International Journal of Application or Innovation in Engineering & Management (IJAIEM)
Web Site: www.ijaiem.org Email: editor@ijaiem.org
Volume 5, Issue 1, January 2016 **ISSN 2319 - 4847**

Acceleration of the RSA Processes based on Parallel Decomposition and Chinese Reminder Theorem

Mohammed Issam Younis[1], Heba Mohammed Fadhil[2] , Zainab Nadhim Jawad [3]

[1] Computer Engineering Department, College of Engineering, Baghdad University, Baghdad, Iraq

[2] Information and Communication Department, Al-Khwarizmi College of Engineering, Baghdad University, Baghdad, Iraq

[3] Engineering Affairs Department, University Presidency, Karbala University , Karbala, Iraq

ABSTRACT

Within current advancement in computer architecture, the trends nowadays involve re-design and re-implement of algorithms to take the advantages of currently available hardware and the applicability of composition. This paper reviews the parallelizing of the RSA Algorithm and adopting the Chinese Remainder Theorem (CRT) to accelerate the decryption process. In addition, this paper proposes variant decompositions to gain extra speed up. The proposed algorithms are implemented using C# programming language. Finally, the practical results demonstrate the many cores' GPU implementation obtained the highest speedup results for both encryption and decryption processes for variant key size and different workload; for the decryption process with CRT, it is noticed that the adopting CRT sequential version gives a speed up gains ~14X. The multi–core gains ~119X speed up; while the many core GPU gains ~433X speed. Thus, CRT gives a significant speed up for the decryption process for all three variant implementations. In addition, in both cases for Multi-cores and Many-cores, the speed up is super due to composition of parallel processing and CRT.

Keywords: RSA, GPU, CRT, DLP, TLP

1. INTRODUCTION

During the last years, the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) have functioned as a secure communication channel on the Internet. Currently, SSL provides confidential data securely and prevents eavesdropping and tampering by random attackers. SSL plays a key role in online banking, E-commerce, and other Internet services to protect passwords and credit card numbers, as well as, social security numbers and other private information [1]. Unfortunately, public key algorithms are not nearly mathematical inexpensive as symmetric encryption algorithms. An intensive study by Zhao et al. of the SSL session shows that more than 90% of the time spent in the encryption operations is in the key exchange of RSA, which involves high costs calculations for sites with high traffic, where the rate of new connections per second can easily reach into thousands. In each SSL connection, the server has to implement the exchange of the encryption key that involves public key encryption. It rapidly becomes a bottleneck when there is a need to create a large number of connections in the server side [2].

Due to its distinctive ability to distribute and manage keys, public key encryption has become the perfect solution to information security [3]. Currently, the servers that rely on public key encryption (such as SSL server) require dealing with a significant number (multiple-precision integer) that requires massive computing power [2].

From the time of its invention in 1978, RSA encryption was investigated extensively for weaknesses. Whereas it is not found effective ways to attacking at any time, due to years of RSA cryptanalysis a wide look at features that offered valuable guidelines for proper use and implementation. Because RSA provides high security and simple to implement, it quickly became the most widely used and commonly public key encryption. However, the expensive RSA encryption in real-time is still a challenge. Finding effective implementation of RSA is one of the important tasks that still needs to be done [4].

Data encryption and decryption are in general complex problems to contain complicated mathematical calculations due to the restrictions demand on computer resources; the processor and memory, especially, when considering the processing of significant amounts of data. Still, in many situations, calculations carried out by encryption algorithms can be divided into a large number of independent parts and implemented on different cores. It has been observed that encryption and decryption of large amounts of data could be decomposed and executed in parallel [5].

Over the past last years, more and more, parallel computing (multi-cores/ many-cores) processors have been overriding sequential ones. The most important engine of processor performance growth had increased parallelism, rather than

increasing clock rate and this tendency is expected to continue. Particularly, today's modern Graphical Processing Units (GPUs) have grown a dimension in terms of performance exceeding traditional Central Processing Unit (CPU) devilishly. Numerous modern computer systems have been made of – besides a CPU –a powerful GPU will perhaps operate idle most of the time and may be used as an inexpensive and immediately available co-processor for many general-purpose applications [6], [7]. Although different applications are executed, the implementations on massively parallel platforms have comparable challenges that require design / redesign algorithms to use wide parallel processing on autonomous data sets [8]. In addition, recent studies regarding the RSA algorithm show that the decryption process is more time consuming than the encryption process. As such, this paper focuses on enhancement RSA encryption/ decryption processes and organized as follows. Section 2 highlights the related works. Section 3 reviews RSA algorithms and Chinese Remainder Theorem (CRT) and gives the design and implementation of the proposed RSA algorithm. Section 4 discusses the results. Finally, Section 5 gives the conclusion and suggestions for future works.

2. RELATED WORKS

With the rapid developments in hardware and software technologies, it seems that the sequential implementation of algorithms is not fast enough. Parallel algorithms, on the other hand, play a significant role in maintaining rapid growth. Not only multi-core processors but also a powerful graphics cards are becoming more and more available [9]. As a result, the researcher focuses on parallelizing both RSA encryption and decryption processes.

Fan et al. introduced an efficient software implementation of the Montgomery multiplication algorithm on a multi-core system. They achieved to speed up of 1.53 and 2.15 when dealing with 256 bits and 1024 bits Montgomery modular multiplication, respectively [10].

Chen and Schaumont investigated the parallelization of the Montgomery multiplication, which is still considered a very time- consuming process in the public key cryptography, and proposes a scalable parallel programming scheme called Parallel Separated Hybrid Scanning (PSHS) to map the Montgomery multiplication to the modern multi-core architecture. pSHS accelerates 2048 bits Montgomery multiplication by 1.97, 3.68, and 6.13 times on two-core, four-core, and eight-core architectures respectively [11].

Baktir and Savas presented an efficient parallel Montgomery multiplication algorithm for software implementations on general-purpose multi-core processors. They achieved to speed up of 0.81 times, 3.37 times and 4.87 times with 2, 4, and 6 cores, respectively [12].

Moss et al. presented the first GPU implementation of 1024 bitsRSA's exponentiation on NVIDIA 7800 GTX GPU [13]. Their experimental results showed that there was a significant latency associated with invoking operations on the GPU, due to the legacy GPU architecture and the application programming interface.

Fleissner proposed a 192 bits Montgomery exponentiation algorithm, which was executed on NVIDIA 7800GTX [14].

Szerwinski and Guneysu employed modular exponentiations of 1024 and 2048 bits based on both Montgomery Coarsely Integrated Operand Scanning (CIOS) and RNS arithmetic by an NVIDIA 8800GTS GPU and the Compute Unified Device Architecture (CUDA) framework [15].

Harrison and Waldron presented a high performance 1024 bits RSA modular exponentiation running on an NVIDIA 8800 GTX, which was established on integers represented in standard radix system and RNS [16].

Fan et al. presented an implementation of the RSA algorithm in parallel using Java for CUDA (JCUDA) and Hadoop. Their experimental results had shown that the RSA algorithm speed improved in comparison to the original method on the CPU only [6].

Neves and Araujo executed 1024 bits RSA decryption on GTX260 GPU [17].

Yao et al. presented an investigation into the implementation and performance of modular exponentiation. They focused on 1024 bits RSA decryption running on an NVIDIA 9600GT and established a peak throughput of 3863 msg/sec which means 5 times improvement over a comparable CPU implementation [18].

Li et al. developed a parallel Montgomery multiplications using CUDA 2.3 platform and NVIDIA GeForce GTX285 GPU. Their results demonstrated that GPU's implementation was ten times faster than CPU's implementation [19].

Zhang et al. implemented the RSA algorithm with modular exponentiation of (512,1024, and 2048) bits, through comparing and analyzing the implementation of GPU and CPU. The research results showed that the GPU implementation was faster 45 times in comparison with multi-core CPU implementation of RSA [20].

Dai introduced Crypto++, which is a free and open source C ++ library of cryptographic algorithms which includes: ciphers, message authentication codes, one-way hash functions, public-key cryptosystems, and key agreement schemes [21].

Finally, Fadhil and Younis discussed the parallel implementation of variable key length RSA algorithm on both multi-cores' CPU and many-cores' GPU [22], [23]. They proposed variant implementations (sequential, Multi-threaded RSA for CPU and Many-core RSA for GPU) using C# programming language and GPU.NET framework. Unlike other

previous works, their implementation supports variable key size up to 8192 bits. their experiments are conducted on a laptop with Intel Core I7 2670QM, 2.20 GHz CPU, and Nvidia GeForce GT630M GPU. The GPU implementation gained approximately 23 speed up factor over the sequential CPU implementation; while the multithread CPU implementation gained only 6 speed up factor over the sequential CPU implementation as far as the latency is concerned. Furthermore, the GPU's implementation achieved throughput (Number of processed messages per second) ~1800 msg/sec, and ~250 msg/sec for 1024 and 2048 bits respectively due to a utilization of both data level parallelism (DLP) and Thread Level Parallelism (TLP). Fix and build from earlier works, the next section discuss three variants implementation of RSA decryption based on CRT.

3. DECOMPOSITION OF RSA CALCULATIONS

Build from our earlier work [22], [23] on construction three variants implementation of the RSA Algorithm (i.e., target to sequential, Muti-cores CPU, and Many-cores GPU). This section give another three variants decryption implementation based on CRT.

3.1 RSA's Exponent calculation

The Montgomery reduction algorithm is used for RSA's exponent calculation for both encryption and decryption processes [14]. The detailed of the Montgomery algorithm is illustrated in Figure 1.

Input: $a,e,n.$
Output : $a^e \bmod n$
Function : $MonExp(a, e, n)$
Step1: $a' = a \cdot r \bmod n$
Step 2: $x' = 1 \cdot r \bmod n$
Step 3: for $i = w - 1$ to 0
a) $x' = MonMul(x', x')$
if $e_i = 1$; then
b) $x' = MonMul(x', a')$
end loop
Step 4: $x = MonMul(x',1)$
return x

Figure 1 Montgomery reduction algorithm [14]

3.2 RSA's decryption acceleration based on CRT

The size of the decryption exponent d and the modulus n is very important because the complexity of the RSA decryption is directly dependent on it. Therefore, to introduce a decryption much faster than modular exponentiation it is prevalent to employ the CRT during decryption. RSA-CRT differs from the standard RSA in key generation and decryption [24], [25]. The RSA-CRT decryption is formalized as follows:

Let p and q be two numbers (co-prime positive integer) such that GCD (p, q) ≡ 1. If a ≡ b (mod p) and a ≡ b (mod q), then a ≡ b (mod p.q) [24].

Since the recipient knows the secret primes p and q, the following modular components can be computed:

1. Calculate d p ≡ d mod(p- 1) and d q ≡ d mod(q-1).
2. Calculate C p ≡ C mod p and C q ≡ C mod q.
3. Calculate Mp ≡ Cpdp mod p and Mq ≡ Cqdq mod q.
4. The final result is calculated as:

M = [((Mq + q- Mp). A) mod q].p+Mp

Where A is known as the multiplicative inverse of q and can be determined by the Euclid's extended algorithm. The decryption speed is about four times faster because the modulus is reduced to half the bit-size of the modulus n. This means that the computations are done with smaller numbers. The variables dp and dq will be referred to as the CRT decryption exponents. Since the primes p and q are only known to the receiver, the CRT decryption algorithm can only be used by the receiver to decrypt a received message [24], [25]. The data structure for storing the private key using CRT is illustrated in Figure 2.

```
public class RSA_CRT
  {
        private BigInteger _N;
        private BigInteger _P;
        private BigInteger _Q;
        private BigInteger _DP;
        private BigInteger _DQ;
        private BigInteger _InverseQ;
        private BigInteger _H;
        private BigInteger _D;
}
```

Figure 2 The data structure of RSA's private key with CRT

3.3 RSA encryption / decryption process

The RSA encryption / decryption process has been applied into three different forms in order to reveal the efficient way to implement the RSA algorithm, which will be explained below. As for the decryption process, it is implemented in two techniques: first without CRT, second with CRT. It should be mentioned that not all protocols that use RSA support the CRT implementation, so the two designs are implemented.

3.3.1 Sequential RSA implementation on the CPU

The details of the sequential implementation are given in Figure 3, which includes public class Montgomery that implements the Montgomery algorithm. It should be mentioned that this class could be reused as basic computing (thread) for multi-core CPUs and as a kernel for GPU.

3.3.2 Parallel RSA implementation on the multi-Cores CPU and many-cores GPU

The main bottleneck of the RSA encryption process is the large size of data. In order to provide a parallel implementation of the RSA, it is desired to have no dependencies between the data. As such, the data can be divided into small portions; each thread can calculate a portion. As a result, this data parallelism method increases the computing speed of RSA. On the thread level, the plaintext or the ciphertext is divided into several portions with the same length. The same encrypt or decrypt operation will be done for each portion, then the encrypt and the decrypt process can be done with multiple threads, each thread only needs to gain the elements which are assigned to it, and run the same encrypt or decrypt function for these elements (in this case Montgomery algorithm). In other words, each thread can independently undertake a modular exponentiation as illustrated in Figure 4.

The details for the multi-cores CPU's implementation and the details for the many-cores GPU's are depicted in Figure 5 and Figure 6 respectively.

With respect to implementation processes, sequential, multi-cores CPU, and many-cores GPU have been implemented using C# programming language and GPU.NET framework. Figure 7 depicts the snapshot of the variant implementations.

Input: Message (M),Public key (e,n),Private key (d,n}))Private key with CRT (d,p,q).
Output : Ciphertext or Plaintext.
Step 1: Chose the key length to encrypt or decrypt with.
Step 2: Generate the keys as mentioned in section 4.2.1.
Public key {e,n}
public struct RSA_Public_Key
Private key {d,n}
public struct RSA_Secret_Key
Private key with CRT {d,p,q}
public class RSA_CRT
Step 3: Insert the data.
Step 4: Divide the data to equal portions (multiple of 64 bits).
Step 5: IF button Encrypt=Enabled,
Then get the Public key {e,n} and define a variable named (item) that takes the data to the encryption process.
For each (item) ,from(i = 0) to (i < list_source.Count)
{
send the data (item) to the encryption function (do Encrypt)
}
Step 6: IF button Decrypt=Enabled and CRT= Flase,
Then get the Private key {d,n}and define a variable named (item) that takes the data to the decryption process.
For each (item) ,from(i = 0) to (i < list_source.Count)
{
send the data (item) to the encryption function (do Decrypt)
}
Step 7: IF button Decrypt=Enabled and CRT=True,
Then get the Private key with CRT {d,p,q} define a variable named (item) that takes the data to the decryption process.
For each (item) ,from(i = 0) to (i < list_source.Count)
{
send the data (item) to the decryption function (RSADecryptPrivateCRT)
}

Figure 3 Sequential RSA algorithm on the CPU

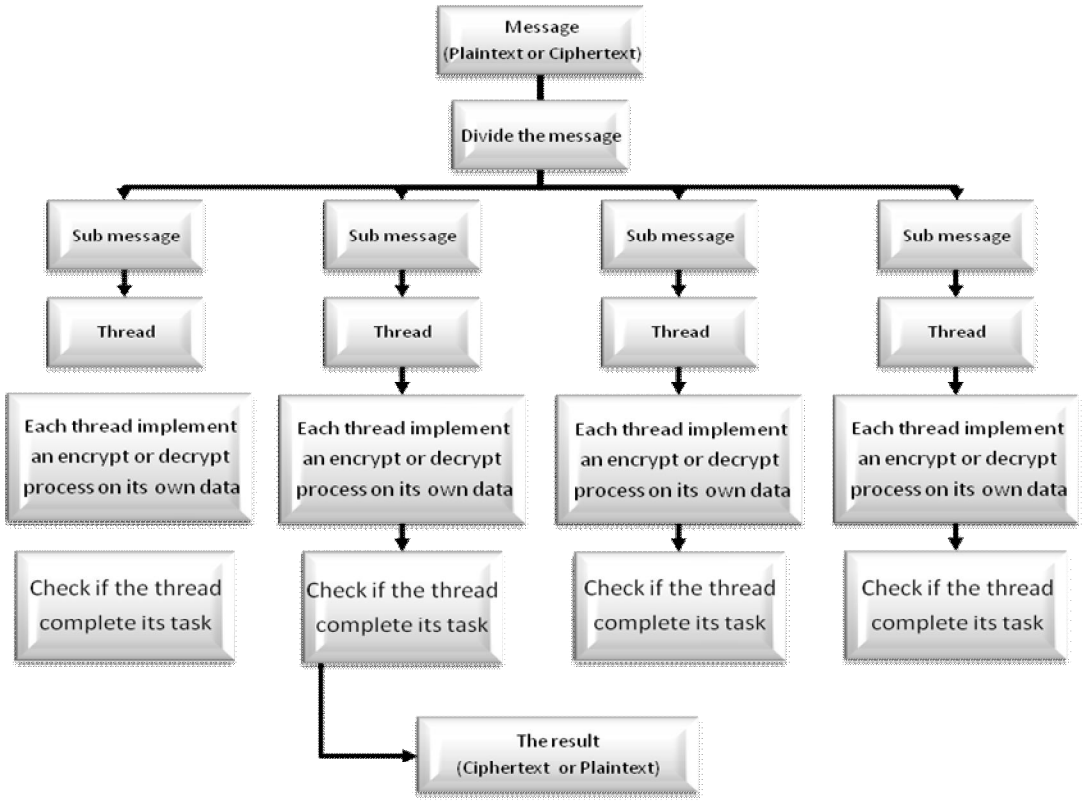

Figure 4 The structure of the thread level execution

Input: Message (M),Public key (e,n),Private key (d,n})Private key with CRT (d,p,q).
Output : Ciphertext or Plaintext.
Step 1: Chose the key length to encrypt or decrypt with
Step 2: Generate the keys
Public key {e,n}
public struct RSA_Public_Key
Private key {d,n}
public struct RSA_Secret_Key
Private key with CRT {d,p,q}
public class RSA_CRT
Step 3: Insert the data .
Step 4: Divide the data to equal portions (multiple of 64 bits).
Step 5: IF button Encrypt=Enabled,
Then get the Public key {e,n}
Step 6: Establish the Parameterized Thread Start (object obj); which represents the method that executes on a System. Threading. Thread,and needs an object that contains data for the thread procedure.
Step 7: Initializes a new instance of the System .Threading .Thread class, specifying a delegate that allows an object to be passed to the thread when the thread is started and specifying the maximum stack size for the thread.
Step 8: Start execution of threads by sending an object containing the data to be used by the method the thread executes(do encrypt function).
Step 9: Check if all the threads finish their task then collect the result.
Step10: IF button Decrypt=Enabled and CRT = Flase,
Then get the Private key {d,n} and repeat step 6&7.
Step11: Start execution of threads by sending an object containing the data to be used by the method the thread executes(do decrypt function).
Step 12: Check if all the threads finish their task then collect the result.
Step13: IF button Decrypt=Enabled and CRT = True,
Then get the Private key {d,p,q} and repeat step 6&7.
Step 14: Start execution of threads by sending an object containing the data to be used by the method the thread executes (RSADecrypt Private CRT function).
Step 15: Check if all the threads finish their task then collect the result.

Figure 5 Parallel RSA algorithm on the Multi-Cores CPU

Input: Message (M),Public key (e,n),Private key (d,n})Private key with CRT (d,p,q).
Output : Ciphertext or Plaintext.
Step 1: Chose the key length to encrypt or decrypt with
Step 2: Generate the keys
Public key {e,n}
public struct RSA_Public_Key
Private key {d,n}
public struct RSA_Secret_Key
Private key with CRT {d,p,q}
public class RSA_CRT
Step 3: Insert the data.
Step 4: Divide the data to equal portions (multiple of 64 bits).
Step 5: IF button Encrypt=Enabled,
Then call the GPU kernel (Reduce_GPU).
Step 6: Set kernel launch parameters (Set grid/ block size for GPU execution).
Launcher.SetGridSize;
Launcher.SetBlockSize ;
Step 7: Each thread has a thread id, so each portion of data is assigned to a thread through it thread id.
Step 8: Then send the data to each thread to do the encryption process.
Step 9: IF button Decrypt=Enabled and CRT = Flase,
Then call the GPU kernel(Reduce_GPU),and repeat step 6&7.
Step 10: Then send the data to each thread to do the decryption process.
Step 11: Check if all the threads finish their task then collect the result.
Step 12: IF button Decrypt=Enabled and CRT = True,
Then call the GPU kernel(RSADecryptPrivateCRT_GPU), and repeat step 6&7.
Step 13: Then send the data to each thread to do the decryption process.
Step 14: Check if all the threads finish their task then collect the result.

Figure 6 Parallel RSA algorithm on the Many-Cores GPU

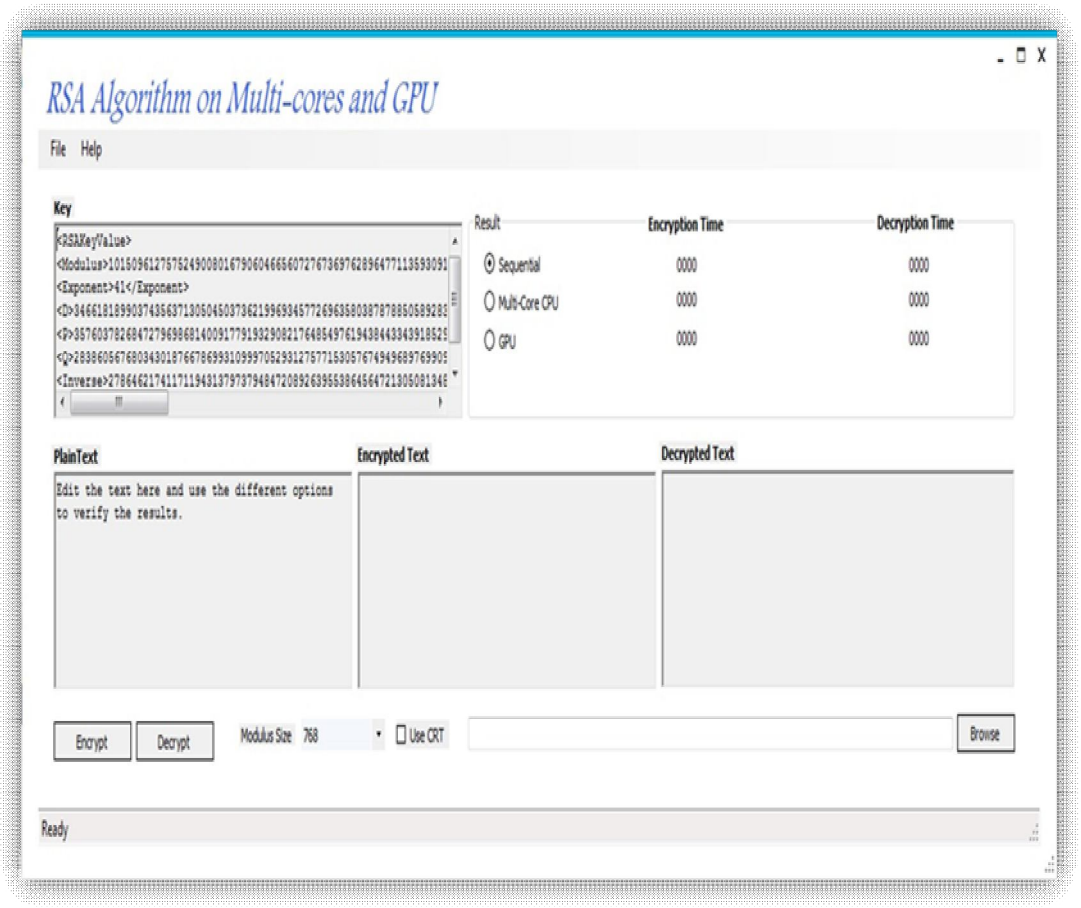

Figure 6 GUI's snapshot of the RSA Implementation.

4. RESULTS AND DISCUSSION

In order to compare the speed up gained of parallelizing RSA on multi-core CPU and many core GPU computing environments against sequential Montgomery implementations, a series of experimental groups are conducted. First, implement the sequential RSA algorithm on the CPU with various key sizes and then record the execution time and data throughput. Second, execute parallel RSA algorithm on multi-core CPU and GPU, and record related results as well, and observe the enhancement of adopting the CRT for decryption process in all cases. Table 1 shows the specifications of the platform that is adopted for evaluation purpose. The test groups are defined as follows:

- **Group 1**: Input messages varied in size that are convenient with the size of the encryption key (one byte less than modulus size).
- **Group 2:** Fixed the load size to be 600 messages to measure the throughput. Here, we are more interested in determining the speed up gain as far as the throughput is concerned. Each message is one byte less than the modulus size.

The results of applying Group1 are tabulated in Table 2. All execution times are measured in milliseconds (ms). Furthermore, the execution time shown in the tables is the average execution time (running the experiment 10 times and take the average execution time). It should be mentioned that during the experimentation the difference between a minimum and maximum time is just less than 100ms which is negligible. All row-by-row cell entries are shaded to refer to the minimum time or maximum speed up. To ensure fair speed up for the parallel implementation, we consider the sequential time of the proposed sequential version. As for the execution time, it is seen that the GPU implementation begins to be faster than the other two implementations when the key size is 3072 bits and higher for the encryption process. Compared to the decryption process, it can be seen that the time was taken to decrypt a message is more than that needed to encrypt one; that is due to the public exponent (e) which is smaller than the private exponent (d). The CRT algorithm is used with the decryption process to further increase the gained speed up; that is due less mathematical calculation, task level parallelism, and hence free resources are available to further computing. With regard to the execution time, the GPU super passes the other two for all key sizes; it can also be inferred that the GPU is more powerful with heavy computations.

Table 1: Specifications of the experiment's platform

Specifications	Platform
Processor	*Intel® Core™ 7-2670QM CPU @ 2.20GHz*
CPU Speed	*2195 MHz*
CPU Cores (Logical)	*8*
RAM	*12GB*
Hard Drive	*750GB*
Graphics Card	*GeForce GT 630M*
Operating System	*Windows 7 64-bit*
Processor Cores	*96*
Number of multiprocessors	*2*
Total amount of global memory	*2048MB*
Total amount of constant memory	*64 KB*
Total amount of shared memory per block	*48 KB*
Total amount of registers available per block	*32768*
Warp size	*32*
Maximum number of threads per block	*1024*
Maximum sizes of each dimension of a block	*1024*1024*64*
Maximum sizes of each dimension of a grid	*65536*65536*65536*
GPU Core speed	*810 MHz*
Memory Interface Width:	*128 bit*
Memory Bandwidth (GB/sec):	*32*

Table 2: The execution time (ms) for encryption/ decryption of message with variable key size

Key Size in bits	Sequential CPU			Multi-cores CPU			Many-cores GPU		
	Encryption	Decryption		Encryption	Decryption		Encryption	Decryption	
		Without CRT	With CRT		Without CRT	With CRT		Without CRT	With CRT
768	0.560	5.030	2.865	0.870	5.460	2.445	1.080	2.420	0.497
1024	0.750	11.460	4.33	0.900	9.880	3.42	1.130	2.840	0.57
2048	1.040	158.339	25.715	2.820	63.003	10.395	1.900	17.290	1.333
3072	2.990	604.494	183.854	4.380	192.931	53.403	2.600	50.122	11.445
4096	6.800	1923.450	409.806	6.220	607.834	67.803	4.640	111.546	15.855
6144	24.801	9671.104	1444.458	12.740	2062.689	193.155	8.630	461.236	22.716
8192	45.822	28628.04	2028.511	20.321	4736.691	241.158	11.760	1244.781	66.115

In order to judge the performance of the parallel implementations, the speed up is calculated for Table 2 using the speed up performance equation:

Speedup=time original / time after enhancement (1)

The speed up is tabulated in Table 3. Where S1 is the speed up for the multi-cores CPU implementation and S2 is the speed up for the many-cores GPU implementation. In addition, in the case of decryption; the speed up is calculated with

respect to original decryption (i.e., without CRT); which is denoted by S0 for sequential decryption. The results of applying Group2 are tabulated in Table 4.

Table 3: The speed up calculation for Group 1

Key Size in bits	Encryption		Decryption Without CRT		Decryption With CRT		
	S_1	S_2	S_1	S_2	S_0	S_1	S_2
768	0.644	0.519	0.921	2.079	1.756	2.057	10.121
1024	0.833	0.664	1.160	4.035	2.647	3.351	20.105
2048	0.369	0.547	2.513	9.158	6.158	15.232	118.784
3072	0.683	1.15	3.133	12.06	3.288	11.32	52.817
4096	1.093	1.466	3.164	17.244	4.694	28.368	121.315
6144	1.947	2.874	4.689	20.968	6.695	50.069	425.74
8192	2.255	3.896	6.044	22.998	14.113	118.711	433.004

According to Table 3, It is clear that the speed up increase linearly as far as the key length is concerned, due to the requirements of intensive computing. The speed up is not very high for the encryption process. Thus, it is recommended to use the sequential RSA implementation for small key size (less than 3072 bits); otherwise, it is recommended to use GPU implementation for higher key lengths as well as for decryption process with any key length. For the decryption process with 8192 bits length (without CRT), it can be seen that the multi-cores CPU only gains ~6X speed up; when the many-cores GPU gains ~23X; so the speed up is much higher with the many core GPU implementation even for small key size. Furthermore, for the decryption process with CRT, it is noticed that the adopting CRT sequential version gives a speed up gains ~14X. The multi–core gains ~119X speed up; while the many core GPU gains ~433X speed. Thus, CRT gives a significant speed up for the decryption process for all three variant implementations. In addition, in both cases for Multi-cores and Many-cores, the speed up is super due to the composition of parallel processing and CRT. As such, it is recommended to use CRT decryption whenever possible.

Table 4: The execution time (ms) for encryption/decryption of 600 messages with variable key size

Key Size in bits	Sequential CPU			Multi-cores CPU			Many-cores GPU		
	Encryption	*Decryption*		*Encryption*	*Decryption*		*Encryption*	*Decryption*	
		Without CRT	*With CRT*		*Without CRT*	*With CRT*		*Without CRT*	*With CRT*
768	707.847	2597.659	953.634	282.596	1078.142	391.582	204.711	172.749	83.773
1024	1262.272	5958.671	1678.251	439.475	2734.996	804.578	431.194	413.883	102.458
2048	3563.504	40569.542	20458.8	1120.078	16557.28	11770.27	1102.763	2504.114	483.916
3072	19415.79	62358.491	20458.79	1144.953	41458.05	15478.14	997.812	9751.023	760.464
4096	27894.54	67892.041	25089.68	2354.102	46852.26	21623.09	1985.574	11024.65	1264.88
6144	70037.62	1006485.8	31254.69	5519.416	55487.37	17254.31	4667.217	14251.07	1724.48
8192	270943.6	685697.77	26789.48	12530.47	63257.72	22036.99	9239.799	17014.54	3987.49

Refers to Table 2, To encrypt one message with 2048 bits key, 1.04 ms is taken for the sequential version which means approximately to encrypt 600 messages that would take 624 ms. But as seen in Table 4, 3563.504 ms are taken which means more time due to looping overhead as well as context switching performed by the operating system. Now, let's see the speed up gain for multi-core version. In order to encrypt one message with 2048 bits key, 2.82 ms is taken for multi-core that means to encrypt 600 messages, 1692 ms would be taken. But as seen in Table 4, 1120.078 ms are taken which means that it is 1.51 times faster than the expected one due to free resources available for computing which can be occupied by available threads and overlapping operations among threads which hide the memory latency significantly. In addition, as far as the speedup and throughput are concerned, the multi-cores CPU implementation with 2048 encryption is ~3.2X faster than the sequential one; whilst it has more latency time for a single message (refer to Table 3 it has slow down by ~X 0.37). Finally, the same observation could be noted for the GPU environment. From

Table (2), 1.9 ms is taken to encrypt one message with 2048 bits key for many core GPU that means to encrypt 600 messages which would take 1140 ms. But as we notice in Table (4), 1102.763 ms are taken which means it is 1.034 times faster than the expected one. Moreover, as far as the speedup and throughput are concerned, the many cores GPU implementation with 2048 encryption is ~3.23X faster than the sequential one; whilst it has more latency time for a single message (refer to Table 3 it has slow down by ~X 0.547). As such, it is recommended to use parallel implementation as far as the throughput is concerned for both encryption and decryption processes. The same observations are valid for variety key sizes. Gained throughput by the GPU exceeds the multi-core and sequential implementations. Furthermore, CRT gives extra throughput due to light computation associated with the decryption process in all three variant implementations as Tabulated in Table 5.

Table 5: The throughput (message per second) for encryption/ decryption processes with variable key size

Key Size in bits	Sequential CPU			Multi-core CPU			Many core GPU		
	Encryption	*Decryption*		*Encryption*	*Decryption*		*Encryption*	*Decryption*	
		Without CRT	*With CRT*		*Without CRT*	*With CRT*		*Without CRT*	*With CRT*
768	848	231	629	2123	557	1532	2931	3473	7162
1024	475	101	358	1365	219	746	1391	1450	5856
2048	168	15	29	536	36	51	544	240	1240
3072	31	10	29	524	15	39	601	62	789
4096	22	9	24	255	13	28	302	54	474
6144	9	0.6	19	109	11	35	129	42	348
8192	2	0.9	22	48	10	27	65	35	150

According to Table 5, unlike the latency enhancement, the throughput decreases as the key length increases due to extensive computation associated with a higher modulus.

5. CONCLUSIONS

This paper has been proposed three variant implementations for RSA algorithm, sequential, multi-cores CPU, and many-cores GPU, of executing modular exponentiation using the Montgomery algorithm. In addition, two variant implementations are done for the decryption process (with and without CRT). According to the practical results, the multi-cores CPU implementation gained speed up for the encryption process more than the speed up for the decryption process. While the GPU implementation also gained speed up for the encryption process, an excellent speed up is gained for the decryption process. These results are gained as far as the latency is concerned. In addition to working in a parallel manner, results show that an extra speed up can be gained by using CRT. Furthermore, additional speed up can be gained as far as the throughput is concerned. Due to overlapping of multithread operation whenever free resources are available. Results reveal that the GPU is appropriate to speed up the RSA algorithm. From our case study on parallelizing RSA algorithm on multi-cores CPU and many-cores GPU by decomposition the algorithm in independent data level and/or task level parts, a noticeable speed up and throughput can be gained. As such, further research on parallelizing different complex computational systems is the forthcoming stream to reflect the current advancement in computer architecture.

References

[1] K. Jang, S. Han, S. Moon, and K. Park, "SSL Shader: Cheap SSL Acceleration with Commodity Processors", The NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation, USENIX Association, San Jose, CA , pp. 1-14, 25-27 April, 2011.

[2] K. Zhao, and X. Chu, "GPUMP: a Multiple-Precision Integer Library for GPUs", IEEE 10th International Conference on Computer and Information Technology (CIT), Bradford, pp. 1164-1168, 29 June-1 July, 2010.

[3] Z. Huang, and S. Li, "Design and Implementation of a Low Power RSA Process for Smartcard", International Journal of Modern Education and Computer Science ,Vol. 3, No. 3, pp. 8-14, 2011.

[4] Y. Ito, K. Nakano, and S. Bo, "The Parallel FDFM Processor Core Approach for CRT-Based RSA Decryption", International Journal of Networking and Computing, Vol. 2, No. 1, pp. 79–96, 2012.

[5] E. Niewiadomska-Szynkiewicz, M. Marks, J. Jantura, and M. Podbielski, "A Hybrid CPU/GPU Cluster for Encryption and Decryption of Large Amounts of Data", Journal of Telecommunications and Information Technology (JTIT), Vol. 3, pp. 32-39, 2012.
[6] W. Fan, X. Chen, and X. Li, "Parallelization of RSA Algorithm Based on Compute Unified Device Architecture", 9th International Conference on Grid and Cooperative Computing (GCC), Nanjing, pp. 174-178, 1-5 November, 2010.
[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing", Proceedings of the IEEE, Vol. 96, No. 5, pp. 879-899, 2008.
[8] S. Antao, J. C. Bajard, and L. Sousa, "RNS-Based Elliptic Curve Point Multiplication for Massive Parallel Architectures", The Computer Journal, Oxford University Press Oxford, UK, Vol. 55, No. 5, pp. 629-647, May, 2012.
[9] P. Lara, F. Borges, R. Portugal, and N. Nedjah, "Parallel Modular Exponentiation Using Load Balancing Without Pre Computation", Journal of Computer and System Sciences, Vol. 78, No. 2, pp. 575–582, 2012.
[10] J. Fan, K. Sakiyama, and I. Verbauwhede, "Montgomery Modular Multiplication Algorithm on Multi-Core Systems", IEEE Workshop on Signal Processing Systems, pp. 261-266, 17-19 October, 2007.
[11] Z. Chen, and P. Schaumont, "A Parallel Implementation of Montgomery Multiplication on Multicore Systems", IEEE Transactions on Computers, Vol. 60, No. 12, pp. 1692-1703, 2011.
[12] S. Baktir, and E. Savas, "Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors", 27th International Symposium on Computer and Information Sciences III, Springer-Verlag London, pp. 467-476, 2013.
[13] A. Moss, D. Page, and N. P. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware", Proceedings of The 11th IMA International Conference on Cryptography and Coding, Cirencester, UK, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, pp. 364-383, December 18-20, 2007.
[14] S. Fleissner, "GPU-Accelerated Montgomery Exponentiation", 7th International Conference Computational Science, Part I – ICCS 07, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, Vol. 4487, pp. 213-220, Beijing, China, 27-30 May, 2007.
[15] R. Szerwinski, and T. Guneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography", 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 08), Springer-Verlag Berlin, Heidelberg, Vol. 5154, pp. 79-99, 2008.
[16] O. Harrison, and J. Waldron, "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware", The 2nd International Conference on Cryptology in Africa, Progress in Cryptology (AFRICACRYPT 09), Gammarth, Tunisia, Lecture Notes in Computer Science, Springer Berlin Heidelberg, Vol. 5580, pp. 350-367, June 21-25, 2009.
[17] S. Neves, and F. Araujo, "On the Performance of GPU Public-Key Cryptography", IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Santa Monica, CA, pp. 133-140, 11-14 September, 2011.
[18] Z. Yao, V. Chenghua, C. Yuwei, and L. Wei, "Efficient Acceleration of RSA Algorithm on GPU", IEEE International Conference on Oxide Materials for Electronic Engineering (OMEE), Lviv, Ukraine, pp. 531-534, 03 - 07 September, 2012.
[19] T. Li, H. Li, and J. Xiang, "A GPU-based Fine-grained Parallel Montgomery Multiplication Algorithm", Recent Advances in Computer Science and Information Engineering, Springer-Verlag GmbH Berlin Heidelberg, Lecture Notes in Electrical Engineering, Vol. 126, pp. 135-143, 2012.
[20] H. Zhang, D. F. Zhang, and X. A. Bi, "Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU", International Journal of Information and Education Technology, Vol. 2, No. 2, pp. 185-187, 2012.
[21] W. Dai, "Crypto++", available at http://www.cryptopp.com, last accessed on December 23, 2015.
[22] H. M. Fadhil, and M.I. Younis, "Parallelizing RSA Algorithm on Multicore CPU and GPU", International Journal of Computer Applications (IJCA), Vol. 87, No. 6, pp. 15-22, February 2014.
[23] H. M. Fadhil, and M.I. Younis, A Multithreading Implementation of RSA Algorithm on Multicore and GPU, LAP Lambert Academic Publishing, Germany, June 29, 2015.
[24] H. K. Haili, and N. Basir, "RSA Decryption Techniques and the Underlying Mathematical Concepts", International Journal of Cryptology Research, Vol. 1, No. 2, pp. 165-177, 2009.
[25] G. N. Shinde, and H. S.Fadewar, "Faster RSA Algorithm for Decryption using Chinese Remainder Theorem", International Conference on Computational and Experimental Engineering and Sciences (ICCES), Vol. 5, No. 4, pp.255-262, 2008.

AUTHORS

Mohammed Issam Younis obtained his BSc in computer engineering from the University of Baghdad in 1997, his MSc degree from the same university in 2001, and his Ph.D. degree from the School of Electrical and Electronics Engineering, USM, Malaysia in 2011. He is currently an associate professor and a Cisco instructor at the Computer Engineering Department, College of Engineering, University of Baghdad. He is also a software-testing expert in the Malaysian Software Engineering Interest Group (MySEIG). His research interests include software engineering, parallel and distributed computing, algorithm design, RFID applications development, embedded systems, networking, and security. Assoc. Prof. Dr. Younis is also a member and Consultant Engineer at the Iraqi Union of Engineers.

Heba Mohammed Fadhil obtained her BSc in computer engineering from the University of Al-Mustansiriya in 2006, and her MSc degree in computer engineering from the University of Baghdad in 2014. She is currently a Lecturer at the Information and Communication Department, Al-Khwarizmi College of Engineering, University of Baghdad. Her research interests include parallel processing, distributed computing, and security. Mrs. Fadhil is also a member at the Iraqi Union of Engineers.

Zainab Nadhim Jawad obtained her BSc in computer engineering from the University of Baghdad in 2004. She is currently an Engineer at the Engineering Affairs Department, University Presidency, Karbala University. Her research interests include algorithm evaluation, parallel computing, and security. Mrs. Jawad is also a member at the Iraqi Union of Engineers.