Parallelizing RSA Algorithm on Multicore CPU and GPU

Heba Mohammed Fadhil
Information and Communication Department,
Al-Khwarizmi College of Engineering, University of Baghdad Al-Jadriyah, Baghdad, Iraq

Mohammed Issam Younis, Ph.D
Computer Engineering Department,
College of Engineering, University of Baghdad
Al-Jadriyah, Baghdad, Iraq

ABSTRACT
Public key algorithms are extensively known to be slower than symmetric key alternatives in the area of cryptographic algorithms for the reason of their basis in modular arithmetic. The most public key algorithm widely used is the RSA. Therefore, how to enhance the speed of RSA algorithm has been the research significant topic in the computer security as well as in computing fields. With remarkable increase in the computing capability of the modern Graphics Processing Unit's (GPUs) as a co-processor of the CPU, one can significantly benefit from the Single Instruction Multiple Thread (SIMT) style of computing. This paper proposes a hybrid system to parallelize the RSA for multicore CPU and many cores GPUs with variable key size. In doing so, three variants implementation for the RSA algorithm are done to facilitate the performance comparison against Crypto++ library and sequential counterpart. 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, additional speedup could be gained as far as the throughput is concerned; the throughput gained for 1024 bits is ~1800 msg/sec; as for 2048 bits is ~250 msg/sec. Due to overlapping of multithread operation whenever free resources are available. The experiments are conducted on a laptop with Intel Core I7-2670QM, 2.20 GHz CPU and Nvidia GeForce GT630M GPU. Results reveal that the GPU is appropriate to speed up the RSA algorithm.

General Terms
Parallel Processing, Parallel Computing, RSA Algorithm, SIMT, Multithreading and Concurrent Computing, Heterogeneous computing.

Keywords
RSA; SIMT; GPU; Parallel algorithms; Heterogeneous computing.

1. INTRODUCTION
Over the past two decades, with the rapid evolution in the area of information technology and the internet have created imaginative applications and technologies along the way. Recently, we can send a multimedia message, or get one from almost anyone around the world in a few seconds through the internet. To guard data transmitted from snooping by someone other than the receiver. It is desired to hide the message before it is sent to a non-secure communication channel. This is achieved through encryption [1] [2]. Due to its distinctive ability to distribute and manage keys, public key encryption has become the perfect solution to information security [3].
Public key algorithms (e.g., RSA algorithm) rely essentially on hard mathematical problems (modular multiplication and modular exponentiation) of very large integers, ranging from 128 to 2048 bits. With such large numbers, the achievement of the calculation process will not be quick or easy to implement [4]. With the rapid developments in hardware and software technologies, it seems that sequential implementation of encryption algorithms are not safer and 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 [5]. Graphics Processing Units (GPUs) have been increasingly used as a powerful accelerator in several high computational demanding applications due to their flexibility and moderate cost [6]. The essential difference between CPUs and GPUs comes from how transistors are composed in the processor. CPUs use large portions of the chip area for caches; while GPUs use most of the area for Arithmetic Logic Units (ALUs) as shown in Figure 1 [7][8].

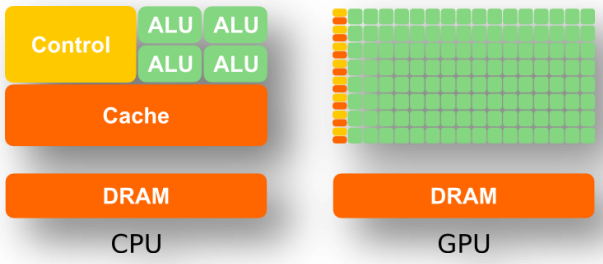


Figure 1. The Different architecture CPU vs. GPU [7]

A demanding need to increase the computational performance in science and engineering headed for heterogeneous computing and highly parallel architectures thus created a strong need for programmers to develop infrastructure in the form of libraries routine to support computing is heterogeneous hardware platforms [9]. Faster executions of public key cryptography and precisely RSA are currently of extreme importance. An RSA operation is a modular exponentiation, which requires repeated modular multiplications. Execution of fast modular multiplication for large integers is the superior concern because it provides the foundation for execution fast modular exponentiation, which is the vital operation of the RSA cryptosystem [10]. Current trends in computing society are to parallelize the sequential algorithms to gain speed up [11]. However, such parallelization is a challenging process. Motivated by such challenge this paper proposes a hybrid system to parallelize the RSA for multicore CPU and many cores GPUs. The remaining of this paper is organized as follows. Section 2 highlights the state of the art of the related works in parallelizing RSA algorithm. Section 3 reviews the RSA algorithm by adopting Montgomery exponentiation. Section 4 highlights the implementation of RSA algorithm and its parallelization. Section 5 shows the results and discusses

them. Finally, Section 6 gives the conclusion and suggestions for future work.

2. RELATED WORK

Recently, many applications have been employed GPU as a real platform to achieve efficient acceleration. To accelerate the RSA encryption/decryption, several researches used a GPU support.

Moss et al. presented the first GPU implementation of a public key primitive; which performed an implementation of 1024 bits exponentiation on NVIDIA 7800 GTX GPU using vector arithmetic in Residue Number Systems (RNS), which allowed them to capitalize the fine-grained SIMD-parallel floating point computation [12]. Their experimental results show that there is a significant latency associated with invoking operations on the GPU, due to overhead imposed by OpenGL shading language, and transfer of data to and from the accelerator. Therefore, the results were not promising, as they were limited by the legacy GPU architecture and interface by that time.

Szerwinski and Güneysu employed modular exponentiations of 1024 and 2048 bits based on both Montgomery Coarsely Integrated Operand Scanning (CIOS) and RNS arithmetic by a NVIDIA 8800GTS GPU and the Compute Unified Device Architecture (CUDA) framework to improve efficient modular exponentiation [13]. Therefore, they were able to compute 813 modular exponentiations per second for RSA with 1024 bits, and 104.3 modular exponentiations per second for RSA with 2048 bits.

Harrison and Waldron presented a high performance RSA 1024 bits modular exponentiation running on an NVIDIA 8800 GTX, which was established on integers represented in standard radix system and RNS, where they obtained a peak throughput of 0.18 ms/op that gives a 4 times improvement over an equivalent CPU implementation [14].

Fleissner proposed a 192 bits Montgomery exponentiation algorithm, indicated as "GPU-MonExp", which was executed on NVIDIA 7800GTX GPU using OpenGL shading language [15]. The performance tests had shown that its implementation run 136-168 times faster than the standard Montgomery exponentiation algorithm.

Neves and Araujo executed RSA-1024 decryption on GT200 GPUs [16]. They found that CIOS, the usual technique to interleave Montgomery multiplication and reduction, is not optimal on the GT200. Nevertheless, both Finely Integrated Product Scanning (FIPS) and Finely Integrated Operand Scanning (FIOS) found to be superior, and FIOS enabled them to reach the best-recorded performances in the GT200 architecture to their date. Their throughput results, for over 20000 RSA-1024 decryptions per second or 41426 512 bits modular exponentiations per second.

Li et al. had developed a fine-grained parallel approach for Montgomery multiplications using CUDA 2.3 platform and NVIDIA GeForce GTX285 GPU [17]. The experiment showed that their implementation could get ten times of acceleration compared to the implementation of comparable algorithm on CPU.

As stated in the researches above, it is has been seen that the key length used for encryption and decryption does not exceed 2048 bits, as well as either of the research conducted a fair comparison between the speed of implementation of the RSA algorithm on the CPU and GPU. As such, this paper gives a study on increasing the length of the key and makes a fair comparison between implementation of the RSA algorithm on the CPU and GPU.

3. RSA ALGORITHM

RSA algorithm, invented by Rivest, Shamir and Adelman in 1978 [18], is one of the famous algorithms for public-key cryptography. It is appropriate for encryption and digital signature. RSA is the utmost far used algorithm in Internet security [19] [10]. In fact, Internet security depends significantly on the security properties of the RSA cryptosystem. Its security depends upon the insolvability of the integer factorization problem and is believed to be vulnerable given sufficiently long keys, such as 1024 bits or more [20].

The RSA algorithm consists of three steps which include key generation, encryption and decryption ones. It is comprised of public and private keys. Messages encrypted with the public key can only be decrypted using the private ones. The RSA algorithm can be summarized in the following steps [18] [21]:

Step 1: Generate randomly two large prime's p and q of approximately the same size, but not too close together. Which are kept secret.

Step 2: Calculate the modulus $n = p*q$. and Calculate: $\varphi(n) = (p-1)(q-1)$; Where $\varphi(n)$ represents the Euler Totient function.

Step 3: Choose a random encryption exponent e less than n such that the GCD $(\varphi(n), e) = 1$, $1 < e < \varphi(n)$.

Step 4: Calculate the decryption exponent d using The Extended Euclidean algorithm: $d.\ e = 1 \bmod \varphi(n)$. Which d is the multiplicative inverse of e modulo $\varphi(n)$.

Step 5: The encryption function is: $E(M) = M^e \bmod n$.

Step 6: The decryption function is: $D(C) = C^d \bmod n$.

Step 7: The RSA keys are: The public key is (n, e), and the private key is (p, q, d).

3.1 Modular Arithmetic in RSA Algorithm

Operands' size is considered essential in mathematic calculation. Therefore, two primes p and q should be chosen to have just about the same bit length to guarantee that any efforts to factor the modulus are computationally infeasible. For security reasons, RSA operands' size needs to be 1024-bits or greater in length which leads to high data throughput rates that are difficult to accomplish [22].

Modular multiplication is used to implement modular exponentiations, which in their turn, are used by several public-key cryptosystems. The performance of public-key cryptosystems is mainly determined by the implementation's efficiency of the modular exponentiation. Therefore, modular multiplication is a vital factor in these systems [22].

3.2 Montgomery Algorithm

Montgomery algorithm presented by Peter Montgomery in 1985, which is the most rare algorithm used in public-key cryptography; serve as an efficient algorithm for modular multiplication and exponentiation operations [23]. Montgomery algorithm allows modular arithmetic to be accomplished efficiently when the modulus is large (1024 bits or more). The Montgomery algorithm consists of two approaches: multiplication and reduction.

Montgomery multiplication is a method for computing ***a. b mod n*** for positive integers *a*, *b*, and *n*. It moderates execution time on a computer when there are large numbers of multiplications to be done with the same modulus n, and with a small number of multipliers. In precise, it is useful to compute $a^e \bmod n$ for a large value of *n*. The amount of multiplications modulo n in such a computation can be completed in a number considerably less than *n* by successively squaring and multiplying according to the pattern of the bits in the binary expression for *n*. Therefore, it eliminates the mod *n* reduction steps and as a result, tends to reduce the size of the timing characteristics.
In common, Montgomery multiplication algorithm computes the Montgomery product as specified by [24].

$$MonMul(a', b') = a'.b'.r^{-1} \pmod n$$

Where the multipliers *a* and *b* are less than the modulus *n*. it is needed to declare another integer *r* which must be greater than *n*, as the gcd (*r*, *n*) = 1. The method, really, changes the reduction modulo *n* to *r*. usually *r* is chosen to be an integral power of 2. Therefore, the reduction modulo *r* is simply a masking operation. If *r* is a reduction modulo power of 2, it should have an odd n, to satisfy the GCD requirement [17]. The computation of ***MonMul (a, b)*** is given in Algorithm 1.

Algorithm 1: Montgomery Multiplication Algorithm [15] [24].

Step 1: Input an odd modulus n and a radix

r = $2^{\lceil \log_2 n \rceil}$, such that GCD (n, r) = 1, an auxiliary value n'= $-n^{-1} \bmod r$, 2 n-residue integers a' and b'.

Step 2: function: MonMul (a', b').

Step 3: Calculate t: = a' .b'.

Step 4: Calculate u: = (t + [t. n' mod r]. n) / r.

Step 5: If u ≥ n then return (u-n) else return u.

Step 6: Output a' .b' $.r^{-1} \pmod n$.

As held from the ***MonMul ( )*** function above, a and b is numbers that represent the n-residues, which can be calculated as follows [24]:

$$a' = a.r \bmod n$$
$$b' = b.r \bmod n$$

The two integer's r^{-1} and n' are calculated, by using the Extended Euclidean algorithm, such that:

$$r\,r^{-1} - n\,n' = 1$$

The final result of the Montgomery multiplication will be in the n-residue as follows [24]:

$$u' = a'.b'.r^{-1} \bmod n$$

Finally, a conversion step has to be performed to transform the result back from the n-residue representation to normal residue representation [24].

$$u = MonMul(u', 1)$$

The computation of the modular exponentiation $x = a^e \bmod N$ using Montgomery multiplications is shown in Algorithm 2.

Algorithm 2: Montgomery Reduction Algorithm [15] [24].

Step 1: Input a, e, n.

Step 2: Function: MonExp (a, e, n).

Step 3: Calculate a'= a .r mod n.

Step 4: Calculate x'= 1 .r mod n.

Step 5: for i = n − 1 down to 0 loop

x' = MonMul(x', x')

If e_i= 1; then

x' = MonMul(x', a')

End loop.

Step 6: x = MonMul(x', 1)

Step 7: return x.

Step 8: Output: $a^e \bmod n$.

4. IMPLEMENTATION OF RSA ALGORITHM

Cryptographic algorithm is recognized as compute-intensive algorithms. Therefore, this section considers four variants implementation, namely: Crypto++ library [25], sequential Montgomery, multithreaded, and GPU based to facilitate the performance comparison among them.

The RSA algorithm consist of three main stages: key, namely: generation, encryption, and decryption. In order to represent such large numbers as 1024 bits and higher the Biginteger Class is used; so the keys are generated according to the step mentioned in Section 3.

- Public key {e,n}

```
public struct RSA_Public_Key
{
  public BigInteger n;
  public BigInteger e;
 }
```

- Privte key

```
public struct RSA_Secret_Key
{
  public BigInteger n;
  public BigInteger d;
  public BigInteger p;
  public BigInteger q;
 }
```

The CPU carries out the key generation. As for the encryption and decryption process it is handled with these four cases:

1. A standard known library that efficiently runs RSA algorithm; Crypto++ library was implemented that uses a standard key size 1024 bits.

2. A sequential implementation of the RSA algorithm runs on the CPU by implementing the Montgomery algorithm.
3. An RSA parallel implementation executed on the multicore CPU.
4. An RSA parallel implementation executed on the many core GPU.

Unlike Crypto++ library, the proposed variant implementations support variable key size as demand. The main bottleneck of the RSA encryption process is the large size of data; so after studying the RSA encryption process; In order to provide a parallel implementation of the RSA, it is desired to have no dependencies between the data. As so, 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.

In the thread level, the plaintext or the cipher text 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 need 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.

The details of sequential implementation are given in Algorithm 3. Algorithm 3 includes public class Montgomery that implements the Montgomery algorithm mentioned in Section 3.2. It should be mentioned that this class could be reused as basic computing (thread) for multicore CPUs and as a kernel for GPU, as depicted in Algorithm 4 and Algorithm 5 respectively.

In the threading structure for GPU, when a kernel is called it will run on a grid. The number of block and threads on a grid can be constructed. On modern GPUs, a thread block may hold up to 1024 threads [26]. Threads can entrance diverse memory locations. Every single thread has a private memory. Every block has a shared memory, which is reachable for every thread within the same block. All threads form different blocks have access to the global memory. However, a kernel could be implemented by multiple blocks of threads, so the overall number of threads is equivalent to the number of threads per block times the number of blocks. As depicted in Figure 2.

Algorithm 3: Sequential RSA implementation on the CPU.

Step 1: Generate the keys as mentioned in section 3.
- Public key {e,n}
  - public struct RSA_Public_Key
- Private key {d,p,q}
  - public struct RSA_Secret_Key

Step 2: Insert the text that will be encrypted from a file or typing it.

Step 3: Send the data to a for loop to do encryption

```
For (int i = 0; i < list_source.Count; i++)
  {
    var item = new {Id = i, Data = list source[i]}
  };
```

Step 4: The encryption process is done using public Encrypt (Big Integer biPlain, RSA_Public_Key rpkKey)

Algorithm 4: Parallel RSA implementation on the multicore CPU.

Step 1: Generate the keys as mentioned in section 3.
- Public key {e,n}
  - public struct RSA_Public_Key
- Private key {d,p,q}
  - public struct RSA_Secret_Key

Step 2: Insert the text that will be encrypted from a file or typing it.

Step 3: Create a pool of threads

```
ThreadStartsList = new
List<ParameterizedThreadStart> ();
```

Step 4: Each thread will take a portion of data to implement encryption on it.

```
For (int i = 0; i < list_source.Count; i++)
{
  ParameterizedThreadStart ts = delegate
  (object o);
   {doEncrypt ((ThreadParameters)o); };
   threadStartsList.Add (ts);
}
```

Step 5: The encryption process can be executed by using:

```
Public Encrypt (BigInteger biPlain,
RSA_Public_Key rpkKey)
```

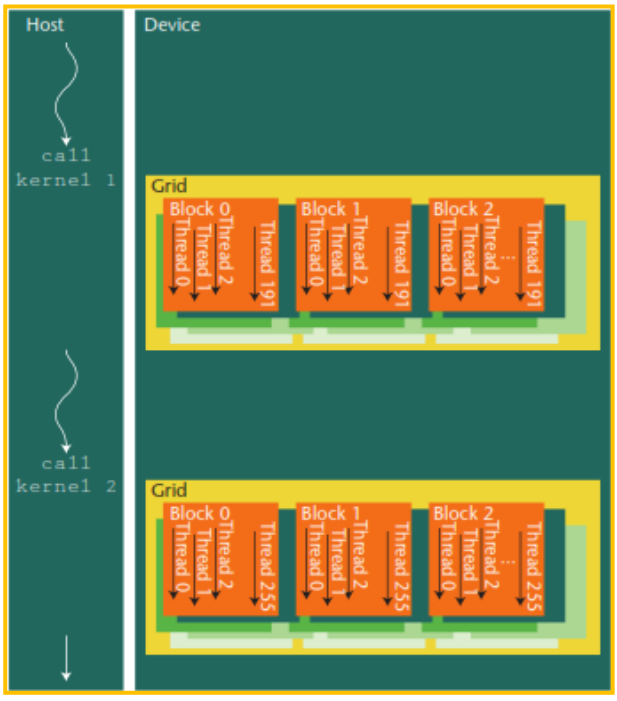


Figure 2. Threading structure [27]

Algorithm 5: Parallel RSA implementation on the many core GPU.

Step 1: Generate the keys as mentioned in section 3.

- Public key {e,n}
  - public struct RSA_Public_Key
- Private key {d,p,q}
  - public struct RSA_Secret_Key

Step 2: Insert the text that will be encrypted from a file or typing it.

Step 3: Set kernel launch parameters (Set grid/ block size for GPU execution).

```
Launcher.SetGridSize (512);
Launcher.SetBlockSize (128);
```

Step 4: Call kernel method (GPU kernel)

```
Reduce_GPU (A, n, m, mPrime);
```

Step 5: Get the thread id and total number of thread.

```
Int ThreadId = BlockDimension.X * Blo
ckIndex.X + ThreadIndex.X;
Int TotalThreads = BlockDimension.X *
GridDimension.X;
```

It should be mentioned that the proposed variants implementations are implemented using C# programming language and GPU.net framework.

5. PERFORMANCE EVALUATION

In order to compare the speedup gain of parallelizing RSA in multicore CPU and GPU computing environments against Crypto++ library and Sequential Montgomery, a series of experimental groups are conducted.

The speed up factor is a measure that captures the comparative benefit of solving a computational problem in parallel. The speed up factor of parallel computation operating on p processors is derived as the following ratio [28]:

$$S_p = \frac{Ts}{Tp} \quad \ldots\ldots\ldots\ldots \textit{(eq. 1)}$$

Where T_s is the execution time taken to perform the computation on one processor and T_p is the execution time needed to perform the same computation using p processors.
Another two important aspects must be considered latency and throughput. Latency is the time taken to process an individual data item through the processing, while throughput is the total number of processed data that occur in a given amount of time [29]. The test groups are defined as follows:

Group 1: the message size is fixed to 760 bits, which is encrypted and decrypted with varied key size from 768 to 8192 bits.

Group 2: input messages varied in size that is convenient with the size of the encryption key (one byte less than modulus size).

Group 3: Varying the block size to be multiple of message size in steps of 50 to 600. Here, we are more interested to determine the speed up gain as far as the throughput is concerned.

The experiment was carried out on a laptop with the following specification:

- CPU: Intel Core I7-2670QM at 2.20GHz clock frequency.
- Memory: 12.0 GB.
- GPU: NVIDIA GeForce GT630M consists of 96 cores.
- System: Windows 7 Home Premium.

The results of applying group1 are tabulated in Table 1 and 2; as the results of applying group2 are tabulated in Table 3 and 4. According to Table 1, it is clear that no significant difference in execution time between the Crypto++ library and our sequential implementation. To ensure fair speed up for the parallel implementation, we consider the sequential time of our sequential version. As for the execution time shown in Table 1 and Table 3, it is seen that the GPU implementation begin to be faster than the other two implementation when the key size is 3072 bits and higher.

Table 1. The execution time (latency) in Milliseconds for encryption of 760 bits message length with variant key size

Key Size in bits	Crypto++	Sequential CPU	Multi - thread CPU	Many core GPU
768	---	0.110	0.87	1.08
1024	0.500	0.130	0.94	0.92
2048	---	0.49	1.28	0.91
3072	---	0.85	1.4	0.8
4096	---	1.54	1.9	0.99
6144	---	4.01	3.13	1.95
8192	---	5.93	3.9	1.980

Table 2. The execution time (latency) in Milliseconds for decryption of 760 bits message length with variant key size

Key Size in bits	Crypto++	Sequential CPU	Multi-thread CPU	Many core GPU
768	---	5.03	5.46	2.42
1024	7.000	8.89	7.89	2.78
2048	---	76.294	38.662	9.27
3072	---	250.034	73.984	23.621
4096	---	411.453	140.378	41.592
6144	---	1727.579	369.301	93.315
8192	---	2664.313	724.961	201.071

From Table 2 and Table 4 it can be seen that the time taken to decrypt a message is more than that needed to encrypt one; that is due the public exponent e is taken small than the private exponent.as for the execution time the GPU exceed the other two for all key size; also it can be inferred that the GPU is more powerful with heavy computations.

Table 3. The execution time in (Milliseconds) for encryption of variant key size

Key Size in bits	Sequential CPU	Multithread CPU	Many core GPU
768	0.56	0.87	1.08
1024	0.75	0.9	1.13
2048	1.04	2.82	1.9
3072	2.99	4.38	2.6
4096	6.8	6.22	4.64
6144	24.801	12.74	8.63
8192	45.822	20.321	11.76

In order to judge the performance of the parallel implementation, the speed up factor is calculated for Table 1, 2, 3, and 4 as explained in equation 1; where S_1 is the speed up factor for the multithread CPU implementation and S_2 is the speed up factor for the many core GPU implementation.

Table 4. The execution time in (Milliseconds) for decryption of variant key size

Key Size in bits	Sequential CPU	Multithread CPU	Many core GPU
768	5.03	5.46	2.42
1024	11.46	9.88	2.84
2048	158.339	63.003	17.29
3072	604.494	192.931	50.122
4096	1923.45	607.834	111.546
6144	9671.104	2062.689	461.236
8192	28628.04	4736.691	1244.781

According to Table 5 and 6, the speed up factor is not very high for the encryption process; however, for the decryption process it can be seen that the multithread only gains ~6X speed up; when the many core GPU gains ~23X; so the speed up is much higher with the many core GPU implementation.

Table 5. The speed up factor calculated for Table 1 and Table 2

Key Size in bits	Encryption		Decryption	
	S_1	S_2	S_1	S_2
768	0.126	0.101	0.921	2.078
1024	0.138	0.141	1.224	3.917
2048	0.382	0.538	1.973	8.230
3072	0.607	1.062	3.379	10.585
4096	0.810	1.555	3.566	12.035
6144	1.281	2.056	4.677	18.513
8192	1.520	2.994	5.166	18.627

Table 6. The speed up factor calculated for Table 3 and Table 4

Key Size in bits	Encryption		Decryption	
	S_1	S_2	S_1	S_2
768	0.643	0.518	0.921	2.078
1024	0.166	0.132	1.159	4.035
2048	0.368	0.547	2.512	9.157
3072	0.682	1.150	3.133	12.06
4096	1.093	1.465	3.164	17.243
6144	1.946	2.873	4.688	20.967
8192	2.254	3.896	6.043	22.998

The results of applying group3 are tabulated in Table 7 and 8; from Table 3 to encrypt one message with 1024 bits key it takes 0.75 ms for sequential version which means to encrypt 600 messages it would take 450 ms but as we see in Table 7 it takes 1262.272 ms which means more time due to looping overhead; now let see the speed up gain for multithread version. From Table 3 to encrypt one message with 1024 bits key it takes 0.9 ms for multithread that means to encrypt 600 messages it would take 540 ms but as we see in Table 7 it takes 439.475 ms which means it is 1.228 times faster than the expected one due to free resources available for computing that could be occupied by available threads. Finally, the same observation could be noted for the GPU environment. From Table 3 to encrypt one message with 1024 bits key it takes 1.13 ms for many core GPU that means to encrypt 600 messages it would take 678 ms but as we see in Table 7 it takes 431.194 ms, which means it is 1.572 times faster.

Table 7. The execution time in (Milliseconds) for encryption of No of messages with 1024 bits key

No. of Message	Crypto++	Sequential CPU	Multi thread CPU	Many core GPU
50	72.04	78.004	29.341	33.321
100	165.1	155.408	59.383	61.603
150	181.06	179.71	105.106	96.675
200	352.04	348.92	131.147	131.177
250	400.213	389.014	153.64	147.64
300	425.69	417.001	160.76	148.63
350	444.11	438	187.09	179.41
400	551.24	544.543	281.446	275.215
600	1301.45	1262.272	439.475	431.194

While for the results from Table 8 of the decryption process; it can be seen that the GPU implementation is approximately 14 time faster, while the multithread CPU is 2.17 faster; this approve the ability of the GPU to deal with large data **.**

Table 8. The execution time in (Milliseconds) for decryption of No of messages with 1024 bits key

No. of Message	Crypto++	Sequential CPU	Multi Thread CPU	Many core GPU
50	623.5	540.63	216.522	47.042
100	1217.051	1136.935	476.607	78.534
150	2360.61	1796.353	580.793	99.445
200	2626.078	2243.748	939.853	145.188
250	2715.921	2390.824	1020.292	157.43
300	3050.214	2986.116	792.691	158.76
350	3560.041	3367.687	1124.443	215.53
400	4740.12	4697.208	1851.426	273.635
600	6021	5958.671	2734.996	413.883

The throughput (message processed per second) for decryption process with key sizes 1024 and 2048 bits are shown in Figures 3, and 4 respectively. The throughput gained by the GPU exceeds the multithread and sequential implementations; the throughput gained for 1024 bits is ~1800 msg/sec; as for 2048 bits is ~250 msg/sec.

6. CONCLUSIONS

This paper proposed three variants implementations of executing modular exponentiation using the Montgomery algorithm, which is the essential computational operation in

cryptosystems like RSA; the 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, additional speedup could be gained as far as the throughput is concerned; the throughput gained for 1024 bits is ~1800 msg/sec; as for 2048 bits is ~250 msg/sec. Due to overlapping of multithread operation whenever free resources are available. Results reveal that the GPU is appropriate to speed up the RSA algorithm. In the future work, we will focus on implementing another cryptographic algorithm like elliptic curve.

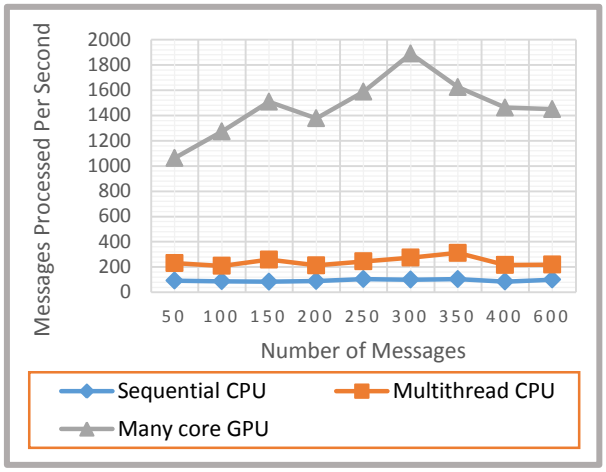
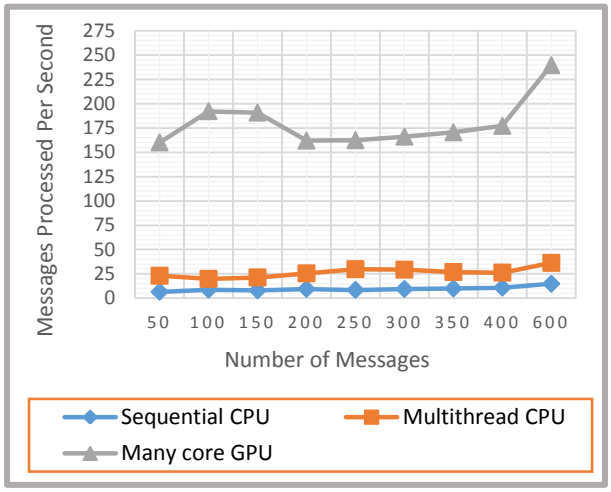


Figure 3. 1024 bits decryption throughput

Figure 4. 2048 bits decryption throughput



7. ACKNOWLEDGMENTS

The authors desire to express their gratitude and thanks to the computer center at University of Baghdad for their support to this work, and offer thanks and appreciation for everyone who assists us to do this work.

8. REFERENCES

[1] Damrudi, M. and Ithnin, N. "State of the Art Practical Parallel Cryptographic Approaches", Australian Journal of Basic and Applied Sciences, Vol. 5, No.7, pp. 660-677, 2011.

[2] Rasool, S.; Qyser, A.; Rizwanullah, M. and Ghori, M. "Secure Data Transmission over Networks", Asian Journal of Computer Science and Information Technology, Vol. 2, No. 8, pp. 257– 261, 2012.

[3] Huang, Z. and Li, S "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] Sepahvandi, S.; Hosseinzadeh, M.; Navi, K. and Jalali, A. "An Improved Exponentiation Algorithm for RSA Cryptosystem", International Conference on Research Challenges in Computer Science, ICRCCS '09, pp.128-132, 2009.

[5] Lara, P.; Borges, F.; Portugal, R. and Nedjah, N. "Parallel modular exponentiation using load balancing without pre computation", Journal of Computer and System Sciences, Vol.78, No.2, pp. 575–582, 2012.

[6] Owens, J.; Houston, M.; Luebke, D.; Green, S.; Stone, J. and Phillips, J. "GPU Computing", Proceedings of the IEEE, Journals & Magazines, Vol. 96, No.5, pp. 879-899, 2008.

[7] Gupta,S. and Babu, M. "Performance Analysis of GPU compared to Single core and Multi-core CPU for Natural Language Applications", International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 2, No. 5, pp.50-53, 2011.

[8] Su, C.; Lan, C.; Huang, L. and Wu, K. "Overview and Comparison of OpenCL and CUDA Technology for GPGPU", IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), pp. 448 – 451, 2012.

[9] Thouti, K. and Sathe, S." Comparison of Open MP and Open CL Parallel Processing Technologies", International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 3, No.4, pp. 56-61, 2012.

[10] Rao, R.; Lakshmi, P.; Shankar, N. "A Novel Modular Multiplication Algorithm and its Application to RSA Decryption", International Journal of Computer Science Issues (IJCSI), Vol. 9, Issue 6, No 3, pp. 303-309, November, 2012.

[11] Hwu, W.; Keutzer, K. and Mattson, T.G. "The Concurrency Challenge", IEEE Design & Test of Computers, Vol. 25, No.4, pp. 312-320, 2008.

[12] Moss, A.; Page, D. and Smart, N. "Toward Acceleration of RSA Using 3D Graphics Hardware", Proceedings of the 11th IMA International Conference on Cryptography and Coding, pp. 364-383, 2007.

[13] Szerwinski, R. and Güneysu, T. "Exploiting the Power of GPUs for Asymmetric Cryptography", 10th International Workshop on Cryptographic Hardware and Embedded Systems – CHES 08 , Washington, D.C., USA, Volume 5154, pp. 79-99, 2008.

[14] Harrison, O. and Waldron, J. "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware", The 2nd International Conference on Cryptology in Africa, Progress in Cryptology (AFRICACRYPT 09), Lecture Notes in Computer Science ,Volume 5580, PP. 350-367, 2009.

[15] Fleissner, S. "GPU-Accelerated Montgomery Exponentiation", 7th International Conference

Computational Science – ICCS 07, Beijing, China, Lecture Notes in Computer Science, Vol. 4487,Springer ,pp. 213-220, 2007.

[16] Neves, S. and Araujo, F. "On the Performance of GPU Public-Key Cryptography", IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 133-140, 2011.

[17] Li, T.; Li, H. and Xiang, J. "A GPU-based Fine-grained Parallel Montgomery Multiplication Algorithm" , Recent Advances in Computer Science and Information Engineering, Vol. 126, pp. 135-143 , 2012.

[18] Rivest, R.; Shamir, A. and Adleman, L. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Published in Magazine Communications of the ACM, New York, NY, USA. Volume 21, Issue 2, pp. 120-126, February 1978.

[19] Alijani, G.; Christy, J.; Craft, H.; Mok, P. and Welsh, J. "Design and Implementation of an Information Security Model for E-Business", Information Systems Education Journal, Vol. 4, No. 4, pp. 1-13, 2006.

[20] Zhao, L.; Iyer, R.; Makineni, S. and Bhuyan, L. "Anatomy and Performance of SSL Processing ", IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 05, pp.197-206, 2005.

[21] Ramachandra Rao, G. A. V.; Lakshmi, P. V.; Ravi Shankar, N." RSA Public Key Cryptosystem using Modular Multiplication " International Journal of Computer Applications**,** Vol. 80, No5, pp.38-42, October 2013.

[22] Selçuk, B. and Erkay, S. "Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors", Computer and Information Sciences III, 27th International Symposium on Computer and Information Sciences, pp. 467-476, 2013.

[23] Montgomery, P. "Modular Multiplication without Trial Division", Mathematics of Computation, Vol. 44, No.170, pp. 519-521, 1985.

[24] Koc, C.; Acar, T. and Kaliski, B. "Analyzing and Comparing Montgomery Multiplication Algorithms", IEEE Micro, Vol. 16, No.3, pp. 26-33, 1996.

[25] Crypto++ web site, available at: http://www.cryptopp.com, last accessed on 6 January 2014.

[26] CUDA C PROGRAMMING GUIDE, 2013, NVIDIA.

[27] Hwu, W.; Rodrigues, C.; Ryoo, S. and Stratton, J." Compute Unified Device Architecture Application Suitability", IEEE Computing in Science & Engineering, Vol.11, No. 3, pp. 16 - 26 2009.

[28] Roosta, S. " Parallel Processing and Parallel Algorithms: Theory and Computation ", ISBN: 0-387-98716-9, Springer Verlag, 2000.

[29] Kermarrec, A.; Bougé, L.and Priol, T. "Euro-Par 2007 Parallel Processing ", 13th International Euro-Par Conference: Lecture Notes in Computer Science, ISBN 978-3-540-74465-8, Vol. 4641, 2007.