Parallel FDFM Approach for Computing GCDs Using the FPGA

Xin Zhou, Koji Nakano, and Yasuaki Ito

Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 JAPAN

Abstract. The main contribution of this paper is to present an FPGA-targeted architecture called the hierarchical GCD cluster, that computes the GCDs of all pairs in a set of numbers. It is designed based on the FDFM (Few DSP slices and Few Memory blocks) approach and consists of 1408 processors equipped with one block RAM and one DSP slice each. Every processor works in parallel and computes the GCDs independently. We have measured the performance of our architecture to compute all pairs of two numbers in RSA moduli. Implementation results show that it runs 0.057μs per one GCD computation of two 1024-bit RSA moduli in a Xilinx Virtex-7 family FPGA XC7VX485T-2. It is 6.0 times faster than the best GPU implementation and 500 times faster than a sequential implementation on the Intel Xeon CPU.

Keywords: FDFM approach, parallel algorithms, DSP slices, block RAMs, RSA cryptosystem

1 Introduction

An FPGA (Field Programmable Gate Array) is an integrated circuit designed to be configured by a designer after manufacturing. It contains an array of programmable logic blocks, and the reconfigurable interconnects allow the blocks to be inter-wired in different configurations. Since any logic circuit can be embedded in an FPGA, it can be used for general-purpose parallel computing [2, 11]. Recent FPGAs have embedded DSP slices and block RAMs. Xilinx Virtex-7 family FPGAs have DSP slices, each of which is equipped with a multiplier, adders/subtracters, logic operators, registers, etc [16]. For example, the DSP slice has a two-input multiplier followed by multiplexers and a three input adder/subtracter/accumulator. It also has pipeline registers between operators to reduce the propagation time. A block RAM is an embedded dual-port memory supporting synchronized read and write operations, and can be configured as a 36k-bit or two 18k-bit dual port RAMs [18]. The main contribution of this paper is to present an architecture for computing the GCD (Greatest Common Divisor) of large two numbers using an FPGA. We employ *the FDFM (Few DSP slices and Few block Memories) approach* [1] to implement parallel GCD computation in the FPGA. The key idea of the FDFM approach is to use few DSP slices and few block RAMs for constituting a processor performing a specific computation.

For example, hardware algorithms for RSA encryption/decryption have been implemented in the FPGA using the FDFM approach [7]. Their implementation using it is better than the conventional approach [12].

One of the applications for benchmarking GCD computation is breaking weak RSA keys. RSA [13] is one the most well-known public-key cryptosystems 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 bits is obtained by 512-bit prime numbers p and q. The decryption key for this encryption key is a pair (n, d) such that $de \equiv 1 \pmod{(p-1)(q-1)}$, 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 $M \equiv M^{ed} \pmod{n}$ always holds for all messages 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, $d \equiv e^{-1} \pmod{(p-1)(q-1)}$ can be computed very easily by extended Euclidean algorithm [3]. 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 of large numbers is very costly, it is not possible to decompose n into p and q in 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 from 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 sharing a prime number *weak RSA keys*. Actually, several public keys collected from the Web includes weak RSA keys [10]. 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 [8]. 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 moduli in the Web. It has been shown that a complicated sequential algorithm can find a pair of weak RSA keys [10]. So, it makes no sense to perform straightforward pairwise computation of GCDs for RSA moduli for the purpose of breaking weak RSA keys. However, bulk computation of GCDs for RSA moduli is useful to measure the performance of the GCD computation. We designed and implemented 1408 GCD processors in a Xilinx Virtex-7 family FPGA XC7VX485T-2, that compute the GCDs in parallel. The implementation results show that, 1408 GCD processors can compute the GCD of one 1024-bit RSA moduli in expected 0.057μs.

Several hardware implementations for computing the GCD on FPGAs have been presented [4, 9]. However, they just implemented Binary Euclidean algorithm to compute the GCD using programmable logic blocks as it is. Hence,

they can support the GCD computation for numbers with very few bits. On the other hand, several previously published papers have presented GPU implementations of Binary Euclidean algorithm in CUDA-enabled GPUs. Fujimoto [5] 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 $10.9\mu s$ per one 1024-bit GCD computation. Scharfglass *et al.* [14] 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 $10.02\mu s$. Later, White [15] 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 $3.15\mu s$. Quite recently, Fujita *et al.* have presented new Euclidean algorithm called Approximate Euclidean algorithm and implemented it in the GPU [6]. Approximate Euclidean algorithm performs perform each 1024-bit GCD computation in $0.346\mu s$ on GeForce GTX 780Ti and $28.6\mu s$ on Intel Xeon X7460 (2.66GHz) CPU. Our implementation of the Hardware Binary Euclidean algorithm performs one 1024-bit GCD computation in $0.057\mu s$ which is 6.0 times faster than the GPU and 500 times faster than the CPU.

2 Euclidean Algorithms for computing GCD

This section first reviews Fast Binary Euclidean algorithm for computing the GCD of two numbers X and Y. Please see [6] for the details. We then show Hardware Binary Euclidean algorithm by modifying Fast Binary Euclidean algorithm, which will be implemented it in an FPGA.

We assume that both input numbers 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 GCD algorithms shown in this paper to handle even input numbers.

Let swap(X,Y) denote a function to exchange the values of X and Y. Further, 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 $X = 11010100$ in binary system, then rshift(X)= 110101. Using swap and rshift functions, we can write Fast 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);
}
```

We will modify Fast Binary Euclidean algorithm to be implemented in the FPGA. We need to read all bits of X and Y to exchange them if we implement function swap as it is. Also, rshift function needs a large barrel shifter. Hence, we should avoid direct implementations of these functions in the FPGA. Instead of function rshift(X), we use rshift$_k(X)$, which removes at most k consecutive 0 bits from the least significant bit of X. In other words, if X has at most k consecutive 0 bits from the least significant bit, all of them are removed. If it has more than k 0 bits, then k 0 bits from the least significant bits are removed. For example, rshift$_2$(11011000)= 110110 and rshift$_2$(11011010)= 1101101 hold. If k is small, rshift$_k(X)$ can be implemented using a small barrel shifter. Using rshift$_k(X)$, we can design an Euclidean-based GCD algorithm for FPGAs as follows:

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

Note that rshift$_k$ function may return an even number. Hence, one of X or Y can be an even number. If this is the case, either $X \leftarrow$rshift$_k(X)$ or $Y \leftarrow$rshift$_k(Y)$ is executed until both of them are odd. Hence, both X and Y are odd, whenever rshift$_k(X - Y)$ is executed, Thus, the argument of rshift$_k$ is always even when it is executed.

If these Binary Euclidean algorithms are executed for two s-bit RSA moduli, the GCD is 1 or $\frac{s}{2}$-bit prime number. Hence, when either X or Y has less than $\frac{s}{2}$ bits during the execution of these Euclidean algorithms, the GCD must be 1. Thus, we can terminate the do-while loop when one of X or Y has less than $\frac{s}{2}$ bits. We call this *early-terminate* technique.

Table 1 shows the average number of iterations of the do-while loop 1024-bit RSA moduli for each values of k of rshift$_k$. Note that $k = \infty$ corresponds to Fast Binary Euclidean algorithm, which performs rshift function that removes all consecutive 0 bits. We can see that the early-terminate technique can reduce the number of iterations by half. Clearly, the number of iterations is smaller for large k. However, rshift$_k(X)$ needs a large barrel shifter for large k. We should select an appropriate value of k that balances the computing time and the used

hardware resources. For our FPGA implementation that we will show later, we have selected $k = 4$.

Table 1. The number of iterations of the do-while loop for 1024-bit RSA moduli

	Hardware Binary Euclidean								Fast Binary
k	1	2	3	4	5	6	7	8	Euclidean
Non-terminate	1445.8	964.3	827.0	772.0	747.1	735.3	729.7	726.8	723.9
Early-terminate	721.1	481.1	412.7	385.2	372.9	367.0	364.2	362.8	361.4

3 A GCD processor for computing the GCD

This section presents a *GCD processor*, which computes the GCD of two moduli by Hardware Binary Euclidean algorithm.

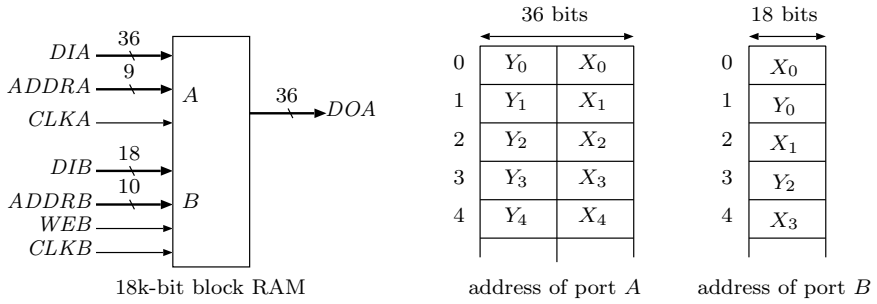

Fig. 1. A block RAM and the memory configuration

Our GCD processor uses one 18k-bit block RAM and one DSP slice in the FPGA. We use a block RAM as a simple dual-port memory [18] with ports A and B as illustrated in Figure 1. Ports A and B are configured as read-only 36-bit mode and write-only 18-bit mode, respectively. In other words, the block RAM is a 512×36-bit memory for port A and 1024×18-bit for port B. Using these ports, we can read 36-bit data and write 18-bit data for different addresses at the same time. Two numbers X and Y of Hardware Binary Euclidean algorithm are stored as 18-bit words. If each of them has 1024 bits each, it is stored in $\lceil \frac{1024}{18} \rceil = 57$ words. Let $X_{56}X_{55} \cdots X_0$ denote 57 words representing X such that $X = \sum_{i=0}^{56} X_i \cdot 2^{18i}$ holds. Similarly, let $Y_{56}Y_{55} \cdots Y_0$ denote those representing Y. In the 18-bit mode, 18-bit data X_i and Y_i can be written in addresses $2i$ and $2i + 1$ $(0 \leq i \leq 56)$, respectively, using port B, as illustrated in Figure 1. They can be read from 36-bit port A such that 36-bit data Y_iX_i $(0 \leq i \leq 56)$ is read from address i.

Figure 2 shows the architecture of a processor for computing the GCD by Hardware Binary Euclidean algorithm. The 36-bit output of the block RAM are connected to the DSP slice through multiplexers. The DSP slice has 18-bit two ports A and B. By two multiplexers, one of X and Y is given to port A, and one of X, Y, and zero (0) is given to port B. Since DSP computes the subtraction $A-B$, it can outputs one of $X-Y$, $Y-X$, $X-0$ $(=X)$, and $Y-0$ $(=Y)$. The subtraction is performed sequentially word-by-word. For example, suppose that we need to compute $Z=X-Y$. Let $Z_{56}Z_{55}\cdots Z_0$ denote 57 words representing Z and show how $X-Y$ are computed. First, X_0 and Y_0 read from the block RAM are transferred to ports A and B of the DSP slice, and X_0-Y_0 is computed and stored in 19-bit register P. The MSB $P[18]$ corresponds to the borrow of the subtraction and the remaining 18 bits of P is equal to Z_0. After that, $X_1-Y_1-P[18]$ is computed and stored in P. This value is equal to Z_1. Repeating the same operation, the DSP slice outputs all words of Z one by one.

Note that, the resulting value of $Z=X-Y$ must be right-shifted appropriately, when it is stored in X or Y. We use a 4-bit shifter which can right-shift Z by 1 bit, 2 bits, 3 bits, or 4 bits. Function rshift_4 can be implemented as follows. We use a 17-bit register to store the output of P, which stores the value of Z. For example, let $Z_0=z_{17}z_{16}\cdots z_0$ and $Z_1=z_{35}z_{34}\cdots z_{18}$ be the first and the second outputs of P. In the next clock cycle just after Z_0 is stored in P, the register stores 17-bit $z_{17}z_{15}\cdots z_1$. In the same time P stores Z_1. Hence, we can have 21 consecutive bits $z_{21}z_{20}\cdots z_1$ of Z by picking four bits in $z_{21}z_{20}z_{19}z_{18}$ in Z_1. The 4-bit shifter selects consecutive 18 bits in 21 bits. For example, it selects $z_{18}z_{17}\cdots z_1$ for 1-bit shift and $z_{21}z_{20}\cdots z_4$ for 4-bit shift. It should be clear that the 4-bit shifter outputs $\text{rshift}_4(X-Y)$. This value is transferred to port B of the block RAM, and the value of X is updated.

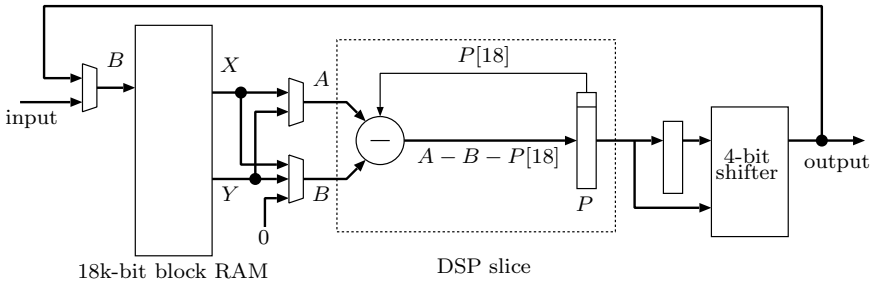

Fig. 2. The architecture of a GCD processor

Let us briefly confirm that the GCD processor can execute Hardware Binary Euclidean algorithm. It should be clear that, by configuring two multiplexers connecting the block RAM and the DSP, we can compute one of $\text{rshift}_4(X-Y)$, $\text{rshift}_4(Y-X)$, $\text{rshift}_4(X)$, and $\text{rshift}_4(Y)$. Also, the resulting value can be

overwritten in X or Y through port B of the block RAM. The conditions "X is even" and "Y is even" can be determined by reading X_0 and Y_0 in the block RAM. Also, "$X \geq Y$" can be decided by reading X and Y from the MSB. More specifically, we check if $X_{56} > Y_{56}$ holds, then "$X \geq Y$" is true. If $X_{56} < Y_{56}$, it is false. If $X_{56} = Y_{56}$, then we need to read and compare X_{55} and Y_{55}. The same operation is repeated until "$X \geq Y$" can be determined. We should note that, with high probability, "$X \geq Y$" can be determined by reading several words of X and Y. Also, during the computation of Hardware Binary Euclidean algorithm, the number of bits of X and Y is decreased. Hence, we use registers to store the current numbers of bits of X and Y. By using them, we can easily determine if $X \neq 0$ and $Y \neq 0$. Further, for early-terminate, we can determine if the number of bits is less than $\frac{s}{2}$.

4 Hierachical GCD cluster

This section presents a hierarchical parallel architecture that we call hierarchical GCD cluster. Since we can embed more than one thousand GCD processors in an FPGA, it makes sense to use multiple servers, each of which controls approximately one hundred GCD processors. The hierarchical GCD cluster consists of multiple GCD clusters, each of which involves multiple GCD processors as illustrated in Figure 3. A single central server controls local servers, each of which maintains GCD processors in the same GCD cluster.

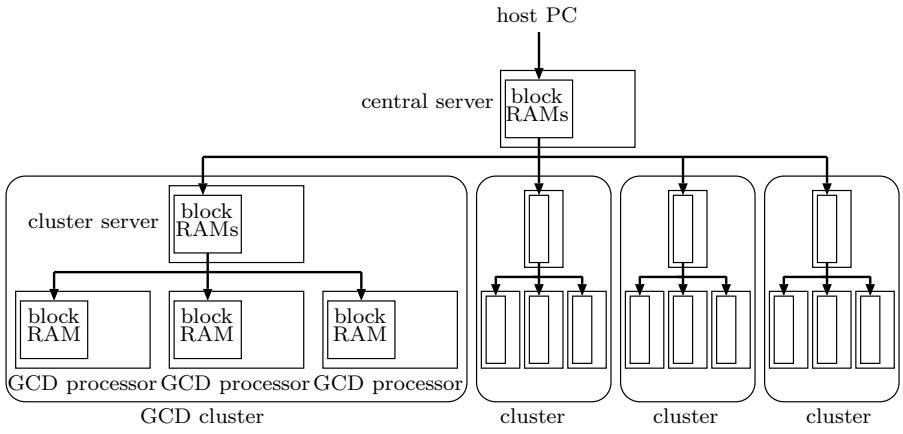

Fig. 3. The architecture of the Hierarchical GCD Cluster

We show how the hierarchical GCD cluster is used to break weak RSA keys. Suppose that we have a lot of moduli collected from the Web and they are stored in the main memory of the host PC. Our goal is to compute all pair of moduli using the hierarchical GCD cluster in an FGPA. For this purpose, we partition

all moduli into groups with m moduli each. The host PC picks two groups and sends them to the central server in the FPGA. Let $N = \{n_0, n_1, \ldots n_{m-1}\}$ and $N' = \{n'_0, n'_1, \ldots n'_{m-1}\}$ denote two groups of m moduli each that the central server in the FPGA has been received. The hierarchical GCD cluster computes $\gcd(n_i, n'_j)$ for all pairs of i and j $(0 \leq i, j \leq m-1)$, and reports the GCDs larger than 1.

Next, we will show how the hierarchical GCD cluster computes the GCDs of N and N' using GCD clusters. Each group of m moduli is partitioned into b blocks of k moduli each, where $m = bk$. Let $N_i = \{n_{ik}, n_{ik+1}, \ldots, n_{(i+1)k-1}\}$ and $N'_i = \{n'_{ik}, n'_{ik+1}, \ldots, n'_{(i+1)k-1}\}$ $(0 \leq i \leq b-1)$ be two sets of k moduli in the i-th groups of sets N and N', respectively. Each cluster is assigned a task to compute the GCDs of all pairs X $(\in N_i)$ and Y $(\in N'_j)$ for a pair i and j $(0 \leq i, j \leq b-1)$. For this purpose, all moduli in N_i and in N'_j are copied from the block RAM in the central server to that in the local server of a GCD cluster. After the local server receives all moduli, the cluster starts computing the GCDs of all pairs X $(\in N_i)$ and Y $(\in N'_j)$. The local server then picks a pair X and Y and copies them to the block RAM of a GCD processor. Upon completion of the copy, the GCD processor starts computing the GCD of X and Y by the Hardware Binary Euclidean algorithm. This procedure is repeated for all GCD processors. If a GCD processor terminates the GCD computation, the local server sends a new pair to it. In this way, the GCDs of all pairs in N_i and N'_j are computed by a GCD cluster. When a GCD cluster completes the computation of all GCDs of a given pair of two groups, the central server picks a new pair i and j and sends all moduli in N_i and in N'_j to the local server. The same operation is repeated until the GCDs of all pairs N and N' are computed.

5 Implementation results in the FPGA

We have implemented a GCD processor for 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli in VC707 evaluation board [19] equipped with the Xilinx Virtex-7 FPGA XC7VX485T-2. Note that one 18k-bit block RAMs can store up to two 9216-bit moduli. Table 2 shows the implementation results. Slice Registers and Slice LUTs (Look-Up-Tables) are hardware resources in CLB (Configurable Logic Block) [17], which are used to implement sequential logics. We can see that the used hardware resources are very few. In particular, we can confirm that only one 18k-bit block RAM and one DSP slice are used. Also, the clock frequency is about 350MHz, which is sufficiently high.

For breaking 1024-bit moduli, a GCD cluster with a local cluster with eight 18k-bit block RAMs and 128 GCD processors is used. Since four 18k-bit block RAMs can store $\lfloor \frac{4 \cdot 1024}{57} \rfloor = 71$ moduli with 1024 bits, each GCD cluster computes the GCDs of $71 \times 71 = 5041$ pairs of blocks stored in block RAMs. Hence, each GCD processor computes the GCDs for expected $\frac{5041}{128} = 39.4$ pairs of 1024-bit moduli. Also we arranged 64 block RAMs to the central server. Since a block of moduli is stored in four block RAMs, we can think that the central server has $8 \times 8 = 64$ pairs of blocks. Thus, each cluster computes the GCDs for moduli

Table 2. Implementation results of one GCD processor for 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18kb block RAMs	Clock Frequency
Available	607200	303600	2800	2060	(MHz)
1024-bit	155	179	1	1	345
2048-bit	160	185	1	1	355
4096-bit	165	192	1	1	343
8192-bit	170	230	1	1	350

in expected $\frac{64}{11} = 5.8$ pairs of blocks. Table 3 shows the implementation results of clusters. Since a cluster server uses eight 18k-bit block RAMs, each GCD cluster with 128 GCD processors involves $128 + 8 = 136$ block RAMs. We have succeeded in implementing a hierarchical GCD luster with 11 GCD clusters and the central server, which uses $11 \cdot 128 = 1408$ DSP slices and $11 \cdot 136 + 64 = 1560$ block RAMs. Unfortunately, due to the overhead for the connection between the central server and GCD clusters, the clock frequency is decreased to 271MHz.

Table 3. Implementation results of the GCD cluster and the hierarchical GCD cluster for 1024-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18kb block RAMs	Clock Frequency
Available	607200	303600	2800	2060	(MHz)
GCD cluster	20548	26306	128	136	320
hierarchical GCD cluster	225176	288215	1408	1560	271

We have evaluated the number of clock cycles to compute all GCDs of $71 \times 71 = 5041$ pairs of 1024-bit moduli by one GCD cluster. For this purpose, we have used RSA moduli generated by OpenSSL Toolkit. By simulation, we have that it takes 859468 clock cycles to compute the GCDs of 5041 pairs. If a GCD cluster operates in 271 MHz as shown in Table 3, the expected computing time is 859468/271MHz =3.17ms. Also, it takes about $71 \times 2 \times 57 = 8094$ clock cycles to transfer a pair of two blocks of 71 moduli each and this overhead is negligible. Since the hierarchical GCD cluster can have up to 11 clusters, we can expect that the GCDs of $5041 \times 11 = 55451$ pairs in the same time. Thus, we can write that one GCD can be computed in expected $3.17\text{ms}/55451 = 0.057\mu\text{s}$.

6 Conclusions

We have presented an Euclidean-based hardware algorithm for computing the GCD. It was implemented in an FPGA using the FDFM approach such that each of 1408 GCD processors computes the GCD of 1024-bit moduli in RSA keys independently. From the implementation results, it can compute the GCD

of two 1024-bit RSA moduli in expected 0.057μs on a Xilinx Virtex-7 family FPGA XC7VX485T-2.

References

1. Ago, Y., Ito, Y., Nakano, K.: An FPGA implementation for neural networks with the FDFM processor core approach. International Journal of Parallel, Emergent and Distributed Systems 28(4), 308–320 (2013)
2. Bordim, J.L., Ito, Y., Nakano, K.: Accelerating the CKY parsing using FPGAs. IEICE Transactions on Information and Systems E86-D(5), 803–810 (May 2003)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein., C.: Introduction to Algorithms. MIT press (2001)
4. Devi, R., Singh, J., Singh, M.: VHDL implementation of GCD processor with built in self test feature. International Journal of Computer Applications 25(2), 50–54 (July 2013)
5. Fujimoto, N.: High throughput multiple-precision GCD on the CUDA architecture. In: Proc. of International Symposium on Signal Processing and Information Technology. pp. 507–512 (Dec 2009)
6. Fujita, T., Nakano, K., Ito, Y.: Bulk gcd computation using a gpu to break weak rsa keys. In: Proc. of International Parallel and Distributed Processing Symposium Workshops. pp. 385–394 (May 2015)
7. Ito, Y., Nakano, K., Bo, S.: The parallel FDFM processor core approach for crt-based rsa decryption. International Journal of Networking and Computing 2(1), 79–96 (Jan 2012)
8. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley (1997)
9. Kohale, S.D., Jasutkar, R.W.: Power optimization of GCD processor using low power Spartan 6 FPGA family. International Journal of Conceptions on Electronics and Communication Engineering 2(1), 1–6 (June 2014)
10. Lenstra, A.K., Hughes, J.P., Augier, M., Bos, J.W., Kleinjung, T., Wachter, C.: Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064 (2012), http://eprint.iacr.org/
11. Nakano, K., Yamagishi, Y.: Hardware n choose k counters with applications to the partial exhaustive search. IEICE Trans. on Information & Systems E88-D(7), 1350–1359 (2005)
12. Nakano, K., Kawakami, K., Shigemoto, K.: RSA encryption and decryption using the redundant number system on the FPGA. In: Proc. of International Symposium on Parallel and Distributed Processing Workshops. pp. 1–8 (May 2009)
13. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21, 120 – 126 (1978)
14. Scharfglass, K., Weng, D., White, J., Lupo, C.: Breaking weak 1024-bit RSA keys with CUDA. In: Proc. of Internatinal Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 207 – 212 (Dec 2012)
15. White, J.R.: PARIS: A PARALLEL RSA-PRIME INSPECTION TOOL. Ph.D. thesis, California Polytechnic State University - San Luis Obispo (June 2013)
16. Xilinx Inc.: 7 Series DSP48E1 Slice User Guide (Nov 2014)
17. Xilinx Inc.: 7 Series FPGAs Configurable Logic Block User Guide (Nov 2014)
18. Xilinx Inc.: 7 Series FPGAs Memory Resources User Guide (Nov 2014)
19. Xilinx Inc.: VC707 Evaluation Board for the Virtex-7 FPGA User Guide (2014)