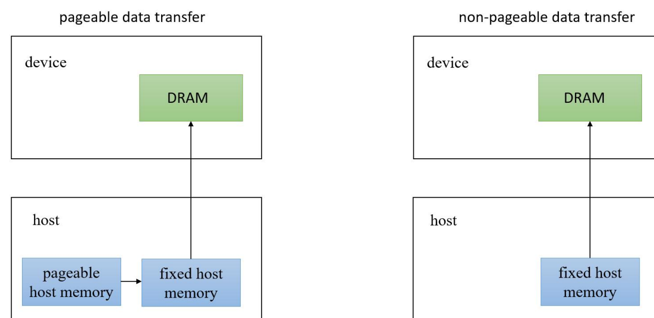

Figure 19 Flow chart of pagable and non-pagable data transfer

Therefore Pinned Memory can be used to improve communication efficiency, essentially forcing the system to do the memory request and release in physical memory, without participating in page swapping, thus improving memory access efficiency. Its can be used to prepare input data on CUDA as it may be faster when transferring across PCI-E (since there is no need to ask the CPU if the cache data is in there).

Advantages of using Pinned Memory: high bandwidth for data transfer from host side to device side; on some devices, it can be mapped to the device address space by zero-copy function and accessed directly from the GPU, eliminating the need for data copying between main memory and video memory.

However, there are also some limitations to using Pinned Memory. Pinned Memory cannot be allocated too much, otherwise it may lead to less physical memory for paging in the operating system, which may degrade the overall system performance. Therefore, how to apply cudaHostAlloc(), cudaFreeHost() to further improve the performance of program memory allocation is one of the subsequent optimization directions in this paper.

5. Conclusion

In recent years, GPU-related fields have developed rapidly, from personal computers to high-performance computer clusters, and GPUs have shown extraordinary acceleration strength12. To accelerate the processing of RSA algorithm, this paper studies the RSA algorithm structure and parallel optimization space, and proposes a parallelization strategy of Barrett algorithm based on GPU to achieve a significant increase in the processing speed of RSA algorithm.

This paper firstly introduces the importance of RSA algorithm and where the bottlenecks are. Second, this paper introduces the process of RSA algorithm, the related theories Montgomery algorithm and Barrett algorithm used in parallel optimization, as well as GPU and CUDA.Then, this paper shows the optimization of the encryption and decryption operations in RSA based on Montgomery algorithm and Barrett reduction algorithm, and the process of parallelizing the optimized algorithm and deploying it to GPUs. Then, by comparing the parallelization strategy running on GPU with the well-known large number library GMP running on CPU, this paper demonstrates the optimization acceleration effect as well as the correctness verification. When the RSA encryption and decryption data volume is large, the parallel optimization scheme of this paper can achieve more than three hundred times speedup, and the optimization effect is remarkable. In addition, this paper analyzes the performance of the program and suggests that further improvements can be made in the program structure and memory usage to obtain better optimization results.

High-performance computing has become a strategic technological high point for countries around the world in the information age13, and it is believed that in the future, not only in the field of cryptography, but also in the fields of energy, education, and medical care, high-performance computing will be a research direction with great potential for development and become an important force to promote the transformation and upgrading of traditional industries[14-15]. The design scheme of this paper successfully deploys the parallelization of Barrett's algorithm to GPU and realizes the parallelized computation of RSA algorithm encryption and decryption process with fast encryption and decryption speed, which achieves higher acceleration ratio compared with the GMP library on traditional CPU. The parallelization scheme in this paper has high practical value in the field of information security, and is also useful for other GPU-related research.

REFERENCES

[1] Ma Yongxin, Zeng Xiaoyang, Wu Min, and Sun Chengshou, "Design of RSA cryptographic coprocessor based on Barrett's modulo multiplication algorithm", Systems Engineering and Electronics Technology, 830-833(2006).

[2] Chen Chuanbo, Zhu Zhongtao, "RSA algorithm application and implementation details", Computer Engineering and 13-14(2006).

[3] Ma Changsha, Hu Aiqun, "A study on improving the speed of RSA algorithm", Information Security and Communication Security,80-82(2010).

[4] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang and Berk Sunar, "Accelerating fully homomorphic encryption using GPU", 2012 IEEE Conference on High Performance Extreme Computing,Waltham,MA,USA,1-5(2012)

[5] Dániel Nagy, Lambert Plavecz, Ferenc Hegedűs, "The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs",Communications in Nonlinear Science and Numerical Simulation,Volume 112,2022,106521,ISSN 1007-5704

[6] Rivest R. L., Shamir A., and Adleman L., "A method for obtaining digital signatures and public-key cryptosystems". commun. Acm, 120-126(1978).

[7] Zhao Lili, Zhang Shengbing, Zhang Meng, Yao Tao, "CUDA-based high-speed FFT computation", Computer Application Research, 1556-1559(2011)

[8] "GPU:Changes Everyting", http://www.nvidia.com/object/gpu.html
[9] Mark Harris, "How to Optimize Data Transfers in CUDA C/C++",Nvida Developer,(2012)
[10] Yao Ping, "CPU/GPU Asynchronous Computing Model on CUDA Platform", University of Science and Technology of China, (2010)
[11] Azadeh M., Ahmad M A., "SSVD: Structural SVD-based image quality assessment", Signal Processing：Image Communication，54-63(2019)
[12] Qiong Chang, Xiang Li, Yun Li, Jun Miyazaki, "Multi-directional Sobel operator kernel on GPUs",Journal of Parallel and Distributed Computing,160-170(2023)
[13] Liao, Xiangke, Xiao Nong, "New high-performance computing systems and technologies", China Science:Information Science, 1175-1210(2016)
[14] Daniel Mira, Eduardo J. Pérez-Sánchez, Ricard Borrell, Guillaume Houzeaux,
"HPC-enabling technologies for high-fidelity combustion simulations",Proceedings of the Combustion Institute,2022,ISSN 1540-7489,
[15] Zhang Shu, Chu Yanli, "GPU high-performance computing with CUDA". China Water Resources and Hydropower Press, (2009).