PhiRSA: Exploiting the Computing Power of Vector Instructions on Intel Xeon Phi for RSA

Yuan Zhao[1,2,3], Wuqiong Pan[1,2(✉)], Jingqiang Lin[1,2], Peng Liu[4], Cong Xue[1,2,3], and Fangyu Zheng[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{yzhao,wqpan,linjq,cxue13,fyzheng}@is.ac.cn
[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
[4] College of Information Sciences and Technology, The Pennsylvania State University, State College, USA
pliu@ist.psu.edu

Abstract. Efficient implementations of public-key cryptographic algorithms on general-purpose computing devices, facilitate the applications of cryptography in communication security. Existing solutions work in two different directions: implementations on GPUs achieve high throughput but great latency, while those on CPUs are with low throughput and small latency. Intel Xeon Phi is the first highly parallel coprocessor of Many Integrated Core (MIC) architecture, with up to 61 cores and one 512-bit Vector Processing Unit (VPU) in each core, which offers the potential to achieve both high throughput and small latency. In this paper, we propose a vector-oriented Montgomery multiplication design based on *vector carry propagation chain* (VCPC) method to fully exploit the computing power of vector instructions on Intel Xeon Phi. Two key features of our design sharply reduce the number of instructions: (1) organizing the additions in Montgomery multiplication to be four VCPCs for saving the overhead of handling carry bits; (2) computing the intermediate scalar variable q in every round without breaking the flow of VCPCs. Furthermore, we offer the optimal Montgomery multiplication implementation of our design on Intel Xeon Phi, which make VPUs fully pipelined and maintain carry bits in vector mask registers. Based on the above, we implement RSA named *PhiRSA* and evaluate it on Intel Xeon Phi 7120P. For 1024, 2048 and 4096-bit RSA, PhiRSA performs 258,370, 41,803 and 5,358 decryptions per second, and the latencies are 0.94, 5.84 and 45.54 ms, respectively. These results achieve 4.1 to 8.5 times performance of the existing RSA implementations on Intel Xeon Phi, exhibit high throughput comparable to those on GPUs but with much less parallel tasks, and small latency comparable to those on CPUs.

Y. Zhao—This work was partially supported by National 973 Program under award No. 2014CB340603 and No. 2013CB338001, Strategy Pilot Project of Chinese Academy of Sciences under award No. XDA06010702.

R. Avanzi and H. Heys (Eds.): SAC 2016, LNCS 10532, pp. 482–500, 2017.
https://doi.org/10.1007/978-3-319-69453-5_26

Keywords: Intel Xeon Phi · Vectorization · Montgomery multiplication · RSA · Performance

1 Introduction

The computing power of general-purpose processors is enhanced by different parallelism designs. Firstly, single-instruction-multiple-data (SIMD) enables the elements of a vector to be processed in parallel. General-purpose CPUs are usually equipped with vector instruction extensions, such as Intel MMX/SSE/AVX, ARM NEON and AMD 3DNow. Graphics processing units (GPUs) follow a different parallelism structure, single-instruction-multiple-thread (SIMT), where thousands of independent threads execute the same instructions concurrently. Finally, simultaneous-multithreading (SMT) is adopted by both CPUs and GPUs, to enable instructions from multiple threads (in a GPU thread block or a CPU core) to be executed in any given pipeline stage at a time.

The GPUs' potential on public-key cryptographic computing has been investigated for several years. Thread-level parallelism and thousands of scalar stream processors in GPUs, produce very high throughput on a great number of simultaneous tasks, but greater latency than the scalar-instruction cryptographic implementations in CPUs [22]. Note that the frequency of GPUs is much lower than that of general-purpose CPUs, for example, Intel Core i7 CPU reaches up to 3.5 GHz while NVIDIA Tesla K20 is only 706 MHz [30]. The deficiency on latency limits the applications of GPUs as public-key cryptographic engines in many scenarios.

In 2012 November, Intel announced the first product family of Many Integrated Core (MIC) architectures, named Intel Xeon Phi. Xeon Phi provides an opportunity to implement public-key algorithms in a high-throughput and low-latency way. For example, Xeon Phi 7120P consists of 61 cores, and each core is shipped with (a) 512-bit SIMD unit, 16-way 32-bit vector instructions, and (b) 4-way SMT unit, 4 hyperthreads on one core for instruction pipelining. Intel Xeon Phi, with the computing power in tera floating-point operations per second (FLOPS), has been applied in the fields of supercomputing, such as molecular dynamics in [25], sparse matrix multiplication in [27] and large integer arithmetic in [8,16]. In fact, similar 512-bit SIMD units are supported in Intel Xeon Skylake and Skylake-E CPUs and will be in Intel Cannonlake CPUs.

This paper presents the first implementation of public-key cryptographic algorithms with 512-bit SIMD instructions on Xeon Phi, called *PhiRSA*. In particular, we evaluate 1024-bit, 2048-bit and 4096-bit RSA on vector instructions. PhiRSA fully exploits the computing power of Xeon Phi 7120P with the following designs. Firstly, to perform 512-bit Montgomery multiplication (see Algorithm 1 for details), the most expensive step of RSA, the intermediate products are organized into four 512-bit vectors; then, these vectors are added using the vector-add-with-carry instruction **vpadcd** in each round of the Montgomery

multiplication's main loop. After n rounds, the corresponding 512-bit vector in each round composes a *vector carry propagation chain* (VCPC). This design exploits the vector mask registers and does not need to handle the carry bits after each addition in a round. Secondly, we exploit vector instructions to compute q (see Algorithm 3 for details), without breaking the flow of VCPCs. When a vector is used to compute q, the carry bit takes effect as the write-mask which is read-only in the operation; therefore, the correct q is obtained in the each round of VCPCs but does not break the chains.

The features of SIMD are fully exploited in PhiRSA, as our design magnifies the advantages of vector instruction extensions of Xeon Phi. Our method outperforms greatly the commonly-used redundant representation method in [3,5,10–12,21]. To avoid handling the carry bits after large-integer addition during Montgomery multiplication, redundant representation stores only 29-bit operands in each 64-bit element of vectors; then, every product of two elements multiplication is 58-bit and the additional 6 bits are used to hold addition carries without overflow. So, it requires extra instructions and vectors to finish the computations.

We implement 1024/2048-bit Montgomery multiplication (and then 2048/4096-bit RSA) based on 512-bit vectors. Two (or four) 512-bit vectors compose a 1024-bit (or 2048-bit) large integer, and the specific vector instruction *valignd* is used to right shift multiple 512-bit vectors of the large integer during the main loop of Montgomery multiplication. The operations of right shift and assignment are performed in only one vector instruction, for each 512-bit vector.

Meanwhile, the benefit of SMT is also kept in PhiRSA. The execution order of vector instructions is manually optimized to fully activate the pipeline of vector processing units (VPUs). When 4 threads are launched to perform RSA computations, the VPU utilization exceeds 90%, that is, almost one instruction is executed in each cycle.

Our contributions are as follows. Firstly, the vector-oriented designs are proposed to fully exploit the computing power of vector instructions for RSA. Secondly, we implement these designs on Intel Xeon Phi 7120P efficiently. To the best of our knowledge, this is the first implementation of public-key cryptography on Intel Xeon Phi. The experimental results exhibit both high throughput and low latency: for 1024-bit, 2048-bit and 4096-bit RSA, PhiRSA achieves the throughput of 258370, 41803 and 5358 decryptions per second with 244 parallel tasks, and the latency of 0.94 ms, 5.84 ms and 45.54 ms, respectively. This throughput is about 40 times of OpenSSL [23] on a single core of Intel Haswell i7-4770R, and the latency is about 5 times. Our throughput is higher than the best implementation [32] on GPUs [32], and the latency is reduced to about 25% only.

The rest of the paper is organized as follows. Section 2 is the related work. The preliminaries about Intel Xeon Phi and Montgomery multiplication are presented in Sect. 3. Section 4 describes the design of our Montgomery multiplication. In Sect. 5, we show how to implement Montgomery multiplication and RSA on Intel

Xeon Phi. In Sect. 6, performance results of our Montgomery multiplication and RSA implementations are given and compared with other works. We conclude in Sect. 7.

2 Related Work

There have been amount of studies using vector instructions to implement large integer multiplication, Montgomery multiplication and public-key cryptography. These works can be classified into three groups. The first group and also the main choice is storing the large integers in vectors horizontally for fine-grained parallel. Intel SSE2 instruction set has been exploited for large integer multiplication in [21] and cryptographic pairing computation in [11]. Redundant representation method proposed in [21] is widely used in vector implementations to help delay the carry propagation. Intel AVX2 instruction set is also applied to modular exponentiation in [12] and Curve25519 implementation in [10]. ARM NEON instruction set is explored to implement Montgomery multiplication in [28], Curve41417 in [3] and RSA in [29]. On Cell platform, an approach to implement Montgomery multiplication is described in [5]. The second group is splitting the Montgomery multiplication into two parts to compute in parallel. This approach is studied in [6] for 2-way vector instruction sets like Intel SSE2 and ARM NEON. The third group is using the vector instructions to carry out multiple tasks in parallel. Computing multiple Montgomery multiplications simultaneously in vector elements is investigated on Intel SSE2 instruction set in [24] and the Cell processor in [4].

Many previous studies have proved that GPUs are suitable for asymmetric cryptography. Most of them are based on the integer computing power of GPU, such as [1,31]. The floating-pointing power is also explored in [2,32]. For 2048-bit RSA GPU implementation, the highest throughput is reported in [32] and the lowest latency is obtained by [31].

Intel Xeon Phi is launched as a brand-new high performance computing platform, which performance has been evaluated in [9]. Large integer multiplication is firstly evaluated on Intel Xeon Phi in [16]. This work implements multiplication by using the usual redundant representation method described in [12]. While the study in [7] firstly implements multiplication and RSA based on the idea of carry propagation and endeavors to minimize memory footprints for reducing memory accesses. However, the results of these two studies are barely satisfactory and the computing power of Intel Xeon Phi has not been fully exploited.

3 Preliminaries

3.1 Overview of Intel Xeon Phi

Intel Xeon Phi comprises of up to 61 cores and every core possesses arithmetic logic units (ALUs) and one 512-bit VPU which provides the major computing power. The cores are in-order and pipelined. Each core supports four hyper-threads to keep the execution units busy and hide memory access latencies.

If the instructions are fully pipelined, the throughput of VPUs gets up to one vector instruction per cycle. There are L1 cache and L2 cache in each core and GDDR5 memory on board. The coprocessor communicates with the host through Peripheral Component Interconnect Express (PCIe) interface. The coprocessor OS based on an open-source Linux kernel runs on the coprocessor to manage resources and process applications. There are two predominant programming models for Intel Xeon Phi, offload execution mode and native execution mode. In native execution mode, the application is cross-compiled and runs directly on the coprocessor OS.

Intel Xeon Phi Instruction Set Architecture [13] introduces 512-bit vector instructions operating on thirty-two 512-bit vector registers (zmm0-zmm31), and offers eight 16-bit vector mask registers (k0-k7) for conditional operations on data elements within vector registers. One vector register consists on either sixteen 32-bit elements or eight 64-bit elements while the vector instructions executive operations on each element. The vector mask registers have many applications, the major is playing as write-mask to protect elements in the destination from updates during the execution of any operations. If a write mask bit is zero, the corresponding destination element is not modified. Vector mask registers can also be used for keeping carry bits, borrow bits and comparison results. Intel Xeon Phi does not support MMX, SSE and AVX instruction set, but introduces amount of novel vector instructions. For example, the vector-add-with-carry instruction *vpadcd* is extremely useful in large integer arithmetic, presented as follows.

$$vpadcd \quad (zmm2/memory), \quad k2, \quad zmm1\{k1\}$$

This instruction performs an element-by-element three-input addition between int32 vector zmm1, a int32 vector in memory or int32 vector zmm2, and the carry bits in k2. The result is written into zmm1, and the carry bits produced by the addition are written into k2. The instruction performing is controlled by the write-mask k1. Some other vector instructions are used in this paper. The instruction *vpmulhud* and *vpmulld* perform element-by-element multiplications between int32 vectors and store the high 32-bit result or the low 32-bit result respectively. The instruction *vpermd* performs an element permutation by using int32 vector elements as source indices. The instruction *valignd* concatenates and shifts right several 32-bit elements from two vectors.

3.2 Montgomery Multiplication

The major computations of RSA are modular multiplication. The modular reduction would be very costly if performing division operations. Montgomery multiplication [20] is proposed to replace division operations by cheaper multiplication and shifting operations. Let M be an odd modulus, $R = 2^n$ and $M < R$, Montgomery multiplication is defined as $MontMul(A,\ B) = A \cdot B \cdot R^{-1} \pmod{M}$. The process of calculating $A \cdot B \pmod{M}$ based on Montgomery multiplication can be computed as follows: $\widetilde{A} = MontMul(A,\ R^2)$, $\widetilde{B} = MontMul(B,\ R^2)$,

then $\widetilde{C} = MontMul(\widetilde{A},\ \widetilde{B})$, finally $C = MontMul(\widetilde{C},\ 1)$, C is the result. If executing a sequence of modular multiplications, such as the modular exponentiation, one modular multiplication only needs to perform one Montgomery multiplication. Koç et al. proposed an interleaved Montgomery multiplication,

Algorithm 1. Montgomery multiplication CIOS method [19]

Input: Modulus M, $R = 2^{nw}$, $R > M$, $\gcd(M,\ R) = 1$, 2^w is radix, n is digits number $0 \leqslant A, B < M$, $B = \sum_{i=0}^{n-1} b_i 2^{iw}$, $\mu = -M^{-1} \bmod 2^w$
Output: $S = A \cdot B \cdot R^{-1} \pmod{M}$, $0 \leqslant S < M$.

```
1: S ← 0
2: for i from 0 to n − 1 do
3:     S ← S + A · b_i
4:     q ← S[0] · μ mod 2^w
5:     S ← S + M · q
6:     S ← S/2^w
7: end for
8: if S ⩾ M then
9:     S ← S − M
10: end if
11: return S
```

named *Coarsely Integrated Operand Scanning* (CIOS) method [19] described in Algorithm 1. This method interleaves multiplication and Montgomery reduction, which is suitable to be implemented by vector instructions.

4 Montgomery Multiplication Design

In this section, we describes Montgomery multiplication design based on vector carry propagation chain (VCPC) method and the computation the intermediate scalar variable q. Then, we analyse the expected performance of our design and compare with the redundant representation method in [12]. Especially, we give out the vector length Montgomery multiplication (Algorithm 3) in Section Appendix as an example.

4.1 Vector Carry Propagation Chain Method

(1) Four VCPCs. As described in Algorithm 1, the main computations of Montgomery Multiplication CIOS Method are $S \leftarrow S + A \cdot b_i$ and $S \leftarrow (S + M \cdot q)$. Note that, this two formulas perform the same computation with different operands. The computing process is to multiply a vector with an element, then add the multiplication product to the sum vector S. We exploit Intel Xeon Phi vector instructions to carry out this computation. The logic instructions are used in this section for better clarification. We use *Mullow*, *Mulhigh* and *Vadc*

standing for *vpmulhud*, *vpmulld* and *vpadcd*. We assume that the length of S is equal to one vector. The formula $S \leftarrow S + A \cdot b_i$ is computed in the following steps.

$$\begin{aligned} T &\leftarrow Broadcast(b_i) \\ L &\leftarrow Mullow(A,\ T) \\ H &\leftarrow Mulhigh(A,\ T) \\ S &\leftarrow S + L \\ S &\leftarrow S + H \end{aligned}$$

The step 1 and step 2 are easy to be carried out, but the products L and H are not the final multiplication product and the least significant element of H is aligned with the 2nd less significant element of L. The step 3 and step 4 must be computed by using *Vadc* instruction. We focus on L and S, extract the corresponding computations from Montgomery multiplication in Algorithm 1. As the integer L in i-th loop is higher one element than the integer L in (i-1)-th loop, and also S shift to the right for a element, so L is aligned to S at all times. We use *Vadc* instruction to overwrite the upper computing process 1 and will demonstrate computing process 2 completes the same calculation as computing process 1:

$$\begin{aligned} &for\ \ i\ \ from\ \ 0\ \ to\ \ n-1 \\ &\qquad T \leftarrow Broadcast(b_i) \\ &\qquad L \leftarrow Mullow(A,\ T) \\ &\qquad S \leftarrow Vadc(S,\ k1,\ L) \\ &\qquad S \leftarrow Rshift(Zero,\ S,\ 1) \end{aligned}$$

$k1$ is a vector mask register for storing carry. *Rshift* is used to stand for *valignd* which concatenates two vector and shifts the whole long vector to the right in 32-bit elements, stores the lower vector to the destination register. We observe that in each loop before performing *Vadc*, S, $k1$ and L are all aligned, so they can be add together by using *Vadc* directly. After performing *Vadc*, the carry bits in $k1$ are propagated forward in a element. While after performing *Rshift* in this loop and *Mullow* in the next loop, L and S are also higher a element than before. So when it is to perform *Vadc* in the next loop, L, S and $k1$ are all aligned, and can be add together by using *Vadc* directly. So computing process 2 has completed the calculation in computing process 1, except has not added the carry k1 back to S after the last round of the loop.

Based on all the above observation, we propose the notion *Vector Carry Propagation Chain* (VCPC), which describes a process: a group of vectors like S, L and a carry like $k1$ are added together in a chain to propagate $k1$ forward in a element after each round. Propagating carry $k1$ only need one *Vadc* instruction in a round of VCPC and adding $k1$ back to S is only performed at the end of VCPC (also known as handing carry). So VCPC very efficiently works out the

carry propagation problem (the carry needs to propagate to the higher element). The strategy of VCPC is just propagating carry forward, delaying to handle it.

We propose a design as named VCPC method to overwrite major computations of Algorithm 1 by using VCPC.

The computing process 3 comprises four VCPCs. The VCPC 1 is *Vadc*(S, k0, L), L is the product of *Mullow*(A, b_i). The VCPC 2 is *Vadc*(S, k1, L), L is the product of *Mullow*(M, q). The VCPC 3 is *Vadc*(S, k2, H), H is the product of *Mulhigh*(A, b_i). The VCPC 4 is *Vadc*(S, k3, H), H is the product of *Mulhigh*(M, q). 4 VCPCs uses 4 vector mask registers k0, k1, k2 and k3 to propagating 4 carries respectively.

$$
\begin{aligned}
&for\ \ i\ \ from\ \ 0\ \ to\ \ n-1\\
&\qquad L \leftarrow Mullow(A,\ b_i)\\
&\qquad S \leftarrow Vadc(S,\ k0,\ L)\\
&\qquad L \leftarrow Mullow(M,\ q)\\
&\qquad S \leftarrow Vadc(S,\ k1,\ L)\\
&\qquad S \leftarrow Rshift(Zero,\ S,\ 1)\\
&\qquad H \leftarrow Mulhigh(A,\ b_i)\\
&\qquad S \leftarrow Vadc(S,\ k2,\ H)\\
&\qquad H \leftarrow Mulhigh(M,\ q)\\
&\qquad S \leftarrow Vadc(S,\ k3,\ H)
\end{aligned}
$$

VCPC 1 and VCPC 2 are performed before S shifting to the right, because the L computed in two VCPCs are aligned to S before shift. While H computed in VCPC 3 and VCPC 4 are aligned to S after shift, so VCPC 3 and VCPC 4 are performed after S shifting to the right. At the start of each round, there are vector S and four carries $k0$, $k1$, $k2$ and $k3$, and $k0$ and $k1$ are aligned to S, the lowest bit of $k2$ and $k3$ are aligned to the second less significant element of S. In the end of each round, S and carries $k0$, $k1$, $k2$ and $k3$ are all move to more significant position in a element, and also maintain that $k0$ and $k1$ are aligned to S, the lowest bit of $k2$ and $k3$ are aligned to the 2nd less significant element of S. So the four VCPCs in computing process 3 can be maintained to end of the loop.

(2) Handling Tail. At the end of our Montgomery multiplication design, handling tail must be performed, which includes two steps: handling carry and reducing S. Handling carry is used to add all the carry vectors produced by VCPCs to the sum vectors S. First, we add carry vectors to a vector which initial value is zero. Then, we add this vector to S. As Intel Xeon Phi does not have the instruction to shift vector mask register, we use the *LMove* to copy the carry to the general purpose register and perform left shift, then copy the carry back to the vector mask register. In the worst case, it will need to perform $s-1$ rounds

move-shift-move operations, but usually only need one round. As presented in Algorithm 1, reducing S is used to ensure the output of Montgomery multiplication is smaller than modulus M. For the constant running time, reducing S is always performed. We use vector-sub-with-borrow instruction *vpsbbd* to perform subtraction just like addition. Handling borrow requires $s-1$ rounds in the worst case, and performs two rounds in a greater chance. For performance reasons, the rounds of move-shift-move are not constant. While attackers can hardly get useful information from the running time of move-shift-move.

4.2 Computing q

The intermediate scalar variable q is produced and used for multiplications in Algorithm 1. q is computed as $q \leftarrow S[0] \cdot \mu \bmod 2^w$, which is very easy to be computed in other Montgomery multiplication design, such as redundant representation method. But in our Montgomery multiplication design, computing q is not easy as the carries are maintaining for propagation.

Note that, q computation is carried out by $Q \leftarrow Mullow(S, U)$, $U = Broadcast(\mu)$, $q = Q[0]$. But if we only perform this operation, q may be not correct. As we analyse the relations between q, four VCPCs and four carries, we can see that VCPC 2 and VCPC 4 need q to compute multiplications, and VCPC 2 propagates $k1$. If $k1$ has not be added to S ($k1$ propagates forward), the $S[0]$ may not be right since without being added carry bit $k1[0]$. So there is a contradiction that computing q requires $k1$ propagation forward, while propagating $k1$ (VCPC2) requires computing q first. If we add zero vector, $k1$ to S to obtain the right S, the $k1$ would propagate forward without adding with the product of *Mullow*(M, q), which will break the VCPC 2, also destruct the VCPC method.

The obvious solution is trial addition which performs an addition to acquire the right $S[0]$ and does not modify $k1$. As depicted in Sect. 3, *vpadcd* is a three-operand instruction, carry k2 (see Sect. 3) is the source operand also the destination operand which means that the old value in $k2$ will be destructed by the new value. So k1 in VCPC2 must be copied for trial addition which need two operations copying of $k1$ and adding the new carry register to S. The drawbacks of trial addition are requiring an extra copy instruction, what's worse, an additional vector mask register (only 8 vector mask registers in a core [26]).

We propose an artistic method to compute q by using write-mask vector. The right counting process of q in VCPC method is $q \leftarrow (S[0] + k1[0]) \cdot \mu$, we rewrite it as $q \leftarrow S[0] \cdot \mu + k1[0] \cdot \mu$. We implements this formula by following two instructions:

$$Q \leftarrow Mullow(S,\ U)$$
$$Q \leftarrow Vadd(Q,\ U)\{k1\}$$

Vadd is a normal vector addition instruction without carry. We use $k1$ as write-mask for *Vadd*. if $k1[0]$ is 1, $Q[0]$ will add a μ; if $k1[0]$ is 0, $Q[0]$ will not be modified, so the value of q is correct. As the write mask is read-only, it would

not be modified. Our method does not require an extra move instruction, also no need for an extra vector mask register. The most interesting idea of our method is using the carry vector as write-mask vector.

4.3 Performance Analysis

In this section, we analyse the performance of our Montgomery multiplication design (VCPC method) and compare with Redundant Representation (RR) method which is presented in [12].

We assume the length of element is w, the number of elements in a vector is n, the length of a vector is s ($s = w * n$) and the length of arguments is l. So the number of rounds in our design is $\lceil l/w \rceil$. And the number of vectors is $\lceil l/s \rceil$. So in every rounds, it needs to perform $2 * \lceil l/s \rceil$ *Mullow*, $2 * \lceil l/s \rceil$ *Mulhigh*, $4 * \lceil l/s \rceil$ *Vadc*, $\lceil l/s \rceil$ *RShift*, 2 operations to compute the intermediate scalar variable q and 2 *Broadcast* (b_i and q), which is equal to $9 * \lceil l/s \rceil + 4$. Therefore, for VCPC method, the total number of instructions is about $\lceil l/w \rceil * (9 * \lceil l/s \rceil + 4)$ (not including handling tail, which does not need many instructions).

RR method has two drawbacks: the first is need double spaces to store all the arguments and temporal variables, the second is the several high bits needed to be reserved for storing carries which can not involve in multiplications. For example, for 1024-bit Montgomery multiplication, RR method divides all the arguments and temporal variables into 29-bit parts for remaining high 6-bit (in 64-bit element) to maintain carries. So RR method need more than double vectors to store the arguments and variables than VCPC method. For RR method, we assume the number of reserve bits is t (in 32-bit element), which generally meets $2 * \lceil l/w \rceil \leqslant 2^t$, otherwise it needs to perform cleanup operation during Montgomery multiplication. The number of rounds in RR method is $\lceil l/(w-t) \rceil$. The number of vectors is $2 * \lceil l/((w-t) * n) \rceil = 2 * \lceil l/(s-t*n) \rceil$. So in every rounds, it need to carry out $4 * \lceil l/(s-t*n) \rceil$ multiplications, $4 * \lceil l/(s-t*n) \rceil$ additions, $2 * \lceil l/(s-t*n) \rceil$ *RShift*, 2 operations to compute q and 2 *Broadcast*, which is equal to $10 * \lceil l/(s-t*n) \rceil + 4$. Hence, for RR method, the total number of instructions is $\lceil l/(w-t) \rceil * (10 * \lceil l/(s-t*n) \rceil + 4)$.

Table 1. Comparison with redundant representation method

	RR method	VCPC method
Vector number	$2 * \lceil l/(s-t*n) \rceil$	$\lceil l/s \rceil$
Instructions/round	$10 * \lceil l/(s-t*n) \rceil + 4$	$9 * \lceil l/s \rceil + 4$
Round	$\lceil l/(w-t) \rceil$	$\lceil l/w \rceil$
Instructions	$\lceil l/(w-t) \rceil * (10 * \lceil l/(s-t*n) \rceil + 4)$	$\lceil l/w \rceil * (9 * \lceil l/s \rceil + 4)$

As presented in Table 1, compared with RR method, VCPC method needs less rounds and less instructions in each round. Consequently, VCPC method

needs less instructions than RR method. For example, we want to compute 1024-bit Montgomery multiplication on Intel Xeon Phi. The element length w is 32, element number in a vector n is 16, the vector length s is 512, the length of arguments l is 1024, the number of reserve bit t is 3 (in 32-bit element). So VCPC method requires 32 rounds, 22 instructions in each round, so that requires about 704 instructions. RR method requires 36 rounds, 34 instructions in each round, so that requires about 1224 instructions. VCPC method only needs a factor of 0.58 instructions than RR method.

5 Implementation

In this section, we describe the implementations of Montgomery multiplication and RSA on Intel Xeon Phi. We choose assembly language instead of intrinsics in C language to implement Montgomery multiplication for fully controlling registers. Besides, we choose native execution mode [26] for our implementations as the ultimate performance can be evaluated in this mode.

5.1 Montgomery Multiplication Implementation

We implement 512-bit, 1024-bit and 2048-bit Montgomery multiplication on Intel Xeon Phi. Although our design provides the scheme with minimal instruction number, the implementation must be optimized to make the execution cycle approach to the instruction number. Two implementation issues are mainly concerned: making VPUs fully pipelined and maintaining carry bits in vector mask registers.

(1) Making VPUs Fully Pipelined. Data-dependencies in the instruction flow may cause pipeline stalls of VPUs. If an instruction about to be executed has to wait for the operands written by the previous instruction for several cycles, in the meantime no other instructions enter the pipeline, the cycles of VPUs will be wasted and performance will be compromised. First of all, we need to investigate the latency of instructions we used. As presented in [15], most vector instructions are four-cycle latency. We measure vector instruction latency by ourselves. The assessment results are presented in Table 2.

Table 2. The latencies of vector instructions on Intel Xeon Phi

Instruction	vpmulhud	vpmulld	vpadcd	vpermd	valignd
Cycles	4	4	4	6	7

As every core of Intel Xeon Phi has four hyperthreads, the four-cycle instructions (*vpmulhud*, *vpmulld* and *vpadcd*) can be fully pipelined, even though the instructions are data-dependent. But data-dependent *vpermd* and *valignd* are

not easily pipelined. We observe that if *vpermd* and *valignd* do not use the data produced by the prior instruction, they can be fully pipelined. As the cores of Intel Xeon Phi are in-order, every vector instruction will be performed in terms of the sequence in the assembly code. So we manually adjust the sequence of instructions in our Montgomery multiplication implementation. The assembly code of one round in 512-bit Montgomery multiplication is presented in ASM Code 1. We insert the data-independent instructions (green ones) into the positions of pipeline stalls (red ones). Then we carry out four threads on one core as each thread executes ASM Code 1 repetitively, the result shows that the execution only requires 12.2 cycles per round which means the utilization of VPUs reaches 98%.

ASM Code 1	Adjusted
1: *vpmulld*	%*zmm*2{*aaaa*}, %*zmm*1, %*zmm*10
2: *vpadcd*	%*zmm*10, %*k*0, %*zmm*0
3: *vpmulld*	%*zmm*0, %*zmm*4, %*zmm*6
4: *vpaddd*	%*zmm*6, %*zmm*4, %*zmm*6{%*k*2}
5: *vpmulhud*	%*zmm*2{*aaaa*}, %*zmm*1, %*zmm*11
6: *vpermd*	%*zmm*6, %*zmm*5, %*zmm*6
7: *vpmulld*	%*zmm*6, %*zmm*3, %*zmm*12
8: *vpadcd*	%*zmm*12, %*k*2, %*zmm*0
9: *vpmulhud*	%*zmm*6, %*zmm*3, %*zmm*13
10: *valignd*	$1, %*zmm*0, %*zmm*5, %*zmm*0
11: *vpadcd*	%*zmm*11, %*k*1, %*zmm*0
12: *vpadcd*	%*zmm*13, %*k*3, %*zmm*0

(2) Maintaining Carry Bits in Vector Mask Registers. As 2048-bit Montgomery multiplication has sixteen VCPCs, it will produce sixteen carry vector every round. However, each core of Intel Xeon Phi has only eight vector mask registers. Although the instruction *kmov* can move data between vector mask registers and general purpose registers, frequent exchanging data will rouse gigantic performance loss. So maintaining carry bits in vector mask registers is essential. We split 2048-bit Montgomery multiplication implementation into four parts and every parts is similar to 1024-bit Montgomery multiplication implementation but without handling tail phase. Outside of these four parts, we need to handle carry bits two times. As every parts have eight VCPCs and the computation of scalar variable q will not break the flow of VCPCs, all carry bits in every round can be kept in vector mask registers.

5.2 RSA Implementation

We apply our Montgomery multiplication implementation to realize PhiRSA based on CRT method [18], which computes m-bit RSA by performing two $(m/2)$-bit Montgomery exponentiations. We also utilize m-ary method [17] to

accelerate Montgomery exponentiations with the precomputed table. 2^5-ary method is applied for 1024-bit RSA and 2^6-ary method is applied for 2048-bit RSA and 4096-bit RSA. To complete CRT method, we implement a schoolbook multiplication, addition and subtraction on Intel Xeon Phi. The differences between implementations of multiplication and Montgomery multiplication are that the multiplication implementation has only two VCPCs, don't need to shift right every round and must save the double size product. Our multiplication implementation is very efficient as it also make VPUs fully pipelined.

6 Experimental Results

In this section, we conduct the experiments to evaluate our Montgomery multiplication implementations and RSA implementations on Intel Xeon Phi 7120P processor (1.33 GHz), and compare with the other studies on Xeon Phi, CPUs and GPUs. The configurations of our evaluation platform are described as follows: the coprocessor is Intel Xeon Phi 7120P, the host CPU is Intel Xeon E5 2697v2, the operating system is RedHat 6.4, and the compiler is Intel Composer XE 2013.

6.1 Implementation Result

We execute 244 threads running on 61 cores and bind four threads to each core for 4-way hyper-threading to avoid the performance loss of the thread migration. *Vector Instruction Number* in table indicates the number of assembly instructions and *Execution Cycles* denotes the real execution time. If VPUs reach the maximum performance that one instruction per cycle, *VPU Utilization* is 100%. We also evaluate the throughput and the latency. *Throughput/Thread* denotes the performance of one thread, which is equal to *Throughput*/244. Table 3 summarizes the performance of 512-bit, 1024-bit and 2048-bit Montgomery multiplication implementations. It shows that VPU Utilizations of all the implementations are above 92%. Note that, *Execution Cycles* are almost four times of *Vector Instruction Number* which dues to four threads performing on one core for pipelining. So *VPU Utilization* are equal to $4(VectorInstructionNumber)/(ExecutionCycles)$. Table 4 shows evaluation results of 1024-bit, 2048-bit and 4096-bit RSA. VPU Utilizations of RSA implementations are also above 90%.

6.2 Comparisons with the Previous Works on Intel Xeon Phi

(1) Comparison with the Implementation of Redundant Representation. The work in [16] applies redundant representation method to implement multiplication, which only provides the number of instructions. As the computation of the schoolbook multiplication is about one half of Montgomery multiplication, we double the instruction number in [16] for a rough comparison.

Table 5 shows that our implementation needs no more than one tenth of instructions compared with their implementation. The major reason is that our Montgomery multiplication design requires less instructions than redundant representation method inherently. Another reason is this generation Intel Xeon Phi (Knights Corner, KNC) does not support the multiplication instruction like *vpmuludq* in AVX2 [14] which needed in redundant representation method. So our design is not only better than redundant representation method but more suitable for Intel Xeon Phi (KNC).

Table 3. Performance of Montgomery multiplication on Intel Xeon Phi

	Montgomery multiplication		
	512-bit	1024-bit	2048-bit
Thread number	244	244	244
Core number	61	61	61
Vector instruction number	218	724	2797
Execution cycles	948	3076	12211
VPU utilization	92%	94%	92%
Throughput ($10^6/s$)	343.78	105.73	26.64
Throughput/thread ($10^6/s$)	1.41	0.43	0.11
Latency (µs)	0.71	2.31	9.16

Table 4. Performance of RSA decryption on Intel Xeon Phi

	RSA decryption					
	1024-bit		2048-bit		4096-bit	
	Window size: 5		Window size: 6		Window size: 6	
Thread number	1	244	1	244	1	244
Core number	1	61	1	61	1	61
Vector instruction number (10^6/op)	0.28	0.28	1.82	1.82	13.7	13.7
Execution cycles (10^6/op)	0.91	1.26	3.97	7.78	29.71	60.66
VPU utilization	31%	90%	46%	94%	46%	90%
Throughput (/s)	1466	258370	336	41803	45	5358
Throughput/thread ($/s$)	1466	1059	336	171	45	22
Latency (ms)	0.68	0.94	2.98	5.84	22.29	45.54

Table 5. Comparisons with the implementation of redundant representation on Intel Xeon Phi

	512-bit MontMul (instructions)	1024-bit MontMul (instructions)	2048-bit MontMul (instructions)
Keliris et al. [12] (Scaled)	3846	9498	28776
Our VCPC method	218	724	2797

(2) Comparison with the Implementation of Carry Propagation. The work in [7] firstly uses carry propagation to implement multiplication and RSA on Intel Xeon Phi 5110P. As described in Table 6, the throughput in [7] is scaled to Intel Xeon Phi 7120P. Our implementations achieve 4.1 to 8.5 times performance of the scaled results. There are three possible reasons: (1) the *Extract* and *Store* operations are very cost; (2) using multiplication to compute Montgomery multiplication is not the best way; (3) intrinsics in C language can not fully control registers.

Table 6. Comparison with the implementation of carry propagation on Intel Xeon Phi

	512-bit RSA-1024 throughput (/s)	1024-bit RSA-2048 throughput (/s)	2048-bit RSA-4096 throughput (/s)
Chang et al. [7] (Scaled)	1310	7217	30282
Our VCPC method	5358	41803	258370

6.3 Comparisons with the Implementations on CPUs and GPUs

Table 7 shows the comparisons with the best implementations on CPUs and GPUs. Compared with CPU implementation in OpenSSL [23] which evaluated on Intel i7 4770R, the throughput of our implementation is about 40 times of a single CPU core, and the latency is about 5 times. The throughput of one core on Intel Xeon Phi is about a factor of 0.6 compared with one CPU core, which is due to the higher frequency of CPU core (3.2 GHz). On GPU platform, the integer implementation in [31] has the lowest latency so far which evaluated on NVIDIA GT 750m, and the floating-pointing implementation in [32] has the highest throughput until now which evaluated on NVIDIA GTX Titan. Compared with [31], the throughput of our implementation is about 7 times, and the latency is no more than 90%. And compared with [32], the throughput of our implementation is about 1.07 times, and the latency is only 26%. So PhiRSA has the advantage on achieving high throughput and small latency simultaneously.

Table 7. Comparisons with the implementations on CPUs and GPUs

RSA decryption	OpenSSL 1.0.1f [23]	Yang et al. [31]	Zheng et al. [32]	Our native implementations
Platform	Intel Haswell i7 4770R	NVIDIA GT 750m	NVIDIA GTX Titan	Intel Xeon Phi 7120P
Core number	4	384	2688	61
Frequency (GHz)	3.2	0.967	0.836	1.33
Computing power (SP GFLOPS)	410	743	4500	2600
RSA-1024 throughput (/s)	25850	34981	-	234981
RSA-2048 throughput (/s)	3427	5244	38975	41803
RSA-4096 throughput (/s)	485	-	-	5358
RSA-1024 latency (ms)	0.16	2.6	-	1.04
RSA-2048 latency (ms)	1.17	6.5	22.47	5.84
RSA-4096 latency (ms)	8.26	-	-	45.54

7 Conclusions

In this contribution, we propose a novel vector-oriented Montgomery multiplication design and implementation to fully exploit the computing power of vector instructions on Intel Xeon Phi. Based on the above, we implement RSA named PhiRSA. PhiRSA is much better than the existing RSA implementations on Intel Xeon Phi which attains 4.1 to 8.5 times performance. Our results also demonstrate that Intel Xeon Phi can be used to achieve both high throughput and small latency for RSA. On Intel Xeon Phi 7120P, PhiRSA achieves high throughput comparable to the implementations on GPUs but with much less parallel tasks, and small latency comparable to the implementations on CPU. PhiRSA and our Montgomery multiplication implementation can be applied to implement other cryptographic algorithms as primitives. We will also integrate PhiRSA into OpenSSL in the future.

A Appendix

Algorithm 3 is vector length Montgomery multiplication.

Algorithm 3. Vector length Montgomery multiplication

Input: 2^w is radix, n is element number, vector size is $s = n * w$
$R = 2^s$, Modulus M is s-bit number, $M < R$,
$\gcd(M,\ R) = 1$, $\mu = -M^{-1} \bmod 2^w$
$A,\ B$ are s-bit number, $0 \leqslant A, B < M$, $B = \sum_{i=0}^{n-1} b_i 2^{iw}$

Output: $S = A \cdot B \cdot R^{-1} \pmod M$, $0 \leqslant S < M$

$k0 \leftarrow 0,\ k1 \leftarrow 0,\ k2 \leftarrow 0,\ k3 \leftarrow 0$
$S \leftarrow 0,\ Zero \leftarrow 0,$
$U \leftarrow Broadcast(\mu)$
/* **VCPC Phase** */
for i from 0 to $n-1$ **do**
  $T \leftarrow Broadcast(b_i)$
  /* $VCPC\ 1: S + k0 = S + k0 + Low(A \cdot b_i)$ */
  $L \leftarrow Mullow(A,\ T)$
  $(S,\ k0) \leftarrow Vadc(S,\ k0,\ L)$
  /* $q \leftarrow (S[0] + k1[0]) \cdot \mu$ */
  $Q \leftarrow Mullow(S,\ U)$
  $Q \leftarrow Vadd(Q,\ U)\{k1\}$
  $Q \leftarrow Broadcast(Q[0])$
  /* $VCPC\ 2: S + k1 = S + k1 + Low(M \cdot q)$ */
  $L \leftarrow Mullow(M,\ Q)$
  $(S,\ k1) \leftarrow Vadc(S,\ k1,\ L)$
  /* $Right\ Shift: S = S \gg 1\ element$ */
  $S \leftarrow RShift(Zero,\ S,\ 1)$
  /* $VCPC\ 3: S + k2 = S + k2 + High(A \cdot b_i)$ */
  $H \leftarrow Mulhigh(A,\ T)$
  $(S,\ k2) \leftarrow Vadc(S,\ k2,\ H)$
  /* $VCPC\ 4: S + k3 = S + k3 + High(M \cdot q)$ */
  $H \leftarrow Mulhigh(M,\ Q)$
  $(S,\ k3) \leftarrow Vadc(S,\ k3,\ H)$
end for
/* **Tail Phase** */
/* *Handling carry* */
$(T,\ k0) \leftarrow Vadc(Zero,\ k0,\ Zero)$
$(S,\ k1) \leftarrow Vadc(S,\ k1,\ T)$
$(T,\ k1) \leftarrow Vadc(Zero,\ k1,\ Zero)$
$(T,\ k2) \leftarrow Vadc(T,\ k2,\ Zero)$
$H \leftarrow Rshift(Zero,\ S,\ 1)$
$(H,\ k3) \leftarrow Vadc(H,\ k3,\ T)$
for i from 0 to $n-2$ **do**
  if $k3 = 0$ **then**
    $BREAK$
  end if
  $(H,\ k3) \leftarrow Vadc(H,\ k3,\ Zero)$
  $k3 \leftarrow Lmove(k3)$
end for
$S \leftarrow Rshift(S,\ zero,\ 1)$
$S \leftarrow Rshift(H,\ S,\ n-1)$
/* *Reducing* */
if $H[n-1] = 1$ **then**
  $(S,\ k0) \leftarrow Vsbb(S,\ k0,\ M)$
  $H \leftarrow Rshift(Zero,\ H,\ n-1)$
  $H \leftarrow Rshift(H,\ S,\ 1)$
  for i from 0 to n-2 **do**
    if $k0 = 0$ **then**
      $BREAK$
    end if
    $(H,\ k0) \leftarrow Vsbb(H,\ k0,\ Zero)$
    $k0 \leftarrow Lmove(k0)$
  end for
end if
$S \leftarrow Rshift(S,\ zero,\ 1)$
$S \leftarrow Rshift(H,\ S,\ n-1)$
return S.

References

1. 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. Workshop rec. SHARCS **9**, 131–144 (2009)
2. 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). doi:10.1007/978-3-642-01001-9_28
3. Bernstein, D.J., Chuengsatiansup, C., Lange, T.: Curve41417: Karatsuba revisited. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 316–334. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44709-3_18
4. Bos, J.W.: High-performance modular multiplication on the cell processor. In: Hasan, M.A., Helleseth, T. (eds.) WAIFI 2010. LNCS, vol. 6087, pp. 7–24. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13797-6_2
5. Bos, J.W., Kaihara, M.E.: Montgomery multiplication on the cell. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 477–485. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14390-8_50
6. Bos, J.W., Montgomery, P.L., Shumow, D., Zaverucha, G.M.: Montgomery multiplication using vector instructions. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 471–489. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43414-7_24
7. Chang, C., Yao, S., Yu, D.: Vectorized big integer operations for cryptosystems on the Intel mic architecture. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), pp. 194–203. IEEE (2015)
8. Chen, J., Watson, W., Chen, M.F.: Efficient GCD computation for big integers on Xeon Phi coprocessor. In: 2014 9th IEEE International Conference on Networking, Architecture, and Storage (NAS), pp. 113–117. IEEE (2014)
9. Fang, J., Varbanescu, A.L., Sips, H., Zhang, L., Che, Y., Xu C.: An empirical study of Intel Xeon Phi. arXiv preprint arXiv:1310.5842 (2013)
10. Faz-Hernández, A., López, J.: Fast implementation of curve25519 using AVX2. In: Lauter, K., Rodríguez-Henríquez, F. (eds.) LATINCRYPT 2015. LNCS, vol. 9230, pp. 329–345. Springer, Cham (2015). doi:10.1007/978-3-319-22174-8_18
11. Grabher, P., Großschädl, J., Page, D.: On software parallel implementation of cryptographic pairings. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 35–50. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04159-4_3
12. Gueron, S., Krasnov, V.: Software implementation of modular exponentiation, using advanced vector instructions architectures. WAIFI **12**, 119–135 (2012)
13. Intel: Intel Xeon Phi coprocessor instruction set architecture reference manual (2012). https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf
14. Intel: Intel 64 and IA-32 architectures software developer's manual, vol. 2 (2a, 2b and 2c): Instruction set reference, a-z (2015)
15. Jeffers, J., Reinders, J.: Intel Xeon Phi coprocessor high-performance programming. Newnes (2013)
16. Keliris, A., Maniatakos, M.: Investigating large integer arithmetic on Intel Xeon Phi SIMD extensions. In: 2014 9th IEEE International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS), pp. 1–6. IEEE (2014)
17. Knuth, D.E.: Seminumerical algorithms, the art of computer programming, vol. 2 (1981)

18. Koç, Ç.K.: High-speed RSA implementation. Technical report, RSA Laboratories (1994)
19. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and comparing montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996)
20. Montgomery, P.L.: Modular multiplication without trial division. Math. comput. **44**(170), 519–521 (1985)
21. Moore, S.: Using streaming SIMD extensions (SSE2) to perform big multiplications. Application note AP-941, Intel Corporation 2000, version 2.0. Order (248606-001) (2000)
22. NVIDIA. Cuda C programming guide 7.5 (2015). http://docs.nvidia.com/cuda/cuda-c-programming-guide/
23. OpenSSL. The open source toolkit for SSL/TLS (2015)
24. Page, D., Smart, N.P.: Parallel cryptographic arithmetic using a redundant montgomery representation. IEEE Trans. Comput. **53**(11), 1474–1482 (2004)
25. Pennycook, S.J., Hughes, C.J., Smelyanskiy, M., Jarvis, S.A.: Exploring SIMD for molecular dynamics, using Intel Xeon Processors and Intel Xeon Phi coprocessors. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1085–1097. IEEE (2013)
26. Rahman, R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, New York (2013)
27. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8384, pp. 559–570. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55224-3_52
28. Seo, H., Liu, Z., Großschädl, J., Choi, J., Kim, H.: Montgomery modular multiplication on ARM-NEON revisited. In: Lee, J., Kim, J. (eds.) ICISC 2014. LNCS, vol. 8949, pp. 328–342. Springer, Cham (2015). doi:10.1007/978-3-319-15943-0_20
29. Seo, H., Liu, Z., Kim, H.: Efficient arithmetic on arm-neon and its application for high-speed RSA implementation
30. Wikipedia. List of NVIDIA graphics processing units (2015). https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
31. Yang, Y., Guan, Z., Sun, H., Chen, Z.: Accelerating RSA with fine-grained parallelism using GPU. In: Lopez, J., Wu, Y. (eds.) ISPEC 2015. LNCS, vol. 9065, pp. 454–468. Springer, Cham (2015). doi:10.1007/978-3-319-17533-1_31
32. Zheng, F., Pan, W., Lin, J., Jing, J., Zhao, Y.: Exploiting the floating-point computing power of GPUs for RSA. In: Chow, S.S.M., Camenisch, J., Hui, L.C.K., Yiu, S.M. (eds.) ISC 2014. LNCS, vol. 8783, pp. 198–215. Springer, Cham (2014). doi:10.1007/978-3-319-13257-0_12