Exploiting the Floating-Point Computing Power of GPUs for RSA

Fangyu Zheng[1,2,3,⋆], Wuqiong Pan[1,2,⋆⋆], Jingqiang Lin[1,2], Jiwu Jing[1,2], and Yuan Zhao[1,2,3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, China
[2] Data Assurance and Communication Security Research Center, CAS, China
[3] University of Chinese Academy of Sciences, China
{fyzheng,wqpan,linjq,jing,zhaoyuan12}@is.ac.cn

Abstract. Asymmetric cryptographic algorithms (e.g., RSA and ECC) have been implemented on Graphics Processing Units (GPUs) for several years. These implementations mainly exploit the highly parallel GPU architecture and port the integer-based algorithms for common CPUs to GPUs, offering high performance. However, the great potential cryptographic computing power of GPUs, especially by the more powerful floating-point instructions, has not been comprehensively investigated in fact. In this paper, we try to fully exploit the floating-point computing power of GPUs for RSA, by various designs, including the floating-point-based Montgomery multiplication algorithm, the optimization for the fundamental operations and the utilization of the latest thread data sharing instruction `shuffle`. The experimental result on NVIDIA GTX Titan of 2048-bit RSA decryption reaches a throughput of 38,975 operations per second, achieves 2.21 times performance of the existing fastest integer-based work and outperforms the previous floating-point-based implementation by a large margin.

Keywords: GPU, CUDA, Floating-Point, Montgomery Multiplication, RSA.

1 Introduction

With the rapid development of e-commerce and cloud computing, the high-density calculation of digital signature algorithms such as the ECC (Elliptic Curve Cryptography) [15, 19] and RSA [30] algorithms is urgently required. However, without significant development in recent years, CPUs are more and more difficult to keep pace with this rapidly expanding demand. Specialized for compute-intensive and high-parallel computation required by graphics rendering,

⋆ This work was partially supported by the National 973 Program of China under award No. 2014CB340603 and the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702.

⋆⋆ Corresponding author.

S.S.M. Chow et al. (Eds.): ISC 2014, LNCS 8783, pp. 198–215, 2014.

GPUs own much more powerful arithmetic capability than CPUs by devoting more transistors to arithmetic processing unit rather than data caching and flow control. With the advent of NVIDIA Compute Unified Device Architecture (CUDA) technology, it is now possible to perform general-purpose computation on GPUs. Due to their powerful arithmetic capability and moderate cost, GPUs are excellent candidates to perform cryptographic acceleration.

Born for high-definition 3D graphics, GPUs require high-speed floating-point processing capability. Therefore, the GPU vendors focus on the development of the floating-point computing power. In NVIDIA GPUs, the floating-point computing power develops rapidly, from 1,300 GFLOPS (Giga Floating-Point Operations Per Second) in 2010 to 5,000 GFLOPS in 2014 [25]. By contrast, the integer multiplication processing power makes slighter progress, achieving only 25% [25, 33] growth.

However, the floating-point instruction set is inconvenient and complicated to realize larger integer modular multiplication which is the core operation of asymmetric cryptography. Furthermore, the floating-point instruction set in the previous GPUs shows no significant performance advantage over the integer one. As far as we know, Bernstein et al. [4] is the first and the only one to utilize the floating-point processing power of CUDA GPUs for asymmetric cryptography. However, compared with their later work [3] based on the integer instruction set, the floating-point-based one only achieves 1/6 performance. Nevertheless, with the rapid development of GPU floating-point processing power, fully utilizing the floating-point processing resource is a great benefit to asymmetric cryptography implementation in GPUs.

Based on the above observations, in this paper, we propose a new approach to implement high-performance RSA by fully exploiting the floating-point processing power in CUDA GPUs. In particular, we propose a floating-point-based large integer representation and modify the Montgomery multiplication algorithm according to it. Also, we flexibly employ the integer instruction set to supplement the deficiency of the floating-point computing. Besides, we are the first to fully utilize the latest instruction `shuffle` to share data between threads, which makes the core algorithm Montgomery multiplication a non-memory-access design, decreasing greatly the performance loss in thread communication. With these improvements, the performance of our RSA implementation increases dramatically compared with the previous works. In NVIDIA GeForce GTX Titan, our resulting 2048-bit RSA implementation reaches 38,975 operations per second, which achieves 2.21 times performance of the existing fastest work and performs 13 times faster than the previous floating-point-based implementation.

The rest of our paper is organized as follows. Section 2 introduces the related work. Section 3 presents the overview of GPU, CUDA, floating-point elementary knowledge, RSA and Montgomery multiplication. Section 4 describes our proposed floating-point-based Montgomery multiplication algorithm in detail. Section 5 shows how to implement RSA decryption in GPUs using our proposed Montgomery multiplication. Section 6 analyses performance of proposed algorithm and compares it with previous work. Section 7 concludes the paper.

2 Related Work

Many previous papers report performance results to demonstrate that the GPU architecture can already be used as an asymmetric cryptography workhorse. Large integer modular multiplication is the heart of asymmetric cryptography. Directed at ECC implementation, [1–3, 5, 18, 28, 32] used various methods to implement modular multiplication. Antão et al. [1, 2] and Pu et al. [28] employed Residue Number System (RNS) to parallelize the modular multiplication into several threads. Bernstein et al. [3] and Leboeuf et al. [18] used one thread to handle a modular multiplication with Montgomery multiplication. Bos et al. [5] and Szerwinski et al. [32] used fast reduction to implement modular multiplication over the Mersenne prime fields [31].

Unlike ECC scheme, RSA calculation requires longer and unfixed modulus and depends on modular exponentiation. Many previous work [9–12, 21, 22, 32], made effort to its implementation. Before NVIDIA CUDA was proposed, Moss et al. [21] mapped RNS arithmetic to the GPU to implement a 1024-bit modular exponentiation. Later in CUDA GPUs, Szerwinski et al. [32] and Harrison et al. [9] developed efficient modular exponentiation by both Montgomery multiplication Coarsely Integrated Operand Scanning (CIOS) method and RNS arithmetic. Jang et al. [11] presented a high-performance SSL acceleration using CUDA GPUs. They parallelized Separated Operand Scanning (SOS) method [17] of the Montgomery multiplication by single limb. Jeffrey et al. [12] used the similar technology to implement 256-bit, 1024-bit and 2048-bit Montgomery multiplication. Neves et al. [22] and Henry et al. [10] used one thread to perform single Montgomery multiplication to economize overhead of thread synchronization and communication, however, their implementations resulted in a high latency.

Note that all above works are based on the CUDA integer computing power. Bernstein et al. is the pioneer to utilize CUDA floating-pointing processing power in asymmetric cryptography implementation [4]. They used 28 SPFs (Single Precision Floating-point) to represent 280-bit integer and implemented the field arithmetic. However, the result was barely satisfactory. Their later work [3] in the same platform GTX 295 using integer instruction set performs almost 6.5 times throughput of [4].

3 Background

In this section, we provide a brief introduction to the basic architecture of modern GPUs, the floating-point arithmetic, the basic knowledge of RSA and the Montgomery multiplication.

3.1 GPU and CUDA

CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of GPU. It is created by NVIDIA and implemented by the GPUs that they produce [25].

Our target platform GTX Titan is a CUDA-compatible GPUs with codename GK-110 [24], which contains 14 streaming multiprocessors (SM). 32 threads (grouped as a *warp*) within one SM can concurrently run in a clock. Following the SIMT (Single Instruction Multiple Threads) architecture, each GPU thread runs one instance of the kernel function. A warp may be preempted when it is stalled due to memory access delay, and the scheduler may switch the runtime context to another available warp. Multiple warps of threads are usually assigned to one SM for better utilization of the pipeline of each SM. These warps are called one *block*. Each SM could access 64KB fast *shared memory*/L1 cache and 64K 32-bit registers. All SMs share 6GB 256-bit wide slow *global memory*, cached read-only *texture memory* and cached read-only *constant memory* [24, 25]. Each SM of GK-110 owns 192 single precision CUDA cores, 64 double precision units, 32 special function units (SFU) and 32 load/store units [24], yielding a throughput of 192 SPF arithmetic, 64 DPF arithmetic, 160 32-bit integer add and 32 32-bit integer multiplication instructions per clock circle [24, 25].

GK-110 also brings a brand new method of data sharing between threads. Previously, shared data between threads requires separated store and load operations to pass data through *shared memory*. First introduced in NVIDIA Kepler architecture, `shuffle` instruction [24, 25] allows threads within a wrap to share data. With shuffle instruction, threads within a wrap can read any value of other thread's in any imaginable permutations [24]. NVIDIA conducted various experiments [26] on the comparison between `shuffle` instruction and *shared memory*, which shows that `shuffle` instruction always gives a better performance than *shared memory*.

3.2 Floating-Point and Integer Arithmetic in GPU

Floating-point arithmetic in CUDA GPUs complies with 754-2008 IEEE Standard for Floating-Point Arithmetic [6]. Among the basic formats which the standard defines, 32-bit binary (single precision floating-point, SPF) and 64-bit binary (double precision floating-point, DPF) are supported in CUDA GPUs.

Table 1. SPF and DPF Basic Formats

	sign	*exp*	biase	*mantissa*
SPF	1 bit	8 bits	0x7F	23 bits
DPF	1 bit	11 bits	0x3FF	52 bits

As demonstrated in Table 1, the real value assumed by a given SPF or DPF data with a sign bit *sign*, a given biased exponent *exp* and a significand precision *mantissa* is $(-1)^{sign} \times 2^{exp-biase} \times 1.mantissa$. Therefore, a SPF and a DPF can respectively represent 24-bit and 53-bit integer.

When executing floating-point multiplication, to avoid precision loss, each SPF or DPF must contain lower than or equal to $\lfloor \frac{24}{2} \rfloor = 12$ or $\lfloor \frac{53}{2} \rfloor = 26$ significant bits. And in NVIDIA Kepler architecture, SPF offers 3 times the multiplication instruction throughput of DPF. Taking all above factors into consideration,

SPF can only provide $(\frac{12}{26})^2 \times 3 = 63.9\%$ multiplication performance of DPF. Therefore, in this paper, we focus on the DPF-based RSA implementation.

3.3 RSA and Montgomery Multiplication

RSA [30] is an asymmetric cryptography algorithm widely used for digital signature, whose core operation is modular exponentiation. In practice, CRT (Chinese Remainder Theorem) [29] technology is widely used to promote the RSA decryption. Instead of calculating k-bit modular exponentiations directly, we can perform two $k/2$-bit modular exponentiations (Equation (1a) & (1b)) and the Mixed-Radix Conversion (MRC) algorithm [16] (Equation (2)) to conduct the RSA decryption.

$$Plain_1 = Cipher^{d \mod (p-1)} \mod p \tag{1a}$$

$$Plain_2 = Cipher^{d \mod (q-1)} \mod q \tag{1b}$$

$$Plain = Plain_2 + [(Plain_1 - Plain_2) \times (q^{-1} \mod p) \mod p] \times q \tag{2}$$

where p and q are $k/2$-bit prime numbers chosen in private key generation ($M = p \times q$). All parameters, p, q, ($d \mod p-1$), ($d \mod q-1$) and ($q^{-1} \mod p$) are part of the RSA private key [13]. Compared with calculating k-bit modular exponentiations directly, the CRT technology gives 3 times performance promotion [29].

Algorithm 1. High-radix Montgomery Multiplication According to [27]

Input:

$M > 2$ with gcd(M,2)=1, , positive integers n, w such that $2^{wn} > 4M$
$M' = -M^{-1} \mod 2^w$, $R^{-1} = (2^{wn})^{-1} \mod M$
Integer multiplicand A, where $0 \leq A < 2M$
Integer multiplier B, where $B = \sum_{i=0}^{n-1} b_i 2^{wi}$, $0 \leq b_i < 2^w$ and $0 \leq B < 2M$

Output:

An integer S such that $S = ABR^{-1} \mod 2M$ and $0 \leq S < 2M$;

1: $S = 0$
2: **for** $i = 0$ *to* $n-1$ **do**
3: $\quad S = S + A \times b_i$
4: $\quad q_i = ((S \mod 2^w) \times M') \mod 2^w$
5: $\quad S = (S + M \times q_i)/2^w$
6: **end for**

In RSA, modular multiplication is the bottleneck of its overall performance. In 1985 Peter L. Montgomery proposed an algorithm [20] to remove the costly division operation from the modular reduction. Let $\bar{A} = AR(mod\ M), \bar{B} = BR(mod\ M)$ be the Montgomeritized form of A, B modulo M, where R and M are coprime and $M \leq R$. Montgomery multiplication defines the multiplication between 2 Montgomeritized form numbers, $MonMul(\bar{A}, \bar{B}) = \bar{A}\bar{B}R^{-1}$

(mod M). 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 2, which lead to a fast division by R.

In 1995, Orup et al. [27] economized the determination of quotients by loosening the restriction for input and output from $[0, M)$ to $[0, 2M)$. Algorithm 1 shows the detailed steps.

4 DPF-Based Montgomery Multiplication

In this section, we propose a DPF-based Montgomery multiplication parallel algorithm in CUDA GPUs. The algorithm includes the large integer representation, the fundamental operations and the parallelism method of Montgomery multiplication.

4.1 Advantage and Challenges of DPF-Based RSA

Large integer modular multiplication is the crucial part of asymmetric cryptographic algorithm and it largely depends on the single-precision multiplication (or multiply-add) especially when using Montgomery multiplication. In GTX Titan, DPF multiplication (or multiply-add) instruction provides double throughput (64/Clock/SM) of the 32-bit integer multiplication instruction (32/Clock/SM) [25]. In CUDA, the product of 32-bit multiplication requires 2 integer multiplication instructions, one for the lower half, the other for the upper half. Thus, DPF can perform about $\frac{64}{32/2} \times (\frac{26}{32})^2 = 2.64$ (DPF supports at most 26-bit multiplication which is discussed in Section 3.2) times multiplication processing power of 32-bit integer in GTX Titan.

However, DPF encounters many problems when used in asymmetric cryptographic algorithm which largely depends on large integer multiplication.

- **Round-off Problem:** Unlike integer, the floating-point add or multiply-add instruction does not support carry flag (CF) bit. When the result of the add instruction is beyond the limit bits of significand (24 for SPF, 53 for DPF), the round-off operation will happen, in which the least significant bits will be left out to keep the significand within the limitation.
- **Inefficient Bitwise Operations:** Floating-point arithmetic does not support bitwise operations which are frequently used in our algorithm. CUDA Math API does support the `__fmod` function [23], which can be employed to extract the least and most significant bits. But it consumes tens of instructions which is extremely inefficient, while using integer native instructions set, the bitwise operation needs only one instruction.
- **Inefficient Add Instruction:** Unlike multiplication instruction, the DPF add instruction is slower than the integer add instruction. In a single clock circle, each SM can execute respectively 64 DPF and 160 integer add instructions [25]. Even taking the instruction word length (53 bits for DPF, 32 bits for integer) into consideration, DPF add instruction only performs nearly

2/3 processing power of the integer one. In our implementation, we use at most 30-bit add operation, which intensifies this performance disadvantage further.

– **Extra Memory and Register File Cost:** A DPF occupies 64-bit memory space, however, only 26 or lower bits are used. In this way, we have to pay $\frac{64-26}{26} = 138\%$ times extra cost in memory access and utilization of register files. While, in integer-based implementation, this issue is not concerned since every bit of a integer is utilized.

4.2 DPF-Based Representation of Large Integer

In Montgomery multiplication, multiply-accumulation operation $s = a \times b + s$ is frequently used. In CUDA, fused multiply-add (FMA) instruction is provided to perform floating-point multiply-add operation, which can be executed in one step with a single rounding. Note that, when each DPF (a and b) contains w ($w \leq 26$) significant bits, we can execute $2^{53}/2^{2w} = 2^{53-2w}$ times of $s = a \times b + s$ (the initial value of s is zero), free from the round-off problem. For example, if we configure 26 significant bits per DPF, after $2^{53-52} = 2$ times of $s = a \times b + s$ operations, s is 53 bits long. In order to continue executing $s = a \times b + s$, we have to restrict s within 52 bits over and over again. This operations is very expensive, due to the complexity of bit extraction. Therefore, we should carefully configure it to get rid of the expensive bit extraction. In Algorithm 1, there are n loops and each loop contains 2 FMA operations for each limb, where $n = \lceil \frac{1024+2}{w} \rceil$. Thus $2 \times \lceil \frac{1024+2}{w} \rceil$ times of FMA operations are needed. Table 2 shows the number of FMAs supported and needed with varying significant bits in each DPF.

Table 2. Choose the Best w for Proposed 1024-bit Montgomery Multiplication

Bits/DPF w	FMAs Supported 2^{53-2w}	FMAs Needed in Algorithm 1 $2 \times \lceil \frac{1024+2}{w} \rceil$	Radix $R = w \times \lceil \frac{1024+2}{w} \rceil$
26	2	80	1040
25	8	84	1050
24	32	86	1032
23	**128**	**90**	**1035**
22	512	94	1034
21	2048	98	1029
20	8192	104	1040

From Table 2, we find that the supported number of FMAs starts to be larger than which is needed, when w is 23. The lower w means we need more instructions to process the whole algorithm. To achieve the best performance, we choose $w = 23$ and the radix $R = 2^{1035}$.

In this paper, we propose 2 kinds of DPF-based large integer representations, Simplified Format and Redundant Format.

Simplified format formats like $A = \sum_{i=0}^{44} 2^{23i} a[i]$, where each limb $a[i]$ contains at most 23 significant bits. It is applied to represent the input of the DPF FMA instruction.

Redundant format formats like $A = \sum_{i=0}^{44} 2^{23i} a[i]$, where each limb $a[i]$ contains at most 53 significant bits. It is applied to accommodate the output of the DPF FMA instruction.

4.3 Fundamental Operation and Corresponding Optimization

In DPF-based Montgomery multiplication, fundamental operations include multiplication, multiply-add, addition and bit extraction.

- **Multiplication:** In our implementation, native multiplication instruction (`mul.f64`) is used to perform multiplication. We require that both multiplicand and multiplier are in Simplified format to avoid round-off problem.
- **Multiply-Add:** In CUDA GPUs, fused multiply-add (`fma.f64`) instruction is provided to perform floating-point $s = a \times b + c$, which can be executed in one step with a single rounding. In our implementation, when using FMA instruction, we require that multiplicand a and multiplier b are both in Simplified format and addend c is in Redundant format but less than $2^{53} - 2^{46}$.
- **Bit Extraction:** In our implementation, we need to extract the most or least significant bits from a DPF. However, as introduced in Section 4.1, the bitwise operation for DPF is inefficient. To promote the performance, we try 2 improvements. The first one is introduced in [8]. Using round-to-zero, we can perform

$$\begin{aligned} x &= a + 2^{52+53-r} \\ u &= x - 2^{52+53-r} \\ v &= a - u \end{aligned}$$

  to extract the most significant r bits u and the least significant $(53 - r)$ bits v from a DPF a. Note that, in $x = a + 2^{52+53-r}$, the least significant $(53 - r)$ bits will be leaved out due to the round-off operation. The second one is converting DPF to integer then using CUDA native 32-bit integer bitwise instruction to handle bit extraction. Through our experiment, we find that the second method always gives a better performance. Therefore, in most of cases, we use the second method to handle bit extraction. There is only one exception when the DPF a is divisible by a 2^r, we can use $a/2^r$ to extract the most significant $53 - r$ bits, which can be executed very fast.
- **Addition:** CUDA GPUs provide the native add instruction to perform addition between two DPFs. But it has two shortcomings: 1. its throughput is low, 64/CLOCK/SM, while, the integer add instruction is 160/CLOCK/SM; 2. it does not provide support for carry flag. Thus, we convert DPF to integer and use CUDA native integer add instruction to handle addition.

4.4 DPF-Based Montgomery Multiplication Algorithm

According to Algorithm 1, in our Montgomery multiplication, $S = ABR^{-1} \pmod{M}$, A, B, M and S are all (1024+1)-bit integer (in fact, M is 1024 bits long, we also represent it as a 1025-bit integer for a common format). As choosing $w = 23$, we need $\lceil \frac{1025}{23} \rceil = 45$ DPF limbs to represent A, B, M and S.

In the previous work [11, 12], Montgomery multiplication is parallelized by single limb, that is, each thread deals with one limb (32-bit or 64-bit integer). The one-limb parallelism causes large cost in the thread synchronization and communication, which decreases greatly the overall throughput. To maximize the throughput, Neves et al. [22] performed one entire Montgomery multiplication in one thread to economize overhead of thread synchronization and communication, however, it results in a high latency, about 150ms for 1024-bit RSA, which is about 40 times of [11].

To make a tradeoff between throughput and latency, in our implementation, we try multiple-limb parallelism, namely, using r threads to compute one Montgomery multiplication and each thread dealing with t limbs, where $t = \lceil \frac{45}{r} \rceil$. The degree of parallelism r can be flexibly configured to offer the maximal throughput with acceptable latency. Additionally, we restrict threads of one Montgomery multiplication within a wrap, in which threads are naturally synchronized free from the overhead of thread synchronization and `shuffle` instruction can be used to share data between threads. To fully occupy thread resource, we choose r, where $r \mid 32$.

In our Montgomery multiplication $S = ABR^{-1}(\mod M)$, the input A, B, M are in Simplified format and the initial value of S is 0. We use 2 phases to handle one Montgomery multiplication, *Computing Phase* and *Converting Phase*. In Computing Phase, we use Algorithm 1 to calculate S, whose result is represented in Redundant format. In Converting Phase, we convert S from Redundant format to Simplified format.

Computing Phase. Algorithm 2 is a t-limb parallelized version of Algorithm 1. Using it, we can complete the Computing Phase.

We divide both the input A, B, M and the output S into $r = \lceil \frac{45}{t} \rceil$ groups and distribute the groups into each thread. For Thread k, the variables with index $tk \sim tk + t - 1$ are processed in it. Note that if the index of certain variable is larger than 44, we pad the variable with zero. In this section, the lowercase variables (index varies from 0 to $t - 1$) indicate the private registers stored in the thread and the uppercase ones (index varies from 0 to $tr - 1$) represent the global variables. For example, $a[j]$ in Thread k represents the $A[tk + j]$. And $v_1 = \texttt{shuffle}(v_2, k)$ means that current thread obtains variable v_2 of Thread k and stores it into variable v_1.

By reference to Algorithm 1, we introduce our parallel Montgomery multiplication algorithm step by step.

(1) $S = S + A \times b_i$: In this step, each Thread k respectively calculates $S[tk : tk+t-1] = S[tk : tk+t-1] + A[tk : tk+t-1] \times B[i]$. In our design, each thread stores a group of B. Thus firstly we need to broadcast the corresponding $B[i]$ from

Algorithm 2. DPF-based parallel 1024-bit Montgomery Multiplication ($S = ABR^{-1} \mod M$) Algorithm: Computing Phase

Input:

Number of processed limbs per thread t ;
Number of threads per Montgomery multiplication r, where $r = \lceil \frac{45}{t} \rceil$;
Thread ID k, where $0 \leq k \leq r-1$;
Radix $R = 23 \times 45$, Modulus M, $M' = -M^{-1} \mod 2^{23}$;
Multiplicand A, where $A = \sum_{i=0}^{rt-1} A[i]2^{23i}$, $0 \leq A[i] < 2^{23}$ and $0 \leq A < 2M$;
Multiplier B, where $B = \sum_{i=0}^{rt-1} B[i]2^{23i}$, $0 \leq B[i] < 2^{23}$ and $0 \leq B < 2M$;
a, b and m consist of t DPF limb, respectively representing $A[tk : tk+t-1]$, $B[tk : tk+t-1]$ and $M[tk : tk+t-1]$

Output:

Redundant-formatted sub-result $s[0 : t-1] = S[tk : tk+t-1]$

1: $S = 0$
2: **for** $i = 0$ *to* 44 **do**
3: $b_i = \texttt{shuffle}(b[i \mod t], \lfloor i/t \rfloor)$
/* *Step (1):* $S = S + A \times b_i$ */
4: **for** $j = 0$ to $t-1$ **do**
5: $s[j] = s[j] + a[j] \times b_i$
6: **end for**
/* *Step (2):* $q_i = ((S \mod 2^w) \times M')(\mod 2^w)$ */
7: **if** $k = 0$ **then**
8: $temp = s[0] \mod 2^{23}$
9: $q_i = temp \times M'$
10: $q_i = q_i \mod 2^{23}$
11: **end if**
12: $q_i = \texttt{shuffle}(q_i, 0)$
/* *Step (3):* $S = S + M \times q_i$ */
13: **for** $j = 0$ to $t-1$ **do**
14: $s[j] = s[j] + m[j] \times q_i$
15: **end for**
/* *Step (4):* $S = S/2^w$ */
16: **if** $k \neq r-1$ **then**
17: $temp = \texttt{shuffle}(s[0], k+1)$
18: **else**
19: $temp = 0$
20: **end if**
21: **if** $k = 0$ **then**
22: $s[0] = s[0] >> 23 + s[1]$
23: **else**
24: $s[0] = s[1]$
25: **end if**
26: **for** $j = 1$ to $t-2$ **do**
27: $s[j] = s[j+1]$
28: **end for**
29: $s[t-1] = temp$
30: **end for**

certain thread to all threads. In the view of single thread, $B[i]$ corresponds to $b[i \bmod t]$ of Thread $\lfloor i/t \rfloor$. We use `shuffle` to conduct this broadcast operation. Then each thread execute $s[j] = s[j] + a[j] \times b_i$ where $j \in [0, t-1]$.

(2) $q_i = ((S \mod 2^w) \times M')(\mod 2^w)$: $S \mod 2^w$ is only related to $S[0]$, which is stored in Thread 0 as $s[0]$. Therefore, in this step, we conduct this calculation only in Thread 0. Note that S is in Redundant format, we should firstly extract the least significant 23 bits *temp* from $S[0]$ then execute $q_i = temp \times M'$. And in next step, q_i will act as a multiplier, hence, we should also extract the least significant 23 bits of q_i.

(3) $S = S + M \times q_i$: In this step, each Thread k respectively calculates $S[tk : tk+t-1] = S[tk : tk+t-1] + M[tk : tk+t-1] \times q_i$. Because q_i is stored only in Thread 0, firstly we should broadcast it to all threads. Similar with Step (1), then each thread executes $s[j] = s[j] + m[j] \times q_i$ where where $j \in [0, t-1]$.

(4) $S = S/2^w$: In this step, each thread conducts a division by 2^w by shifting right operation. In the view of the overall structure, shifting right can done by executing $S[k] = S[k+1]$ $(0 \le k \le rt-2)$ and padding $S[rt-1]$ with zero. In the view of single thread, there are 2 noteworthy points. The first point is that Thread k needs to execute $S[tk+t-1] = S[t(k+1)]$ but $S[t(k+1)]$ is stored in Thread $(k+1)$, thus Thread $(k+1)$ needs to propagate its stored $S[t(k+1)]$ to Thread j. Note that $S[t(k+1)]$ is corresponding to $s[0]$ of Thread $k+1$ and to avoid to override variable Thread k uses *temp* to store it. The second point is that we represent S in Redundant format, when executing $S = S/2^w$ $(w = 23)$, the upper-30-bit of the least significant limb $S[0]$ needs to be stored. Thus, in Thread 0, we use $s[0] = s[0] >> 23 + s[1]$ instead of $s[0] = s[1]$ in other threads. $S[0]/2^{23}$ is at most (53-23)=30 bits long, and due to the right shift operation, the $S[0]$ in each loop is not the same. Thus $S[0]$ can be only accumulated within $45 \times 2 \times 2^{23} \times 2^{23} + 2^{30} < 2^{53}$, which does not cause round-off problem. Note that $S[0]$ is divisible by 2^{23}, as introduced in Section 4.3, we use $S[0]/2^{23}$ to extract the most significant 30 bits.

After Computing Phase, S is in Redundant format. Next, we use Converting Phase to convert S into Simplified format.

Converting Phase. In Converting Phase, we convert S from Redundant format to Simplified format: every $S[k]$ adds the *carry* ($S[0]$ does not execute this addition) and holds the least significant 23 bits of the sum and propagates the most significant 30 bits as new *carry* to $S[k+1]$. However, this method is serial, the calculation of every $S[k]$ depends on the *carry* that $S[k-1]$ produces, which does not comply with the architecture of GPU. In practice, we use parallelized method to accomplish Converting Phase, which is shown in Algorithm 3.

Algorithm 3 uses symbol $split(c) = (h, l) = (\lfloor \frac{c}{2^{23}} \rfloor, c \mod 2^{23})$ to denote that we divide 53-bit number c into 30 most significant bits $h$and 23 least significant bits l. Firstly, all threads execute a chain addition for its $s[0] \sim s[t-1]$ and store the *carry* that the last additions produces. Then every Thread $k-1$ (except Thread $(r-1)$) propagates the stored *carry* to Thread k using `shuffle` instruction, then repeats chain addition with the propagated *carry*. This step continues until *carry* of every thread is zero. We use the CUDA `__any()` voting

Algorithm 3. DPF-based parallel Montgomery Multiplication ($S = ABR^{-1} \bmod M$) Algorithm: Converting Phase

Input:
Thread ID k
Number of processed limbs per thread t ;
Number of threads per Montgomery multiplication r, where $r = \lceil \frac{45}{t} \rceil$;
Redundant-formatted sub-result $s[0 : t-1] = S[tk : tk + t - 1]$
Output:
Simplified-formatted sub-result $s[0 : t-1] = S[tk : tk + t - 1]$

```
1: carry = 0
2: for j = 0 to t − 1 do
3:    s[j] = s[j] + carry
4:    (carry, s[j]) = split(s[j])
5: end for
6: while carry of any thread is non-zero do
7:    if k = 0 then
8:       carry = 0
9:    else
10:      carry = shuffle(carry, k − 1)
11:   end if
12:   for j = 0 to t − 1 do
13:      s[j] = s[j] + carry
14:      (carry, s[j]) = split(s[j])
15:   end for
16: end while
```

instruction to check *carry* of each thread. The number of the iterations is $(r-1)$ in the worst case, but for most cases it takes one or two. Compared with the serialism method, we can save over 75% execution time in Converting Phase using the parallelism method.

After Converting Phase, S is in Simplified format. An entire Montgomery multiplication is completed.

5 RSA Implementation

This section introduces how to implement Montgomery exponentiation using our proposed Montgomery multiplication. We also discuss the CRT computation.

5.1 Montgomery Exponentiation

Using CRT, We need to perform 2 1024-bit Montgomery exponentiation $S = X^Y R \ (mod\ M)$ to accomplish 2048-bit RSA decryption. In Montgomery exponentiation, we represent the exponent Y in integer. Similar with our processing of shared data B in Montgomery multiplication $S = ABR^{-1}$, each thread stores a piece of Y and uses `shuffle` to broadcast Y from certain thread to all threads.

With the binary square-and-multiply method, the expected number of modular multiplications is $3k/2$ for k-bit modular exponentiation. The number can be reduced with m-ary method given by [14] that scans multiple bits, instead of one bit of the exponent. We have implemented 2^6-ary method and reduced the number of modular multiplications from 1536 to 1259, achieving 17.9% improvement. Using 2^6-ary method, we need to store $(2^6 - 2)$ pre-processed results ($X^2R \sim X^{63}R$) into memory. The memory space that pre-processed results needed (about 512KB) is far more than the size of *shared memory* (at most 48KB), thus we have to store them into *global memory*. Global memory load and store consume hundreds of clock circles. To improve memory access efficiency, we first convert the Simplified formatted pre-processed results into 32-bit integer format then store the integer-formatted results in global memory. This optimization saves about 50% memory access latency.

5.2 CRT Computation

In our first implementation, GPU only took charge of the Montgomery exponentiation. And the CRT computation (Equation (2)) was offloaded to CPU (Intel E3-1230 v2) using GNU multiple precision (GMP) arithmetic library [7]. But we find the low efficiency of CPU computing power greatly limits the performance of the entire algorithm, which occupies about 15% of the execution time. Thus we integrate the CRT computation into GPU. For CRT computation, we additionally implement a modular subtraction and a multiply-add function. Both functions are parallelized in the threads which take charge of Montgomery exponentiation. The design results in that the CRT computation occupies only about 1% execution time and offers the independence of CPU computing capability. The detailed schematic diagram is shown in Fig. 1.

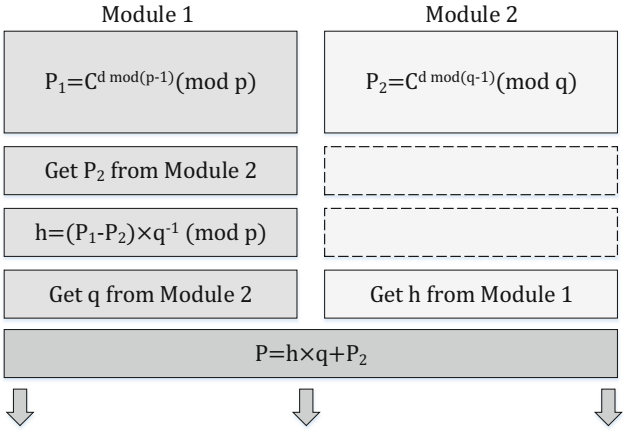

Fig. 1. Intra-GPU CRT implementation

6 Performance Evaluation

In this section we discuss the implementation performance and summarize the results for the proposed algorithm. Relative assessment is also presented by considering related implementation.

6.1 Implementation Result

Using the DPF-based Montgomery multiplication algorithm and RSA implementation respectively described in Section 4 and Section 5, we implement 2048-bit RSA decryption in NVIDIA GTX Titan.

Table 3 summarizes the performance of our DPF-based RSA implementation. *Thrds/RSA*=$t \times 2$ indicates we use t threads to process a Montgomery multiplication and $t \times 2$ threads to process one RSA decryption. Theoretically, *Thrds/RSA* can vary from 2×2 to 16×2, however, when using 2 threads to handle a DPF-based Montgomery multiplication, the number of registers per thread surpasses the GPU hardware limitation, 255 registers per thread, thus many variables need to be stored in memory, which decreases greatly the overall performance. Thus we restrict *Thrds/RSA* from 4×2 to 16×2.

Table 3. Performance of 2048-bit RSA Encryption

Thrds/RSA	Throughput (/s)	Latency (ms)	RSAs/launch	MonMul Throughput (10^6/s)
4×2	38,975	22.47	896	110.5
8×2	36,265	18.53	672	102.8
16×2	27,451	16.32	448	78.4

Varying the number of RSA decryptions per launch, we record the performance respectively for 3 configurations of *Thrds/RSA*. We also record its corresponding *Latency* and *RSAs/launch*, which represent respectively how much time single GPU calculation costs and how many RSA encryptions it contains. For comparison with the work which only implements modular multiplication, we also measure our performance of 1024-bit modular multiplication, which is demonstrated in the *MonMul Throughput* column.

From Table 3, we can find that the less *Thrds/RSA* gives higher throughput due to the decreased computational cost for data traffic between threads. When *Thrds/RSA* is 4×2, our implementation gives the highest throughput of 38,975 RSA decryptions per second.

6.2 Performance Comparison

Proposed vs. Floating-point-based Implementation. Bernstein et al. [4] employed the floating-point arithmetic to implement 280-bit Montgomery multiplication, thus, we scale its performance by floating-point processing power.

Table 4 shows performance of the work [4] and ours. Note that the work [4] implemented the 280-bit modular multiplication, thus we multiply its performance by $(\frac{280}{1024})^2$ as the performance of the 1024-bit one. "1024-bit MulMod (scaled)" indicates the performance scaled by the platform floating-point processing power.

Table 4. Performance Comparison of Bernstein et al. [4] and ours

	GTX 295 Bernstein et al. [4]	GTX Titan Proposed
Floating-point Processing Power (GFLOPS)	1788	4500
Scaling Factor	0.397	1
280-bit MulMod ($\times 10^6$)	41.88	-
1024-bit MulMod ($\times 10^6$)	3.13	110.5
1024-bit MulMod (scaled) ($\times 10^6$)	7.89	110.5

Our implementation achieves 13 times speedup than [4]. Part of our performance promotion results from the advantage DPF achieves over SPF as discussed in Section 3.2. The reason why they did not utilize DPF is that GTX 295 they used does not support DPF instructions. The second reason of our advantage comes from the process of Montgomery multiplication. Bernstein et al. used Separated Operand Scanning (SOS) Montgomery multiplication method which is known inefficient [17]. And they utilized only 28 threads of a wrap (consists of 32 threads), which wastes 1/8 processing power. The third reason is that we used the CUDA latest `shuffle` instruction to share data between threads, while [4] used *shared memory*. As Section 3.1 introduced, the `shuffle` instruction gives a better performance than shared memory. The last reason lies in that Bernstein et al. [4] used floating-point arithmetic to process all operations, some of which are more efficient using integer instruction such as bit extraction. By contrast, we flexibly utilize integer instructions to accomplish these operations.

Proposed vs. Integer-based Implementation. For fair comparison, we firstly evaluate the CUDA platform of each work, and scale their performance into our GPU hardware GTX Titan. [10–12, 22, 32] are all based on integer arithmetic. Thus we scale their CUDA platform performance based on the integer processing power. The parameters in Table 5 origin from [25] and [33], but the integer processing power is not given directly. Taking SM Number, processing power of each SM and shader clock into consideration, we calculate integer multiplication instruction throughput *Int Mul*. Among them, 8800GTS and GTX 260 support only 24-bit multiply instruction, while, the other platforms support 32-bit multiply instruction. Hence, we adjust their integer multiply processing capability by a correction parameter $(\frac{24}{32})^2$ (unadjusted data is in parenthesis). *Scaling Factor* is defined as *Int Mul* ratio between the corresponding CUDA platform and GTX Titan.

Table 5 summarizes the resulting performance of each work. We divide each resulting performance by the corresponding *Scaling Factor* listed in Table 5

as the scaled performance. Note that the RSA key length of Neves et al. [22] is 1024 bits, while ours is 2048 bits, we multiply it by an additional factor $1/4 \times 1/2 = 1/8$ (1/4 for the performance of modular multiplication, 1/2 for the half bits of the exponent). And Szerwinski et al. [32] accomplished non-CRT 1024-bit Montgomery exponentiation with 813/s throughput, thus we divide it by an additional factor 2 as the performance of the CRT 2048-bit RSA decryption.

Table 5. Throughput and Latency of Operations per second

	Szerwinski et al. [32]	Neves et al. [22]	Henry et al. [10]	Jang et al. [11]	Robinson et al. [12]	Ours
CUDA platform	8800GTS	GTX 260	M2050	GTX 580	GTX Titan	
SM Number	16	24	14	16	14	
Shader Clock (GHz)	1.625	1.242	1.150	1.544	0.836	
Int Mul/SM (/Clock)	8 (24-bit)	8 (24-bit)	16	16	32	
Int Mul (G/s)	117 (208)	134 (238)	258	395	375	
Scaling Factor	0.312	0.357	0.688	1.053	1	
1024-bit MulMod (10^6/s)	-	-	11.1	-	49.8	110.5
MulMod (scaled) (10^6/s)	-	-	16.1	-	49.8	110.5
RSA-1024(/s)	813(non-CRT)	41,426	-	-	-	-
RSA-2048(/s)	406.5		-	12,044	-	38,975
RSA (scaled)(/s)	1,303	14,504	-	11,438	-	38,975

From Table 5, we can see that our modular multiplication implementation outperforms the others by a great margin. We achieve nearly 6 times speedup than the work [10], and even at the same CUDA platform, we obtain 221% performance of the work [12]. Our RSA implementation also shows a great performance advantage, 269% performance of the next fasted work [22]. Note that 1024-bit RSA decryption of [22] have latency of about 150ms, while 2048-bit RSA decryption of ours reaches 22.47ms.

The great performance advantage lies mainly in the utilization of floating-point processing power and the superior handling of Montgomery multiplication. Besides, compared with the work using multiple threads to process a Montgomery multiplication [11, 12], another vital reason is that we use more efficient `shuffle` instruction to handle data sharing instead of shared memory and employ more reasonable degree of thread parallelism to economize the overhead of thread communication.

7 Conclusion

In this contribution, we propose a brand new approach to implement high-performance RSA cryptosystems in latest CUDA GPUs by utilizing the powerful

floating-point computing resource. Our results demonstrate that the floating-point computing resource is a more competitive candidate for the asymmetric cryptography implementation in CUDA GPUs. In the NVIDIA GTX Titan platform, our 2048-bit RSA decryption achieves 2.21 times performance of the existing fastest integer-based work and performs 13 times faster than the previous floating-point-based implementation. We will try to apply these designs to other asymmetric cryptography, such as the floating-point-based ECC implementation.

References

1. Antão, S., Bajard, J.C., Sousa, L.: Elliptic curve point multiplication on GPUs. In: IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP), pp. 192–199. IEEE (2010)
2. Antão, S., Bajard, J.C., Sousa, L.: RNS-Based elliptic curve point multiplication for massive parallel architectures. The Computer Journal 55(5), 629–647 (2012)
3. Bernstein, D.J., Chen, H.C., Chen, M.S., Cheng, C.M., Hsiao, C.H., Lange, T., Lin, Z.C., Yang, B.Y.: The billion-mulmod-per-second PC. In: Workshop Record of SHARCS, vol. 9, pp. 131–144 (2009)
4. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on graphics cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)
5. Bos, J.W.: Low-latency elliptic curve scalar multiplication. International Journal of Parallel Programming 40(5), 532–550 (2012)
6. IEEE Standards Committee, et al.: 754-2008 IEEE standard for floating-point arithmetic. IEEE Computer Society Std. 2008 (2008)
7. Granlund, T.: the gmp development team. gnu mp: The gnu multiple precision arithmetic library, 5.1 (2013)
8. Hankerson, D., Vanstone, S., Menezes, A.J.: Guide to elliptic curve cryptography. Springer (2004)
9. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 350–367. Springer, Heidelberg (2009)
10. Henry, R., Goldberg, I.: Solving discrete logarithms in smooth-order groups with CUDA. In: Workshop Record of SHARCS, pp. 101–118. Citeseer (2012)
11. Jang, K., Han, S., Han, S., Moon, S., Park, K.: Sslshader: Cheap ssl acceleration with commodity processors. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, p. 1. USENIX Association (2011)
12. Jeffrey, A., Robinson, B.D.: Fast GPU Based Modular Multiplication, `http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4156_montgomery_multiplication_CUDA_concurrent.pdf`
13. Jonsson, J., Kaliski, B.: Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1 (2003)
14. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms, vol. 2, p. 116. Addison-Wesley, Reading (1981)
15. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)
16. Koç, C.K.: High-speed RSA implementation. Technical report, RSA Laboratories (1994)

17. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro 16(3), 26–33 (1996)
18. Leboeuf, K., Muscedere, R., Ahmadi, M.: A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography. In: 2013 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2593–2596. IEEE (2013)
19. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
20. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
21. Moss, A., Page, D., Smart, N.P.: Toward acceleration of RSA using 3D graphics hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 364–383. Springer, Heidelberg (2007)
22. Neves, S., Araujo, F.: On the performance of GPU public-key cryptography. In: 2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 133–140. IEEE (2011)
23. NVIDIA: NVIDIA CUDA Math API, http://docs.nvidia.com/cuda/cuda-math-api/index.html#axzz308wmibga
24. NVIDIA: NVIDIA GeForce Kepler GK110 Writepaper, http://159.226.251.229/videoplayer/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf?ich_u_r_i=e1d64c09bd2771cfc26f9ac8922d9e6d&ich_s_t_a_r_t=0&ich_k_e_y=14450689257506632824 71&ich_t_y_p_e=1&ich_d_i_s_k_i_d=1&ich_u_n_i_t=1
25. NVIDIA: CUDA C Programming Guide 5.5 (2013), http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
26. NVIDIA: Shuffle: Tips and Tricks (2013), http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf
27. Orup, H.: Simplifying quotient determination in high-radix modular multiplication. In: Proceedings of the 12th Symposium on Computer Arithmetic, pp. 193–199. IEEE (1995)
28. Pu, S., Liu, J.-C.: EAGL: An Elliptic Curve Arithmetic GPU-Based Library for Bilinear Pairing. In: Cao, Z., Zhang, F. (eds.) Pairing 2013. LNCS, vol. 8365, pp. 1–19. Springer, Heidelberg (2014)
29. Quisquater, J.J., Couvreur, C.: Fast decipherment algorithm for RSA public-key cryptosystem. Electronics Letters 18(21), 905–907 (1982)
30. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
31. Solinas, J.A.: Generalized mersenne numbers. Citeseer (1999)
32. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
33. Wikipedia: Wikipedia: List of NVIDIA graphics processing units (2014), http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units