2015 IEEE International Parallel and Distributed Processing Symposium Workshops

Bulk GCD Computation Using a GPU to Break Weak RSA Keys

Toru Fujita, Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—RSA is one the most well-known public-key cryptosystems widely used for secure data transfer. An RSA encryption key includes a modulus n which is the product of two large prime numbers p and q. If an RSA modulus n can be decomposed into p and q, the corresponding decryption key can be computed easily from them and the original message can be obtained using it. RSA cryptosystem relies on the hardness of factorization of RSA modulus. Suppose that we have a lot of encryption keys collected from the Web. If some of them are inappropriately generated so that they share the same prime number, then they can be decomposed by computing their GCD (Greatest Common Divisor). Actually, a previously published investigation showed that a certain ratio of RSA moduli in encryption keys in the Web are sharing prime numbers. We may find such weak RSA moduli n by computing the GCD of many pairs of RSA moduli. The main contribution of this paper is to present a new Euclidean algorithm for computing the GCD of all pairs of encryption moduli. The idea of our new Euclidean algorithm that we call Approximate Euclidean algorithm is to compute an approximation of quotient by just one 64-bit division and to use it for reducing the number of iterations of the Euclidean algorithm. We also present an implementation of Approximate Euclidean algorithm optimized for CUDA-enabled GPUs. The experimental results show that our implementation for 1024-bit GCD on GeForce GTX 780Ti runs more than 80 times faster than the Intel Xeon CPU implementation. Further, our GPU implementation is more than 9 times faster than the best known published GCD computation using the same generation GPU.**

I. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1], [4]–[7]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], [9], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [10], since they have thousands of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [8]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [5], [6], [9]–[11]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the shared memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the throughput between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory. Also, the latency of the global memory access is several hundred clock cycles, while that of the shared memory access is quite small [8]. Hence, we should minimize the memory access to the global memory to maximize the performance.

RSA [12] is the most well-known public-key cryptosystem widely used for secure data transfer. RSA cryptosystem uses an encryption key open to the public and a secret decryption key. An encryption key is a pair (n, e) of modulus n and exponent e such that $n = pq$ for two distinct large prime numbers p and q and e ($< (p-1)(q-1)$) and $(p-1)(q-1)$ are coprime. For example, for 1024-bit RSA cryptosystem, modulus n with 1024 bit obtained by 512-bit prime numbers p and q. The decryption key for this encryption key is a pair (n, d) such that $de = 1 \pmod{(p-1)(q-1)}$, that is, d is the multiplicative inverse of $e \pmod{(p-1)(q-1)}$. For a public encryption key (n, e), a message M ($0 \leq M \leq n-1$) is converted to the cipher message $C = M^e \bmod n$. Since $M = M^{ed} \bmod n$ always holds for all message M, the cipher message C can be converted to the original message M by computing $C^d \bmod n$. If the values of p and q are available, $d = e^{-1} \pmod{(p-1)(q-1)}$ can be computed very easily by extended Euclidean algorithm [13]. However, to obtain p and q from an encryption key (n, e), we need to decompose n into p and q. Since the computation of factorization is very costly, it is not possible to decompose n into p and q in a practical computing time. RSA cryptosystem relies on the hardness of factorization of a large number.

Suppose that we have a set of many RSA encryption keys collected in the Web. If some of moduli in encryption keys are generated by inappropriate implementation of a random prime number generator, they may share or reuse the same prime number. We call RSA keys share a prime number *weak*

/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.54

RSA keys. Actually, several public keys collected from the Web includes weak RSA keys [14]. If two moduli share a prime number, they can be decomposed by computing the GCD (Greatest Common Divisor). More specifically, if two distinct moduli n_1 and n_2 share a moduli p then the GCD of n_1 and n_2 is equal to p. It is well known that the GCD can be computed very easily by Euclidean algorithms [15]. Once we have the GCD p, we can decompose n_1 into p and $\frac{n_1}{p}$ and n_2 into p and $\frac{n_2}{p}$. Hence, we may break weak RSA keys by computing the GCDs of all pairs of two moduli in the Web. The main goal of this paper is to develop an efficient algorithm for computing the GCD to break weak RSA keys using the GPU.

It is well known that Euclidean algorithm [15] can compute the GCD of two numbers very efficiently. Original Euclidean algorithm repeats modulo computation of two numbers until one of them reaches zero and the other one stores the GCD. However, modulo computation of large numbers takes a lot of time. Hence, Binary Euclidean algorithm [16], which does not use modulo computation, is often used to compute the GCD. Basically, Binary Euclidean algorithm repeats subtraction of two numbers and arithmetic shifts until one of them reaches zero. Binary Euclidean algorithm needs more iterations than original Euclidean algorithm, but the computation of each iteration of Binary Euclidean algorithm takes less time than that of original Euclidean algorithm. Totally, Binary Euclidean algorithm runs faster than original Euclidean algorithm, and it is commonly used to compute the GCD.

The main contribution of this paper is to present a new Euclidean algorithm for computing the GCD. The idea of our new Euclidean algorithm that we call *Approximate Euclidean algorithm* is to compute a good approximation of quotient by simple 64-bit division and to use it for reducing the number of iterations of the Euclidean algorithm. It runs much faster than original Euclidean algorithm and Binary Euclidean algorithm. We also present an implementation of Approximate Euclidean algorithm optimized for CUDA-enabled GPUs.

In our previous papers [17], [18], we have shown that the bulk execution of an oblivious sequential algorithm can be implemented in CUDA-enabled GPUs very efficiently. A sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. *The bulk execution* of a sequential algorithm is to execute it for many different inputs in turn or at the same time. Suppose that each input of the bulk execution is assigned to a CUDA thread and each CUDA thread executes the sequential algorithm for an assigned input. Since each thread access the same address at each time unit, memory access to the global memory is coalesced if the input and the work space is arranged in the CUDA global memory in column-wise. Hence, this implementation runs very fast.

We will show that our Approximate Euclidean algorithm is *semi-oblivious* in the sense that an address accessed at each of almost all time units. In other words, semi-oblivious algorithm may access different addresses in few time units. If each of the CUDA threads executes a semi-oblivious sequential algorithm for the bulk execution, then they may perform non-coalesced access to the global memory. However, if the ratio of non-coalesced access is small enough, the bulk execution of a semi-oblivious sequential algorithm still runs efficiently on the GPU. We will show that Approximation Euclidean algorithm is semi-oblivions, and their implementations on CUDA-enabled GPUs run fast.

We have also implemented Euclidean algorithms on Xeon X7460 (2.66GHz) CPU and GeForce GTX 780 Ti GPU. The experimental results show that, for computing the GCD of two RSA moduli, Approximate Euclidean algorithm runs faster than the other Euclidean algorithms both on the Intel CPU and the GPU. More specifically, Approximate Euclidean algorithm for computing 1024-bit GCD runs 1.68 times and 8.46 times faster than Binary Euclidean algorithm on the CPU and on GPU, respectively. Also, Approximation Euclidean algorithm for 1024-bit GCD on the GPU achieves a speedup of factor 81.7 over the CPU implementation.

Several previously published papers have presented GPU implementations of Binary Euclidean algorithm in CUDA-enabled GPU. Fujimoto [19] has implemented Binary Euclidean algorithm using CUDA and evaluated the performance on GeForce GTX285 GPU. The experimental results show that the GCDs for 131072 pairs of 1024-bit numbers can be computed in 1.431932 seconds. Hence, his implementation runs 10.9 microseconds per one 1024-bit GCD computation. Scharfglass et al. [20] have presented a GPU implementation of Binary Euclidean algorithm. It performs the GCD computation of all 199990000 pairs of 20000 RSA moduli with 1024 bits in 2005.09 seconds using GeForce GTX 480 GPU. Thus, their implementation performs each 1024-bit GCD computation in 10.02 microseconds. Quite recently, White [21] has showed that the same computation can be performed in 63.0 seconds on Tesla K20Xm. It follows that it computes each 1024-bit GCD in 3.15 microseconds. Our implementation of Approximate Euclidean algorithm can perform each 1024-bit GCD computation in 0.346 microseconds on GeForce GTX 780Ti. Hence, it runs 9 times faster than the previously published best known implementation using the same generation GPU.

This paper is organized as follows. We first review several Euclidean algorithms in Section II. We then go on to present our Approximate Euclidean algorithm that computes an approximation of quotient of two numbers in Section III. In Section IV, we show an efficient implementation of Euclidean algorithms for large numbers. Since large numbers must be stored in multiple words in the memory, we must be careful to reduce the number of memory access. We show that each iteration of Euclidean algorithm can be done by reading two numbers and writing one number. Section V shows Euclidean algorithms for computing the GCD for RSA moduli. Since RSA moduli are the product of two prime numbers, we can modify Euclidean algorithms for any number to handle only RSA moduli. We also evaluate the number of iterations of Euclidean algorithms and show that Approximate Euclidean algorithm performs fewer iterations. In Section VI, we show how we implement Euclidean algorithms in CUDA-enabled GPUs for breaking weak RSA keys. Finally, we show experimental results of the performance of Euclidean algorithms both on the CPU and on the GPU in Section VII. They show that Approximate Euclidean algorithm run faster than the others. Section VIII concludes our work.

II. Euclidean Algorithms for GCD

The main purpose of this section is to review classical Euclidean algorithm for computing the GCD of two numbers X and Y. For simplicity, we assume that both inputs X and Y are odd and $X \geq Y$ holds. Hence, the GCD of X and Y is always odd. If one of them is odd and the other is even, say X is odd and Y is even, then gcd(X, Y)=gcd$(X, \frac{Y}{2})$ holds. Thus, we can convert Y into odd numbers by removing consecutive 0 bits from the least significant bit of Y. Also, from gcd(X, Y)=2$\cdot$ gcd$(\frac{X}{2}, \frac{Y}{2})$, we can obtain a factor of 2 in the GCD of X and Y if both X and Y are even. Thus, it should have no difficulty to modify all GCD algorithms shown in this paper to handle even input numbers. For later reference, let s denote the number of inputs bits of X and Y.

Let swap(X,Y) denote a function to exchange the values of X and Y. We can write a standard Euclidean algorithm for computing the GCD of X and Y as follows:

```
[Original Euclidean algorithm]
gcd(X,Y){
    do {
        X ← X mod Y; // X < Y always holds
        swap(X,Y); // X > Y always holds
    } while(Y ≠ 0)
    return(X);
}
```

We will show that Original Euclidean algorithm runs no more than $2s$ iterations of the do-loop. If $X < 2Y$, then X will store $X-Y$, which is less than $\frac{X}{2}$. Otherwise, X will store the value less than Y, which is no more than $\frac{X}{2}$. Hence, the value of X is halved or smaller and thus the number of bits in X is decreased by one or more. Since the number of bits of one of the two numbers is decreased by one, Original Euclidean algorithm performs no more than $2s$ iterations.

Since modulo computation is costly, Binary Euclidean algorithm, which does not execute it, is often used to compute the GCD efficiently:

```
[Binary Euclidean algorithm]
gcd(X,Y){
    do{
        if(X is even) X ← X/2;
        else if(Y is even) Y ← Y/2;
        else X ← (X−Y)/2 // X − Y is always even
        if(X < Y) swap(X, Y);
    } while (Y ≠ 0)
    return(X);
}
```

If X (or Y) is even, then the number of bits in X (or in Y) is decreased by one. If both X and Y are odd, the number of bits in X is decreased by one or more. Hence, the number of iterations of the do-loop of Binary Euclidean algorithm is also no more than $2s$.

For the reader's benefits, we use both the decimal system and the binary system to represent numbers. For example, numbers 223 in the decimal system or 11011111 in the binary system is denoted by "223", "1101,1111", or "1101,1111 (223)." Note that Binary Euclidean algorithm removes 0 in the least significant bit. We can reduce the number of iterations of the do-loop by removing consecutive 0 bits. Let rshift(X) be a function returning the number obtained by removing consecutive 0 bits from the least significant bit of X. For example, if rshift$(1101, 0100) = 0011, 0101$. Note that X is always odd after executing rshift(X) if it is non-zero. Using the rshift function, we can accelerate Binary Euclidean algorithm as follows:

```
[Fast Binary Euclidean algorithm]
gcd(X,Y){
    do {
        X ←rshift(X − Y);
        if(X < Y) swap(X, Y)
    } while (Y ≠ 0)
    return(X);
}
```

In each iteration of Fast Binary Euclidean algorithm, the number of bits in X or in Y can be decreased by one or more. Hence, for any input numbers, the number of iterations of the do-while loop of Fast Binary Euclidean algorithm is no larger than that of Binary Euclidean algorithm.

Table I shows an example of computation performed by Binary Euclidean algorithm and Fast Binary Euclidean algorithm for $X = 1111, 1110, 1101, 1100, 1011(1043915)$ and $Y = 1011, 1011, 1011, 1011, 1011(768955)$. We can confirm that the output $0101(5)$, which is equal to the GCD of X and Y. Euclidean algorithm computes the GCD in 24 iterations, while Fast Euclidean Algorithm runs only 16 iterations.

Using the idea of removing consecutive 0 bits used in Fast Binary Euclidean Algorithm, we can accelerate Original Euclidean algorithm. Let " div " denote quotient operator such that $X \text{ div } Y = \lfloor \frac{X}{Y} \rfloor$, that is, the rounded-down integer of $\frac{X}{Y}$. Clearly, we have $X \bmod Y = X - Y \cdot (X \text{ div } Y)$. Thus, we can rewrite the original Euclidean algorithm as follows:

```
[Original Euclidean algorithm using div ]
gcd(X,Y){
  do {
    Q ← X div Y;
    X ← X − Y · Q;
    swap(X,Y);
  } while(Y ≠ 0);
  return(X);
}
```

If X is even, then we can reduce the number of bits in X by rshift(X). Since X and Y are odd, $X - Y \cdot Q$ is even when Q is odd. However, if Q is even then $X - Y \cdot Q$ is odd and rshift does not remove 0 bits. Hence, we should decrease Q by one if Q is even. Using this idea, we can further accelerate Original Enclidean algorithm as follows:

```
[Fast Euclidean algorithm]
gcd(X,Y){
  do {
    Q ← X div Y;
```

TABLE I. AN EXAMPLE OF COMPUTATION PERFORMED BY BINARY EUCLIDEAN ALGORITHM AND FAST BINARY EUCLIDEAN ALGORITHM

		Binary Euclidean algorithm	Fast Binary Euclidean algo.
1	X	1111,1110,1101,1100,1011	1111,1110,1101,1100,1011
	Y	1011,1011,1011,1011,1011	1011,1011,1011,1011,1011
2	X	1011,1011,1011,1011,1011	1011,1011,1011,1011,1011
	Y	0010,0001,1001,0000,1000	0100,0011,0010,0001
3	X	1011,1011,1011,1011,1011	0101,1011,1100,0100,1101
	Y	0001,0000,1100,1000,0100	0100,0011,0010,0001
		⋮	
13	X	0001,0010,1001,1101	0101,0101
	Y	1100,0010,0001	0001,1001
14	X	1100,0010,0001	0001,1001
	Y	0011,0011,1110	1111
15	X	1100,0010,0001	1111
	Y	0001,1001,1111	0101
16	X	0101,0100,0001	0101
	Y	0001,1001,1111	0101
17	X	0001,1101,0001	
	Y	0001,1001,1111	
18	X	0001,1001,1111	
	Y	0001,1001	
19	X	1100,0011	
	Y	0001,1001	
20	X	0101,0101	
	Y	0001,1001	
21	X	0001,1110	
	Y	0001,1001	
22	X	0001,1001	
	Y	1111	
23	X	1111	
	Y	0101	
24	X	0101	
	Y	0101	
	X	0101	
	Y	0000	

```
    if(Q is even) Q ← Q − 1
    X ←rshift(X − Y · Q);
    if(X < Y) swap(X,Y);
  } while(Y ≠ 0);
  return(X);
}
```

Note that, X may be larger than Y after executing $X \leftarrow X - Y \cdot Q$. For example, if $X = 15$ and $Y = 7$, then $X \operatorname{div} Y = 2$. Hence, $X = 15 - 7 \cdot (2 - 1) = 8$ and $X > Y$ holds. Thus, we need to compare X and Y and exchange them if $X > Y$, to guarantee that $X \geq Y$ holds for the next iteration.

Table II shows an example of computation performed by Original Euclidean algorithm and Fast Euclidean algorithm for the same input numbers X and Y as Table I. We can see that they perform fewer iterations than Binary Euclidean algorithm and Fast Binary Euclidean algorithm. Also, Fast Euclidean algorithm performs fewer iterations than Original Euclidean algorithm. However, for some input numbers, Fast Euclidean algorithm performs more iterations than Original Euclidean algorithm. For example, if $X = 39$ and $Y = 9$, then the GCD is computed as follows. Original Euclidean algorithm runs 2 iterations: $(39, 9) \rightarrow (9, 3) \rightarrow (3, 0)$. Fast Euclidean algorithm runs 3 iterations: $(39, 9) \rightarrow (12, 9) \rightarrow (9, 3) \rightarrow (3, 0)$. Although such examples exist, Fast Euclidean algorithm takes fewer iterations than Original Euclidean algorithm for most input numbers.

TABLE II. AN EXAMPLE OF COMPUTATION PERFORMED BY ORIGINAL EUCLIDEAN ALGORITHM AND FAST EUCLIDEAN ALGORITHM

		Original Euclidean algorithm		Fast Euclidean Algorithm	
		X&Y	Q	X&Y	Q
1	X	1111,1110,1101,1100,1011	1	1111,1110,1101,1100,1011	1
	Y	1011,1011,1011,1011,1011		1011,1011,1011,1011,1011	
2	X	1011,1011,1011,1011,1011	2	1011,1011,1011,1011,1011	43
	Y	0100,0011,0010,0001,0000		0100,0011,0010,0001	
3	X	0100,0011,0010,0001,0000	1	0100,0011,0010,0001	9
	Y	0011,0101,0111,1001,1011		0111,0101,0011	
4	X	0011,0101,0111,1001,1011	3	0111,0101,0011	11
	Y	1101,1010,0111,0101		1001,1011	
5	X	1101,1010,0111,0101	1	1001,1011	1
	Y	1100,1000,0011,1100		0101,0101	
6	X	1100,1000,0011,1100	10	0101,0101	1
	Y	0001,0010,0011,1001		0010,0011	
7	X	0001,0010,0011,1001	1	0010,0011	1
	Y	0001,0010,0000,0010		0001,1001	
8	X	0001,0010,0000,0010	83	0001,1001	5
	Y	0011,0111		0101	
9	X	0011,0111	1	0101	
	Y	0010,1101		0000	
10	X	0010,1101	4		
	Y	1010			
11	X	1010	2		
	Y	0101			
	X	0101			
	Y	0000			

III. APPROXIMATE EUCLIDEAN ALGORITHM FOR GCD

The main purpose of this section is to show our new Euclidean algorithm called *Approximate Euclidean algorithm*.

Approximate Euclidean algorithm is based on Fast Euclidean algorithm presented in the previous section. The computation of quotient for large numbers performed by Fast Euclidean algorithm is costly. Our new idea is to find a good approximation of quotient by small computing costs. We assume that X and Y are stored in multiple d-bit words, and let $D = 2^d$. Approximate Euclidean Algorithm is described as follows:

```
[Approximate Euclidean Algorithm]
gcd(X, Y){
  do {
    (α, β) ← approx(X, Y);
    if(β = 0){
      if(α is even) α = α − 1; //α is odd
      X ← rshift(X − Y · α); //Y · α is odd
    } else X ← rshift(X − Y · α · D^β + Y); //α · D^β is even
    if(X < Y) swap(X, Y);
  } while (Y ≠ 0);
  return(X);
}
```

In this algorithm, approx(X, Y) is a function to compute a pair (α, β) such that $\alpha \cdot D^\beta$ ($\leq Q$) is a good approximation of $Q = X \operatorname{div} Y$, and the computing cost of approx(X, Y) is much smaller than that of $X \operatorname{div} Y$. Also, to guarantee that X is even, $X - Y \cdot (\alpha \cdot D^\beta - 1)$ is computed if $\alpha \cdot D^\beta$ is even. Note that if $\alpha \cdot D^\beta$ is always 1, that is, $(\alpha, \beta) = (1, 0)$ then Approximate Euclidean algorithm is the same as Fast Binary Euclidean algorithm. Since the value of $\alpha \cdot D^\beta$ can be more than 1, the number of iterations in Approximate Euclidean Algorithm may be smaller than Binary Euclidean algorithms.

We first show the idea of implementation of approx(X, Y). Suppose that X and Y are represented by l_X d-bit words $x_1x_2\cdots x_{l_X}$ and l_Y d-bit words $y_1y_2\cdots y_{l_Y}$. In other words,

$$X = x_1D^{l_X-1} + x_2D^{l_X-2} + \cdots + x_{l_X}D^0$$

and

$$Y = y_1D^{l_X-1} + y_2D^{l_X-2} + \cdots + y_{l_Y}D^0$$

hold. It should be clear that $l_X \geq l_Y$ always holds from $X \geq Y$. Let $\langle x_1x_2\rangle (= x_1 \cdot D + x_2)$ and $\langle y_1y_2\rangle (= y_1 \cdot D + y_2)$ be integers represented most significant two d-bit words of X and Y. Basically, approx(X, Y) returns a pair $(\langle x_1x_2\rangle \text{ div } (\langle y_1y_2\rangle + 1), l_X - l_Y)$. Hence, $\alpha \cdot D^\beta = (\langle x_1x_2\rangle \text{ div } (\langle y_1y_2\rangle+1)) \cdot D^{l_X-l_Y}$ is used as an approximation of $Q = X \text{ div } Y$. Also, it is guaranteed that $\alpha \cdot D^\beta \leq Q$. Thus, $X - Y \cdot \alpha \cdot D^\beta$ is always non-negative.

We show an example using 4-bit words, that is, $d = 4$. Let $X = 1101, 1001, 0000, 0011(55555)$, and $Y = 0100, 1101, 0010(1234)$. If this is the case, $l_X = 4$, $l_Y = 3$, $\langle x_1x_2\rangle = 1101, 1001(217)$ and $\langle y_1y_2\rangle = 0100, 1101(77)$. Hence we have, $\langle x_1x_2\rangle \text{ div } (\langle y_1y_2\rangle+1) = 217 \text{ div } (77+1) = 2$ and $l_X - l_Y = 1$. Thus, approx(X, Y) returns $(\alpha, \beta) = (2, 1)$ and we have $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$, which approximates $X \text{ div } Y = 45$.

Using this idea, the following function approx computes a pair (α, β):

```
approx(X, Y){
  if(l_X ≤ 2)
    return (X div Y, 0); // Case 1
  if(l_Y = 1) {
    if(x_1 ≥ y_1)
      return (x_1 div y_1, l_X − 1); // Case 2-A
    else
      return (⟨x_1x_2⟩ div y_1, l_X − 2); // Case 2-B
  }
  if(l_Y = 2) {
    if(⟨x_1x_2⟩ ≥ ⟨y_1y_2⟩)
      return (⟨x_1x_2⟩ div ⟨y_1y_2⟩, l_X − 2);// Case 3-A
    else
      return (⟨x_1x_2⟩ div (y_1 + 1), l_X − 3); // Case 3-B
  }
  if(⟨x_1x_2⟩ > ⟨y_1y_2⟩)
    return (⟨x_1x_2⟩ div (⟨y_1y_2⟩ + 1), l_X − l_Y); // Case 4-A
  if(l_X > l_Y)
    return (⟨x_1x_2⟩ div (y_1 + 1)), l_X − l_Y − 1); // Case 4-B
  return (1, 0); // Case 4-C
}
```

The reader should have no difficulty to confirm that operands of " div " have at most 2 words, that is, $2d$ bits. Also, the resulting value of " div " has at most d bits.

Let us see how approx(X, Y) computes (α, β). It has four cases determined by the values of l_X and l_Y. We will show that function approx outputs a good approximation $\alpha \cdot D^\beta$ of $X \text{ div } Y$ for each cases

Case 1: X has 1 or 2 words

Clearly, Y also has 1 or 2 words from $X \geq Y$. Hence, approx outputs $(X \text{ div } Y, 0)$ and we have $\alpha \cdot D^\beta = X \text{ div } Y$. Example: If $X = 1101, 1111(223)$ and $Y = 0010, 1101(45)$ then approx outputs $(223 \text{ div } 45, 0) = (4, 0)$.

Case 2: X has more than 2 words and Y has 1 word. Case 2 has two sub-cases as follows:

Case 2-A: If $x_1 \geq y_1$ then approx outputs $(x_1 \text{ div } y_1, l_X - 1)$

Example: If $X = 1001, 0010, 1001(2345)$ and $Y = y_1 = 0100(4)$ then $x_1 = 1001(9)$ and $x_1 \geq y_1$ hold. If this is the case, approx outputs $(9 \text{ div } 4, 3 - 1) = (2, 2)$. We can confirm that $\alpha \cdot D^\beta = 2 \cdot 16^2 = 512$ approximates $X \text{ div } Y = 2345 \text{ div } 4 = 586$.

Case 2-B: If $x_1 < y_1$ then approx outputs $(\langle x_1, x_2\rangle \text{ div } y_1, l_X - 2)$

Example: If $X = 0100, 1101, 0010(1234)$ and $Y = 1100(12)$ then $x_1 = 0100(4)$ and $\langle x_1, x_2\rangle = 0100, 1101(77)$ hold. Hence, $x_1 < y_1$ is satisfied and approx outputs $(77 \text{ div } 12, 3 - 2) = (6, 1)$. We can confirm that $\alpha \cdot D^\beta = 6 \cdot 16^1 = 96$ approximates $X \text{ div } Y = 1234 \text{ div } 12 = 102$.

Case 3: X has more than 2 words and Y has 2 words. Case 3 has two sub-cases as follows:

Case 3-A: If $\langle x_1x_2\rangle \geq \langle y_1y_2\rangle$ then approx outputs $(\langle x_1x_2\rangle \text{ div } \langle y_1y_2\rangle, l_X - l_Y)$.

Example: If $X = 1001, 0010, 1001(2345)$ and $Y = 0011, 1011(59)$ then $\langle x_1x_2\rangle = 1001, 0010(146)$. Hence $\langle x_1x_2\rangle \geq \langle y_1y_2\rangle$ is satisfied and approx outputs $(146 \text{ div } 59, 3 - 2) = (2, 1)$.

We can confirm that $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$ approximates $X \text{ div } Y = 2345 \text{ div } 59 = 39$.

Case 3-B: If $\langle x_1x_2\rangle < \langle y_1y_2\rangle$ then approx outputs $(\langle x_1x_2\rangle \text{ div } (y_1 + 1), l_X - 3)$.

Example: If $X = 1001, 0010, 1001(2345)$ and $Y = 1110, 0111(231)$ then $\langle x_1x_2\rangle = 1001, 0010(146)$ and $y_1 = 1110(14)$. Since $\langle x_1x_2\rangle < \langle y_1y_2\rangle$ satisfied, approx outputs $(146 \text{ div } (14+1), 3-3) = (9, 0)$. We can confirm that $\alpha \cdot D^\beta = 9 \cdot 16^0 = 9$ approximates $X \text{ div } Y = 2345 \text{ div } 231 = 10$.

Case 4: Both X and Y have more than 2 words. Case 4 has three sub-cases as follows:

Case 4-A: If $\langle x_1x_2\rangle > \langle y_1y_2\rangle$ then approx outputs $(\langle x_1x_2\rangle \text{ div } (\langle y_1y_2\rangle + 1), l_X - l_Y)$. Note that, from $\langle x_1x_2\rangle > \langle y_1y_2\rangle$, we always have $\langle y_1y_2\rangle \leq D^2 - 1$. Hence $\langle y_1y_2\rangle$ has at most $2d$ bits.

Example: If $X = 1101, 0100, 0011, 0001(54321)$ and $Y = 0100, 1101, 0010(1234)$ then $\langle x_1x_2\rangle = 1101, 0100(212)$ and $\langle y_1y_2\rangle = 0100, 1101(77)$ Since $\langle x_1x_2\rangle > \langle y_1y_2\rangle$ is satisfied, approx outputs $(212 \text{ div } (77 + 1), 4 - 3) = (2, 1)$. We can confirm that $\alpha \cdot D^\beta = 2 \cdot 16^1 = 32$ approximates $X \text{ div } Y = 54321 \text{ div } 1234 = 44$.

Case 4-B: If $\langle x_1x_2\rangle \leq \langle y_1y_2\rangle$ and $l_X > l_Y$ then approx outputs $(\langle x_1x_2\rangle \text{ div } (y_1 + 1), l_X - l_Y - 1)$.

Example: If If $X = 1101, 0100, 0011, 0001(54321)$ and $Y = 1111, 1010, 0000(4000)$ then $\langle x_1x_2\rangle = 1101, 0100(212)$ and $\langle y_1y_2\rangle = 1111, 1010(250)$ hold. Hence, $\langle x_1x_2\rangle \leq \langle y_1y_2\rangle$ holds. Since $y_1 = 1111(15)$, approx outputs $(212 \text{ div } (15 + 1), 4-3-1) = (13, 0)$. We can confirm that $\alpha \cdot D^\beta = 13 \cdot 16^0 = 13$ approximates $X \text{ div } Y = 54321 \text{ div } 4000 = 13$.

Case 4-C: If this is the case, $\langle x_1x_2\rangle \leq \langle y_1y_2\rangle$ and $l_X \leq l_Y$ hold. Recall that $X \geq Y$ and thus $\langle x_1x_2\rangle = \langle y_1y_2\rangle$ and $l_X = l_Y$ must be satisfied. Hence the values of X and Y are almost the same and it makes sense to return $(1, 0)$ and $\alpha \cdot D^\beta = 1 \cdot 16^0 = 1$ if this is the case.

TABLE III. AN EXAMPLE OF COMPUTATION PERFORMED BY APPROXIMATE EUCLIDEAN ALGORITHM

		X & Y	CASE	(α, β)
1	X	1111,1110,1101,1100,1011	4-A	$(1, 0)$
	Y	1011,1011,1011,1011,1011		
	X	1011,1011,1011,1011,1011	4-A	$(2, 1)$
	Y	0100,0011,0010,0001		
3	X	1110,0110,1010,1111	4-A	$(3, 0)$
	Y	0100,0011,0010,0001		
4	X	0100,0011,0010,0001	4-B	$(7, 0)$
	Y	0111,0101,0011		
5	X	0111,0101,0011	4-A	$(1, 0)$
	Y	0011,1111,0111		
6	X	0011,1111,0111	3-B	$(3, 0)$
	Y	1101,0111		
7	X	1101,0111	1	$(1, 0)$
	Y	1011,1001		
8	X	1011,1001	1	$(11, 0)$
	Y	1111		
9	X	1111	1	$(3, 0)$
	Y	0101		
	X	0101		
	Y	0000		

Table III shows an example of computation performed by approximate Euclidean algorithm for 4-bit words, that is, $d = 4$ and $D = 16$. It computes the GCD for the same inputs used in Tables I and II in 9 steps. The values used to compute α in approx are underlined. We can confirm that Approximate Euclidean algorithm outputs $0101(5)$, the GCD of X and Y correctly.

Recall that Fast Euclidean algorithm computes the exact value of quotient $Q = X \text{ div } Y$. On the other hand, Approximate Euclidean algorithm uses an approximation $\alpha \cdot D^{\beta}$ of quotient Q. Hence, Approximate Euclidean algorithm may take more iterations than Fast Euclidean algorithm. Actually, from Tables II and III, we can see that Fast Euclidean and Approximate Euclidean performs 8 and 9 iterations, respectively, for the same input numbers.

Further, we should note that approx returns $\beta = 0$ with very high probability. As we will show later, it returns $\beta > 0$ and "rshift$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$" is executed with probability less than 10^{-8} when $d = 32$.

IV. EFFICIENT IMPLEMENTATION OF EUCLIDEAN ALGORITHMS FOR LARGE NUMBERS

This section is devoted to show the details of implementations of Euclidean algorithms for large numbers. We assume that all numbers are stored in d-bit words. Hence, a number with s bits is stored in $\frac{s}{d}$ words. For example, a 512-bit number is stored in sixteen 32-bit words. Since Euclidean algorithms operates large numbers stored in multiple words, naive implementations perform a lot of redundant memory access operations. We will show how we implement fundamental operations used in Binary Euclidean algorithm, Fast Binary Euclidean algorithm, and Approximate Euclidean algorithm. We will show that $3\frac{s}{d} + O(1)$ memory access operations are performed in each iteration of Binary Euclidean algorithm and Fast Binary Euclidean algorithm if X and Y with s bits are stored in d-bit words. More specifically, each iteration essentially performs three operations, reading from X, reading from Y, and writing in X, each of which involves $\frac{s}{d}$ memory access operations. Also, additional $O(1)$ reading operations

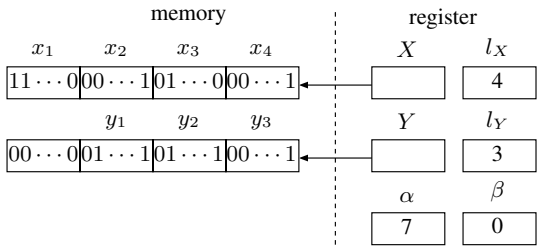

Fig. 1. Implementation of X and Y

are performed for X and Y. Further, Approximate Euclidean algorithm performs $3\frac{s}{d} + O(1)$ memory access operations in each iteration with very high probability, and $4\frac{s}{d} + O(1)$ memory access operations with very low probability.

Figure 1 illustrates how X and Y are implemented. Two s-bit numbers X and Y are stored in arrays of $\frac{s}{d}$ words. Two registers are used to store pointers that specify arrays for X and Y. Also, the values of l_X, l_Y, α, and β are stored in registers. We assume that $d = 32$ and show how each operation in Binary Euclidean algorithm, Fast Binary Euclidean algorithm, and Approximate Euclidean algorithm can be performed.

X is even : The result of the condition can be determined by reading the least significant word of X.

$Y \neq 0$: This condition is equivalent to "$l_Y > 0$." Hence no reading operation to Y is necessary.

$X < Y$: If $l_X < l_Y$ then $X < Y$ is true and if $l_X > l_Y$ then $X < Y$ is false. Thus, access to the memory is not necessary if $l_X \neq l_Y$. If $l_X = l_Y$ then we need to compare the values of X and Y from the most significant word. More specifically, x_1 and y_1 are read from the memory. If $x_1 < y_1$ then $X < Y$ is true and if $x_1 > y_1$ then $X < Y$ is false. If $x_1 = y_1$ then x_2 and y_2 are read from the memory and they are compared in the same way. If x_2 and y_2 takes 32-bit random values, then $x_2 \neq y_2$ with probability $1-2^{-32}$. Hence, the result of $X < Y$ can be determined without reading x_3 and y_3 with very high probability. If this is the case, only four words x_1, x_2, y_1 and y_2 in the memory are accessed.

swap(X, Y): This can be done by exchanging the pointer variables for X and Y. Thus, access to the values of X and Y is not necessary for swap(X, Y).

approx(X, Y): The value of approx(X, Y) can be determined by those of l_X, l_Y, x_1, x_2, y_1, and y_2. Hence, approx(X, Y) accesses at most four words x_1, x_2, y_1 and y_2 in the memory.

$X \leftarrow$rshift($X - Y \cdot \alpha$): This operation can also be done by reading words of X and Y and writing words of X from the least significant word. For example, this operation for X with four 32-bit words x_1, x_2, x_3, x_4 and Y for three 32-bit words y_1, y_2, y_3 as illustrated in Figure 1 can be performed using a 64-bit temporary register variable z and a 16-bit temporary register variable r as follows:

$z \leftarrow x_4 + (x_3 << 32) - y_3 \cdot \alpha$
$r \leftarrow$ the number of consective 0 bits in z from the LSB
$x_4 \leftarrow (z >> r)$&0xFFFFFFFF
$z \leftarrow (z >> 32) + (x_2 << 32) - y_2 \cdot \alpha$

$x_3 \leftarrow (z >> r)$&0xFFFFFFFF
$z \leftarrow (z >> 32) + (x_1 << 32) - y_1 \cdot \alpha$
$x_2 \leftarrow (z >> r)$&0xFFFFFFFF
$x_1 \leftarrow z >> (r + 32)$

Clearly, each word in X and Y is read once, each word in X is written once. Note that this algorithm works only if $r \leq 32$. The reader should have no difficulty to modify this algorithm that works correctly even if $r > 32$. Further, update operations "$X \leftarrow \frac{X}{2}$", "$X \leftarrow \frac{X-Y}{2}$", "$X \leftarrow$rshift$(X - Y)$", and can be implemented in the same way.

$\boldsymbol{X \leftarrow}$**rshift**$\boldsymbol{(X - Y \cdot \alpha \cdot D^{\beta} + Y)}$: This can be done in a similar way to "$X \leftarrow$rshift$(X - Y \cdot \alpha)$". However, we need to perform additional reading operations from Y to compute "$+Y$."

From above implementations of fundamental operations, each iteration of Binary Euclidean algorithm and Fast Binary Euclidean algorithm can be done in $3\frac{s}{d} + O(1)$ memory access operations if s-bit X and Y stored in d-bit words. Since $X \leftarrow$rshift$(X - Y \cdot \alpha \cdot D^{\beta} + Y)$ needs additional reading operations, that of Approximate Euclidean algorithm can be done in $3\frac{s}{d} + O(1)$ memory access operations if function approx returns $\beta = 0$, and in $4\frac{s}{d} + O(1)$ memory access operations if it returns $\beta > 0$. We should note that function approx returns $\beta = 0$ with very high probability. As we will show later, the probability that $\beta > 0$ is so small that we can say that each iteration of Approximate Euclidean algorithm performs $3\frac{s}{d} + O(1)$ memory access operations in practice.

V. Euclidean algorithms for RSA moduli

Recall that our target is breaking weak RSA keys by computing the GCDs of RSA moduli. Hence, we can assume that numbers given for which we compute the GCD are products of two large prime numbers. We assume that computing devices such as CPU and GPU for GCD computation can have memory of 32/64-bit words and registers of 32/64-bit words and supports 64-bit fundamental arithmetic and logic operations. Hence, we set $d = 32$ for our Approximate Euclidean algorithm.

Since we are interested in factorizing two s-bit RSA moduli by Euclidean algorithms, we can terminate algorithm when Y has less than $\frac{s}{2}$ bits. Once Y ($\neq 0$) has less than $\frac{s}{2}$ bits, the input numbers are coprime. If they have the same $\frac{s}{2}$-bit factor, X stores it and $Y = 0$ when Euclidean algorithm terminates. We say that Euclidean algorithm is early-terminate if it terminates when Y has less than $\frac{s}{2}$ bits, and non-terminate if it continue the computation until Y reaches 0. For example, early-terminate Approximate Euclidean algorithm can be written as follows:

```
[Early-terminate Approximate Euclidean algorithm]
gcd(X, Y){
  do {
    (α, β) ← approx(X, Y);
    if(β = 0){
      if(α is even) α = α − 1; //α is odd
      X ← rshift(X − Y · α); //Y · α is odd
    } else X ← rshift(X − Y · α · D^β + Y); //α · D^β is even
    if(X < Y) swap(X, Y);
  } while (Y has at least s/2 bits);
  if(Y has less than s/2 bits) return(1);
  else return(X);
}
```

Also, approx(X, Y) is performed for X and Y with at least $\frac{s}{2}$ bits. Hence, approx(X, Y) is executed only for Case 4 and program source codes for Cases 1, 2, and 3 can be omitted.

Table IV shows the average number of iterations of do-while loops performed by Euclidean algorithms, (A) Original Euclidean algorithm, (B) Fast Euclidean algorithm, (C) Binary Euclidean algorithm, (D) Fast Binary Euclidean algorithm, (E) Approximate Euclidean algorithm for 10000 pairs of 512-bit, 1024-bit, 2048-bit, and 4096-bit RSA moduli. Encryption moduli are generated using OpenSSL Toolkit [22]. The number of iterations are evaluated both for non-terminate and early-terminate versions of Euclidean algorithms.

Each iteration of (A) and (B) is very costly, because they computes quotient/modulo of two s-bit numbers. On the other hand, (C) and (D) involves no division/multiplication operation. Further, each iteration of (E) involves one division of two 64-bit numbers, and $\frac{s}{32}$ repetitions 32-bit multiplications. Hence, the computation of each iteration takes more time than that of (C) and (D). However, they perform the same memory access operations to X and Y. Thus, if memory access latency is large like GPUs, the computing time of each iteration of (E) is just little larger than that of (C) and (D). Hence, it makes sense to evaluate and compare the number of iterations of (C), (D), and (E).

From Table IV, we can see that 1. the early-terminate versions of Euclidean algorithm reduce the number of iterations to half, 2. the number of iterations is proportional to the length of input RSA moduli, 3. the number of iterations of (E) is about a half of (D) and about a quarter of (C), and 4. the number of iterations of (B) is exactly the same as that of (E). To see the small difference of (B) and (E), the table also show the average value of (E)-(B), that is, the number of iterations of (E) minus that of (B). Quite surprisingly, their difference is only 0.001%-0.002%. Recall that (B) computes the exact quotient by division of two large numbers, while (E) computes an approximation by 64-bit division. Hence, we can say that approximated quotient is sufficient for computing the GCD.

We should also note that the value of β computed by function approx in Approximate Euclidean algorithm is zero with very high probability. More specifically, if Approximate Euclidean algorithm is executed to compute the GCD for 4096-bit moduli on 32-bit word device, function approx returns non-zero β 1191 times out of 201277617364 calls, that is, it returns $\beta = 0$ with probability more than $1 - 10^{-8}$.

VI. CUDA implementation of breaking weak RSA code

We first show that Approximate Euclidean algorithms are semi-oblivious. We then show how CUDA blocks are arranged to execute Euclidean algorithms.

Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input [18] .

TABLE IV. THE NUMBER OF ITERATIONS PERFORMED BY EUCLIDEAN ALGORITHMS

		Non-terminate				Early-terminate			
		512	1024	2048	4096	512	1024	2048	4096
(A)	Original Euclidean algorithm	299.2	598.4	1197.1	2392.7	149.9	299.3	598.8	1196.5
(B)	Fast Euclidean algorithm	190.5	380.8	761.8	1523.1	95.2	190.3	380.9	761.6
(C)	Binary Euclidean algorithm	722.2	1445.1	2890.8	5782.5	361.2	722.8	1445.8	2891.8
(D)	Fast Binary Euclidean algorithm	362.3	723.6	1446.5	2892.8	180.4	361.0	722.4	1445.5
(E)	Approximate Euclidean algorithm	190.5	380.8	761.8	1523.2	95.2	190.3	380.9	761.6
	(E)−(B)	0.0032	0.0035	0.0157	0.0152	0.0014	0.0009	0.0089	0.0048

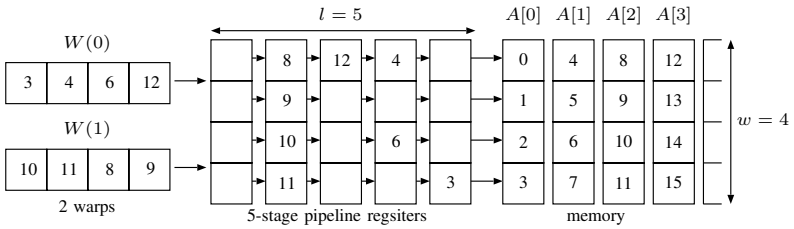

Fig. 2. The UMM with width $w = 4$ and latency $l = 5$

More specifically, there exists a function $a : \{0, 1, \ldots, t-1\} \rightarrow \mathcal{N}$, where t is the running time of the algorithm and $\mathcal{N}$ is a set of all non-negative integers such that, for any input of the algorithm, it accesses address $a(i)$ or does not access the memory at each time i ($0 \leq i \leq t-1$). In other words, at each time i ($0 \leq i \leq t-1$), it never accesses an address other than $a(i)$. Suppose that we need to execute a sequential algorithm for many different inputs on a single CPU in turn or on a parallel machine at the same time. We call such computation *bulk execution*.

For theoretical performance analysis of Approximate Euclidean algorithm, we first define *the UMM (the Unified Memory Machine)* [23], [24] which captures the essence of the global memory access of CUDA-enabled GPUs. We the go on to show that the bulk execution oblivious algorithms can be implemented very efficiently on the UMM. Let us define the UMM with width w and latency l. The memory of the UMM is partitioned into address groups $A[0], A[1], \ldots$ such that each $A[j]$ ($j \geq 0$) involves $j \cdot w, j \cdot w + 1, \ldots, (j+1) \cdot w - 1$. The reader should refer to Figure 2 that illustrates address groups for $w = 4$. Also, the memory access is performed through l-stage pipeline registers as illustrated in Figure 2. Let p be the number of threads of the UMM and $T(0)$, $T(1)$, ..., $T(p-1)$ be the p threads. We assume that p is a multiple of w. The p threads are partitioned into $\frac{p}{w}$ groups called *warps* with w threads each. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w+1), \ldots, T((i+1) \cdot w - 1)\}$. Warps are dispatched for the memory access in turn, and w threads in a warp try to access the memory at the same time. More specifically, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests the memory access. If no thread in a warp needs the memory access, such warp is not dispatched for the memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send the memory access requests, one request per thread, to the memory banks

For the memory access, each warp sends the memory access requests to the memory banks through the l-stage pipeline registers. We assume that each stage can store the memory access requests destined for the same address group. For example, since the memory access requests by $W(0)$ are separated in three address groups in the figure, they occupy three stages of the pipeline registers. Also, those by $W(1)$ are in the same address group, they occupy only one stage. In general, if the memory access requests by a warp are destined for d address groups, they occupy d stages. For simplicity, we assume that the memory access is completed as soon as the request reaches the last pipeline stage. Thus, all memory access requests by $W(0)$ and $W(1)$ in the figure are completed in 3(address groups)$+1$(address group)$+5$(latency)$-1 = 8$ time units. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

We will show that the bulk execution of an oblivious sequential algorithm can be done efficiently on the UMM. Without loss of generality, we can assume that an oblivious sequential algorithm works on a 1-dimensional array b of size n. If p threads on the UMM perform the bulk execution, the global memory stores p arrays of b. We use *column-wise arrangement* to allocate p arrays as illustrated in Figure 3. More specifically, let $b_j[i]$ denote the i-th element of b for thread j. Each $b_j[i]$ is allocated in address $j \cdot p + i$. If all threads execute a same oblivious algorithm, then they access the same address at each time unit. In other words, if a sequential algorithm access address i at some time unit, p threads access $b_0[j], b_1[j], \ldots, b_{p-1}[j]$ at the same time. Clearly, they are arranged in addresses $j \cdot p + 0, j \cdot p + 1, \ldots, j \cdot p + (p-1)$ in the same row of the 2-dimensional array. Hence, they are in consecutive addresses and memory access by p threads is always coalesced.

Let us evaluate the computing time for the bulk execution of an oblivious sequential algorithm on the UMM. Let t be the running time of an oblivious sequential algorithm and p be the number of inputs and the number of thread. For each memory access of the obvious sequential algorithm p threads performs coalesced memory access. Since they are in $\frac{p}{w}$ address groups, it can be completed in $\frac{p}{w} + l - 1$ time units. Since the oblivious sequential algorithm performs at most t

$b_0[0]$	$b_1[0]$	$b_2[0]$	$b_3[0]$	$b_4[0]$	$b_5[0]$	$b_6[0]$	$b_7[0]$
$b_0[1]$	$b_1[1]$	$b_2[1]$	$b_3[1]$	$b_4[1]$	$b_5[1]$	$b_6[1]$	$b_7[1]$
$b_0[2]$	$b_1[2]$	$b_2[2]$	$b_3[2]$	$b_4[2]$	$b_5[2]$	$b_6[2]$	$b_7[2]$
$b_0[3]$	$b_1[3]$	$b_2[3]$	$b_3[3]$	$b_4[3]$	$b_5[3]$	$b_6[3]$	$b_7[3]$

Fig. 3. Column-wise arrangement for $p = 8$ arrays of size $n = 4$ each

memory access operations, p threads on the UMM terminates in $(\frac{p}{w} + l - 1) \cdot t = O(\frac{pt}{w} + lt)$ time units. Thus we have,

Theorem 1: The bulk execution of an oblivious sequential algorithm runs $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm.

In our previous paper [18], we have proved that Theorem 1 is time-optimal.

Unfortunately, Approximate Euclidean algorithm is not oblivious. However, we will show that our Approximate Euclidean algorithm is *semi-oblivious* in the sense that an address accessed at each of almost all time units. In other words, semi-oblivious algorithm may access different addresses in few time units. If each of the CUDA threads executes a semi-oblivious sequential algorithm for the bulk execution, then they may perform non-coalesced access to the global memory. However, if the ratio of non-coalesced access is small enough, the bulk execution of a semi-oblivious sequential algorithm still runs efficiently on the GPU. In Approximate Euclidean algorithm, the computation of $X \leftarrow$rshift$(X - Y \cdot \alpha)$ is oblivious. It reads X and Y from the least significant words, and the resulting values are stored in X from the least significant words. This computation performs $3\frac{s}{d}$ memory access operations. Approximate Euclidean algorithm performs additional $O(1)$ memory access operations for function approx and for determining condition of if-else statements. These memory access operations may not be oblivious. However, the number of memory access operations is much smaller than $3\frac{s}{d}$, we can say that Approximate Euclidean algorithm is semi-oblivious.

Suppose that we have m moduli $n_0, n_1, \ldots, n_{m-1}$ of s bits each in RSA encryption keys. We are interested in computing of the GCD of all $\frac{m(m-1)}{2}$ pairs of m moduli. We partition m moduli into $\frac{m}{r}$ groups of r moduli. Let $n_{i,k}$ $(0 \leq i \leq \frac{m}{r} - 1, 0 \leq k \leq r - 1)$ denote the k-th modulus in i-th group, that is, $n_{i \cdot r+k}$. We use $\frac{m^2}{r^2}$ CUDA blocks. CUDA block (i, j) $(0 \leq i, j \leq \frac{m}{r} - 1)$ computes the GCD of moduli one in the i-th group and the other in j-th group using r threads. Each k-th thread in CUDA block (i, j) computes gcd$(n_{i,k}, n_{j,0})$, gcd$(n_{i,k}, n_{j,1})$, ..., gcd$(n_{i,k}, n_{j,r-1})$, one by one.

Let A_k and B_k be $\frac{s}{32}$-word variables of thread k $(0 \leq k \leq r - 1)$ to store input moduli. The details of the program for CUDA block (i, j) $(0 \leq i, j \leq \frac{m}{r} - 1)$ are spelled out as follows:

Each thread k $(0 \leq k \leq r - 1)$ works in parallel

```
if(i < j){
  for u ← 0 to r − 1 do {
    A_k ← n_{i,k};
    B_k ← n_{j,u};
    gcd(A_k, B_k);
  }
} else if(i = j) {
  for u ← k + 1 to r − 1 do {
    A_k ← n_{i,k};
    B_k ← n_{i,u};
    gcd(A_k, B_k);
  }
}
```

Note that CUDA block (i, j) such that $i > j$ terminates immediately. If $i = j$ then the GCD of $n_{i,k}$ and $n_{i,u}$ such that $k < u$ is computed. Thus, $\frac{m^2}{r^2}$ CUDA blocks combined, the GCDs of all $\frac{m(m-1)}{2}$ pairs of m moduli are computed.

VII. Experimental results

This section shows the running time of Euclidean algorithms. We have used Xeon X7460 (2.66GHz) CPU for executing a sequential Euclidean algorithms and GeForce GTX 780 Ti GPU for evaluating the CUDA implementations. In our CUDA implementation, we use CUDA blocks with 64 threads in which each thread computes GCDs of 64 pairs of RSA moduli. We have used local memory arranged in the global memory to store X and Y.

Table V shows the time for computing one GCD in microseconds when all $\frac{16384 \cdot 16383}{2} = 134209536$ pairs of 16K $(= 16384)$ RSA moduli with 512, 1024, 2048, and 4096 bits. The moduli are generated by OpenSSL Toolkit [22]. It also shows the speedup ratio of the GPU over the CPU. For example, the CPU computes the GCD of two 1024-bit moduli in 28.6 microseconds, while the GPU computes it 0.346 microseconds. Hence, the execution time ratio of the GPU over the CPU is 82.7.

From the table, we can see that Approximate Euclidean algorithm is faster than the others. Since Euclidean algorithms are semi-oblivious, the speedup ratio CPU/GPU is enough large. However, the execution time ratio CPU/GPU of Binary Euclidean algorithm is rather smaller than the others. This is due to the branch divergence of a CUDA C program for Binary Euclidean algorithm. Since CUDA architecture is based on SIMT (Single Instruction Multiple Threads), all threads in a warp must execute the same instruction in each clock cycle. Hence, if CUDA C program has a branch using a if-else statement, then the instructions for the true case are executed first and then those for the false case are executed. Note that, if all threads execute the instructions for the same case, those for the other case are not executed. Binary Euclidean algorithm has a if-else if-else statement to select one of the three cases: (X, Y) is (even, odd), (odd, even), and (odd, odd). Since the instructions for these three cases are executed sequentially, and the branch divergence degenerates the performance of Binary Euclidean algorithm. On the other hand, we can ignore the branch divergence of Approximate Euclidean algorithm. Approximate Euclidean algorithm has if-else statement to select two cases: $\beta = 0$ or $\beta > 0$, where β is the value computed by function approx. However, $\beta > 0$

TABLE V. THE PERFORMANCE OF EUCLIDEAN ALGORITHMS: ONE GCD COMPUTING TIME IN MICROSECONDS AND THE EXECUTION TIME RATIO WHEN ALL PAIRS OF 16K MODULI ARE COMPUTED

			Non-terminate				Early-terminate			
			512	1024	2048	4096	512	1024	2048	4096
CPU	(C)	Binary Euclidean algorithm	25.7	81.0	279	1040	17.1	56.2	200	771
	(D)	Fast Binary Euclidean algorithm	16.9	49.7	166	624	10.8	33.6	117	448
	(E)	Approximate Euclidean algorithm	14.8	43.4	140	499	9.40	28.6	96.4	357
GPU	(C)	Binary Euclidean algorithm	0.460	3.54	15.8	66.8	0.410	2.93	12.5	50.6
	(D)	Fast Binary Euclidean algorithm	0.137	0.683	3.01	11.9	0.105	0.583	2.32	9.11
	(E)	Approximate Euclidean algorithm	0.115	0.437	1.75	6.69	0.0773	0.346	1.33	5.01
CPU/GPU	(C)	Binary Euclidean algorithm	55.8	22.9	17.7	15.6	41.6	19.2	16.0	15.2
	(D)	Fast Binary Euclidean algorithm	124	72.7	55.1	52.4	102	57.6	50.5	49.2
	(E)	Approximate Euclidean algorithm	129	99.3	79.8	74.6	122	82.7	72.8	71.2

with probability less than 10^{-8} if $d = 32$. Hence all threads executes instructions for the case of $\beta = 0$ with very high probability, and those for $\beta > 0$ are not executed. Further, the 64-bit division operation for function approx and 32-bit multiplications for rshift$(X - Y \cdot \alpha)$ takes a lot of time on the CPU. On the other hand, on the GPU, time for these operations are hidden by large memory access latency. Hence, the GPU implementation for Approximate Euclidean algorithm achieves much higher speedup ratio over the CPU.

We should also note that the time for transferring moduli from the host PC to the GPU is so small that we can omit it. For example, 16K 4096-bit moduli can be transferred in 0.002 seconds while all Euclidean algorithms for them runs at least 600 seconds.

VIII. CONCLUSION

We have presented a new Euclidean algorithm for computing the GCD of all pairs of encryption moduli to break weak RSA keys. The idea of our new Euclidean algorithm that we call Approximate Euclidean algorithm is to compute an approximation of quotient by just one 64-bit division and to use it for reducing the number of iterations of the Euclidean algorithm. We also present an implementation of Approximate Euclidean algorithm optimized for CUDA-enabled GPUs. The experimental results show that our implementation for 1024-bit GCD on GeForce GTX 780Ti runs more than 80 times faster than the Intel Xeon CPU implementation. Also, our GPU implementation is more than 9 times faster than the best known published GCD computation using the same generation GPU.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 153–159.

[3] Y. Takeuchi, D. Takafuji, Y. Ito, and K. Nakano, "Ascii art generation using the local exhaustive search on the GPU," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 194–200.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[5] A. Kasagi, K. Nakano, and Y. Ito, "Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU," *IEICE Transactions on Information and Systems*, vol. Vol. E96-D, no. 12, pp. 2617–2625, Dec. 2013.

[6] ——, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing (ICPP)*, Oct. 2013, pp. 1–10.

[7] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2012, pp. 94–102.

[8] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.

[9] ——, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[10] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[11] K. Nakano, "Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models," *IEICE Trans. on Information and Systems*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[12] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120 – 126, 1978.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein., *Introduction to Algorithms*. MIT press, 2001.

[14] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, Whit is right," Cryptology ePrint Archive, Report 2012/064, 2012. [Online]. Available: http://eprint.iacr.org/

[15] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

[16] J. Stein, "Computational problems associated with racah algebra," *Journal of Computational Physics*, vol. 1, no. 3, Feb. 1967.

[17] D. Takafuji, K. Nakano, and Y. Ito, "A CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, Aug. 2014, pp. 178–191.

[18] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, "Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2014, pp. 586–595.

[19] N. Fujimoto, "High throughput multiple-precision GCD on the CUDA architecture," in *Proc. of International Symposium on Signal Processing and Information Technology*, Dec. 2009, pp. 507–512.

[20] K. Scharfglass, D. Weng, J. White, and C. Lupo, "Breaking weak 1024-bit RSA keys with CUDA," in *Proc. of Internatinal Conference of Breaking weak 1024-bit RSA keys with CUDA*, Dec. 2012, pp. 207 – 212.

[21] J. R. White, "PARIS: A parallel RSA-prime inspection tool," Ph.D. dissertation, California Polytechnic State University - San Luis Obispo, June 2013.

[22] "OpenSSL: The open source toolkit for SSL/TLS." [Online]. Available: https://www.openssl.org/

[23] K. Nakano, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

[24] ——, "Sequential memory access on the unified memory machine with application to the dynamic programming," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 85–94.