Hindawi
Security and Communication Networks
Volume 2017, Article ID 3508786, 15 pages
https://doi.org/10.1155/2017/3508786

Research Article

Utilizing the Double-Precision Floating-Point Computing Power of GPUs for RSA Acceleration

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

[1]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2]*Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China*
[3]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*

Correspondence should be addressed to Fangyu Zheng; fyzheng@is.ac.cn

Received 23 May 2017; Accepted 8 August 2017; Published 17 September 2017

Academic Editor: Zhe Liu

Asymmetric cryptographic algorithm (e.g., RSA and Elliptic Curve Cryptography) implementations on Graphics Processing Units (GPUs) have been researched for over a decade. The basic idea of most previous contributions is exploiting the highly parallel GPU architecture and porting the integer-based algorithms from general-purpose CPUs to GPUs, to offer 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 fully exploit the floating-point computing power of GPUs, by various designs, including the floating-point-based Montgomery multiplication/exponentiation algorithm and Chinese Remainder Theorem (CRT) implementation in GPU. And for practical usage of the proposed algorithm, a new method is performed to convert the input/output between octet strings and floating-point numbers, fully utilizing GPUs and further promoting the overall performance by about 5%. The performance of RSA-2048/3072/4096 decryption on NVIDIA GeForce GTX TITAN reaches 42,211/12,151/5,790 operations per second, respectively, which achieves 13 times the performance of the previous fastest floating-point-based implementation (published in Eurocrypt 2009). The RSA-4096 decryption precedes the existing fastest integer-based result by 23%.

1. Introduction

With the rapid development of e-commerce and cloud computing, the high-density calculation of digital signature and asymmetric cryptographic algorithms such as the Elliptic Curve Cryptography (ECC) [1, 2] and RSA [3] algorithms is urgently required. However, without significant development in recent years, CPUs are more and more difficult to keep pace with such rapidly increasing demands. Specialized for the compute-intensive and high-parallel computations required by graphics rendering, GPUs own much more powerful computation 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 computing capability and moderate cost, many researchers resort to GPUs to perform cryptographic acceleration.

Born for high-definition 3D graphics, GPUs require high-speed floating-point processing capability; thus the floating-point units residing in GPUs achieve dramatical increase. From 2010 to the present, floating-point computing power of CUDA GPUs grows almost 10 times, from 1,345/665.6 (Fermi architecture) Giga Floating-point Operations Per Second (GFLOPS) to 10,609/5304 (Pacal architecture) GFLOPS for single/double-precision floating-point arithmetic. By contrast, integer multiplication arithmetic of NVIDIA GPUs gains only 25% performance improvement from Fermi to Kepler architecture; the latest Maxwell and Pascal architectures even do not provide dedicated device instructions for 32-bit integer multiply and multiply-add arithmetic [4].

However, the floating-point instruction set is inconvenient to realize large integer modular multiplication which

is the core operation of asymmetric cryptography. More importantly, the floating-point instruction set in the previous GPUs shows no significant performance advantage over the integer one. To the best of our knowledge, Bernstein et al. [5] are 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 [6] based on the integer instruction set, the floating-point-based one only achieves about 1/6 performance. Nevertheless, with the rapid development of GPU floating-point processing power, fully utilizing the floating-point processing resource will bring great benefits to the 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 double-precision floating-point (DPF) processing power in CUDA GPUs. In particular, we propose a DPF-based large integer representation and adapt the Montgomery multiplication algorithm to it. Also, we flexibly employ the integer instruction set to supplement the deficiency of the DPF computing power. Besides, we fully utilize the latest *shuffle* instruction to share data between threads, which makes the core algorithm Montgomery multiplication a non-memory-access design, decreasing greatly the performance loss in the thread communication. Additionally, a method is implemented to apply the proposed DPF-based RSA decryption algorithm in practical scenarios where the input and output shall be in bit-format, improving further the overall performance by about 5%, by decreasing data transfer consumption using smaller data format and leveraging GPUs to promote the efficiency of data conversion.

With these improvements, in GTX TITAN, the performance of the proposed RSA implementation increases dramatically compared with the previous works. In particular, the experimental results of RSA-2048/3072/4096 decryption with CRT reach the throughput of 42,211/12,151/5,790 operations per second and achieve 13 times the performance of the previous floating-point-based implementation [5], and RSA-4096 decryption is 1.23 times the performance of the existing fastest integer-based one [7].

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 demonstrate that the GPU architecture can be used as an asymmetric cryptography workhorse. For ECC, Antão et al. [8, 9] and Pu and Liu [10] employed the Residue Number System (RNS) to parallelize the modular multiplication into several threads. Bernstein et al. [6] and Leboeuf et al. [11] used one thread to handle a modular multiplication with Montgomery multiplication. Pan et al. [12], Zheng et al. [13], Bos [14], and Szerwinski and Güneysu [15] used fast reduction to implement modular multiplication over the Mersenne prime fields [16].

Unlike ECC scheme, RSA calculation requires longer and unfixed modulus and depends on modular exponentiation. Before NVIDIA CUDA appeared on the market, Moss et al. [17] mapped RNS arithmetic to the GPU to implement a 1024-bit modular exponentiation. Later in CUDA GPUs, Szerwinski and Güneysu [15] and Harrison and Waldron [18] developed efficient modular exponentiation by both Montgomery multiplication Coarsely Integrated Operand Scanning (CIOS) method and the RNS arithmetic. Jang et al. [19] presented a high-performance SSL acceleration using CUDA GPUs. They parallelized the Separated Operand Scanning (SOS) method [20] of the Montgomery multiplication by single limb. Jeffrey and Robinson [21] used the similar technology to implement 256-bit, 1024-bit, and 2048-bit Montgomery multiplication in GTX TITAN. Neves and Araujo [22] and Henry and Goldberg [23] used one thread to perform single Montgomery multiplication to economize the overhead of thread synchronization and communication; however, their implementations resulted in a very high latency. Emmart and Weems [7] applied one thread to a row oriented multiplication for 1024-bit modular exponentiation and a distributed model based on CIOS method for 2048-bit modular exponentiation. Yang [24] used an Integrated Operand Scanning (IOS) method with single limb or two limbs for Montgomery multiplication algorithm to implement RSA-1024/2048 decryption.

Note that all above works are based on the CUDA integer computing power. Bernstein et al. are the pioneers to utilize CUDA floating-pointing processing power in asymmetric cryptography implementations [5]. They used 28 single precision floating-points (SPFs) to represent 280-bit integer and implemented the field arithmetic. However, the result was barely satisfactory. Their later work [6] in the same platform GTX 295 using the integer instruction set performed almost 6.5 times the throughput of [5].

3. Background

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

3.1. GPU and CUDA. CUDA is a parallel computing platform and programming model that enables dramatical increase in computing performance by harnessing the power of GPU. It is created by NVIDIA and implemented by the GPUs which they produce [4].

The target platform, GTX TITAN, is a CUDA-compatible GPU (codename GK-110) with the CUDA Compute Capability 3.5 [25], which contains 14 streaming multiprocessors (SMs). 32 threads (grouped as a *warp*) within one SM concurrently run in a clock. Following the Single Instruction

Multiple Threads (SIMT) 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 has 64 KB on-chip memory which can be configured as fast *shared memory* and L1 cache; the maximum allocation of shared memory is 48 KB [25]. Each SM also possesses 64 K 32-bit registers. All SMs share 6 GB 256-bit wide slow *global memory*, cached read-only *texture memory,* and cached read-only *constant memory* [4, 25]. Each SM of GK-110 owns 192 single precision CUDA cores, 64 double-precision units, 32 special function units (SFUs), and 32 load/store units [25], 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 [4, 25].

NVIDIA GPUs of Compute Capability 3.x or later bring a 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 the NVIDIA Kepler architecture, *shuffle* instruction [4, 25] allows threads within a warp to share data. With shuffle instruction, threads within a warp can read any value of other threads in any imaginable permutations [25]. NVIDIA conducted various experiments [26] on the comparison between shuffle instruction and shared memory, which show that shuffle instruction always gives a better performance than shared memory. This instruction is available in GTX TITAN.

3.2. Integer and Floating-Point Arithmetic in GPU. NVIDIA GPUs with CUDA Compute Capability 3.x support both integer and floating-point arithmetic instruction sets. This section introduces some most concerned instructions in asymmetric cryptographic algorithm implementations, including add and multiply(-add) operations.

Integer. Integer arithmetic instructions `add.cc`, `addc`, `mad.cc`, and `madc` are provided to perform multiprecision add and multiply-add operations, which reference an implicitly specified condition code register (`CC`) having a single carry flag bit (`CC.CF`) holding carry-in/carry-out [27]. With this native support for multiprecision arithmetic, most of the previous works [6, 8–11, 13–15, 18, 19, 21–23] chose to use integer instructions to implement large integer arithmetic in asymmetric cryptographic algorithms.

One noteworthy point is that multiply or multiply-add instructions of NVIDIA GPUs with Compute Capability 3.x have a unique feature: when calculating the 32-bit multiplication, the whole 64-bit product cannot be obtained using single instruction but requires two independent instructions (one is for lower-32-bit half and the other for upper-32-bit half). Although the whole multiplication "instruction" (`mul.wide`) is provided which is used in [22, 23], it is a *virtual* instruction but not the native instruction, which is broken into 2 native instructions (`mul.lo.` and `mul.hi`) when running on the 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

Floating-Point. Floating-point arithmetic instructions in CUDA GPUs comply with 754-2008 IEEE Standard for Floating-Point Arithmetic [28]. Among five basic formats which the standard defines, 32-bit binary (i.e., SPF) and 64-bit binary (i.e., DPF) are supported in NVIDIA GPUs. 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 $(-1)^{\text{sign}} \times 2^{\text{exp}-\text{biase}} \times 1.\text{mantissa}$. Therefore, each SPF or DPF can, respectively, represent 24-bit or 53-bit integer.

Add and multiply(-add) operation instructions for floating-point are natively supported. In particular, floating-point multiply-add operation is always implemented by fused multiply-add (`fma`) instruction, which is executed in one instruction with single rounding.

Unlike integer instructions, the floating-point add or multiply-add instructions do 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 happens, in which the least significant bits are left out to keep the significand within the limitation. This round-off causes precision-loss, which is intolerable in cryptographic calculation. Thus in algorithm design, all operations should be carefully scheduled to avoid it.

Table 2 compares the computing power of integer, SPF and DPF in the target platform GTX TITAN. It is found that integer is more advantageous in add operation than floating-point, but the multiply(-add) operation of DPF is 2.6 times the performance of integer.

3.3. RSA and Montgomery Multiplication. RSA [3] is an algorithm widely used for digital signature and asymmetric encryption, whose core operation is modular exponentiation. And in practical scenarios, CRT [29] is widely used to promote the RSA decryption. Instead of calculating a $2n$-bit modular exponentiation directly, two n-bit modular exponentiations ((1a) and (1b)) and the Mixed-Radix Conversion (MRC) algorithm [30] (2) are sequently performed to conduct the RSA decryption:

$$P_1 = C^{d \bmod (p-1)} \bmod p; \tag{1a}$$

$$P_2 = C^{d \bmod (q-1)} \bmod q; \tag{1b}$$

$$P = P_2 + \left[(P_1 - P_2) \cdot \left(q^{-1} \bmod p\right) \bmod p\right] \cdot q, \tag{2}$$

where p and q are n-bit prime numbers chosen in private key generation ($M = p \times q$). All parameters, p, q, ($d \bmod p - 1$), ($d \bmod q - 1$), and ($q^{-1} \bmod p$) are parts of the RSA

Table 2: Computing power comparison among 32-bit integer, SPF, and DPF arithmetic instruction in GTX TITAN.

Instruction type	Parameter	Data format		
		Integer	SPF	DPF
Add	Significant bits w	32	23[1]	52[1]
	Instructions/SM/CLOCK i	160	192	64
	Bits/SM/CLOCK $w \cdot i$	5120	4416	3328
Multiply(-add)	Significant bits w	32	12[2]	26[2]
	Instructions/SM/CLOCK i	32	192	64
	Bits/SM/CLOCK $w^2 \cdot i$	16384[3]	27648	43264

[1]Addend and adder should be lower than $24 - 1 = 23$ (SPF) or $53 - 1 = 52$ (DPF) bits to avoid round-off problem. [2]Multiplicand and multiplier should be lower than $\lfloor 24/2 \rfloor = 12$ (SPF) or $\lfloor 53/2 \rfloor = 26$ (DPF) bits to avoid round-off problem. [3]Since the product of two 32-bit integers requires 2 multiplication instructions, "Bits/SM/CLOCK" for "Integer" is $(w^2 \cdot i)/2$.

private key [31]. Compared with calculating $2n$-bit modular exponentiations directly, the CRT technology gives 3 times the performance promotion [29].

Even with CRT technology, the bottleneck restricting the overall performance of RSA lies in modular multiplication. In 1985, Montgomery proposed an algorithm [32] to remove the costly division operation from the modular reduction. Let $\overline{A} = AR(\mathrm{mod}\, M)$ and $\overline{B} = BR(\mathrm{mod}\, M)$ be the Montgomeritized forms 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, $\mathrm{MonMul}(\overline{A}, \overline{B}) = \overline{A}\,\overline{B}R^{-1}(\mathrm{mod}\, M)$. 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 leads to a fast division by R.

A series of improvement for the Montgomery multiplication got public since its foundation. In 1995, Orup [33] 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

We aim to implement $2n$-bit RSA decryption with CRT introduced in Section 3.3; thus n-bit Montgomery multiplication shall be implemented first. This section proposes a DPF-based Montgomery multiplication parallel calculation scheme directed on CUDA GPUs, including the large integer representation, the fundamental operations, and the parallelism method of Montgomery multiplication.

4.1. Advantages and Challenges of DPF-Based RSA. DPF instruction set has a huge advantage in asymmetric cryptographic algorithm acceleration, as demonstrated in Table 2. Since RSA builds on large integer multiplication, such a huge advantage becomes a decisive point to compete with the integer instruction set.

However, it encounters many problems when exploiting its theoretical superior multiplication computing power.

(i) *Round-Off Problem*. Due to the round-off problem, every detail should be taken into careful consideration, which makes it very difficult and complicated to design and implement the algorithm.

(ii) *Nonsupport for Carry Flag*. Lack of support for carry flag makes it very inconvenient and inefficient to perform multiprecision add or multiply-add operation. Instead of using only one carry-flag-supported integer instruction, the carry has to be manually handled via multiple add or multiply-add instructions.

(iii) *Inefficient Add Instruction*. From Table 2, it can be found that, unlike multiplication instruction, the DPF add instruction is slower than the integer add instruction. Moreover, nonsupport for carry flag makes it perform even worse in multiprecision add operation.

(iv) *Inefficient Bitwise Operations*. Floating-point arithmetic does not support bitwise operations which are frequently used. CUDA Math API does support the `_fmod` function [34], 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.

(v) *Extra Memory and Register File Cost*. A DPF occupies 64-bit memory space; however, only 26 or lower bits are used. In this way, $(64 - 26)/26 = 138\%$ times extra cost in memory access and utilization of register files have to be overconsumed. In integer-based implementation, this issue is not concerned since every bit of an integer is utilized.

Before the introduction to the proposed algorithm, several symbols are defined in Table 3 for better reading of the following sections.

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.

Input:
$M > 2$ with $gcd(M, 2) = 1$, positive integers g, w such that $2^{wg} > 4M$;
$M' = -M^{-1} \bmod 2^w$, $R^{-1} = (2^{wg})^{-1}$
$\bmod M$;
Integer multiplicand A, where $0 \le A < 2M$;
Integer multiplier B, where $B = \sum_{i=0}^{g-1} b_i 2^{wi}$, $0 \le b_i < 2^w$ and $0 \le B < 2M$;
Output:
An integer S such that $S = ABR^{-1} \bmod 2M$ and $0 \le S < 2M$;
(1) $S = 0$
(2) **for** $i = 0$ *to* $g - 1$ **do**
(3) $S = S + A \times b_i$
(4) $q_i = ((S \bmod 2^w) \times M')(\bmod 2^w)$
(5) $S = (S + M \times q_i)/2^w$
(6) **end for**

Algorithm 1: High-radix Montgomery multiplication without determination of quotients accordingly (CIOS).

Table 3: Symbol explanation.

Symbol	Explanation
k	Thread ID where $0 \le k \le r - 1$
w	Number of significant bits in each DPF limb
n	The bit length of the modulus
v	The window size of fixed window exponentiation algorithm
u	Number of RSA per block
l	Number of DPF limbs of the DPF modulus, where $l = \lceil (n + 1)/w \rceil$
r	Number of threads per Montgomery multiplication
t	Number of DPF limbs per thread, where $t = \lceil l/r \rceil$

Table 4: Maximal w for different bit length.

n length (bit)	1024	1536	2048
w length (bit)	23	22	22

When each DPF (a and b) contains w ($w \le 26$) significant bits, $(2^{53} - 1)/(2^w - 1)^2$ times of $s = a \times b + s$ (the initial value of s is zero) can be executed, free from the round-off problem.

Note that, in Algorithm 1, there are g loops and each loop contains 2 `fma` operations for each limb, where $g = \lceil (n + 2)/w \rceil$, where n stands for the bit length of the modulus. Thus $2 \times \lceil (n + 2)/w \rceil$ times of `fma` operations are needed in total. The following requirement should be met for w, where

$$\frac{2^{53} - 1}{(2^w - 1)^2} \ge 2 \times \left\lceil \frac{n + 2}{w} \right\rceil, \tag{3}$$

so that the number of the supported `fma` surpasses the required ones. And the lower w means more instructions are required to process the whole algorithm. From Formula (3), to achieve the best performance, w is chosen as shown in Table 4.

In this contribution, two kinds of DPF-based large integer representations are proposed, *Simplified format* and *Redundant format*.

(i) *Simplified format* is like $A = \sum_{i=0}^{l-1} 2^{wi} a[i]$, where each limb $a[i]$ contains at most w significant bits. It is applied to represent the input of the `fma` instruction.

(ii) *Redundant format* is like $A = \sum_{i=0}^{l-1} 2^{wi} a[i]$, where each limb $a[i]$ contains at most **53** significant bits. It is applied to accommodate the output of the `fma` instruction.

4.3. Fundamental Operation and Corresponding Optimization. In DPF-based Montgomery multiplication, the fundamental operations include multiplication, multiply-add, addition, and bit extraction.

(i) *Multiplication*. In the implementation, the native multiplication instruction (`mul`) is used to perform multiplication. It is required that both multiplicand and multiplier are in Simplified format to avoid round-off problem.

(ii) *Multiply-Add*. In CUDA GPUs, `fma` instruction is provided to perform floating-point $s = a \times b + c$, which is executed in one step with a single rounding. In the implementation, when using `fma` instruction, it is required that multiplicand a and multiplier b are both in Simplified format and addend c is in Redundant format but less than $(2^{53} - 2^{46})$.

(iii) *Bit Extraction*. The algorithm needs 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. Two attempts of improvements are made to promote the performance. The first one

is introduced in [35]. Using round-to-zero, we can perform

$$x = a + 2^{52+53-r}$$
$$u = x - 2^{52+53-r} \quad (4)$$
$$v = a - u$$

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 left 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 the experiments, it is found that the second method always gives a better performance. Therefore, in most of cases, the second method is used to handle bit extraction. There is only one exception when the DPF a is divisible by a 2^r, DPF division $a/2^r$ is used to extract the most significant $53-r$ bits, which can be executed very fast.

(iv) *Addition*. CUDA GPUs provide the native add instruction to perform addition between two DPFs. But as aforementioned, it is inefficient and does not provide support for carry flag. Thus, DPF is first converted into integer; then CUDA native integer add instruction is used to handle addition.

4.4. DPF-Based Montgomery Multiplication Algorithm. With reference to Algorithm 1, in the Montgomery multiplication, $S = ABR^{-1} (\mathrm{mod}\ M)$, A, B, M, and S are all $(n+1)$-bit integer (in fact, M is n bits long, and it is also represented as a $(n+1)$-bit integer for a common format). As choosing w as limb size, $l = \lceil (n+1)/w \rceil$ DPF limbs are required to represent A, B, M, and S.

In the previous works [19, 21], 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 and Araujo [22] performed one entire Montgomery multiplication in one thread to economize overhead of thread synchronization and communication, however, resulting in a high latency, about 150 ms for 1024-bit RSA, which is about 40 times of [19].

To make a tradeoff between throughput and latency, in the implementation, we try multiple-limb parallelism, namely, using r threads to compute one Montgomery multiplication and each thread dealing with t limbs, where $t = \lceil l/r \rceil$. 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 warp, 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, r shall be a divisor of the warp size (i.e., 32).

Input:
k: Thread ID, where $0 \le k \le r - 1$;
h: Loops num, where $h = \lceil n/w \rceil$;
$A[0 : rt - 1]$: Multiplicand, where $0 \le A[i] < 2^w$;
$B[0 : rt - 1]$: Multiplier, where $0 \le B[i] < 2^w$;
$M[0 : rt - 1]$: Modulus, where $0 \le M[i] < 2^w$;
M': Integer, where $MM' = -1 \bmod 2^w$;
$a[0 : t - 1] = A[tk : tk + t - 1]$
$b[0 : t - 1] = B[tk : tk + t - 1]$
$m[0 : t - 1] = M[tk : tk + t - 1]$
Output:
Redundant-format sub-result $s[0 : t - 1] = S[tk : tk + t - 1]$, where $S = ABR^{-1} \bmod M$;

```
(1) S = 0
(2) for i = 0 to (h − 1) do
(3)     b_i = shuffle(b[i mod t], ⌊i/t⌋)
        * Step (1): S = S + A × b_i *
(4)     for j = 0 to t − 1 do
(5)         s[j] = s[j] + a[j] × b_i
(6)     end for
        * Step (2): q_i = ((S mod 2^w) × M') mod 2^w *
(7)     if k = 0 then
(8)         temp = s[0] mod 2^w
(9)         q_i = temp × M'
(10)        q_i = q_i mod 2^w
(11)    end if
(12)    q_i = shuffle(q_i, 0)
        * Step (3): S = S + M × q_i *
(13)    for j = 0 to t − 1 do
(14)        s[j] = s[j] + m[j] × q_i
(15)    end for
        * Step (4): S = S/2^w *
(16)    temp = (k ≠ r − 1)?shuffle(s[0], k + 1) : 0
(17)    s[0] = (k = 0)?(s[0] ≫ w + s[1]) : (s[1])
(18)    for j = 1 to t − 2 do
(19)        s[j] = s[j + 1]
(20)    end for
(21)    s[t − 1] = temp
(22) end for
```

Algorithm 2: DPF-based parallel Montgomery multiplication ($S = ABR^{-1} \bmod M$) algorithm: Computing Phase.

In the proposed Montgomery multiplication $S = ABR^{-1} (\mathrm{mod} M)$, the inputs A, B, and M are in Simplified format and the initial value of S is 0. Two phases are employed to handle one Montgomery multiplication, *Computing Phase* and *Converting Phase*. In Computing Phase, Algorithm 2 is responsible to calculate S, whose result is represented in Redundant format. Then in Converting Phase, S is converted from Redundant format to Simplified format applying Algorithm 3.

4.4.1. Computing Phase. Algorithm 2 is a t-limb parallelized version of Algorithm 1. In the algorithm, both the inputs A, B, and M and the output S are divided equally into $r = \lceil l/t \rceil$ groups and each group is assigned into one single thread. In each Thread k, the variables with index $tk \sim tk + t - 1$ are processed. Note that if the index of certain variable is

Input:
k: Thread ID;
t: Number of processed limbs per thread;
r: Number of threads per Montgomery multiplication, where $r = \lceil l/t \rceil$;
$s[0 : t-1]$: Redundant-format sub-result, where $s[0 : t-1] = S[tk : tk+t-1]$;
Output:
$s[0 : t-1]$: Simplified-format sub-result, where $s[0 : t-1] = S[tk : tk+t-1]$;
(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) carry = $(k = 0)?0$: shuffle(carry, $k-1$)
(8) **for** $j = 0$ to $t-1$ **do**
(9) $s[j] = s[j]$ + carry
(10) (carry, $s[j]$) = split($s[j]$)
(11) **end for**
(12) **end while**

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

larger than $l-1$, the variable shall be padded 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 $A[tk+j]$. And $v_1 = \text{shuffle}(v_2, k)$ means that current thread obtains variable v_2 of Thread k and stores it into variable v_1 using the shuffle instruction.

With reference to Algorithm 1, we introduce the proposed parallel Montgomery multiplication algorithm step by step.

(1) $S = S + A \times b_i$: in this step, each Thread k calculates $S[tk : tk+t-1] = S[tk : tk+t-1] + A[tk : tk+t-1] \times B[i]$. Note that each thread stores a group of B. Thus for each Loop i, firstly the corresponding $B[i]$ shall be broadcast from certain thread to all threads. In the view of single thread, $B[i]$ corresponds to $b[i \bmod t]$ of Thread $\lfloor i/t \rfloor$. The shuffle instruction is used to conduct this broadcast operation. Then each thread executes $s[j] = s[j] + a[j] \times b_i$, where $j \in [0, t-1]$.

(2) $q_i = ((S \bmod 2^w) \times M')(\bmod 2^w)$: $S \bmod 2^w$ is only related to $S[0]$, which is stored in Thread 0 as $s[0]$. Therefore, in this step, this calculation is only conducted in Thread 0, while other threads are idle. Note that S is in Redundant format, and the least significant w bits temp of $S[0]$ shall be firstly extracted before executing $q_i = \text{temp} \times M'$. And in next step, q_i will act as a multiplier; hence then, the same bit extraction shall be also applied to q_i.

(3) $S = S + M \times q_i$: in this step, each Thread k 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, similar to Step (1), it should also be broadcast to all threads. Then each thread executes $s[j] = s[j] + m[j] \times q_i$, 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 whole algorithm, shifting right can be done by executing $S[k] = S[k+1]$ $(0 \le k \le rt-2)$ and padding $S[rt-1]$ with zero. While 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. The second point is that S is represented in Redundant format; when executing $S = S/2^w$, the upper $(53-w)$ bits of the least significant limb $S[0]$ need to be stored. Thus, in Thread 0, $s[0] = s[0] \gg w + s[1]$ is calculated instead of $s[0] = s[1]$ in other threads. $S[0]/2^w$ is at most $(53-w)$ bits long, and due to the shift-right operation, $S[0]$ in each loop is not the same. Thus $S[0]$ can be only accumulated within $\lceil (n+2)/w \rceil \times 2 \times (2^w - 1)^2 + 2^{53-w} < 2^{53}$, which does not cause round-off problem. Note that $S[0]$ is divisible by 2^w; as introduced in Section 4.3, a simple division $S[0]/2^w$ can be used to efficiently extract the most significant $(53-w)$ bits.

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

4.4.2. Converting Phase. In Converting Phase, S is converted from Redundant format to Simplified format: every $S[k]$ adds the carry ($S[0]$ does not execute this addition) and holds the least significant w bits of the sum and propagates the most significant $(53-w)$ bits as new carry to $S[k+1]$. However, this method is serial, and the calculation of every $S[k]$ depends on the carry that $S[k-1]$ produces, which does not comply with the parallelism architecture of GPU. In practice, parallelized method is applied to accomplish Converting Phase, which is shown in Algorithm 3.

Algorithm 3 uses symbol $\text{split}(c) = (h, l) = (\lfloor c/2^w \rfloor, c \bmod 2^w)$ to denote that 53-bit integer c is divided into $(53-w)$ most significant bits h and w least significant bits l. Firstly, all threads execute a chain addition for its $s[0] \sim s[t-1]$

and store the final carry. Then every Thread $k-1$ (except Thread $(r-1)$) propagates the stored carry to Thread k using shuffle instruction and then repeats chain addition with the propagated carry. This step continues until carry of every thread is zero, which can be checked by the CUDA `_any()` voting instruction [4]. 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, over 75% execution time would be economized 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 the techniques on the implementation of Montgomery exponentiation, CRT computation, and pre-/postcomputation format conversion.

5.1. Montgomery Exponentiation. As introduced in Section 3.3, RSA with CRT technology requires two n-bit Montgomery exponentiation $S = X^Y R(\text{mod } M)$ to accomplish $2n$-bit RSA decryption. With the binary square-and-multiply method, the expected number of modular multiplications is $3n/2$ for n-bit modular exponentiation. The number can be reduced with m-ary method given by [36] that scans multiple bits, instead of one bit of the exponent. Jang et al. [19] used sliding window technology Constant Length Nonzero Windows (CLNW) [37] to reduce the number of Montgomery multiplications further. But it is not suitable for our design, in which an entire warp may contain more than one Montgomery multiplication and each Montgomery multiplication has different exponentiation which leads to different scanning step size and execution logic. These differences would cause warp divergence, largely decreasing the overall performance. And m-ary method is timing-attack-proof, the calculation time of which would not be affected by bit pattern. For all the above reasons, we choose to employ m-ary method not the CLNW method.

In m-ary method, where $m = 2^v$, the window size v shall be chosen properly to decline the number of modular multiplications $(n+\lceil n/v\rceil+2^v)$ and memory access $(\lceil n/v\rceil+2^v)$, and the window size $v = 6$ is employed as it offers the least computational cost and memory access. In this contribution, 2^6-ary method is implemented and reduces the number of modular multiplications from 1536 to 1259 for 1024-bit modular exponentiation, achieving 17.9% improvement. In $m = 2^6$-ary method, $(2^6 - 2)$ precompute table $(X^2R \sim X^{63}R)$ needs to be stored into memory for each Montgomery exponentiation. The memory space required (about 512 KB) is far more than the size of *shared memory* (at most 48 KB); thus they have to be stored into *global memory*. Global memory load and store operations consume hundreds of clock circles. To improve memory access efficiency, the Simplified format precompute tables are converted into w-bit integer format using GPU format-convert instruction and then stored in global memory. This optimization economizes about 50% memory access consuming.

Input:
X: Simplified-format Base number;
Y: n-bit Integer Exponent;
M: Simplified-format Modulus;
R: Radix, $R = 2^{w\times l}$;
Output:
$S = X^Y R \bmod M$;
(1) $T = X$
(2) **for** $i = 2$ to $2^6 - 1$ **do**
(3) $T =$`MonMul`(T, T, M)
(4) Store $X^i = T$ in *Global Memory*
(5) **end for**
(6) Decompose n-bit Y into 6-bit limb $Y[i]$, where $Y = \sum_{i=0}^{\lceil n/6\rceil-1=z} Y[i]2^{6i}$, then assign them into r threads equally.
(7) $T = X^{Y[z]}$
(8) **for** $i = z - 1$ to 0 **do**
(9) Use shuffle to obtian $Y[i]$
(10) Repeat $T =$`MonMul`(T, T, M) for 6 times
(11) $T =$`MonMul`$(X^{Y[i]}, T, M)$
(12) **end for**
(13) $S = T$

Algorithm 4: Parallel n-bit Montgomery exponentiation ($S = X^Y R \bmod M$) algorithm (window size $v = 6$).

In Montgomery exponentiation, the exponent Y is represented in integer. Similar to the processing of shared data B in Montgomery multiplication $S = ABR^{-1} \bmod M$, each thread stores a piece of Y and uses shuffle to broadcast Y from certain thread to all threads.

Algorithm 4 shows how to conduct Montgomery exponentiation where $S =$`MonMul`(A, B, M) means calculating $S = ABR^{-1} \bmod M$ using Algorithms 2 and 3.

5.2. CRT Computation. In the first implementation, GPU only took charge of the Montgomery exponentiation. And the CRT computation (2) was offloaded to CPU using GNU multiple precision (GMP) arithmetic library [38]. 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 make attempt to integrate the CRT computation into GPU.

For CRT computation, a modular subtraction and a multiply-add function are additionally implemented. 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 scheme is shown in Figure 1.

5.3. Pre- and Postcomputation Format Conversion. RSA algorithm is built on large integer arithmetic, whose input and output are both large integers. The Digital Signature Standard (FIPS PUB 186-4) [39] by National Institute of Standards and Technology (NIST) regulates how to conduct conversion between bit-string and integer. Unfortunately, our DPF-based

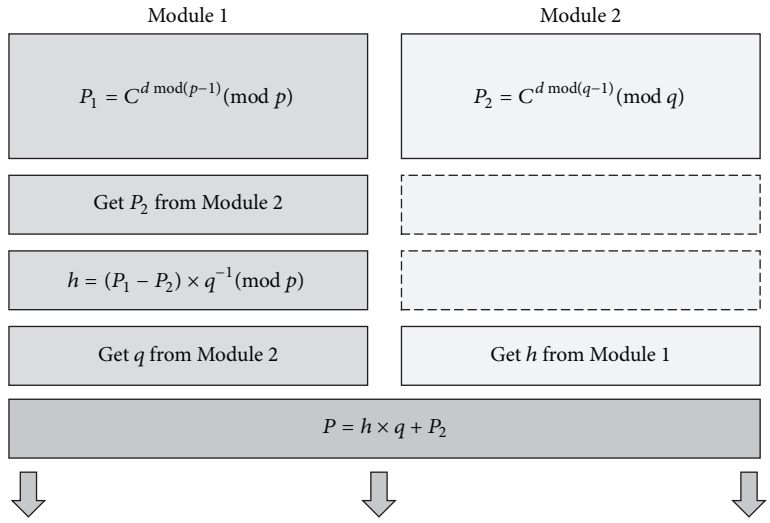

Figure 1: Intra-GPU CRT implementation.

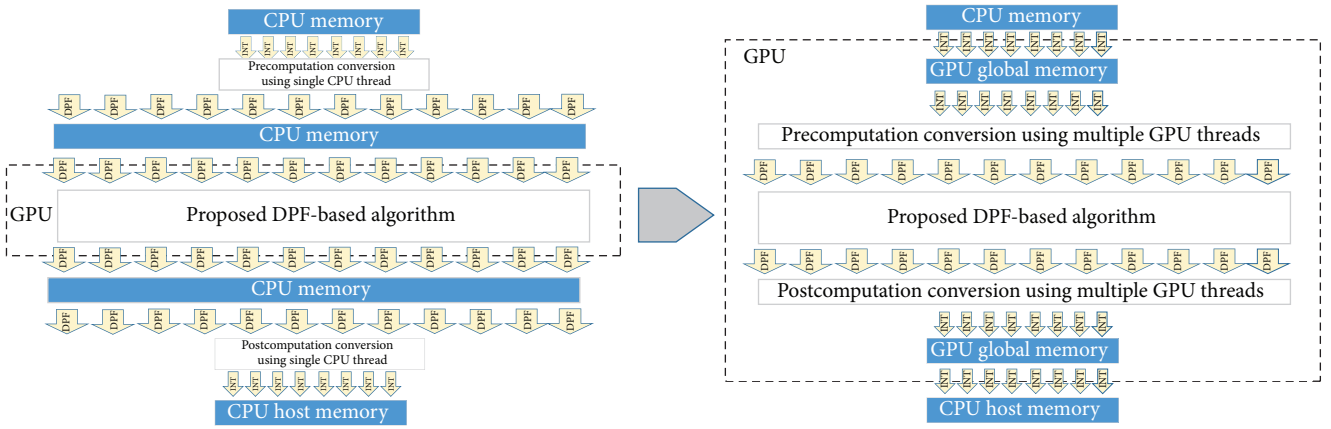

Figure 2: Two designs of pre- and postcomputation format conversion.

large integer representation is not consistent with it. Thus before and after applying the proposed DPF-based RSA algorithm, the conversion between DPF and 32-bit integer shall be conducted.

The first attempt was made to conduct the conversion in CPU, which is easy to implement, as shown in Figure 2(a): before and after GPU kernel, converting its DPF-format input and output from or to bit-string using bitwise and format conversion instructions of CPU. However, this strategy has two significant drawbacks.

Firstly, coded in DPF, the input and output require much more data transfer (`cudaMemcpy`) between GPU and CPU. Specifically, since each Simplified format DPF has only 23-bit ($w = 23$) significand, $\lceil 2048/23 \rceil = 90$ DPFs needs to store a 2048-bit integer, which occupies $90 \times 64 = 5760$ bits and causes $(5760 - 2048)/2048 = 181\%$ extra data transfer overhead. In fact, the overhead is quite high. As introduced in Section 3.3, for k-bit RSA decryption with CRT technology, it requires $3.5k$ bits for input and k bits for output, $4.5k$ bits in total. Assume 896 2048-bit DPF-based RSA decryptions are calculated once, at rate of 6 GB/s for data transfer through PCIe between GPU and CPU; it requires $(896 \times 4.5 \times 2048)/(6 \times 10^9 \times 8) \times (1 + 181\%) = 0.483$ ms for `cudaMemcpy`, while the total duration for RSA-2048 decryption is only about 20 ms.

Secondly, converting data format in CPU is inefficient. The simplest way is using serial operations to convert before and after GPU kernel in CPU. For example, we complete 896 RSA-2048 transformations which take over 1.1 ms using CPU's (Intel E5-2697 v2) single core in the experiment. But for GPU, it is adept in accomplishing conversion with thousands threads, and the GPU threads can be used to accomplish such amount of RSA in about 0.1 ms in GTX TITAN.

With full consideration for above two drawbacks, we turn to transfer data between GPU and CPU using integer rather

Table 5: Target platform configuration.

CPU	Intel Xeon CPU E5-2697v2 at 2.7 GHz
GPU	GeForce GTX TITAN
OS	Ubuntu 12.04
Tool chain	CUDA 6.0

than DPF, and in GPU kernel $r \times 2$ threads for each RSA decryption accomplish data conversion before and after RSA decryption calculation, as shown in Figure 2(b).

Precomputation Conversion. First, CPU transfers the 32-bit-integer-format inputs to global memory of GPU. According to its own DPFs needed to process in Algorithms 2 and 3, each CUDA thread extracts bits from the corresponding 32-bit integers, reconstructs them in w-bit-integer format using bitwise instructions, and finally converts w-bit-integer-format integer into Simplified format DPFs using integer-to-DPF instruction as the input of the proposed algorithm.

Postcomputation Conversion. The postcomputation conversion is a reverse procedure of the precomputation conversion, converting the Simplified format outputs into 32-bit-integer-format, then returning them to CPU.

This method largely reduces the cost of data transfer and leverages the computing power of GPUs. For 2048-bit RSA decryption, the overall latency decreases by up to 1.2 ms (5%).

6. Performance Evaluation

This section presents the implementation performance and summarizes the results for the proposed algorithm. Relative assessment is also presented by considering related implementation. The hardware and software configuration used in the experiment are listed in Table 5.

6.1. Experiment Result. Applying the DPF-based Montgomery multiplication algorithm and RSA implementation, respectively, described in Sections 4 and 5, RSA-2048/3072/4096 decryptions are implemented in NVIDIA GTX TITAN, respectively.

Several configuration parameters may affect the performance of the kernel, including the following:

(i) *Batch Size*: the number of RSA decryptions per GPU kernel launch

(ii) *Threads/Block*: the number of CUDA threads contained in a CUDA block

(iii) *Threads/RSA*: the number of CUDA threads assigned for each RSA decryption

(iv) *Regs/Thread*: the maximum number of registers assigned for each CUDA thread; both Regs/Thread and Threads/Block × Regs/Thread should be restricted within the GPU hardware limitation (i.e., 255 32-bit registers per thread and 65536 per CUDA block in GTX TITAN).

Table 6 demonstrates the performance for RSA-2048/3072/4096 with a set of combinations for all the above configurations.

Note that the configuration *Batch Size* is chosen from a small number to the hardware limitation. Figure 3 summarizes the impact of *Batch Size* on the performance, which indicates the larger *Batch Size* can always lead to the better performance. It is easy to understand, because GPUs pipeline the instructions to leverage instruction-level parallelism [4], and configuring as large *Batch Size* as possible can keep the pipeline busy most of time and fully utilize the computing resource in GPUs, thereby yielding better performance. As the throughput is the most concerned factor in this contribution, Table 6 chooses the maximum *Batch Size* supported by the GPU hardware for assessment (half of the maximum *Batch Size* is also provided for comparison), although some lower configurations of *Batch Size* bring a lower latency with an almost equivalent throughput.

Another important factor on performance is Threads/RSA. In Table 6, Column Threads/RSA = $t \times 2$ indicates t threads are used to process a Montgomery multiplication and $t \times 2$ threads to process one RSA decryption. As discussed in Section 4.4, theoretically, parallelizing single computing task into too many threads would result in poor throughput; meanwhile, parallelism of too few threads may lead to a high latency. Towards the practical usage and also emphasizing a high throughput, in the experiment, on the premise of an acceptable latency, as few Threads/RSA as possible are applied to offer a high throughput.

However, even without the consideration for acceptable latency, at some points, due to the hardware limitation for register file per thread, parallelism using too few threads may not promote but even largely decrease the overall throughput. An example of the experiment is RSA-4096 with Threads/RSA = 4×2 as shown in Table 6. For RSA-2048, the throughput of Threads/RSA = 4×2 always precedes Threads/RSA = 8×2. However, for RSA-4096 with Threads/RSA = 4×2, the growing number of variables required exceeds tremendously the hardware limitation (when Batch Size is 14×64, 127 registers per thread in GTX TITAN); thus many variables have to spill into off-chip local memory which is hundreds times slower than registers. In this way, the throughput is much less than Threads/RSA = 8×2. In fact, the proposed DPF-based algorithm is more sensitive for the number of available registers since it consumes more register file than normal integer-based one as specified in Section 4.1.

To summarize, for best practice of the proposed DPF-based algorithm, many factors should be comprehensively taken into consideration to suit both properties of GPU hardware and RSA modulus size, especially Threads/RSA and Regs/Thread with higher and higher requirement of RSA modulus size (7680 bits and more).

6.2. Performance Comparison. This section provides two groups of comparisons depending on different implementation mechanisms, floating-point-based and integer-based.

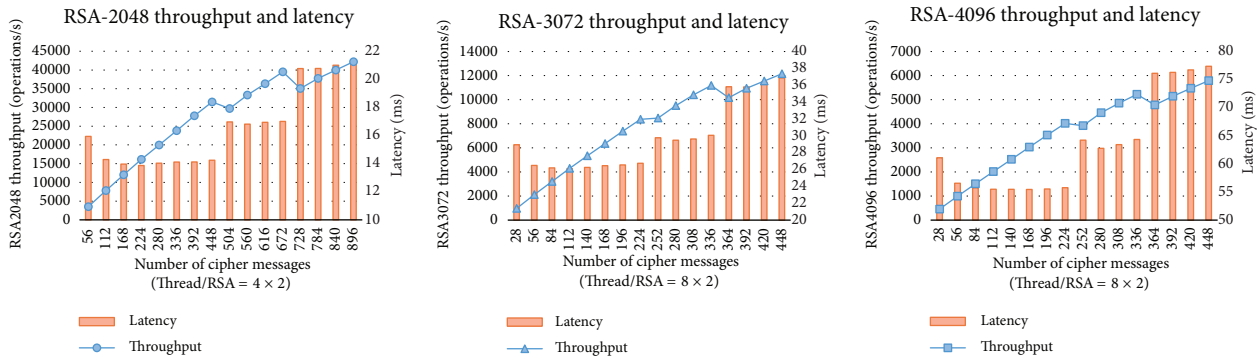

Figure 3: RSA-2048/3072/4096.

Table 6: Performance of RSA decryption.

Model	Threads/RSA	Batch Size	Regs/Thread	Threads/Block	Throughput (ops/s)	Latency (ms)
RSA-2048	4 × 2	14 × 64	127	512	**42,211**	21.22
	8 × 2	14 × 32			34,400	13.02
	4 × 2	14 × 32	255	256	31,095	14.41
	8 × 2	14 × 16			20,744	**10.80**
RSA-3072	4 × 2	14 × 64	127	512	10,642	84.19
	8 × 2	14 × 32			**12,151**	36.86
	4 × 2	14 × 32	255	256	10,555	42.44
	8 × 2	14 × 16			8,393	**26.69**
RSA-4096	4 × 2	14 × 64	127	512	821	1092.00
	8 × 2	14 × 32			**5,790**	77.37
	4 × 2	14 × 32	255	256	4,022	111.39
	8 × 2	14 × 16			4,147	**54.01**

6.2.1. Proposed versus Floating-Point-Based Implementation. Bernstein et al. [5] employed the floating-point arithmetic to implement 280-bit Montgomery multiplication. Table 7 shows performance of the work [5] and ours. Note that the work [5] only implemented the 280-bit modular multiplication; thus its performance is scaled by $(280/1024)^2$ as the performance of the 1024-bit one, and it is further scaled by the difference of the floating processing power of the corresponding GPU. The scaled result is shown in the row "1024-bit MulMod (scaled)."

Table 7 demonstrates the resulting implementation achieves 13 times speedup compared to the performance of [5]. Part of the 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 the advantage comes from the process of Montgomery multiplication. Bernstein et al. used Separated Operand Scanning (SOS) Montgomery multiplication method which is known inefficient [20]. And they utilized only 28 threads of a wrap (consisting 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 [5] 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. [5] 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.

6.2.2. Proposed versus Integer-Based Implementation. Previous works [7, 19, 21–23] are all integer-based. Thus we scale their CUDA platform performance based on the integer processing power. The parameters in Table 8 origin from [4, 40], 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, 8800 GTS 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 $(24/32)^2$ (unadjusted data is in parenthesis). *Throughput Scaling Factor* is defined as *Int Mul* ratio between the corresponding CUDA platform and GTX

Table 7: Performance comparison of Bernstein et al. [5].

	Bernstein et al. [5]	Proposed
Platform	GTX 295	GTX TITAN
GFLOPS	1788	4500
Scaling factor	0.397	1
280-bit MulMod (ops/s)	41.88×10^6	—
1024-bit MulMod (ops/s)	—	110.5×10^6
1024-bit MulMod (scaled) (ops/s)	7.89×10^6	110.5×10^6

TITAN. And *Throughput Scaling Factor* is also defined, as the Shader Clock ratio.

Table 8 summarizes the resulting performance of each work. We divide each resulting performance by the corresponding *Scaling Factor* listed in Table 8 as the scaled performance. Note that the RSA key length of Neves and Araujo [22] is 1024 bits, while ours is 2048 bits, and 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).

From Table 8, at the modular multiplication level, the proposed implementation outperforms the others by a great margin for floating-point-based RSA. We achieve nearly 6 times speedup compared to the work [23], and even at the same CUDA platform, we obtain 221% performance of the work [21].

At RSA implementation level, the proposed implementation also shows a great performance advantage, 291% performance of the work [22] for RSA-2048. Note that RSA-1024 decryption of [22] has latency of about 150 ms, while 2048-bit RSA decryption of ours reaches 21.52 ms when it reaches the throughput peak. Yang [24] reported the latency for RSA-2048 throughput of 5,244 (scaled as 31,782) is 195.27 ms (scaled as 225.60 ms). The throughput of the proposed RSA-2048 implementation is 132% performance of Yang's work, but the latency is 9.4% of their work [24]. Our peak RSA-2048 throughput is slightly slower than the fastest integer-based implementation [7], while our latency is 3 times faster. For RSA-4096 decryption, Emmart and Weems use distributed model [7] based on CIOS method with distributed integer values to report a throughput of 5257 RSA-4096 decryption on a GTX780Ti, scaling the performance to a GTX TITAN is 4,693, and our work is 1.23 times faster.

The performance advantage lies mainly in the utilization of floating-point processing power and the superior handling of Montgomery multiplication, which overcomes the problems addressed in Section 4.1. The DPF-based representation of large integer and Montgomery multiplication is carefully scheduled to avoid *round-off problem* and decrease largely the number of the expensive *carry flag handling* processes. For the *inefficient add instruction and bitwise operations* of DPF, the integer instruction set is flexibly employed to supplement the deficiency. And precompute table storage optimization (in Section 5.1) and the design of pre- and postcomputation format conversion (in Section 5.3) mitigate the performance influence brought by *extra memory and register file cost*.

Besides, compared with the works using multiple threads to process a Montgomery multiplication [19, 21], 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.

6.3. Discussion. In 2016 GPU Technology Conference, NVIDIA released the latest NVIDIA's Tesla P100 accelerator using the Pascal GP100 which has a huge improvement in floating-point computing power, especially DPF. Tesla P100 can deliver 5.3 TFLOPS of DPF performance which is 3.4 times our target platform [41]. Meanwhile the number of the registers for each CUDA core is doubled [41], which is essential for DPF-based RSA implementation. The performance of proposed DPF-based algorithm can be improved by at least three times based on all above improvements, which will further widen the gap between the DPF-based and the traditional integer-based algorithms.

7. Conclusion

In this contribution, we propose a brand new approach to implement high-performance RSA cryptosystems in the 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 NVIDIA GeForce GTX TITAN, our resulting RSA-2048 decryption reaches a throughput of 42,211 operations per second, which achieves 13 times the performance of the previous floating-point-based implementation. For RSA-3072/4096 decryption, our peak throughput is 12,151 and 5,790, and RSA-4096 implementation is 1.23 times faster than the best integer-based result. We also hope our endeavor can shed light on future research and inspire more case studies on GPU using floating-point computing power as well as other asymmetric cryptography. We will apply these designs to arithmetic, such as the floating-point-based ECC implementation, and exploit the floating-point power of Tesla P100.

Table 8: Throughput of operations per second.

	Neves and Araujo [22]	Henry and Goldberg [23]	Jang et al. [19]	Emmart and Weems [7]	Yang [24]	Jeffrey and Robinson [21]	Ours
CUDA platform	GTX 260	M2050	GTX 580	GTX780Ti	GT 750 m	GTX TITAN	
SM number	24	14	16	15	2	14	
Shader Clock (GHz)	1.242	1.150	1.544	0.876	0.967	0.837	
Int Mul/SM (/Clock)	8 (24-bit)	16	16	32	32	32	
Int Mul (G/s)	134 (238)	258	395	420	62	375	
Throughput scaling factor	0.357	0.688	1.053	1.12	0.165	1	
Latency scaling factor	1.484	1.374	1.845	1.047	1.155	1	
1024-bit MulMod (ops/s)	—	11.1×10^6	—	—	—	49.8×10^6	110.5×10^6
MulMod (scaled) (ops/s)	—	16.1×10^6	—	—	—	49.8×10^6	110.5×10^6
RSA-1024 (ops/s)	41,426	—	—	—	34,981	—	—
RSA-2048 (ops/s)	—	—	12,044	62,365	5,244[1]	—	42,211[2]
RSA-2048 (ms)	—	—	13.83	60.07	195.27[1]	—	21.22
RSA-2048 (scaled) (ops/s)	14,504	—	11,438	55,683	31,782	—	42,211[2]
RSA-2048 (scaled) (ms)	—	—	25.51	62.87	225.60	—	21.22
RSA-4096 (ops/s)	—	—	—	5,257	—	—	5,790[3]
RSA-4096 (scaled) (ops/s)	—	—	—	4,693	—	—	5,790[3]

[1]Yang et al. also report the latency of RSA-2048 decryption is 6.5 ms (after scaled 6.8 ms) when the Batch Size is 1, at the moment the throughput is 154. [2]The peak 2048-bit RSA throughput, when Threads/RSA is 4×2, window size is 6, Max Reg. is 127, and Batch Size is 14×64. [3]The peak 4096-bit RSA throughput, when Threads/RSA is 8×2, window size is 6, Max Reg. is 127, and Batch Size is 14×32.

Disclosure

A preliminary version of this paper appeared under the title Exploiting the Floating-Point Computing Power of GPUs for RSA, in Proc. Information Security, 17th International Conference, ISC 2014, Hong Kong, October 12–14, 2014 [42] (Best Student Paper Award). Dr. Yuan Zhao participated in this work when he studied in Chinese Academy of Sciences and now works in Huawei Technologies, Beijing, China.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

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.

References

[1] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[2] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology (CRYPTO '85)*, H. C. Williams, Ed., vol. 218 of *Lecture Notes in Computer Science*, pp. 417–426, Springer, 1986.

[3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the Association for Computing Machinery*, vol. 21, no. 2, pp. 120–126, 1978.

[4] 2013, NVIDIA: CUDA C programming guide 8.0, https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[5] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on Graphics Cards," in *Advances in Cryptology-EUROCRYPT*, pp. 483–501, Springer, Berlin, 2009.

[6] D. J. Bernstein, H. C. Chen, M. S. Chen et al., "The Billion-Mulmod-Per-Second PC," in *Workshop record of SHARCS*, vol. 9, pp. 131–144, 2009.

[7] N. Emmart and C. Weems, "Pushing the performance envelope of modular exponentiation across multiple generations of gpus," in *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*, pp. 166–176, May 2015.

[8] S. Antão, J.-C. Bajard, and L. Sousa, "Elliptic curve point multiplication on GPUs," in *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '10)*, pp. 192–199, July 2010.

[9] S. Antão, J.-C. Bajard, and L. Sousa, "RNS-based elliptic curve point multiplication for massive parallel architectures," *Computer Journal*, vol. 55, no. 5, pp. 629–647, 2012.

[10] S. Pu and J.-C. Liu, "EAGL: An elliptic curve arithmetic GPU-based library for bilinear pairing," in *Pairing-Based Cryptography–Pairing*, pp. 1–19, Springer, 2014.

[11] K. Leboeuf, R. Muscedere, and M. Ahmadi, "A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography," in *Proceedings of the 2013 IEEE International Symposium on Circuits and Systems (ISCAS '13)*, pp. 2593–2596, May 2013.

[12] W. Pan, F. Zheng, Y. Zhao, W. T. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with GPU acceleration," *IEEE Transactions on Information Forensics and Security*, pp. 111–122, 2017.

[13] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the Potential of GPUs for Modular Multiplication in ECC," in *Information Security Applications - 15th International Workshop*, pp. 295–306, Springer International Publishing, 2014.

[14] J. W. Bos, "Low-latency elliptic curve scalar multiplication," *International Journal of Parallel Programming*, vol. 40, no. 5, pp. 532–550, 2012.

[15] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Cryptographic Hardware and Embedded Systems–CHES*, pp. 79–99, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] J. A. Solinas, Generalized mersenne numbers. Citeseer, 1999.

[17] A. Moss, D. Page, and N. P. Smart, "Toward acceleration of RSA using 3D graphics hardware," in *Cryptography and coding*, vol. 4887 of *Lecture Notes in Comput. Sci.*, pp. 364–383, Springer, Berlin, 2007.

[18] O. Harrison and J. Waldron, "Efficient acceleration of asymmetric cryptography on graphics hardware," in *Progress in Cryptology–AFRICACRYPT*, pp. 350–367, Springer, 2009.

[19] K. Jang, S. Han, S. Moon, and K. Park, "Sslshader: cheap ssl acceleration with commodity processors," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[20] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.

[21] A. Jeffrey and B. D. Robinson, 2014, Fast GPU based modular multiplication. http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4156_montgomery_multiplication_CUDA_concurrent.pdf.

[22] S. Neves and F. Araujo, "On the performance of GPU public-key cryptography," in *Proceedings of the 22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2011*, pp. 133–140, September 2011.

[23] R. Henry and I. Goldberg, "Solving discrete logarithms in smooth-order groups with CUDA," in *Workshop Record of SHARCS*, pp. 101–118, 2012.

[24] Y. Yang, "Accelerating RSA with fine-grained parallelism using GPU," in *Information Security Practice and Experience*, J. Lopez and Y. Wu, Eds., vol. 9065 of *Lecture Notes in Computer Science*, Springer, 2015.

[25] 2012, NVIDIA: NVIDIA Kepler GK110 whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[26] 2013, NVIDIA: Shuffle: tips and tricks. http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf.

[27] 2016, NVIDIA: Parallel thread execution ISA version 5.0, http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz4SFUtDRZT.

[28] IEEE Standards Committee, "754-2008 - IEEE standard for floating-point arithmetic," pp. 1–58, 2008.

[29] J. J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronics letters*, vol. 18, no. 21, pp. 905–907, 1982.

[30] C. K. Koç, "High-speed RSA implementation," Tech. Rep., RSA Laboratories, 1994.

[31] J. Jonsson and B. Kaliski, Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1. The Internet Society, 2003.

[32] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[33] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proceedings of the 1995 IEEE 12th Symposium on Computer Arithmetic*, pp. 193–199, July 1995.

[34] 2016, NVIDIA: NVIDIA CUDA math API. http://docs.nvidia.com/cuda/cuda-math-api/index.html#axzz308wmibga.

[35] D. Hankerson, S. Vanstone, and A. J. Menezes, *Guide to Elliptic Curve Cryptography*, Springer, New York, NY, USA, 2004.

[36] D. E. Knuth, *The art of computer programming: seminumerical algorithms*, Addison-Wesley Publishing Co., Reading, Mass, USA, 1981.

[37] C. K. Koç, "Analysis of sliding window techniques for exponentiation," *Computers and Mathematics with Applications*, vol. 30, no. 10, pp. 17–24, 1995.

[38] T. Granlund, The gmp development team. gnu mp: The gnu multiple precision arithmetic library, 5.1., 2013.

[39] P. Gallagher and C. Kerry, FIPS Pub 186-4: Digital signature standard. DSS. NIST, 2013.

[40] 2016, Wikipedia: Wikipedia:List of NVIDIA graphics processing units. http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units.

[41] 2016, NVIDIA: NVIDIA Tesla P100 whitepaper, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[42] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the floating-point computing power of GPUs for RSA," in *Proceedings of the Information Security - 17th International Conference (ISC '14)*, pp. 198–215, 2014.