Implementation of Secure, Fast and Efficient RSA algorithm on GPU

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

in

Information Security

Submitted By

Sonam Mahajan

(801233022)

Under the supervision of:

Dr. Maninder Singh
Associate Professor

THAPAR UNIVERSITY

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

July 2014

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Implementation of Secure, Fast and Efficient RSA algorithm on GPU"** in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Maninder Singh** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Sonam

(Sonam Mahajan)
Roll No. 801233022

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Maninder Singh)
Associate Professor
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to Dr. Maninder Singh, my mentor and thesis supervisor for his constant support and motivation. He had been instrumental in guiding me throughout the thesis with his valuable insights, constructive criticism and interminable encouragement.

I would like to thank all the faculty members and staff of Computer Science and Engineering department who were always there at the end of the hour and provided all the help and facilities, which I required, for the completion of this work.

I offer my deepest gratitude to my family for their support and affection and for believing in me always. I also want to thank my colleagues , who have given me moral support and their relentless advice throughout the completion of this work

(Sonam Mahajan)

Roll No. 801233022

ABSTRACT

In the field of cryptography, public key algorithms satisfy all the four requirements of security: Confidentiality, Integrity, Non-repudiation, Authentication as compared to the conventional cryptosystem. Public key cryptography is widely known to be slower than symmetric key alternatives for massive computations of modular arithmetic. The modular arithmetic and modular exponentiation makes RSA computationally expensive when compared to symmetric algorithms. Therefore, to make a more efficient, secure and faster implementation of RSA algorithms is publicly concerned.

With the development of the GPGPU (General-purpose computing on graphics processing units) field, more and more computing problems are solved by using the parallel property of GPU (Graphics Processing Unit). NVIDIA's proprietary CUDA and the Khronos Group's open standard OpenCL are the two frameworks designed to exploit the massive parallelism .

The target in this work is to develop a secure, faster and efficient RSA using CUDA and OpenCL. Finally, this work will compare the performance between the CUDA-RSA and CPU-RSA, OpenCL-RSA and CPU-RSA and CUDA-RSA and OpenCL-RSA for small and large prime numbers, in order to look into the possibility of improving the performance of RSA algorithms

TABLE OF CONTENTS

CERTIFICATE…… i
ACKNOWLEDGEMENT…… ii
ABSTRACT…….. iii
TABLE OF CONTENTS…….. iv
LIST OF FIGURES…… vi
LIST OF TABLES…… viii

CHAPTER 1: INTRODUCTION…….. 1

CHAPTER 2: LITERATURE REVIEW…….. 5
2.1 RSA Algorithm…… 5
2.1.1 Operation…… 5
2.1.2 Modular Exponential……7
2.1.3 Advantages and Disadvantages of RSA Algorithm……. 10
2.2 GPU…….. 10
2.2.1 Idea Behind GPU Programming…….. 10
2.2.2 Technology Trends Showcasing CPU Speed Remaining Flat..11
2.3 CUDA……... 14
2.3.1 Programming Interface …….. 15
2.3.2Memory Hierarchy…….. 17
2.3.3Program in CUDA…… 18
2.4 OpenCL System Model…….. 19
2.5 Porting CUDA Application to OpenCL Program……21
2.5.1Terminology…….. 22
2.5.2 Writing Kernels: Qualifiers……. 22
2.5.3 Writing Kernels: Indexing…….. 23
2.5.4 Writing Kernels: Synchronization…… 23
2.5.5 Important API calls……. 24
2.6 CUDA versus OpenCL……. 24
2.7 Existing Work…….. 25
2.8 Chapter Summary…… 28

CHAPTER 3: PROBLEM STATEMENT ……. 29

3.1 Objectives ……. 29

CHAPTER 4: IMPLEMENTATION…… 30

4.1 Algorithm for CUDA Kernel…….. 30

4.1.1 Integration of C with CUDA API…….. 32

4.2 Algorithm for OpenCL Kernel……... 32

4.2.1OpenCL Host Code in Detail……34

4.3 Chapter Summary…… 38

CHAPTER 5: EXPERIMENTAL RESULTS……. 39

5.1 Case 1: Analysis of CUDA-RSA and CPU-RSA for Small Prime Numbers…… 39

5.2 Case 2: Analysis of OpenCL-RSA and CPU-RSA for Small Prime Numbers……. 43

5.3 Analysis of the Speedup Graph of the Above Two Cases……. 46

5.4 Case 3: Analysis of CUDA-RSA and OpenCL-RSA for Large Prime Numbers…… 47

5.5 Chapter Summary…….. 50

CHAPTER 6: CONCLUSIONS AND FUTURE SCOPE……. 51

6.1 Discussion……. 51

6.2 Conclusion……. 51

6.3 Future Scope…… 52

REFERENCES…… 53

LIST OF PUBLICATIONS …….. 56

LIST OF FIGURES

Figure 1.1 Working of GPU Acceleration Process…… 2
Figure 1.2 Comparison of Multicore Machines and GPUs(Nvidia)…… 3
Figure 2.1 RSA Key Generation Process…… 6
Figure 2.2 RSA Key Generation Example…… 6
Figure 2.3 Figure Showing the Basic Modular Operations……... 7
Figure 2.4 Figure Explaining the Naive Modular Exponentiation Process…….. 7
Figure 2.5 Algorithm for Right-to-Left Binary Modular Exponentiation…… 8
Figure 2.6 Algorithm for Left-to-Right Binary Modular Exponentiation…… 8
Figure 2.7 Left-to-Right k-aray Exponentiation…… 9
Figure 2.8 Algorithm for Sliding Window Exponential…… 9
Figure 2.9 Figure Showing the Feature Size of Transistors with Time…… 11
Figure 2.10 Figure Showing the CPU Speed Remaining Flat…….. 12
Figure 2.11 Figure Showing the Computational Power and Memory Bandwidth with Time …… 14
Figure 2.12 CUDA System Model……. 15
Figure 2.13 Kernel Code and Execution Process of SIMD Addition Instruction in CUDA ……. 16
Figure 2.14 Memory Hierarchy of CUDA…… 17
Figure 2.15 Flow of CUDA Program…….. 18
Figure 2.16 OpenCL Platform Model…… 19
Figure 2.17 OpenCL Execution Model……. 19
Figure 2.18 Figure Explaining the Card Game Analogy……. 21
Figure 2.19 General Terminology…… 22
Figure 2.20 Porting of Kernel: Qualifiers…… 22
Figure 2.21 Porting of Kernel: Indexing…….. 23
Figure 2.22 Synchronization of the Kernels…… 23
Figure 2.23 Important API Calls……. 24
Figure 4.1 CUDA-C Host Code Integration with Kernel Code…… 32
Figure 4.2 Accessing of Platforms and Device Ids in CUDA…….. 34
Figure 4.3 Creation of Context in OpenCL…… 34

Figure 4.4 Creation of Command Queue……. 35
Figure 4.5 Creation of File in OpenCL…… 35
Figure 4.6 Creating and Building Programs in OpenCL…… 35
Figure 4.7 Setting of Kernel Arguments…….. 36
Figure 4.8 Managing Data Transfer in OpenCL…….. 37
Figure 4.9 Performing Operation in OpenCL …….. 37
Figure 4.10 Copying Back the Results to the Host……37
Figure 4.11 Wait for the Successful Completion of the Results…….. 38
Figure 5.1 Comparison of CUDA-RSA and CPU-RSA for 512 Bytes Data……40
Figure 5.2 Comparison of CUDA-RSA and CPU-RSA for 2048 Bytes Data…….. 40
Figure 5.3 Comparison of CUDA-RSA and CPU-RSA for 8192 Bytes Data…….. 41
Figure 5.4 Comparison of CUDA-RSA and CPU-RSA for 32784 Bytes Data…… 41
Figure 5.5 Graph Showing the Comparison of CUDA-RSA and CPU-RSA for Small Prime Numbers…… 43
Figure 5.6 Comparison of OpenCL-RSA and CPU-RSA for 512 Bytes Data……. 43
Figure 5.7 Comparison of OpenCL-RSA and CPU-RSA for 2048 Bytes Data…… 44
Figure 5.8 Comparison of OpenCL-RSA and CPU-RSA for 8192 Bytes Data…… 44
Figure 5.9 Comparison of OpenCL-RSA and CPU-RSA for 32784 Bytes Data…. 45
Figure 5.10 Graph Showing the Comparison of OpenCL-RSA and CPU-RSA…. 46
Figure 5.11 Graph Showing the Speedup Comparison of CPU/CUDA-RSA and CPU/OpenCL-RSA …… 47
Figure 5.12 Comparison of CUDA-RSA and OpenCL-RSA for 512 Bytes Data… 47
Figure 5.13 Comparison of CUDA-RSA and OpenCL-RSA for 2048 Bytes Data.. 48
Figure 5.14 Comparison of CUDA-RSA and OpenCL-RSA for 8192 Bytes Data.. 48
Figure 5.15 Comparison of CUDA-RSA and OpenCL-RSA for 32784 Bytes Data……. 49
Figure 5.16 Comparison of CUDA-RSA and OpenCL-RSA for Large Prime Numbers. …….. 50

LIST OF TABLES

Table 2.1 Components of OpenCL Execution Model……..20
Table 2.2 Card Game analogy……..21
Table 5.1 Comparison of CUDA-RSA and CPU-RSA for Different Data Size……..42
Table 5.2 Comparison of OpenCL-RSA and CPU-RSA for Small Prime Numbers …...45
Table 5.3 Comparison of CUDA-RSA and OpenCL-RSA for Large Prime Numbers…49

CHAPTER 1

INTRODUCTION

In today's world, there is a great demand of parallel computing or programming so as to achieve high performance in terms of time required to process computationally expensive data units. To achieve parallelism on CPU (Central Processing Unit) is not possible but it can be achieved by SIMD (Single Instruction Multiple Data) on General Purpose Graphic Processing Unit (GPGPU) along with the integration of Central Processing Unit (CPU). Research on implementing security algorithms using GPGPU is a latest trend these days as security algorithms are computationally rich and idea to harness the performance in terms of speed, time using GPGPU is appropriate.

In this work RSA algorithm is implemented using the same techniques of GPGPU and using Compute Unified Device Architecture (CUDA) and OpenCL as a programming model to utilize the parallel architecture of Graphical Processing Unit (GPU). CUDA and OpenCL makes use of use multithreading technique.

RSA is a public key encryption scheme. The problem with this security algorithm lies with the factoring of large integers also known as factoring problem. RSA is an asymmetric key encryption scheme. It uses two different keys for encryption and decryption. The public key that is publicly known and is used for encryption . Private key can only decrypt the messages encrypted by public key.

GPUs are also considered as desktop supercomputers for its ability to provide high computation power at low lost. The three key features of GPU as: low cost, high computation power, high availability made it possible to use GPUs for many general purpose computation. One can harness the compute power close to 1 Tera Flops (TFLOPs) at a high cost. To achieve high parallelism that too at a low time there require a high understanding of handling the flow of serial execution using pipeline in the earlier, GPGPU model. CUDA and OpenCL are the two programming interfaces used to leverage massive parallelism. CUDA is specific to NVIDIA GPU's and OpenCL provides a portable language that can program CPU, GPU and other devices from different vendors too. So CUDA from NVIDIA and OpenCL both provide

heterogeneous programming model. This parallel hardware can be used along with CPU. In GPU programming model such as CUDA, compiler and runtime can extract the parallelism automatically.

Cryptographic operations are highly computationally expensive and hence consume a large amount of CPU cycles from the system. Such examples are web servers. To incorporate with this and to offload cryptographic operations, web servers are typically deployed in systems using cryptographic accelerator cards. Though this saves system CPU cycles from complex and computationally intensive application logic but it also results in difficult deployment scenarios. So, the possibility of offloading the massive computationally rich operations to GPU would lighten the CPU load to a great extent and hence the saving can be used for other applications. The idea also provides a significant performance increase. This technique is known as "**GPU ACCELERATED COMPUTING**"[29] . The idea will be clear from the Figure. 1.1

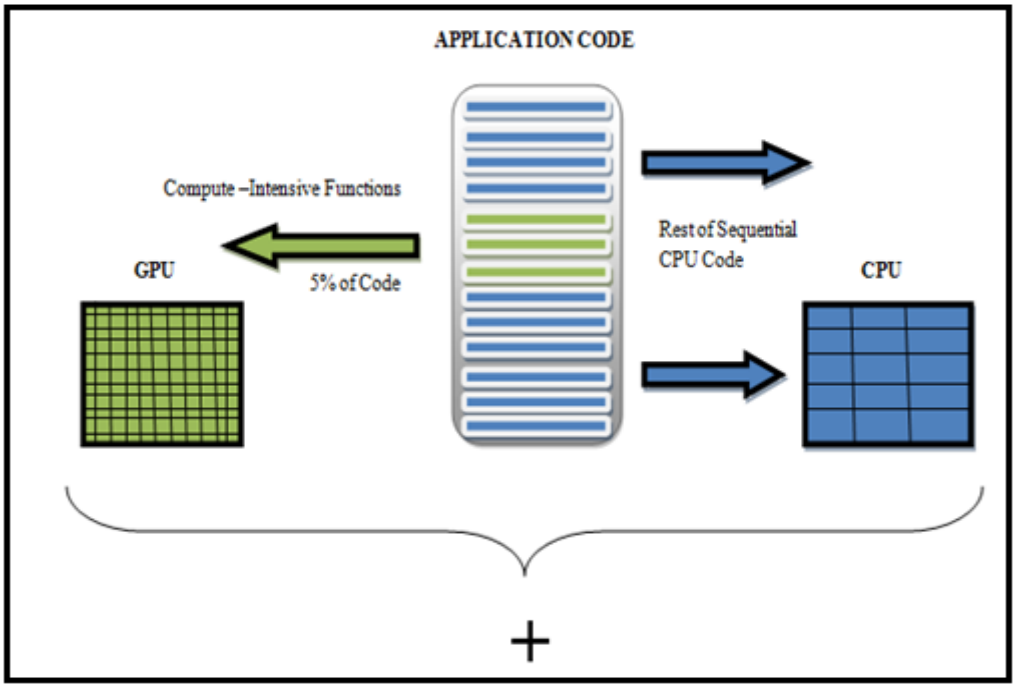

Figure 1.1 Working of GPU Acceleration Process.

CPU and GPU can be compared on the basis of how the tasks are processed. CPU has few optimized cores for sequential processing whereas GPU has a large number of smaller and more efficient cores that can handle multiple tasks efficiently and simultaneously. Figure 1.2 shows the basic difference between multicore machines and GPUs(Nvidia)[28]

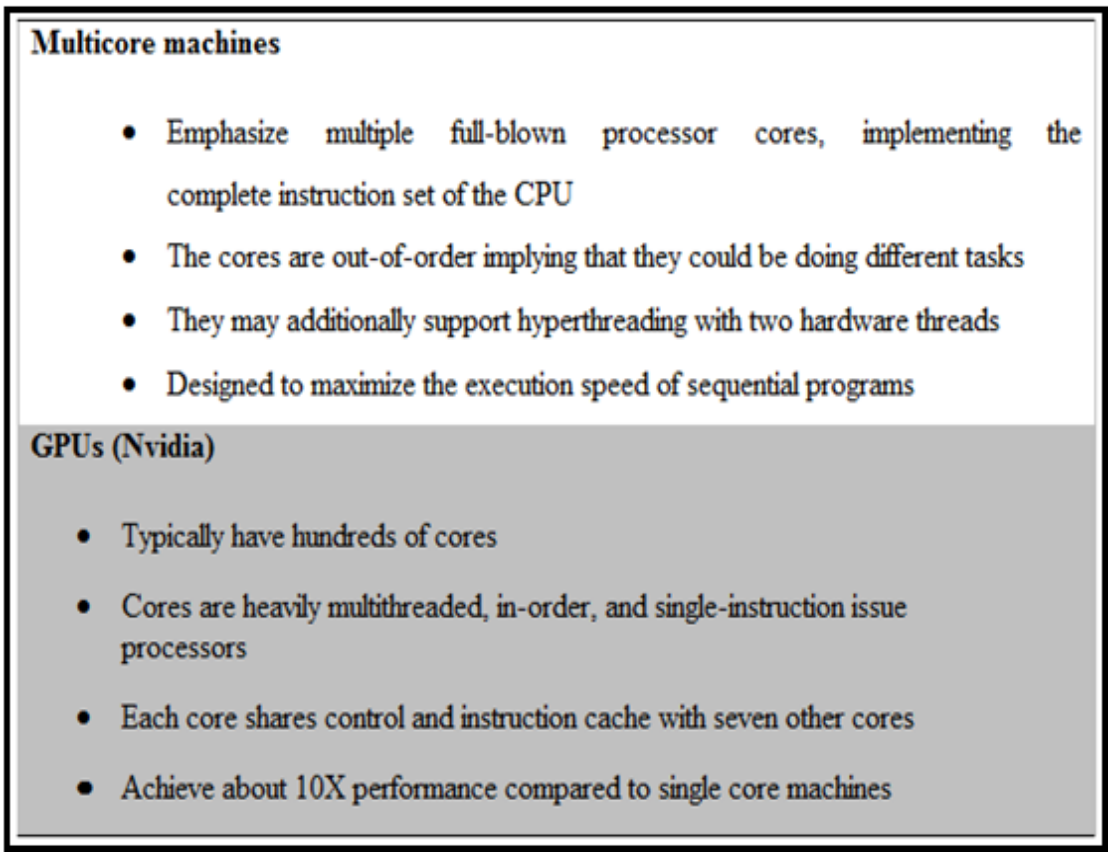

Figure 1.2 Comparison of Multicore Machines and GPUs(Nvidia)

Our objective of this work is to parallelize the RSA algorithm using OpenCL and CUDA and analysis of performance gain between the two platforms used. CPU implementation of RSA algorithm consumes a lot of CPU cycles as RSA algorithm is computationally rich. CPU process RSA encryption on plaintext one by one as cores is limited. But on the other hand, GPU has many cores and can parallelize the plaintext by dividing the plaintext among different cores and these cores process data at the same time. This results in massive parallel computation of the computationally rich part of RSA algorithm i.e., modulo exponential function that results in performance boost of the algorithm.

There always exists a trade off between security and efficiency while implementing the public key cryptosystem. RSA algorithm requires a lot of computation for high level of security. Complex computation provides a high level of security but fails in level of efficiency and hence makes it slower.

This work implement a more efficient, secure and faster RSA algorithm using parallel programming by parallelizing the more computationally expensive function of RSA algorithm i.e., modular multiplication and modular exponentiation.

In this work by using the unique properties of NVIDIA CUDA-C and OpenCL, RSA algorithm is designed for both platforms and the performance of both is compared with the CPU RSA one by one. Finally the project will compare the performance between the NVIDIA CUDA RSA and OpenCL RSA and performance gain is concluded.

The remaining sections of this report is organised as follows:

In Chapter 2, the review of the state of art of RSA and GPU programming is done. This chapter explain the traditional RSA algorithm and describes the computationally rich part of the algorithm i.e., modular exponential function and modular multiplication. This chapter also explain the history of GPU programming along with the various programming models (CUDA and OpenCL) used in the project.

In Chapter 3, formulation of Problem Statement and the gap analysis is done between the existing work and the desired work. Discussion is also done regarding the justification of the question and answer.

In Chapter 4, implementation of the problem formulated in Chapter 3 is done. This chapter includes design of RSA algorithm on three different platforms, sequential RSA that make use of CPU only, CUDA RSA that conjugate NVIDIA GPU with CPU and finally OpenCL RSA . This chapter mainly focuses on the kernel implementation of CUDA RSA and OpenCL RSA.

In Chapter 5, discussion is made on the implementation done in the Chapter 4 and also explains the results derived from it.

In Chapter 6, conclusion of the work done along with the summary of contribution and future work is given.

CHAPTER 2

LITERATURE SURVEY

2.1 RSA Algorithm

Encryption is the process of encoding the original text so that others cannot understand the original data until and unless the message is intended for him. Standard encryption methods basically have following flaws:

I. Establishment of the secure channel is required for the exchange of the key .Sender needs to exchange the decoding key with the receiver only then receiver will be able to decode the message; and
II. No guarantee of the sender i.e., one cannot guarantee about the sender of a given message.

RSA is a public key encryption scheme and is invented by (Ron Rivest, Adi Shamir, and Leonard Adleman). RSA is used for both signing and encryption and is a benchmark in the field of public-key cryptography. Many electronic commerce protocols are dependent on RSA for security. RSA with long key size and up-to-date is considered secure. RSA is an asymmetric key encryption scheme and uses two different keys for encryption and decryption [27]

2.1.1 Operation

The three steps of RSA algorithm – Key Generation, Encryption and Decryption[27] are explained as follows:

Key Generation

As explained above RSA is an asymmetric key encryption scheme. It uses two different keys, one for encryption and one for decryption. The public key encrypts the messages and is publicly known. Private key can only decrypt the encrypted messages. The whole RSA algorithm rely on the value of n,which acts as modulus during key generation. The value of n is calculated by choosing two random prime numbers . The keys for RSA algorithm are generated as follows:

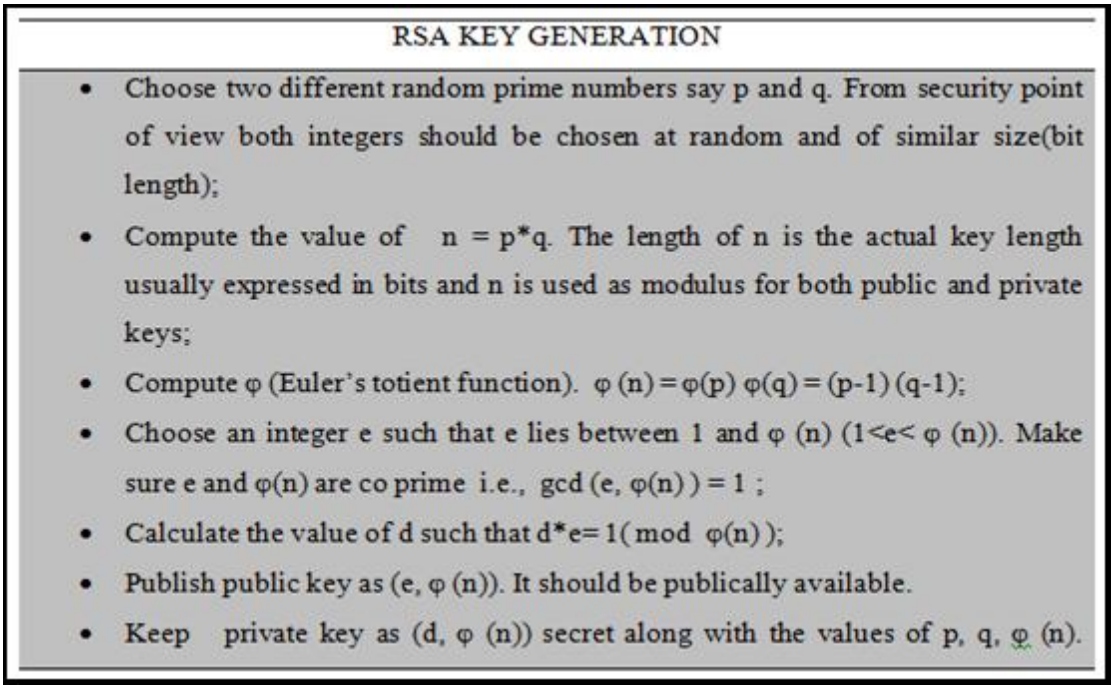

Figure 2.1 RSA Key Generation Process.

Encryption process

Alice transmits its public key (n,e) to the Bob and keeps the private key secret. Suppose Bob wishes to send message M to Alice. He first convert the message M to an integer m, such that m is between 0 and n using padding scheme. Finally cipher text c is calculated as: $C = m^e \bmod n$. This cipher text is then transmitted to Alice.

Decryption process

Alice can decode the cipher text using her private key component d as :

$m = C^d \bmod n$ .Given m, original message M can be recovered. The complete process of RSA key generation is shown in the below Figure 2.2.

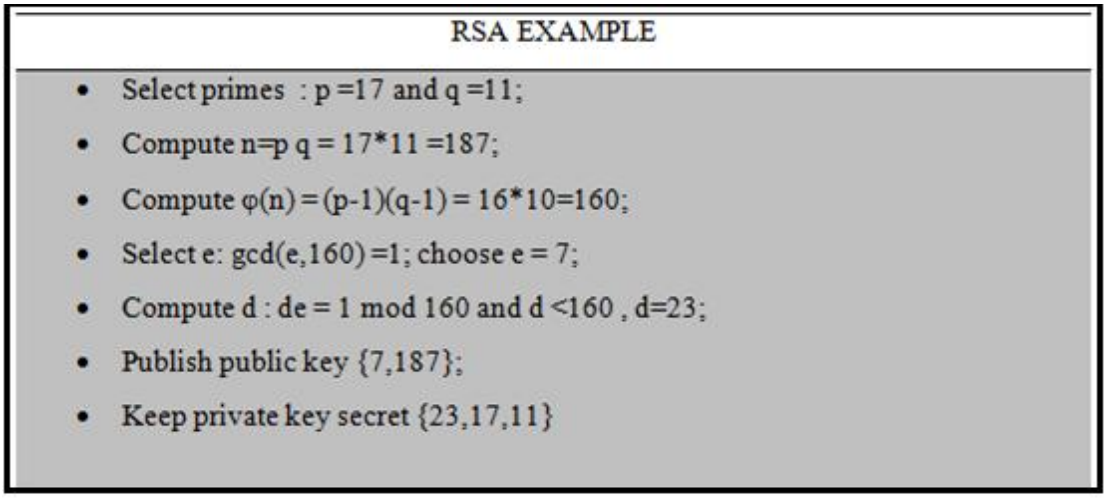

Figure 2.2 RSA Key Generation Example

2.1.2 Modular Exponential

Modular Arithmetic

Public key cryptography is computationally expensive as it mostly includes raising a large power to a base and then reducing the result using modulo function. This process is known as modular exponential[19]. In practical, RSA algorithm should be fast, i.e., modular exponential function should be fast and efficient. The simple modular operations are given in the Figure 2.3.

$$(u + v) \bmod m = ((u \bmod m) + (v \bmod m)) \bmod m$$
$$(u - v) \bmod m = ((u \bmod m) - (v \bmod m)) \bmod m$$
$$(u * v) \bmod m = ((u \bmod m) * (v \bmod m)) \bmod m$$

Figure 2.3 Figure Showing the Basic Modular Operations.

Naive Modular Exponentiation

In this method modular multiplications are applied repeatedly. For example: with g=4, e=13 and m= 497, the naive modular exponential will solve g^e mod m as shown in Figure 2.4 and gives the result as 445. This method is not efficient as it performs e-1 modular multiplications. In cryptography the security depends on the larger value of e and efficiency depends how efficiently these modular multiplications and modular exponential functions are solved. The plaintext, cipher-text or even partial cipher text is supposed to be of large in value and it will require a large amount of modular multiplications in naive algorithm[30].

1. e=1, c= 4 mod 497 = 4.
2. e=2, c=(4 _ 4) mod 497 = 16 mod 497 = 16.
3. e=3, c=(16 _ 4) mod 497 = 64 mod 497 = 64.
4. e=4, c=(64 _ 4) mod 497 = 256 mod 497 = 256.
5. e=5, c=(256 _ 4) mod 497 = 1024 mod 497 = 30.
6. e=6, c=(30 _ 4) mod 497 = 120 mod 497 = 120.
7. e=7, c=(120 _ 4) mod 497 = 480 mod 497 = 480.
8. e=8, c=(480 _ 4) mod 497 = 1920 mod 497 = 429.
9. e=9, c=(429 _ 4) mod 497 = 1716 mod 497 = 225.
10. e=10, c=(225 _ 4) mod 497 = 900 mod 497 = 403.
11. e=11, c=(403 _ 4) mod 497 = 1612 mod 497 = 121.
12. e=12, c=(121 _ 4) mod 497 = 484 mod 497 = 484.
13. e=13, c=(484 _ 4) mod 497 = 1936 mod 497 = 445.

Figure 2.4 Figure Explaining the Naive Modular Exponentiation Process.

Repeated Square-and-Multiply Methods

Algorithm: Right-to-left binary modular exponentiation

Input: an element g and integer $e \geq 1$, and a modulus m.
Output: $g^e \bmod m$.

1. A = 1; S = g; E = e.
2. While $E \neq 0$ do the following:
   2.1. If E is odd, then A = (A .S) mod m, E = E - 1.
   2.2. E = E/2.
   2.3. If $E \neq 0$, then S = (S . S) mod m.
3. Return (A).

Figure 2.5 Algorithm for Right-to-Left Binary Modular Exponentiation

It makes use of the fact that if the e value is even, then the modular exponential is calculated as $g^e \bmod m = (g^{e/2} * g^{e/2}) \bmod m$,hence by reducing the amount of modular multiplications to 2z where z is the number of bits when converting e to binary form. The algorithm [19] comes with two forms as shown below.

I. Right-to-left binary modular exponential
II. Left-to-right binary modular exponential

This algorithm works efficiently for large value of e. The algorithms for two different forms of repeated square-and-multiply methods are given in Figure 2.5 and Figure 2.6

Algorithm: Left-to-right binary modular exponentiation

Input: an element g and a positive integer $e = (e_t\ e_{t-1}\ \_\ \_\ \_\ e_1\ e_0)_2$,and a modulus m.
Output: $g^e \bmod m$.

1. A = 1.
2. For i from t down to 0 do the following:
   2.1. A = (A . A) mod m.
   2.2. If $e_i = 1$, then A = (A . g) mod m.
3. Return (A).

Figure 2.6 Algorithm for Left-to-Right Binary Modular Exponentiation.

Left –to-right k-ary exponentiation

This algorithm is the generalization of the previous left-to-right binary modular exponentiation described previously. In this algorithm[19] each iteration processes more than one bit of the exponent and works efficiently when pre-computations are done in advance and is used again and again.

Algorithm: Left-to-right k-ary modular exponentiation

Input: g and e $=(e_t\ e_{t-1}\_\ \_\ \_\ e_1\ e_0)_b$, where $b=2^k$ for some $k\geq 1$ and a modulus m.
Output: g^e mod m

1. Precomputation.
1.1. $g_0 = 1$.
1.2. For i from 1 to $(2^k - 1)$ do: $g_i = (g_{i-1} . g)$

2. A = 1.

3. For i from t down to 0 do the following:
3.1. $A = (A^{2^k})$ mod m
3.2. $A = (A . g^{e_i})$ mod m.

4. Return (A).

Figure 2.7 Left-to-Right k-aray Exponentiation.

Sliding Window Exponential

The main idea behind this algorithm[19] is the reduction of pre-computations as compared to the above discussed algorithm and hence ultimately reduction in the average number of multiplications done.

Algorithm: Sliding-window exponential

Input: g and e = $(e_t\ e_{t-1}.....e_1\ e_0)_2$, with $e_t = 1$, an integer $k \geq 1$, and a modulus m.

Output: g^e mod m.

1.Precomputation.

1.1. $g_1 = g$, $g_2 = g^2$

1.2. For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} = (g_{2i-1} . g_2)$ mod m.

2. A = 1, i = t

3. While $i \geq 0$ do the following:

3.1. If $e_i = 0$ then do: $A = A^2$ mod m , i = i-1.

3.2. Otherwise ($e_i \neq 0$) , find the longest bitstring $e_i\ e_{i-1}.....e_l$ such that $i-l+1 \leq k$ and $e_l = 1$, and do the following: $A = A^{2^{i-l+1}} . g_{(e_i\ e_{i-1}.....e_l)_2}$, i=l-1.

4. Return (A).

Figure 2.8 Algorithm for Sliding Window Exponential.

2.1.3 Advantages and Disadvantages of RSA algorithm

The RSA algorithm is known for its increased security and convenience. In RSA algorithm private keys are neither transmitted or nor revealed to anyone. By contrast in secret-key system keys are exchanged either manually or with the help of communication channel. This is because in secret-key system same key is used for encryption and decryption and there exist a chance that an enemy can retrieve this secret key during transmission

Public-key systems can provide digital signatures[25]. These digital signatures cannot be repudiated. By contrast in secret-key systems authentication requires either sharing of some important secrets or the involvement of third party. This can result in sender repudiating the previously authenticated message by claiming that the shared secret is compromised any one of the party involved

Speed is the main disadvantage of public-key cryptography. Because there is always a trade off between efficiency and security. For efficient RSA, key size should be small but small key size leads to many security holes. There exist many secret-key methods that are faster than the asymmetric encryption method. In reality public-key encryption is used to supplement the private-key encryption.

2.2 GPU

2.2.1 Idea Behind GPU Programming

There are three ways for digging hole faster[23]:

Dig faster – With little more efforts remove one shovelful of dirt per second rather than one shovelful of dirt per two seconds. This obviously will help but there exist a limit to this speeding up ultimately shovel will explode with time.

Buy a shovel that is more productive i.e., a shovel with four or even five blades instead of one. Obviously it is an interesting approach. But again, there exists diminishing returns in making shovel better. It is not likely one can use 20-30 bladed shovel.

Hire more diggers - This is the most productive approach of all, that will help to its best in the hole digging project.

As it is clear from the above discussion that efforts are given to build a faster processor in parallel. From first method of digging holes faster, it actually points how to run processor with faster clock speed so as to spend less time for each computation. However, in a modern processor increase in clock speed results in the increase of power consumption.

From second method of buying a more productive shovel it actually means increase of work on each step of clock cycle by increasing the number of processors. But single processor also has a limit on amount of work per second.

Third method of hiring more diggers, refer to parallel computing having emphasis on more diggers with many shovels instead of one fast digger with one fast and efficient shovel. Try should be to get work done using a large number of weaker processor rather than one with less amount of powerful processors.

2.2.2 Technology Trends Showcasing CPU Speed Remaining Flat

GPU is an interesting processor today. To understand the popularity of the GPU, it is must to understand the technology trends these days. The world has gone parallel. Transistors are the basic unit of modern processors and the size of these transistors is getting smaller and smaller each year.

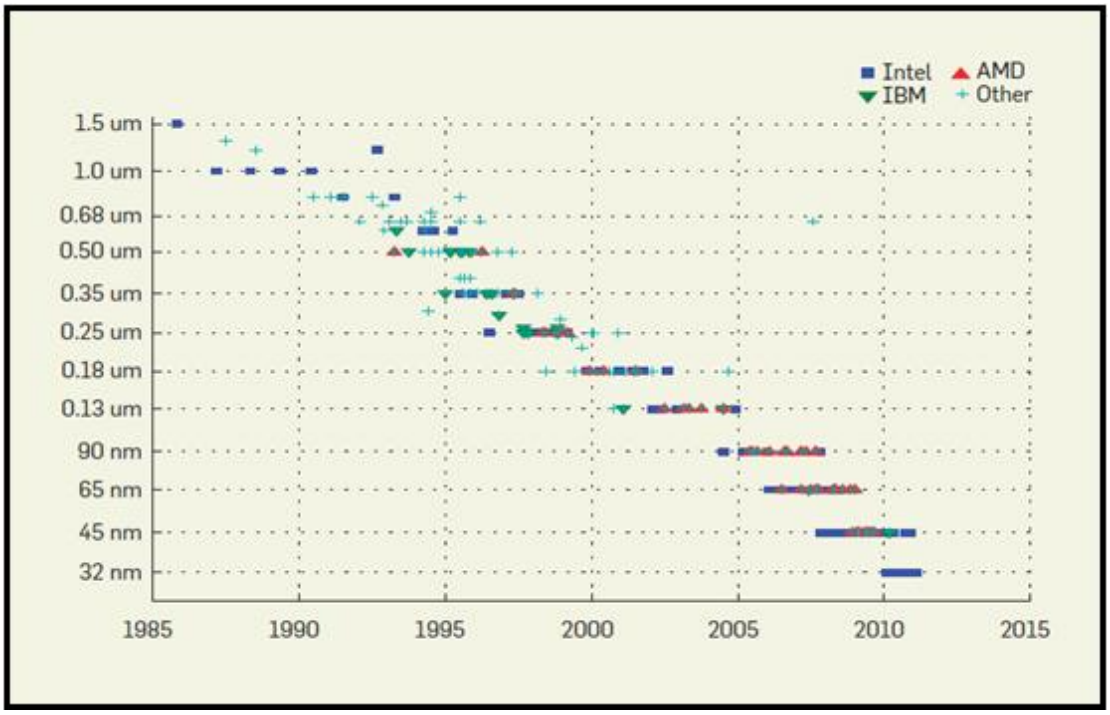

Figure 2.9 Figure Showing the Feature Size of Transistors with Time.[23]

This graph is from Stanford CPUDB project. It shows the feature size of processors over time. X axis shows time parameter[23]. And Y axis is the feature size, means how big transistors are. It is clear from the graph that the feature size is getting smaller and smaller every generation. The decrease in feature size results in smaller transistors that run faster and consume less power. As a result one can have more processors on a single chip. The consequence results in more and more computational resources every year. Historically, the improvement of transistors would result in increase of the processor clock rates every year.

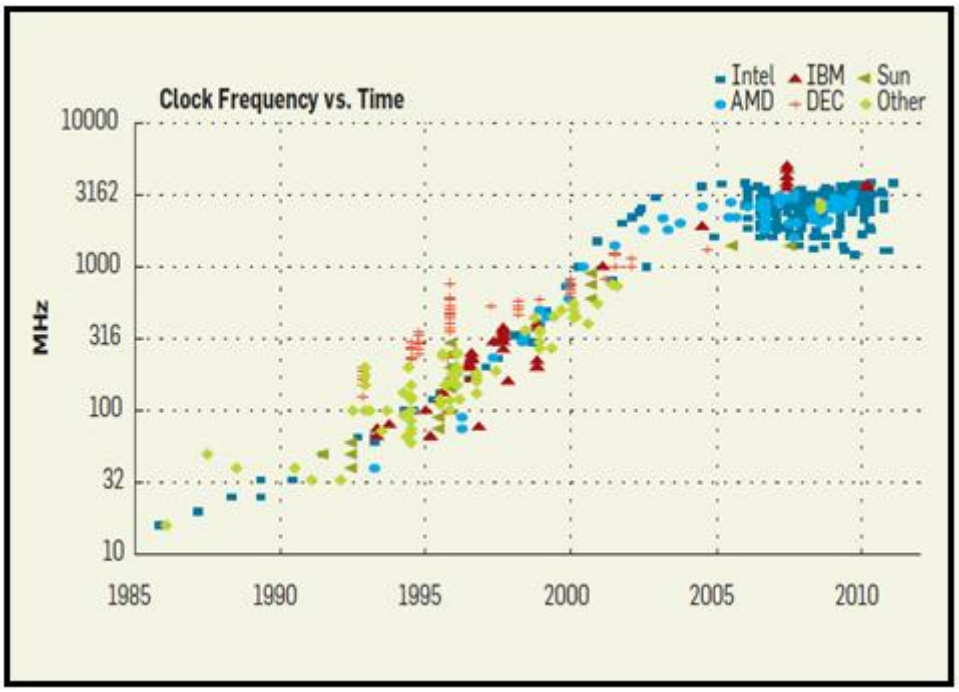

Figure 2.10 Figure Showing the CPU Speed Remaining Flat.[24]

This graph shows the clock speeds over years with time on X axis and clock frequency on Y axis. It is clear from the above graph that over many years, clock speed continues to go up. Over the last decade, the clock speeds have essentially remained constant. The reason behind the constant clock speed does not mean that the transistors have stopped getting faster with reduction in size. The process of reduction in size and hence the output is faster transistors and even consume less energy. The actual problem is heat that is the resultant of running this big amount of transistors. It is tough to keep all these processors cool [24].

Power is the driving factor behind the designing of processors at all scales whether it is about designing of mobile phone or a largest super computer. It is clear that idea of

making processor depending upon a single faster processor is not good and worthy. The resultant of this approach would be a processor with heating effect. Instead, processor designers now focus more on building smaller and more efficient processors considering the power factor rather than focusing simply on building faster less-efficient processors.

2.2.3 Emergence of GPU

Traditional CPU like processors have a very complicated control hardware, which no doubt add on flexibility but is also costly in terms of power and design complexity. This is the reason behind the inefficiency of traditional CPUs[22]. Solution for this problem is to build simple control structures which support more computations.

In GPU, data path is the outcome of large number of compute units in parallel fashion. These compute units alone are simple and small with more power efficiency. So, it is easy for a single chip to accommodate a large number of it. The complexity is in programming them together in an efficient way to solve massive computations.

Traditional CPU's focuses on optimizing the latency with an effort to reduce the execution time of per task. GPU's rather than optimizing latency focuses on optimizing the throughput. This is no doubt a different approach but result is a revolution in the computer industry.

GPUs are rich in compute units and these compute units work together to perform large computation. GPUs have simpler control complexity with a large amount of compute power and resultant of it is GPU programming model.

GPU’s explicit parallel programming model already gives a sense to programmer that GPU has lots of processors and effort is only to efficiently program all these processors together to get the best output. There is no restriction like in case of CPU, programming using only one processor and dependency on magic compiler for mapping of work onto many different processors.

GPUs are built to focus on optimizing the throughput rather than latency [22]. The latency of single individual computation is accepted in exchange for large amount of other computations that are processed in parallel per second. GPUs are best suited for the environment where throughput is the most important metric.

From the point of view of a software developer, 8 core Intel Ivy Bridge processor, has 8 cores, each core has 8-wide AVX vector operations, each core supports two simultaneously running threads, on multiply those together and it will give 128 way parallelism. A simple C program that will run sequentially will not be able to exploit the GPU hardware and less make use of even less than 1% of it. No doubt serial programming is far easy than the parallel programming , no matter what the target is but this additional complexity is worthwhile for the potential performance gains.

2.3 CUDA

According to the market demand, the GPU today are highly parallel supporting multithreading with a large number of cores [21]. The GPUs are known for its tremendous computational power and large memory bandwidth. This is shown in Figure 2.11. The reason behind this is that GPU are specially designed for intense and highly parallel computation.

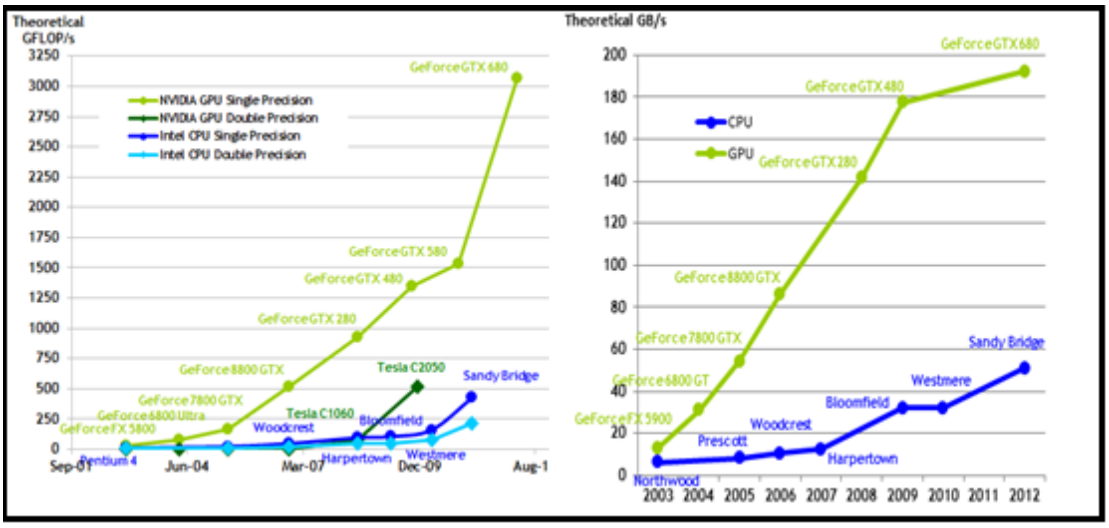

Figure 2.11 Figure Showing the Computational Power and Memory Bandwidth with Time.[21]

CUDA stands for Compute Unified Device Architecture and is used to program GPU, CPU and other NVIDIA devices. It was released in 2007. CUDA has a software environment that support various languages and application program interfaces. In CUDA, CUDA-enabled GPU are known as devices and CPU is known as host , where devices acts as co-processor.GPU has its separate memory and processor. To exploit the CUDA for parallelism, data needs to be transferred between the host computer memory and CUDA devices memory space. So, input-output time should be considered for calculating performance results. With CUDA programmers can run

thousands of threads parallel and keep them busy at the same time. As a result applications can deliver a significant performance increase.

2.3.1 Programming Interface

A kernel[19] is a function that is called from the host. It is executed on the CUDA enabled devices by a large number of threads simultaneously. In CUDA user configure the number of threads that run parallel on different set of data. This model is known as Single Program Multiple Data(SPMD). Same kernel function will be executed on each thread but data element is single and different. Thread Index and Block Index distinguish each thread and block respectively from other threads and blocks in a grid. Thread Index and Block Index are used to determine the data element the thread will access. Parallel computation in CUDA is organized with the abstraction of threads, blocks and grids.

Threads execute the kernel. Each thread has a unique thread index and is used to access data elements. Different threads run together and can process the entire data set. Multiple MPs make SPs (stream processors) and each SP can handle threads within a block[18].

Group of threads make a **block**. Threads in block can run both parallel or serial and can be synchronized using synchronization function. Synchronization is the process in which one threads stops at a certain point to wait for others threads to reach there and process further. Collection of multiprocessor (MPs) makes a GPU chip. Each MP can handle blocks only within a grid. A block is confined to a single MP. Group of blocks make a **grid**. Blocks cannot be synchronized. A single GPU chip can handle an entire

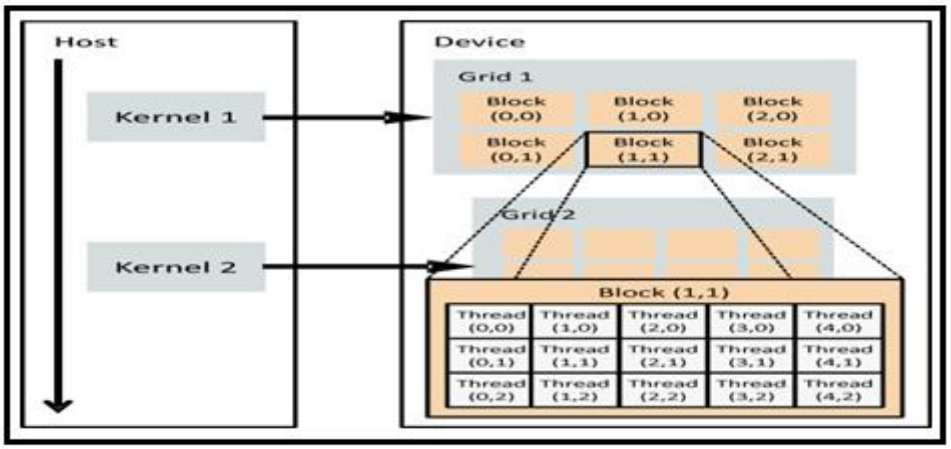

Figure 2.12 CUDA System Model.

Kernel makes asynchronous function calls. Synchronization can be both implicit and explicit depending upon the function requirement. Synchronization function is used for explicit synchronization and implicitly it is done whenever host access or request for device memory. Synchronization acts as a barrier in both cases and it blocks the host thread. It keeps blocking the calling host thread till the previous kernel finish their work[26].

GPU acts as a co-processor to CPU. Both CPU and GPU work simultaneously. After the initialization of the host and the CUDA device array, if there exists kernel function call, the kernel will start running as the CUDA device is found idle. Kernel execution depends on the function arguments and execution configuration. At the same time the host itself is not idle, it proceed to the next line of code after the kernel launch function. Hence, both CPU and CUDA device are working simultaneously and running their own separate programs. If there exist another kernel function call by the host, it will wait until the execution of the previous launched kernel function on the device.

CUDA executes SIMD(Single Instruction Multiple Data [16]) instructions. We write this kernel or device code for addition of two numbers on multiple data sets and Figure 2.13 shows the kernel code and actual processing of the kernel code on device.

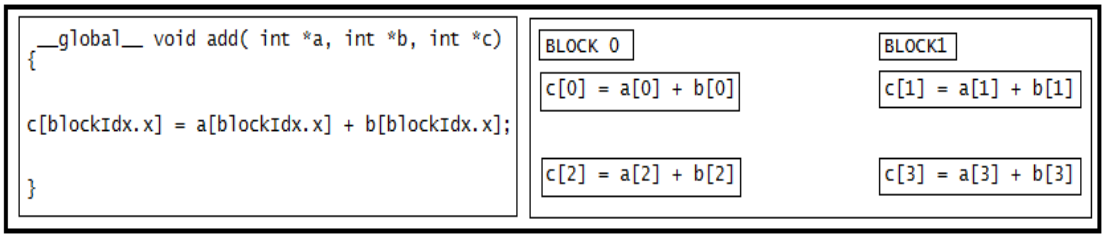

Figure 2.13 Kernel Code and Execution Process of SIMD Addition Instruction in CUDA.

Threads within a block run in SIMD fashion and group of threads within a block is known as warps. Wrap size is the number of threads within a wrap and are executed in SIMD fashion. Same instruction with different data set is broadcasted to each thread within a warp. There exists time division within active warps. To maximize the use of the computational resources thread scheduler perform switching periodically between warps. The order of execution between warps and blocks is not fixed. GPU process a large number of threads simultaneously and suppress the memory access costs by using thread scheduling.

2.3.2 Memory Hierarchy

The memory hierarchy [15] of CUDA-enabled GPU is as follows:

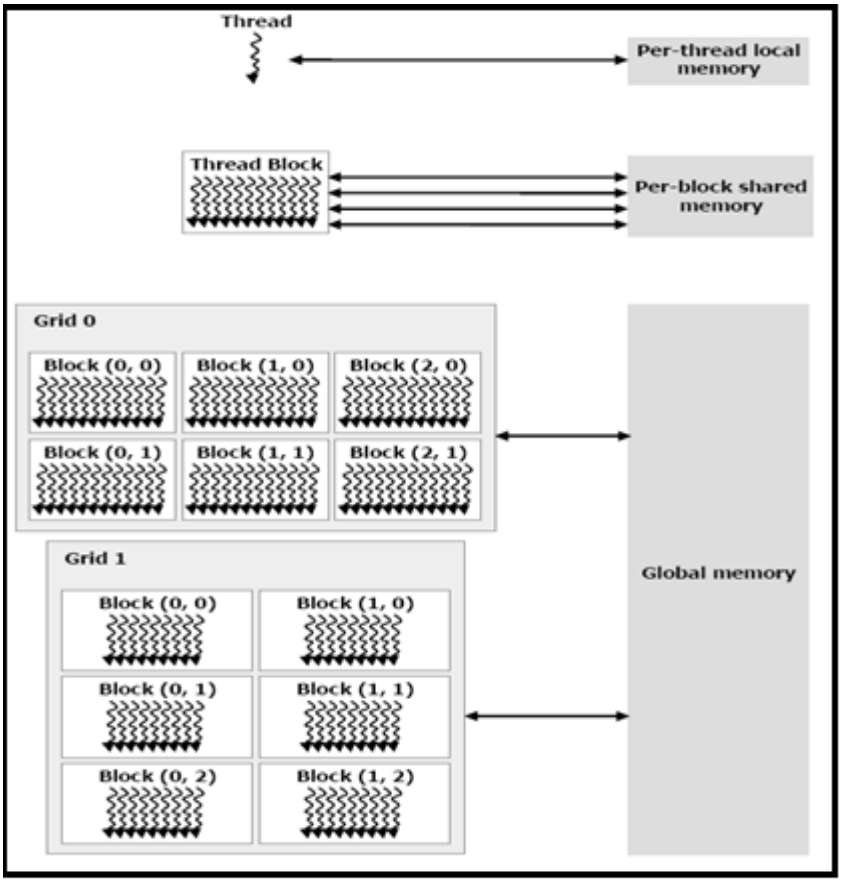

Figure 2.14 Memory Hierarchy of CUDA.

a) **Global memory**: This is known as read and write memory or device memory. CUDA enabled GPU with compute capability of 2.x are slow but now are cached. These are typically implemented in DRAM and has a high access latency of 400-800 cycles. This memory has a finite access bandwidth and has a potential of traffic congestion. It gives throughput of 177 GB/s.

b) **Texture cache:** It is the cache memory and is available within each MP. This is the read-only memory and can accommodate global memory data[20].

c) **Constant cache**: This is also a read-only memory and is available within each MP. In this memory, constants and kernel arguments are stored. It is a slow memory with a cache of (8 kb).

d) **Shared memory:** This is an extremely fast, highly parallel memory with (16 kb per MP) and is restricted to a block. It exchanges data between threads in a block. There exist more chances of bank conflicts.

e) **Registers:** It is the fastest memory and is only accessible by the local threads.

f) **Local memory:** Local memory does not exist physically. It is an abstraction to the local scope of a thread. It is actually put in the global memory by the compiler. It is the slow memory but now available with cache.

2.3.3 Program in CUDA

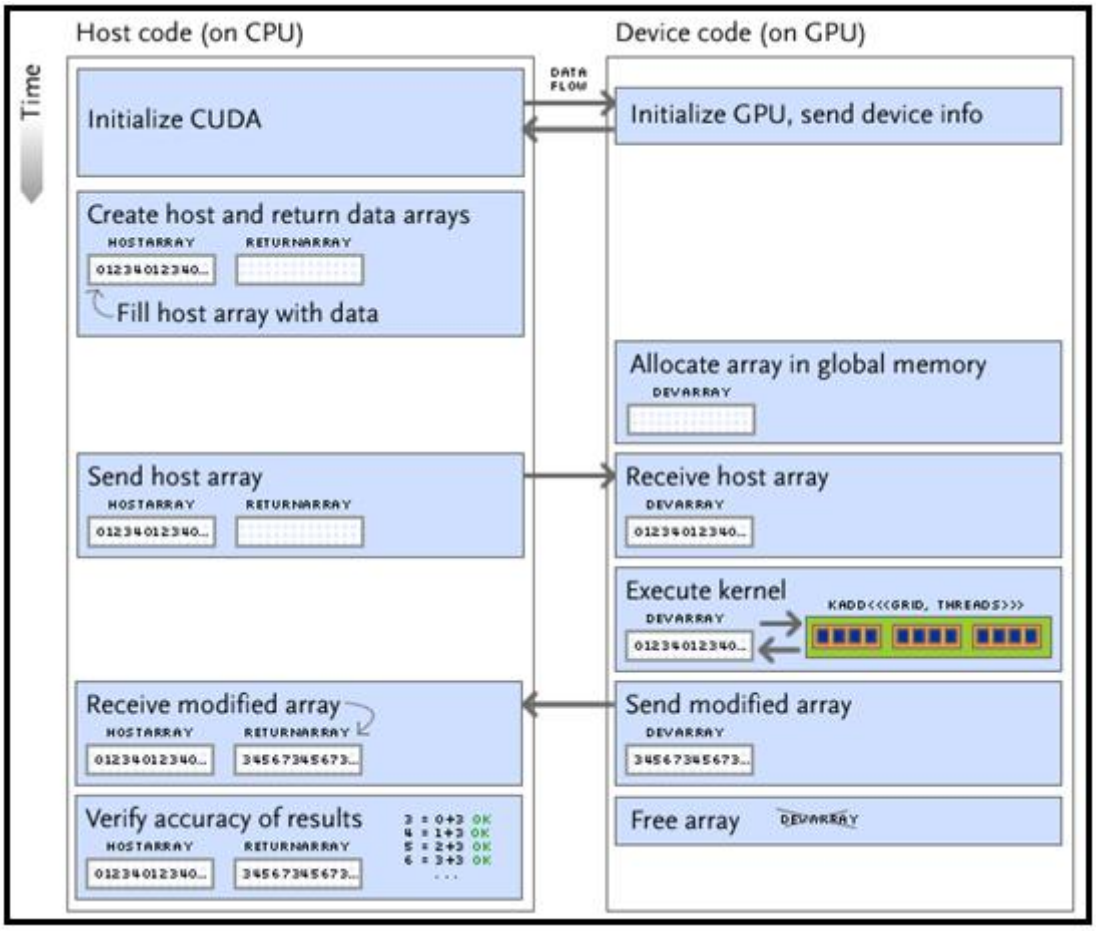

Figure 2.15 Flow of CUDA Program

To write a program in CUDA, kernel function is identified first .This Kernel function is called from the host. After that initialization of CUDA on both CPU and GPU is done and relevant device information is send to the host. Following that, host arrays and return data arrays are created and simultaneously host arrays are filled with data. After the creation of arrays, next step is to allocate memory for the arrays and send the host arrays to the device[14].Finally kernel is executed and the modified array or results are send back to the host for verification of the results and further processing . After the kernel execution is completed last step is to free the array. Figure 2.15 gives a detail overview of the CUDA program from kernel creation to kernel execution.

2.4 OpenCL System Model

OpenCL provides a framework for heterogeneous programming [12], i.e., one can program CPUs, GPUs and other computing devices simultaneously. It supports software development by providing a runtime system along with API and libraries. The important OpenCL models are explained below:

Platform Model

This model consists of a host device and is connected to one or more OpenCL compliant devices. An OpenCL device has many compute units and these compute units are divided into many processing elements. Processing elements perform computations on a device[13]. Host application is executed by the host device . OpenCL architecture follows SIMD and are executed by the compute unit's processing elements[17].

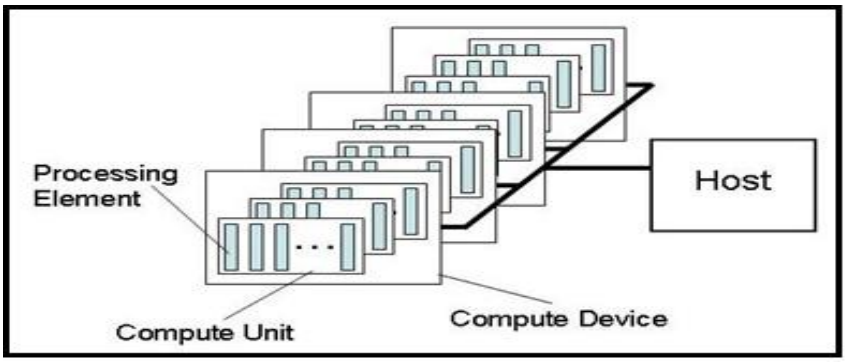

Figure 2.16 OpenCL Platform Model.

Execution Model

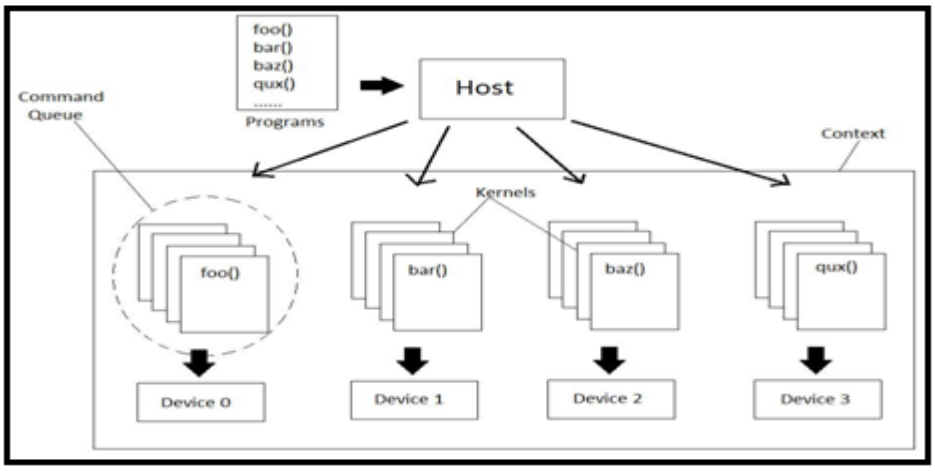

Figure 2.17 OpenCL Execution Model.

Execution model [12]explains how an application runs on the OpenCL. In OpenCL application runs on a Host and Host further submit the work to the OpenCL complaint devices. Table 2.1 and Figure 2.17. explain the CUDA execution model .

Table 2.1 Components of OpenCL Execution Model.

Work-Item	**It is the basic unit of work on an OpenCL devices**
Kernel	It is the code for a work-item(basically a C function)
Program	Collection of kernels and other functions (analogous to a dynamic library)
Context	The environment in which the work-item executes; includes devices, and their memories and command queues.
Command Queue	Queue used by the host application to submit work to a device(e.g., kernel execution instances) • Work is queued in-order, one queue per device • Work can be executed in-order or out-of order.

Memory Model

Memory management in OpenCL is done explicitly. One must move the data from host memory to global memory[12] , global memory to local memory and then finally back to the host memory along with the results.[13] OpenCL has following four types of memory:

a) Private memory: Each work-item has its own private memory.
b) Local memory: This memory type is shared within a group.
c) Global/Constant memory: This memory type is visible to all work-groups.
d) Host memory: It is the CPU memory.

Analogy : OpenCL processing using game of cards.

The OpenCL is more like a poker game[12]. In a poker game dealer sitting on a table along with players and distribute a set of cards to each player. On analysing their cards players further decide their actions without interaction with others. Players also request for additional cards. Dealer process the request and takes the control on completion of the game.

Dealer in the analogy is the OpenCL host with each player as device, card table as context and card as kernel. Command queue is the each player's hand. Table 2.2 explains the whole analogy.

Table 2.2 Card Game analogy.

Card game	OpenCL application
The dealer sits at a card table and determines who the players are.	The host selects devices and places them in a context.
The dealer selects cards from a deck and deals them to each player. Each player's cards form a hand.	The host selects kernels from a program. It adds kernels to each device's command queue.
Each player looks at their hand and decides what actions to take.	Each device processes the kernels that are sent through the command queue.
The dealer responds to players' requests during the game.	The host receives events from the devices and invokes event-handling routines.
The game ends, and the dealer looks at each player's hand to determine who won.	Once the devices are finished, the host receives and processes the output data.

The above card analogy is best understood from the below card game of four players where each hand receives four cards. Compare the Figure 2.18 with the previous Figure 2.17, for the clarity of the analogy.

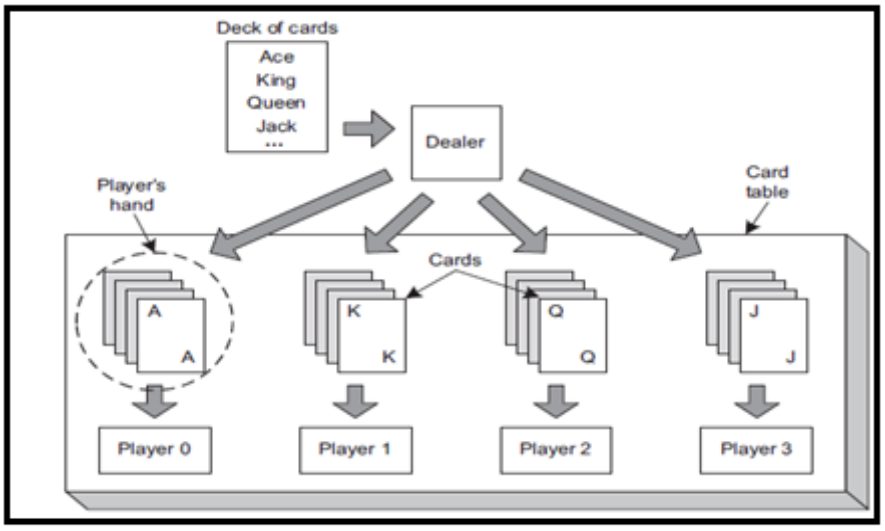

Figure 2.18 Figure Explaining the Card Game Analogy.

2.5 Porting CUDA Application to OpenCL Program

One can easily convert the CUDA-C application to OpenCL program. One requirement is to find equivalent syntax for various built-in functions and keywords in

kernel. Secondly, runtime API calls should be converted to equivalent API calls in OpenCL[11].

2.5.1Terminology

CUDA-C Terminology	OpenCL terminology
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Figure 2.19 General Terminology

The above Figure 2.19 shows the general terminology for various computations and memory spaces in CUDA-C and OpenCL. The terms are quite similar[11].

2.5.2 Writing Kernels: Qualifiers

Qualifiers for kernel functions are annotated with __. The difference between the two is __global__functions are called from host not from device in CUDA-C and __device __ functions are not callable from host but in OpenCL entry points functions are __kernel functions and are callable from device including CPU device and annotation is required for non-entry point functions[11].

CUDA-C Terminology	OpenCL terminology
__global__function(callable from host,not callable from device)	__kernel function (callable from device,including CPU device)
__device__ function(not callable from host)	No annotation necessary
__constant__variable declaration	__constant variable declaration
__device __variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

Figure 2.20 Porting of Kernel: Qualifiers.

2.5.3 Writing Kernels : Indexing

The various indexing functions required for CUDA and OpenCL are given in the below Figure 2.21. CUDA provides a special terminology for such purpose but in OpenCL these are carried by function calls[11].

CUDA-C terminology	OpenCL terminology
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
No direct equivalent. Combine blockDim,blockIdx and threadIdx to calculate a global index	get_global_id()
No direct equivalent. Combine gridDim and blockDim to calculate the global size.	get_global_size()

Figure 2.21 Porting of Kernel: Indexing.

2.5.4 Writing Kernels: Synchronization

The various synchronization functions are given in Figure 2.22.__synthreads() and barrier() perform the synchronization function in CUDA-C and OpenCL respectively. Synchronization provides a mechanism to synchronize all the work-items in a work-group[11].

CUDA-C terminology	OpenCL terminology
__syncthreads()	barrier()
__threadfence()	No direct equivalent
__threadfence_block()	mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE)
No direct equivalent	read_mem_fence()
No direct equivalent	write_mem_fence()
CUDA-C terminology	OpenCL terminology
CUdevice	cl_device_id
CUcontext	cl_context
CUmodule	cl_program
CUfunction	cl_kernel
CUdeviceptr	cl_mem
No direct equivalent	cl_command_queue

Figure 2.22 Synchronization of the Kernels.

2.5.5 Important API calls

CUDA-C terminology	OpenCL terminology
cuInit()	No OpenCL initialization required
cuDeviceGet()	clGetContextInfo()
cuCtxCreate()	clCreateContextFromType()
No direct quivalent	clCreateCommandQueue()
cuModuleLoad()	clCreateProgramWithSource() or clCreateProgramWithBinary()
No direct equivalent. CUDA programs are compiled off-line	clBuildProgram()
cuModuleGetFunction()	clCreateKernel()
cuMemAlloc()	clCreateBuffer()
cuMemcpyHtoD()	clEnqueueWriteBuffer()
cuMemcpyDtoH()	clEnqueueReadBuffer()
cuFuncSetBlockShape()	No direct equivalent[functionality in clEnqueueNDRangeKernel()]
cuParamSeti()	clSetKernelArg()
cuParamSetSize()	No direct equivalent[functionality in clSetKernelArg()]
cuLaunchGrid()	clEnqueueNDRangeKernel()
cuMemFree()	clReleaseMemObj()

Figure 2.23 Important API Calls.

2.6 CUDA versus OpenCL

Vendors: CUDA implementations have only one vendor , namely NVIDIA Corporation but there are many vendors having OpenCL implementations. The list includes AMD, Nvidia,Apple, Intel, IBM , Altera and Portable OpenCL.

Code Portability: Efforts are required to re-write OpenCL kernel code while switching from one device to another though the host code remains the same[9].

Capabilities:

a) Since OpenCL does not support pinned host memory which causes the penalty while host-device data transfer.
b) CUDA lacks in synchronization flexibility when compared to OpenCL.
c) CUDA is rich in mature tools like CUBLAS and CUFTT
d) CUDA provides C API whereas OpenCL provides C++ bindings
e) OpenCL follows C99 and does not allow C++ constructs in GPU code.
f) Command Queue in CUDA cannot en-queue CPU pointer functions.

Accessibilities and convenience: CUDA is more accessible and a convenient approach as compared to OpenCL. Integration of host code with kernel code is easy and smooth in CUDA whereas in OpenCL productivity is less during the start stage.

2.7 Existing Work

Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA [8]

In Feburary 2008, Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk and Wen-mei W. Hwu discusses the features and generalization techniques of GeForce 8800 GTX processors . As GPUs have evolved a lot as a coprocessor to CPU and has gained a lot of attention because of its easier programmability and handling compute intensive part of the program thus offloading the CPU. This in return has led to a great research in GPGPU of mapping the applications to GPU hardware. This work utilize the multithreading to best possible so as to overcome the expense of global memory latency. This requires balance of number of blocks and resources utilized along with the number of threads that are supposed to run in parallel. Using the various optimizations techniques on memory such utilized a speed up of 10.5X to 457X is obtained in kernel codes of various applications so analysed and between 1.16X to 431X speedup for complete application so analysed.

Parallelization of RSA Algorithm Based on Compute Unified Device Architecture [7]

In Nov 2010, Wenjun Fan , Xudong Chen, Xuefeng Li,implemented the parallel RSA . Computer security domain's research topic is mainly to find ways to enhance the speed of RSA algorithm. With the popularity of GPU as to handle and accelerate massive computations gives an idea to parallelize using GPU as a co-processor of the CPU. NVIDIA's CUDA can solve the purpose. This paper presents parallelized implementation of RSA algorithm using JCUDA and Hadoop. Results demonstrate the enhanced speed of RSA algorithm.

A performance comparison of CUDA and OpenCL[6]

In May 2011, Kamran Karimi, Neil G. Dickson and Firas Hamze implemented a project on performance comparison of the two GPUs platform. CUDA and OpenCL

are the two frameworks used for parallel or GPU programming where CUDA is specific to NVIDIA GPU's and OpenCL provides a portable language that can program CPU, GPU and other devices from different vendors too. OpenCL entail performance lack. This paper uses identical kernels from a Quantum Monte Carlo application as to compare the CUDA and OpenCL in terms of various factors such as performance gain, latency, speed etc. Porting of CUDA kernel to OpenCL kernel using NVIDIA compiler requires only minimal modifications. In contrast, compilation requires more modification using ATI's build tools. Data transfer time to and from GPU is measured and compared along with kernel execution time for the both platforms. Application execution time is also measured and compared.

A Comprehensive Performance Comparison of CUDA and OpenCL[5]

In September 2011, Jianbin Fang, Varbanescu,A.L, Sips,H. implemented a project in which comprehensive performance analysis is done between the two programming platforms using 16 benchmarks some from synthetic world and some from real world. Various parameters are used for the analysis of performance gaps such as compilers so used by the underlying hardware, programming model, architectural details and also various optimization strategies. In this paper results shows that the CUDA performance for most of the applications gives the performance boost of up to 30% as compared to OpenCL. But the reason behind this is unfair comparisons. On Fair comparisons, same can be expected from OpenCL. So, definition of fair comparisons is given along with the guidelines for analysis. Portability of OpenCL is justified by running the various benchmarks so selected previously on the others platforms. This paper concludes that the OpenCL can be a better choice or good alternative to CUDA on the basis of portability. As portability do not affect the performance of the benchmark applications.

On the performance of GPU public-key cryptography [4]

In September 2011, Neves,S. Araujo,F analysed the performance of public-key cryptography using GPU . Popularity of GPU's has increased over the last few years because of its capability to handle and accelerate large massive computations. In this paper modular exponential function is studied that make the base of modern public-key cryptography algorithms. The project is carried on NVIDIA GT200 architecture and fastest and best modular exponential function is used for GPU. Part of the

performance results are from Montgomery multiplications and are from general techniques. Throughput results with over 20000 RSA-1024 decryptions per second or 41426 512-bit modular exponentiations per second**,** presents a significant speedup in contrast to previous GPU implementations. There is no significant latency penalty. This paper shows that the GPU performs worse on performance/watt ratio.

PARIS: A Parallel RSA Prime inspection tool [3]

In June 2013, Joseph R. White implemented a tool that presents a method for validating RSA keys for poor prime numbers. Use of poor prime numbers or small prime numbers is a fundamental problem of the RSA algorithm that leads to many security holes. As there always exists a trade-off between speed of RSA algorithm and key-length. For efficient RSA key length should be small and that leads to security holes despite of mathematical soundness of RSA algorithm. PARIS uses NVIDIA's CUDA framework and GTX 480 and Tesla K20Xm for speedup comparison. It can perform greatest common divisor calculations between all keys. It offers a 27.5 and 33.9 times speedup using GTX 480 and Tesla K20Xm respectively. That is why this validation tool is used practically these days.

Efficient acceleration about encryption and decryption of RSA algorithm on GPU [2]

In August 2013, YAN Chenghua and ZHNG Zhiming studied the encryption and decryption of RSA algorithm using GPU programming by parallelising the different compute intensive functions such as modular exponential and modular multiplications along with other parallel levels. Along with GPU implemented RSA, serial RSA algorithm is such developed and compared. Results such obtained shows a acceleration of four times and also that the algorithm works best for small amount of data . For large data value serial algorithm is best suitable.

Parallelising RSA algorithm on multicore CPU and GPU[1]

In February 2014**,** Heba Mohammed Fadhil and Mohammed Issam Younis works on the parallelizing public key algorithms because they are slow as compared to symmetric key algorithms. By using the GPU as a co-processor to CPU and using the SIMT(Single Instruction Multiple Thread) concept they parallelize the RSA

algorithm. This paper proposes an algorithm for multicore CPU and Multicore GPU for varying key size and implements three variants of the RSA algorithm so as to compare it with the Crypto++ library and traditional sequential RSA algorithm. The GPU RSA obtains an acceleration of 23X as compared to the traditional sequential counterpart where the multithread CPU RSA implementation has a speedup of 6X as compared to the sequential CPU RSA implementation with latency as a factor.The gain in throughput for 1024 bits is 1800 message per second , for 2048 bits is 250 message per second. The results so obtained clarifies that GPU is appropriate of RSA speedup.

2.8 Chapter Summary

RSA is the most popular public key encryption-decryption algorithm. The problem with RSA algorithm is its slow speed as compared to its counterpart symmetric key cryptographic algorithms. The modular exponential and modular multiplication part of RSA algorithm makes it slow. So there exists a trade-off between security and efficiency in RSA algorithm. This trade-off can be overcome by GPU programming i.e., by parallelizing the compute intensive part of the RSA algorithm. CUDA and OpenCL are the two frameworks for GPU programming. By GPU programming the modular exponential and modular multiplication can be handled by thousands and thousands of threads that will process the data simultaneously. Thus by using GPU as a co-processor to CPU , CPU can be offloaded from computationally expensive part of RSA algorithm and hence can be made available for other simple calculations.

CHAPTER 3

PROBLEM STATEMENT

As per the literature survey, public-key cryptography algorithms are computationally expensive .Modular exponential and modular multiplications requires massive computations. RSA, one of the known public key algorithm despite of the mathematically soundness is not efficient and secure. RSA algorithm has a trade-off between security and efficiency. To have secure RSA, one require large key size and large key size means more computations, making RSA slow. If efficient RSA with small key size is used, it will make RSA algorithm vulnerable to many security attacks.

Theoretically, by making GPU as a co-processor to CPU and offloading the CPU from massive computations, this problem can be solved .The computationally expensive part of the RSA algorithm i.e., modular exponential and modular multiplication is make to run on the GPU. This GPU implementation should work efficiently as compared to traditional sequential RSA algorithm. This work analyse RSA algorithm on two different GPU frameworks, OpenCL and CUDA. CUDA is NVIDIA proprietary framework and OpenCL is an open standard for GPU programming. This work develops a secure and efficient RSA algorithm.

This work is implemented using NVIDIA GT-630M having 96 processing cores with graphics clock speed of 0.8GHz and the CPU clock speed of 2.4 GHz. Theoretically GPU-RSA can run 32 (96*0.8/2.4) times faster than the CPU implementation .

3.1 Objectives

1. To implement a secure and efficient RSA algorithm using CUDA and OpenCL.
2. To make GPU as a co-processor to CPU hence offloading the compute intensive functions of RSA algorithm.
3. To analyse the performance gain between the traditional sequential RSA algorithm and GPU version of RSA algorithm for small and large prime numbers.
4. To design and develop faster and efficient implementation of RSA algorithm among the sequential CPU, CUDA and OpenCL.

CHAPTER 4

IMPLEMENTATION

This work runs RSA algorithm on GPU and analyse the respective performance gain as compared to traditional sequential algorithm and uses two different platforms for parallelizing RSA, CUDA-C and OpenCL. The objective is to find out the faster and efficient implementation of the RSA among the three.

The most computationally expensive part of the RSA algorithm is

I. Modular multiplication
II. Modular exponentiation

Therefore this work parallelize the computationally expensive part of the RSA algorithm, so as to develop efficient and faster RSA algorithm by making GPU as a co-processor to CPU. As a result a large number of threads can be executed simultaneously.

Theoretically, the RSA algorithm implemented on the GPU using CUDA-C and OpenCL should work better than the traditional sequential RSA algorithm because the former case makes use of a large number of parallel processors to execute the RSA.

This chapter explain the implementation of the kernel and host implementation of the RSA algorithm, showing how C integrate along with CUDA driver API and OpenCL API for various different task such as

I. Kernel loading and settings before launch
II. Initialization of implicit code
III. CUDA and OpenCL context management
IV. CUDA and OpenCL module management
V. Kernel configuration along with the parameter passing

4.1 Algorithm for CUDA Kernel

In Algorithm 1, arguments are passed as pointer to global rsa(). After that the thread index are assigned following the declaration of the variables. Each thread in a block is

assigned a unique id and is known as thread index. Threads follow the SIMT fashion in which single instruction is processed by multiple threads in parallel simultaneously. Step 4, checks the condition whether the total number of threads to be executed is greater than the number of threads assigned. If the condition is true, mod() function of Algorithm 2 is called for further processing.

Algorithm 1:

```
            INPUT : Function to calculate modular exponential
            with *num , *key , *n and *results as input integer
            variables to __global__ void mod()
            OUTPUT: num**y mod n
Step 1:     Start
Step 2:     Declare integer variables temp, total_no_of_threads
Step 3:     Assign thread index
            i<-- threadIdx.x + blockDim.x*blockIdx.x
Step 4:     If(i< total_no_of_threads)
            temp <-- mod(num[i],*key,*n)
            atomicExch(&result[i],temp)
Step 5:     END
```

Algorithm 2:

```
            INPUT : Function to calculate modular exponential
            with g,e,n as input variable to __device__long long
            int mod()
            OUTPUT: g^e mod n
Step 1:     Start
Step 2:     Declare variable a,ret.size
Step 3:     a <-- (g%n) * (g%n)
Step 4:     ret <-- 1
Step 5:     size <-- e/2
Step 6:     If (e==0)
                  return (g%n)
            Else
                  while( true)
                        if(size > 0.5)
                        ret <-- (ret * a) % n
                        size <-- size - 1.0
                        else if ( size== 0.5)
                        ret <-- (ret * (g%n))%n
                        break
                        else
                        break
            return ret
Step 7:     END
```

Algorithm 2, process the mod() function which accepts three parameters. The result of Algorithm 2 is the encrypted text that is further send to the host as result[] array. __global__ keywords used in Algorithm 1, explains the execution of the rsa() on the device and is called from the host. Pointers variables are used because rsa() is supposed to run on device and for that variables must point to device memory. To accomplish this, memory is allocated on the GPU.

This proposed algorithm also process large values that built-in data types cannot support.

4.1.1 Integration of C with CUDA API

```
cudaGetDeviceCount(&devcount);
 printf("%d CUDA devices found",devcount);
 if(devcount>0)
 cudaSetDevice(1);
 int *dev_num,*dev_key,*dev_den;
 unsigned int *dev_res;
 unsigned  int res[3]={1,1,1};
 cudaMalloc( (void **)&dev_num, nsize*sizeof(int));
 cudaMalloc( (void **)&dev_key,sizeof(int));
 cudaMalloc( (void **)&dev_den, sizeof(int));
 cudaMalloc( (void **)&dev_res, nsize*sizeof(unsigned int));
 ts = GetTickCount();

cudaMemcpy(dev_num,num,nsize*sizeof(int),cudaMemcpyHostToDevic
e);
 cudaMemcpy(dev_key,&key,sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(dev_den,&den,sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(dev_res,res,nsize*sizeof(unsigned
int),cudaMemcpyHostToDevice);
 rsa<<<nblocks,nthreads>>>(dev_num,dev_key,dev_den,dev_res);
cudaMemcpy(res,dev_res,nsize*sizeof(unsigned
int),cudaMemcpyDeviceToHost);
cudaFree(dev_num);
cudaFree(dev_key);
cudaFree(dev_den);
cudaFree(dev_res);
```

Figure 4.1 CUDA-C Host Code Integration with Kernel Code.

The above is the small part of the host code showing the integration of CPU and GPU.

1. Check for available GPU devices;
2. Initialize host and device copies of variables;
3. Allocate space for device copies of variables using cudaMalloc();
4. Setup input values;
5. Copy input values to device using cudaMemcpy;
6. Launch kernel on GPU;
7. Copy result back to host using cudaMemcpy();
8. Cleanup.

4.2 Algorithm for OpenCL Kernel

Algorithm 1:

In Algorithm 1, arguments are passed as pointer to __kernel__ void RSA(). After that the thread index are assigned following the declaration of the variables. Each thread in

a block is assigned a unique id and is known as thread index. Threads follow the SIMT fashion in which single instruction is processed by multiple threads in parallel simultaneously. Input size is then calculated. Step 8, checks the condition whether the total number of threads to be executed is greater than the number of threads assigned. If the condition is true, mod() function of Algorithm 2 is called for further processing.

```
                INPUT : Function to calculate modular exponential
                with *num , *key , *n and *results as input global
                integer variables to __kernel__ void RSA()
                OUTPUT: num^key mod n
Step 1:         Start
Step 2:         Declare integer variables
                temp,i,Data,m,r,ret,no_of_elements
Step 3:         Current thread value
                i<-- get_global_id(0)
Step 4:         Calculate the input size
                m<-- get_global_size(0)
Step 5:         iData <--num[i]
Step 6:         r<--1
Step 7:         *ret = &r
Step 8:         If(i< no_of_elements)
                mod(num[i],*key,*n,*ret)
                result[i] = *ret
Step 9:         END
```

Algorithm 2:

```
                INPUT : Function to calculate modular exponential
                with g,e,n as input variable to __kernel__ void
                mod()
                OUTPUT: g^e mod n
Step 1:         Start
Step 2:         Declare variable a,ret.size
Step 3:         a <-- (g%n) * (g%n)
Step 4:         ret <-- 1
Step 5:         size <-- e/2
Step 6:         If (e==0)
                      return (g%n)
                Else
                      while( true)
                            if(size > 0.5f)
                            ret <-- (ret * a) % n
                            size <-- size - 1.0f
                            else if ( size== 0.5f)
                            ret <-- (ret * (g%n))%n
                            break
                            else
                            break
                return ret
Step 7:         END
```

Algorithm 2, process the mod() function which accepts four parameters. The result of Algorithm 2 is the encrypted text that is further send to the host as result[] array.

__kernel__ keywords used in Algorithm 1, explains the execution of the RSA() on the device and is called from the host. Pointers variables are used because rsa() is supposed to run on device and for that variables must point to device memory. To accomplish this, memory is allocated on the GPU.

This proposed algorithm also process large values that built-in data types cannot support.

4.2.1 OpenCL Host Code in Detail

Step1: Access the platforms and Device IDs

```
// Fetch the Platform and Device IDs; we only want one.

         error=clGetPlatformIDs(1, &platform, &platforms);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

        error=clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
&device, &devices);
        if (error != CL_SUCCESS)
        {
                printf("\n Error number %d", error);
          }
```

Figure 4.2 Accessing of Platforms and Device Ids in CUDA

While coding the program the underlying hardware along with the number of graphic cards a computer has is not known to the user. To make it possible, first the platforms to work further upon them are accessed. Once, the platforms available are known to the user, next step is to access the connected device provided by that particular vendor.

Step 2: Creation of the Context

Creating context is necessary to identify the set of devices that are supposed to work together and further helps in the creation of the command queue.

```
// CREATE CONTEXT
       cl_context context=clCreateContext(NULL, 1, &device,
NULL, NULL, &error);
         if (error != CL_SUCCESS) {
                 printf("\n Error number %d", error);
         }
```

Figure 4.3 Creation of Context in OpenCL.

Step 3: Creation of command queue

```
//CREATE COMMAND QUEUE
        cl_command_queue  cq  =  clCreateCommandQueue(context,
device, 0, &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
```

Figure 4.4 Creation of Command Queue.

Command queues are used to send kernels to the device. Command queues accomplish the task of sending kernel to device by allowing the host for it.

Step 4: Creation of the File

```
//CREATE FILE
          FILE *fil=fopen("rsa.cl","r");
          char *src;
        src = (char *)malloc(MAX_SOURCE_SIZE);
        srcsize=fread(src, 1, MAX_SOURCE_SIZE, fil);
          fclose(fil);
          const char *srcptr[]={src};
```

Figure 4.5 Creation of File in OpenCL.

Step 5: Creating and Building programs

```
// Submit the source code of the example kernel to OpenCL

cl_program prog=clCreateProgramWithSource(context, 1, srcptr,
&srcsize, &error);
        if (error != CL_SUCCESS)
        {
                printf("\n Error number %d", error);
        }

// and compile it (after this we could extract the compiled
version)
 error=clBuildProgram(prog, 1, &device, NULL, NULL, NULL);

        if ( error != CL_SUCCESS ) {
                printf( "Error on buildProgram " );
                printf("\n Error number %d", error);
                fprintf( stdout, "\nRequestingInfo\n" );
clGetProgramBuildInfo( prog, device, CL_PROGRAM_BUILD_LOG,
8096, build_c, NULL );
printf( "Build Log for %s_program:\n%s\n", "example", build_c
);
        }
```

Figure 4.6 Creating and Building Programs in OpenCL.

Program is a group of kernels and is represented by cl_program data structure. Creation and building of a program helps in deploying of kernel to the devices.

Step 6: Package functions in kernels

Kernels are the data structure that is used to package the functions after the compilation and linking process of a program. Kernels are deployable and can be sent to a device using command queue. In this step functions are firstly packaged and then finally the parameters for the kernel are mapped.

```
// get a handle and map parameters for the kernel
        cl_kernel k_example=clCreateKernel(prog, "RSA", &error);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

        error = clSetKernelArg(k_example, 0, sizeof(mem1),
&mem1);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

        error = clSetKernelArg(k_example, 1, sizeof(mem2),
&mem2);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

          error = clSetKernelArg(k_example, 2, sizeof(mem3),
&mem3);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

          error = clSetKernelArg(k_example, 3, sizeof(mem4),
&mem4);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
```

Figure 4.7 Setting of Kernel Arguments.

Step 7: Managing Data Transfer

This step shows the method of transferring the data from host memory to a buffer. After creating the target buffer ,input data is send to the OpenCL using clEnqueueWriteBuffer. In Figure 4.8 , mem1, mem2 , mem3 are the device memory spaces for buf, dev_key and dev_den respectively. Host's input data which is used for further kernel processing is stored in mem1, mem2 and mem3.

```
// Target buffer just so we show we got the data from OpenCL

 int buf2[262144];

// Send input data to OpenCL (async, don't alter the buffer!)

error=clEnqueueWriteBuffer(cq, mem1, CL_FALSE, 0, worksize, buf,
0, NULL, NULL);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

error=clEnqueueWriteBuffer(cq, mem2, CL_FALSE, 0, worksize,
dev_key, 0, NULL, NULL);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }

error=clEnqueueWriteBuffer(cq, mem3, CL_FALSE, 0, worksize,
dev_den, 0, NULL, NULL);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
```

Figure 4.8 Managing Data Transfer in OpenCL.

Step 8: Perform operation:

```
// Perform the operation

const size_t global_worksize=length;
error=clEnqueueNDRangeKernel(cq, k_example, 1, NULL,
&global_worksize, NULL, 0, NULL, NULL);
if (error != CL_SUCCESS) {
printf("\n Error number %d", error);
        }
```

Figure 4.9 Performing Operation in OpenCL.

This step focus on the data partitioning using the clEnqueueNDRangeKernel(). GPU programming process a large amount of data and better distribute the data, faster will be the execution and more efficient the application will be. This task is done by the above given function that places the kernel for execution in command queue. It also distributes the task among the processing resources of the devices in work.

Step 9: Reading result back to the host

```
// Read the result back into buf2

error=clEnqueueReadBuffer(cq, mem4, CL_FALSE, 0, worksize, buf2,
0, NULL, NULL);
if (error != CL_SUCCESS) {
printf("\n Error number %d", error);
        }
```

Figure 4.10 Copying Back the Results to the Host.

Step 10: Check for the successful completion

```
// Await completion of all the above

        error=clFinish(cq);
        if (error != CL_SUCCESS) {
                printf("\n Error number %d", error);
        }
```

Figure 4.11 Wait for the Successful Completion of the Results.

This is the last step which checks whether the task has completed successfully. Successful completion depicts the successful kernel execution.

The above steps are used while developing the RSA algorithm in OpenCL

4.3 Chapter Summary

This chapter explain the implementation of the work. Firstly RSA algorithm for both platforms are discussed followed by the host code integration with CUDA-C and explanation of OpenCL host code in detail. Testing and experimental results are discussed in Chapter 5.

CHAPTER 5

EXPERIMENTAL RESULTS

This chapter explains the testing and results of the work implemented in the previous chapter taking three test cases as follows:

1. In First test case, the RSA encryption algorithm for small prime numbers(n=131*137) on two platforms NVIDIA-CUDA using CUDA-C and sequentially using CPU only is analysed;
2. In Second test case, the RSA encryption algorithm for small prime numbers(n=131*137) parallel using OpenCL(GPU+CPU) and sequentially using CPU only is analysed;
3. In Third test case RSA encryption algorithm is developed using CUDA-C and OpenCL and the execution time of CUDA-C RSA and OpenCL RSA is compared.

For each test case, first the snapshots are discussed and then the results and their performance is analysed using table followed by the graphs. The computer used for testing has an Intel(R) Core(TM) i3-2370M 2.4GHZ CPU with 4 GB RAM. The whole work is done on Windows 7OS. Nvidia GeForce GT 630M is used as a graphic card with 512MB memory, and a 2GHZ DDR2 memory[10]. Visual Studio 2010 is used for developing and testing the RSA algorithm .The code has been tested for:

I. Values of messages between 0 and 800. It accommodate the complete ASCII table;
II. 8 bit key value.

5.1 Case 1: Analysis of CUDA-RSA and CPU-RSA for Small Prime Numbers

In this test case, results are analysed for parallelized algorithm developed for CUDA and traditional RSA that runs sequentially using CPU only. The whole results are carried for small value of n with value of (n=131*137). The further sections will elaborate the explanation of the analysis using snapshots for few data sets and then a

table for all data sets on which the results are analysed. Finally the results can be visually analysed from the graphs. Figure 5.1- Figure 5.4 compare the CUDA-RSA and CPU-RSA for 512, 2048, 8192, 32784 bytes data respectively. Execution time is compared between the two RSA algorithm taking Block Size, Threads as parameters for CUDA RSA. The Figures below will explain that the execution time varies depending upon the various factors.

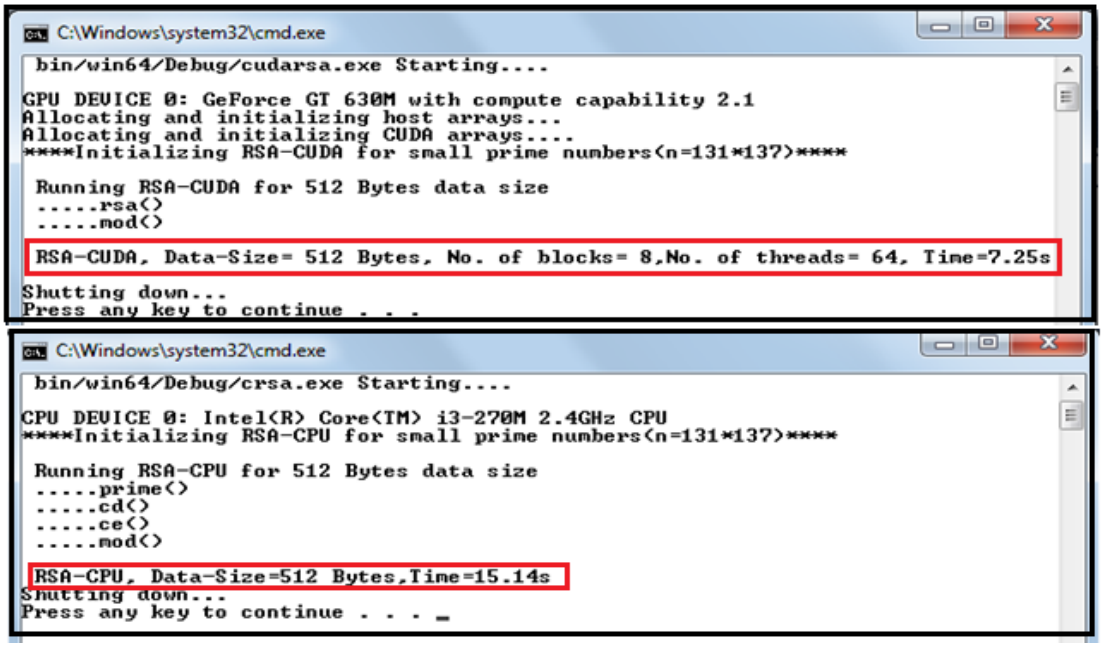

Figure 5.1 Comparison of CUDA-RSA and CPU-RSA for 512 Bytes Data

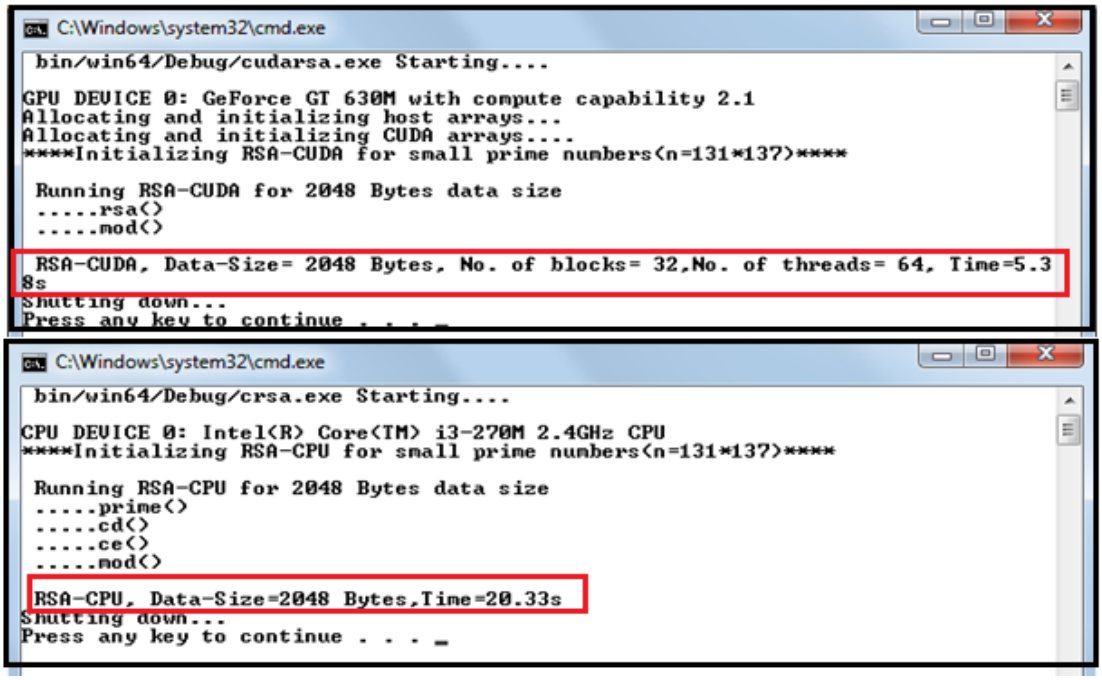

Figure 5.2 Comparison of CUDA-RSA and CPU-RSA for 2048 Bytes Data

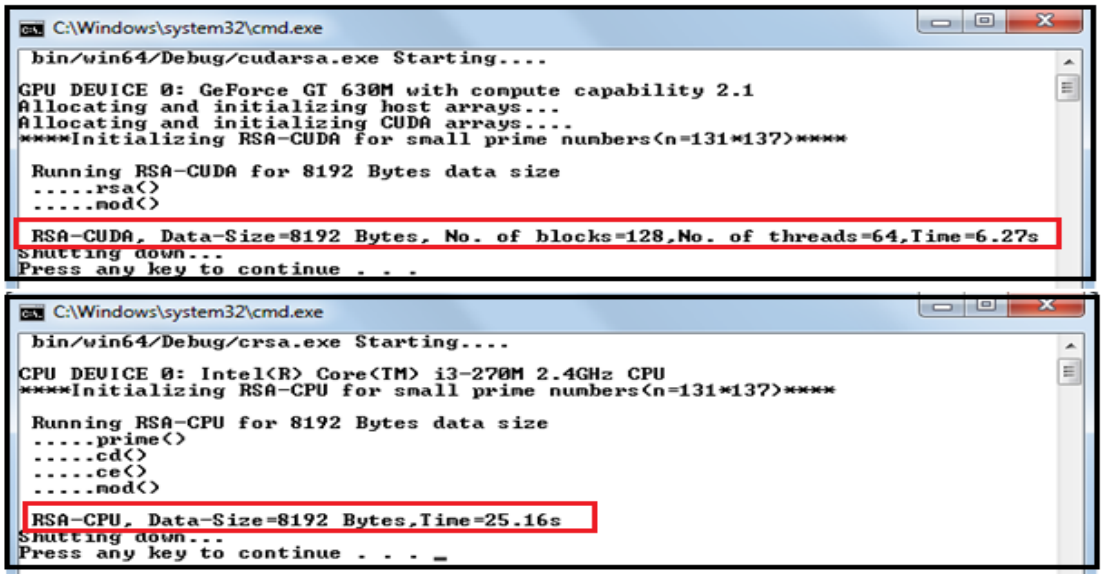

Figure 5.3 Comparison of CUDA-RSA and CPU-RSA for 8192 Bytes Data

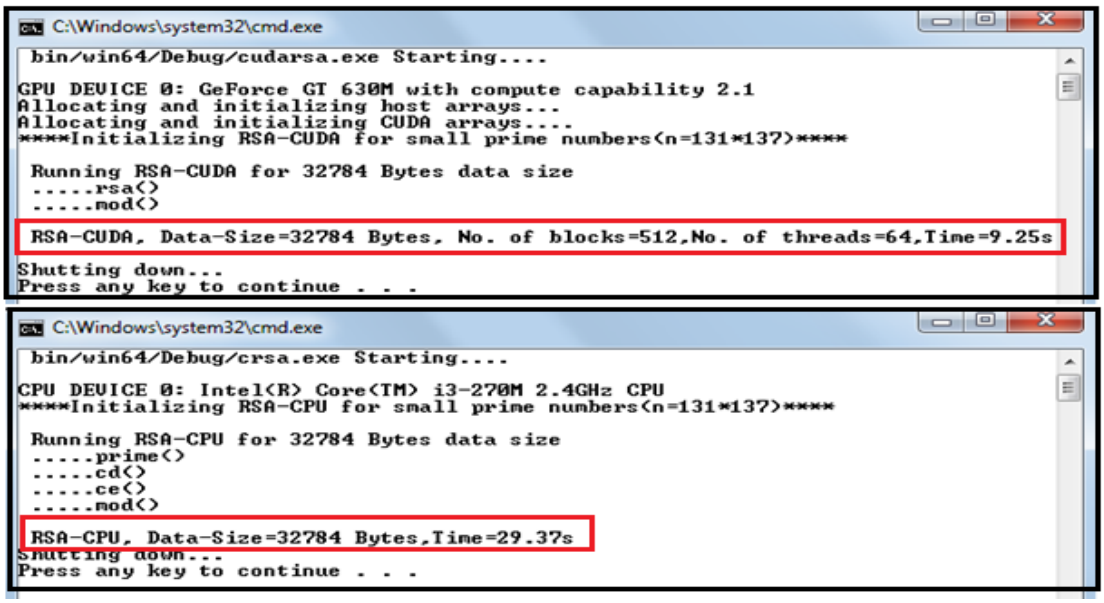

Figure 5.4 Comparison of CUDA-RSA and CPU-RSA for 32784 Bytes Data

As cleared from the Figure 5.1-Figure 5.4. that the noticeable performance gain is analysed between the two different RSA algorithms ,one that run sequentially using CPU only and the other that use GPU as a co-processor to CPU. The snapshots give a clear view that running RSA on GPU gives a performance gain. On small data sets the performance gain is not up to the mark but as the data load increases there exists a huge difference in the execution time of the both. So, the running of RSA algorithm using GPU as a co-processor to the CPU is worthy and as stated in the problem formulation, the results so received are as expected. The detailed analysis of the

algorithm using two different approaches can be analysed using the Table 5.1 given below.

Table 5.1 Comparison of CUDA-RSA and CPU-RSA for Different Data Size

Data size(bytes)	**No. of blocks**	**No. of threads**	**CUDA-RSA(T_1)**	**C-RSA(T_2)**	**Speedup(T_2/T_1)**
256	4	64	7.56	12.56	1.66
512	8	64	7.25	15.14	2.08
1024	16	64	6.86	17.60	2.56
2048	32	64	5.38	20.33	3.77
4096	64	64	5.68	22.64	3.98
8192	128	64	6.27	25.16	4.01
16392	256	64	7.21	26.66	3.69
32784	512	64	9.25	29.37	3.17

In this test case results are analysed on 8 different amount of data sets ranging from (256-32784) bytes. The number of threads used in a single block is constant. Each block runs 64 threads simultaneously, or each block executes kernel function or encryption operation on 64 different data sets simultaneously. The number of blocks used varies accordingly the data size, so as to get as efficient encryption as possible.
As, results from the above Table 5.1 shows that when CUDA-RSA is run for small data size i.e., 256 bytes the speedup ratio is not much. As the data size increases the execution time of the CUDA RSA decreases and that of traditional RSA keeps on increasing, as expected.

The shocking result is running CUDA-RSA on 256 bytes gives execution time of 7.56 seconds and running the same on a larger data amount of 16392 gives the execution time of 7.21 seconds. From here, it is clear that where sequential time keeps on increasing with the increase of data size from 12.56 to 26.66, there is no increase in execution time on the other side. Hence, CUDA-RSA performs best for larger data set.

The algorithm is run 20 times on the same data set and all the results given in the Table 5.1 are calculated by taking the average of the results ,so as to achieve the high level of accuracy.

The result given in the Table 5.1 can be best analysed from Figure 5.5.

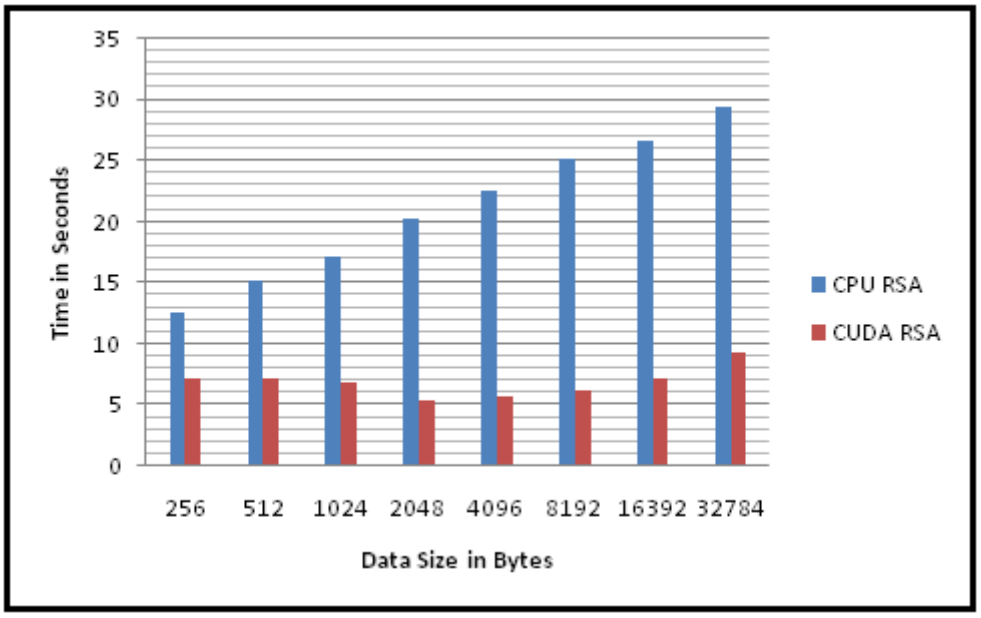

Figure 5.5. Graph Showing the Comparison of CUDA-RSA and CPU-RSA for Small Prime Numbers

5.2 Case 2: Analysis of OpenCL-RSA and CPU-RSA for Small Prime Numbers

In this test case, results are analysed for parallelized algorithm developed for OpenCL and traditional RSA that run sequentially using CPU only. The whole results are carried for small value of n with value of (n=131*137). The further sections will elaborate the explanation of the analysis using snapshots for few data sets and then a table for all data set on which the results are analysed and finally the results can be visually analysed from the graphs same as done earlier.

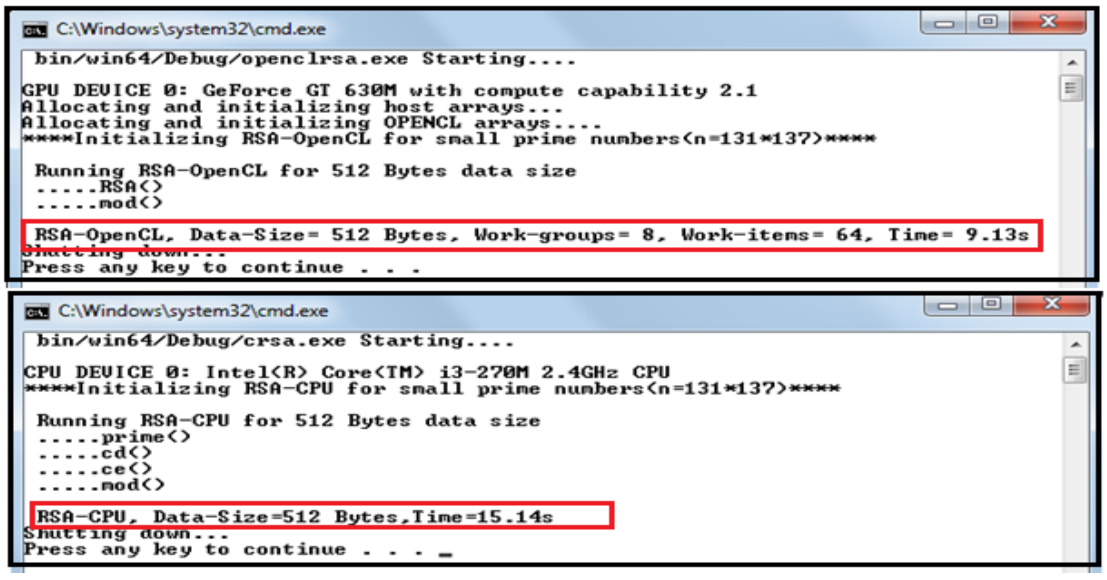

Figure 5.6 Comparison of OpenCL-RSA and CPU-RSA for 512 Bytes Data

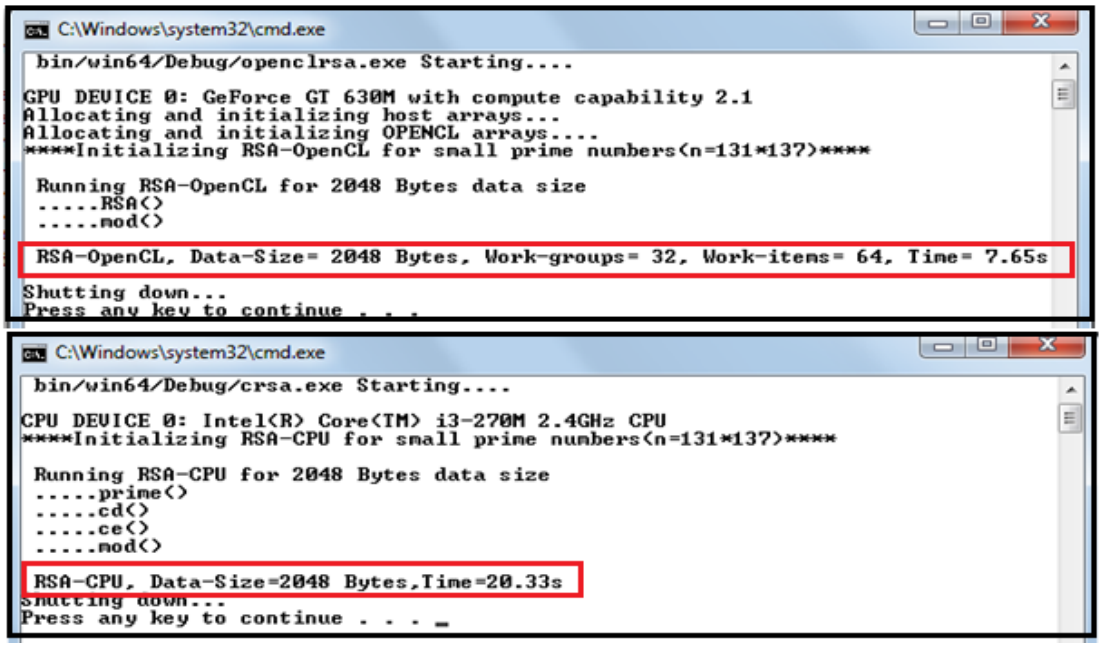

Figure 5.7 Comparison of OpenCL-RSA and CPU-RSA for 2048 Bytes Data

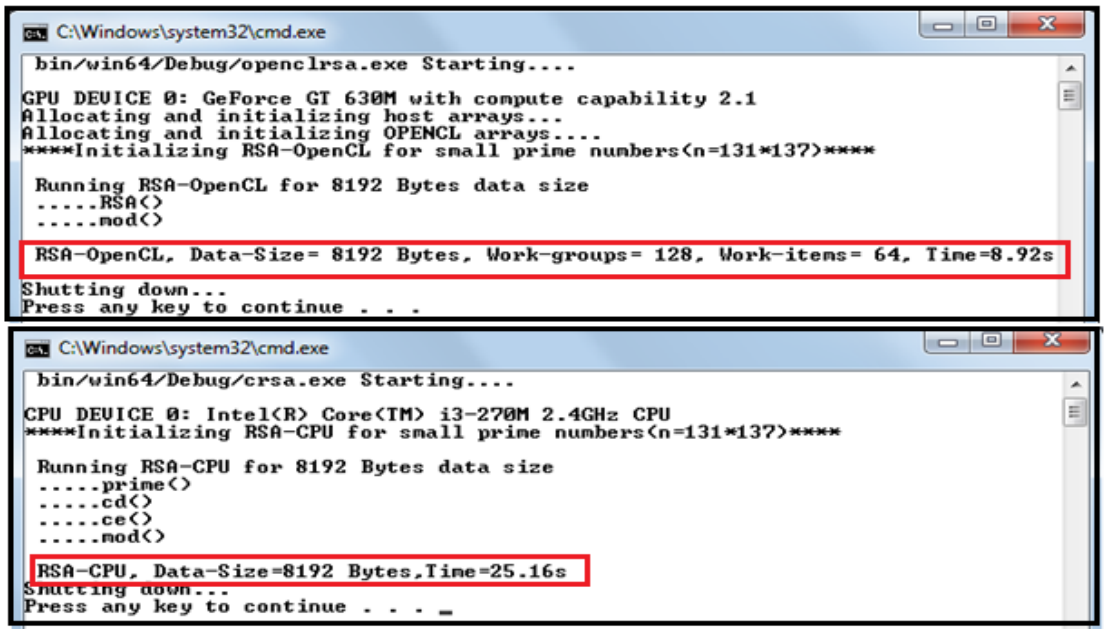

Figure 5.8 Comparison of OpenCL-RSA and CPU-RSA for 8192 Bytes Data

As cleared from the above Figure 5.6-Figure 5.9 that the noticeable performance gain is again analysed between the two different RSA algorithms ,one that run sequentially using CPU only and the other that use GPU as a co-processor to CPU i.e., using OpenCL this time. The Figure 5.6-Figure 5.9 gives a clear view of what is stated in the problem statement. Running RSA on GPU using OpenCL gives a performance gain. But on small data sets the performance gain is not up to the mark. As the data load increases there exists a huge difference in the execution time of the both.

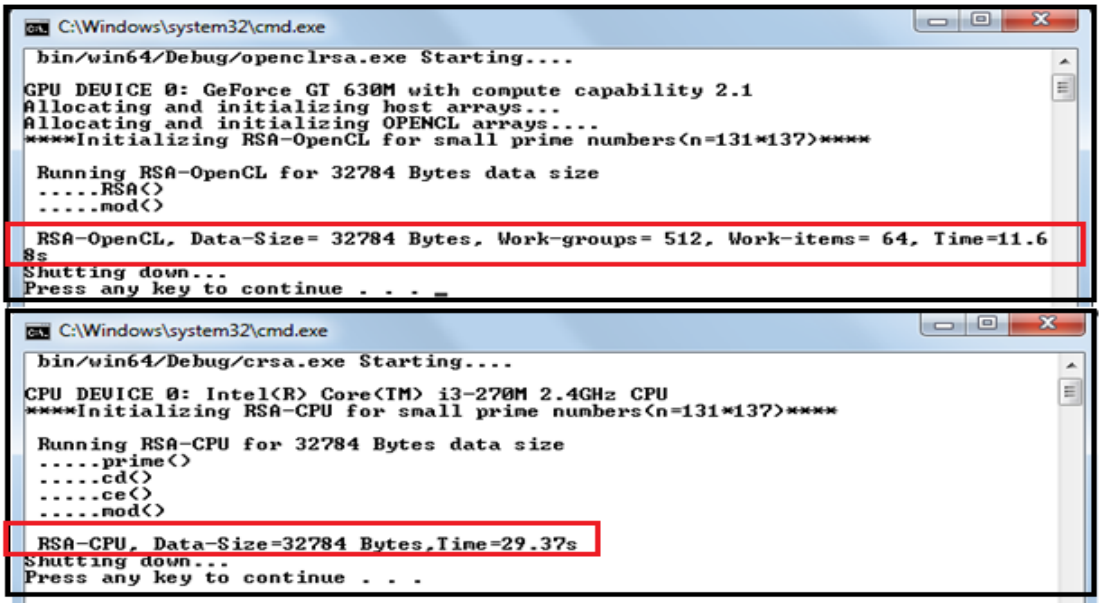

Figure 5.9 Comparison of OpenCL-RSA and CPU-RSA for 32784 Bytes Data

So, the running of RSA algorithm using GPU as a co-processor to the CPU is worthy and as stated in the problem formulation, the results so received are as expected. The detailed analysis of the algorithm using two different approaches can be analysed using the Table 5.2 given below.

Table 5.2 Comparison of OpenCL-RSA and CPU-RSA for Small Prime Numbers

Data size(bytes)	No. of work-groups	No. of work-items	OpenCL RSA(T_1)	C RSA(T_2)	Speedup(T_2/T_1)
256	4	64	9.45	12.56	1.32
512	8	64	9.13	15.14	1.65
1024	16	64	8.23	17.60	2.13
2048	32	64	7.65	20.33	2.65
4096	64	64	7.78	22.64	2.91
8192	128	64	8.92	25.16	2.82
16392	256	64	9.13	26.66	2.92
32784	512	64	11.68	29.37	2.51

In this test case, tests are carried for 8 different amount of data sets ranging from (256-32784) bytes. The number of work-items used in a single work-group is constant. Each work-group operates on 64 work-items simultaneously, or each work-group executes kernel function or encryption operation on 64 different data sets simultaneously. The number of work-group used varies accordingly the data size, so as to get as efficient encryption as possible.

Table5.2 shows that when we run OpenCL-RSA for small data size i.e., for 256 bytes,

the speedup ratio is not much. As the data size increases the execution time of the OpenCL RSA decreases and that of traditional RSA keeps on increasing, as expected.

The shocking result is that running OpenCL-RSA on 512 bytes gives execution time of 9.13 seconds and running the same on a larger data amount of 16392 gives the execution time of 9.13 seconds. It means execution time of (16392-512=15880) bytes of data is negligible or zero.

Sequential time keeps on increasing with the increase of data size from 15.14 to 26.64, there is no increase in execution time on the other side. Hence, OpenCL-RSA performs best for larger data set.

The algorithm is run 20 times on the same data set and all the results given in the Table 5.2 are calculated by taking the average of the results, so as to achieve the high level of accuracy.

The result given in the Table 5.2 can be best analysed using the graph given below.

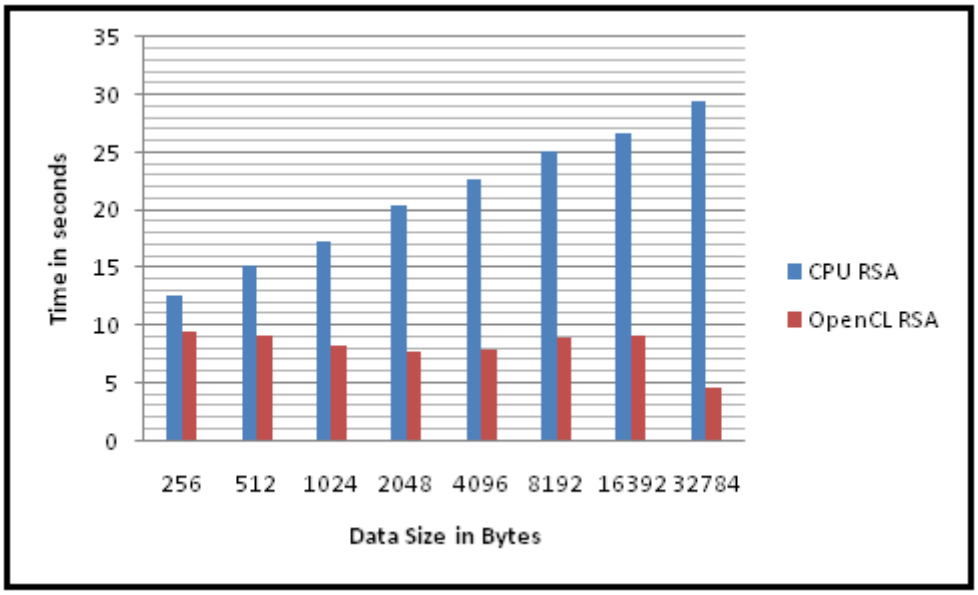

Figure 5.10 Graph Showing the Comparison of OpenCL-RSA and CPU-RSA

5.3 Analysis of the Speedup Graph of the Above Two Cases

In this section the speedup so calculated is analysed using the graph. The speedup is calculated using the below formula:

$$\frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

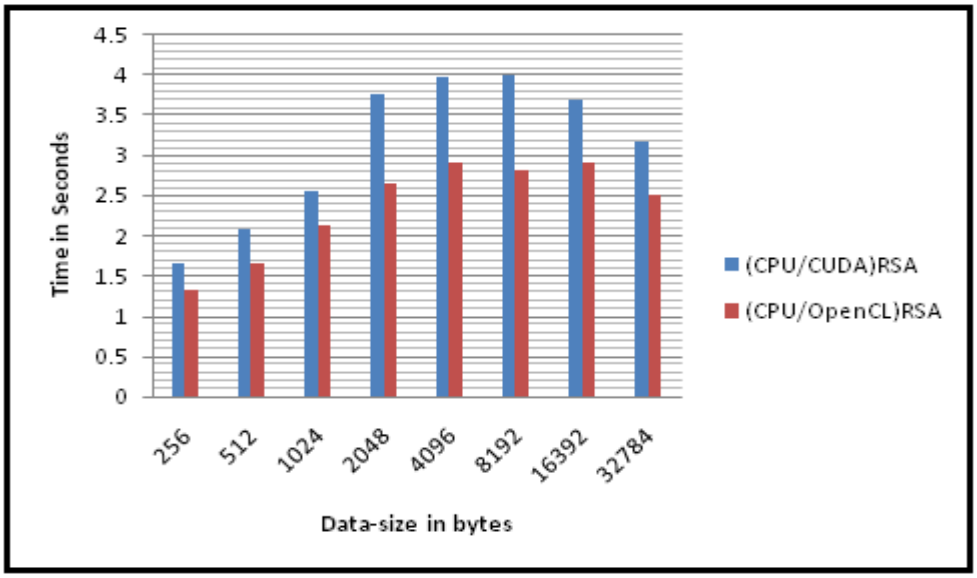

Figure 5.11 Graph Showing the Speedup Comparison of CPU/CUDA-RSA and CPU/OpenCL-RSA

5.4 Case 3: Analysis of CUDA-RSA and OpenCL-RSA for Large Prime Numbers

In this test case, results are analysed for parallelized algorithm developed for OpenCL and CUDA. The whole results are carried for large value of n with value of (n=1009*509). The further sections will elaborate the explanation of the analysis using snapshots for few data sets and then a table for all data set on which the results are analysed and finally the results can be visually analysed from the graphs same as done earlier.

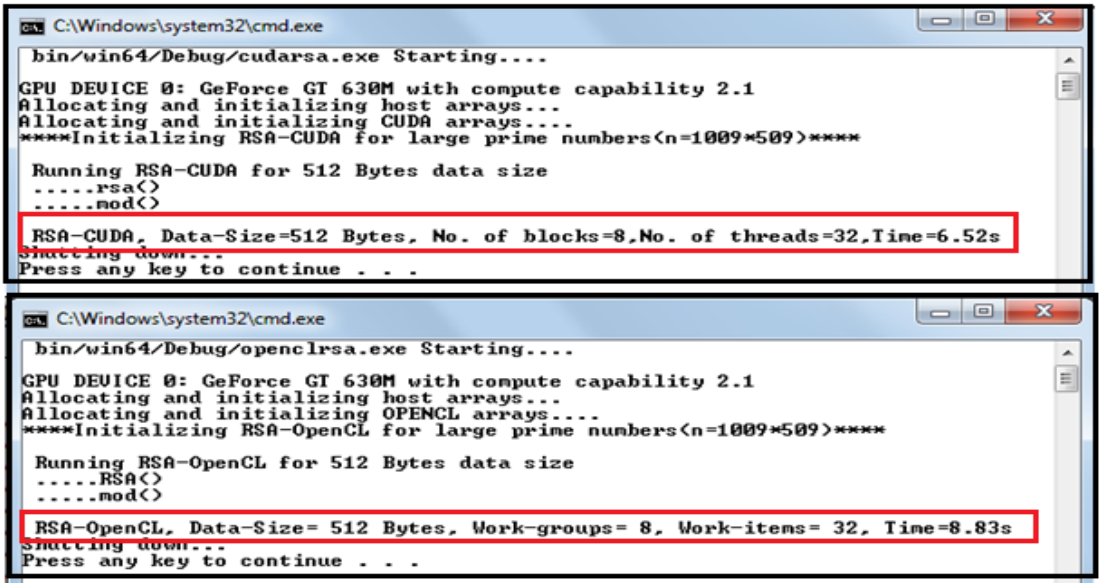

Figure 5.12 Comparison of CUDA-RSA and OpenCL-RSA for 512 Bytes Data

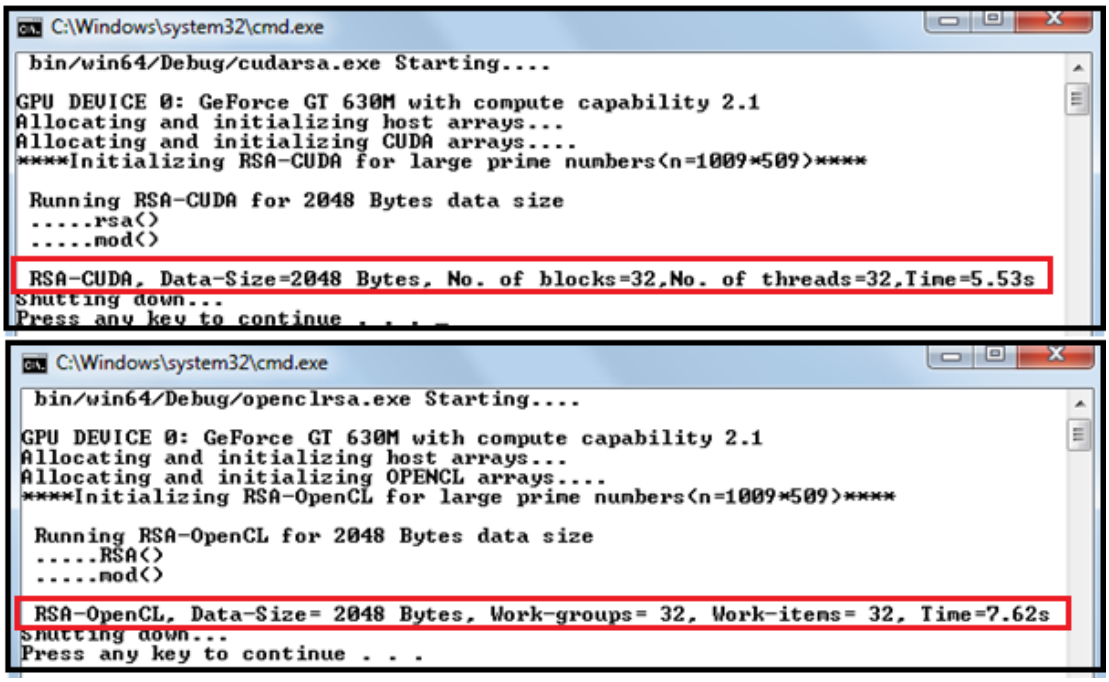

Figure 5.13 Comparison of CUDA-RSA and OpenCL-RSA for 2048 Bytes Data

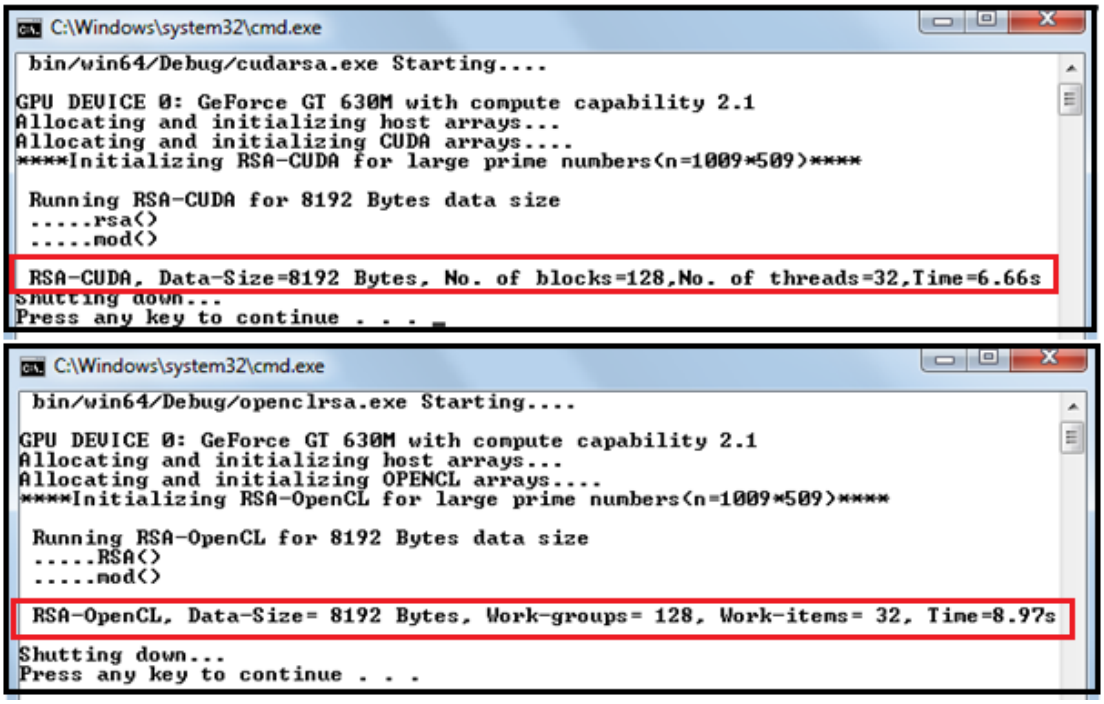

Figure 5.14 Comparison of CUDA-RSA and OpenCL-RSA for 8192 bytes data

As cleared from the Figure 5.12-Figure 5.15 that the noticeable performance gain is analysed between the two different RSA algorithms on GPU, one that run using CUDA-C and the other that use OpenCL. The snapshot gives a clear view that running RSA on CUDA gives less execution time as compared to OpenCL-RSA. So, running of RSA algorithm using CUDA-C as a co-processor to the CPU is more

efficient than the OpenCL .The detailed analysis of the algorithm using two different GPU platforms can be analysed using the Table 5.3.

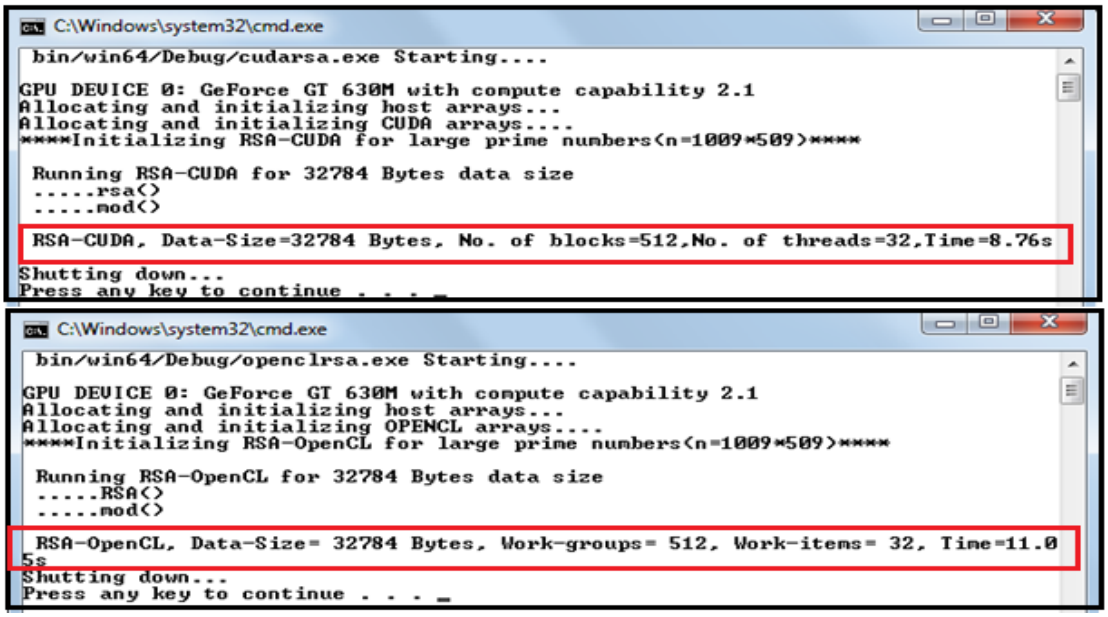

Figure 5.15 Comparison of CUDA-RSA and OpenCL-RSA for 32784 Bytes Data

Table 5.3 Comparison of CUDA-RSA and OpenCL-RSA for Large Prime Numbers

Data size(bytes)	No. of work-groups/blocks	No. of work-items/threads	CUDA RSA	OpenCL-RSA
256	4	32	6.08	8.57
512	8	32	6.52	8.83
1024	16	32	6.69	9.01
2048	32	32	5.53	7.62
4096	64	32	6.58	8.88
8192	128	32	6.66	8.97
16392	256	32	7.81	9.86
32784	512	32	8.76	11.05

In this test case, tests are carried for 8 different amount of data sets ranging from (256-32784) bytes. The number of work-items/threads used in a single work-group/block is constant. Each work-group/block operates on 32 work-items/threads simultaneously, or each work-group/block executes kernel function or encryption operation on 64 different data sets simultaneously. The number of work-group/blocks used varies accordingly the data size, so as to get as efficient encryption as possible.

In this tests are carried on different amount of data-size only to analyse which GPU platforms works efficiently for RSA algorithm.

As, cleared from the above Table 5.3 that RSA works efficiently on CUDA-C and takes less time as compared to OpenCL-RSA. Both the GPU scenarios follow the same trend of better performance on large amount of data size. Execution time of 512 bytes data size is same as the execution time of 8192 bytes of data using CUDA-C RSA. Same in case of OpenCL, the execution time of 256 bytes of data is nearly equivalent to the execution time of 8192 bytes of data.

But when the execution time is compared for each data set , the execution time of RSA using CUDA is less than the execution time of RSA using OpenCL and this can be best visualised using the below graph.

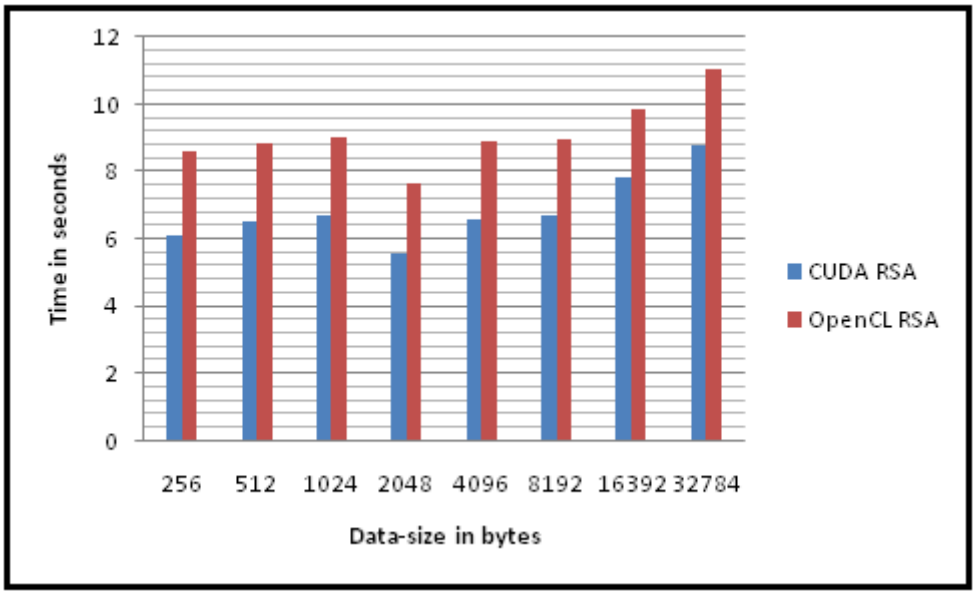

Figure 5.16 Comparison of CUDA-RSA and OpenCL-RSA for Large Prime Numbers.

5.5 Chapter Summary

This chapter analyse the RSA algorithm using three different frameworks. GPU implementations of RSA algorithm using CUDA and OpenCL works efficiently as compared to the traditional sequential algorithm as expected and stated in the problem statement.

GPU implementation of RSA works best for large data size. For small data size GPU implementation do not gives better results. CUDA-RSA and OpenCL RSA algorithm works efficiently on large amount of data size with noticeable performance gain.

On comparison between CUDA RSA and OpenCL RSA for small and large value of prime numbers CUDA works more efficiently as compared to the OpenCL.

CHAPTER 6

CONCLUSIONS AND FUTURE SCOPE

6.1 Discussion

This work is implemented using NVIDIA GT-630M having 96 processing cores with graphics clock speed of 0.8GHz and the CPU clock speed of 2.4 GHz. Theoretically GPU-RSA can run 32 (96*0.8/2.4) times faster than the CPU implementation. However, results shows that the CUDA RSA can run 4.01 times faster than the CPU and OpenCL RSA can run 2.92 times faster than the CPU implementation.

Factors that influence the performance of OpenCL and CUDA program are as follows:

1. Many of the computations are still done sequentially. Such as Key Generation process, testing of prime numbers and all.
2. Access speed also affect,as different memories have different access speeds. Data transfer time from global memory to shared memory and vice-versa add on the extra execution time. Global memory speed is very slow and to achieve better performance gain ,global memory data should be transferred to shared memory first if the memory is accessed many times. Furthermore, only threads within a block can share the shared memory resulting in transferring of the temporary results so calculated again back to global memory, if threads of other blocks require that results.
3. In GPU implementation, threads are time sliced. The kernel implementation require more threads than the available processors. Whereas in CPU there is no such problem of time slicing. Time slicing using round robin in GPU implementation add on the extra run time.
4. Synchronization of threads also add on the extra run time

6.2 Conclusions

This work analyse the RSA algorithm using three different frameworks. GPU implementations of RSA algorithm using CUDA and OpenCL works efficiently as

compared to the traditional sequential algorithm as expected and stated in the problem statement.

GPU implementation of RSA works best for large data size. For small data size GPU implementation do not gives better results. CUDA-RSA and OpenCL RSA algorithm works efficiently on large amount of data size with noticeable performance gain.

On comparison between CUDA RSA and OpenCL RSA for small and large value of prime numbers CUDA works more efficiently as compared to the OpenCL.

The work concludes that using GPU as a co-processor to CPU, Secure, Fast and Efficient RSA algorithm can be developed which otherwise is not possible.

6.3 Future Scope

The parallelized RSA algorithm developed, works for 8-bit key. It can be implemented using large bit size making the algorithm more efficient and secure.

Also the algorithm so developed only focuses on the computationally expensive part of the algorithm such as modular exponential and modular multiplication .The same approach can be used for various sequential functions used such as key generation, determination of prime number etc. It will raise the performance gain towards the theoretical expected performance gain.

Same approach can be applied to decryption process in the further research work.

REFRENCES

[1] H. Fadhil , M. Younis , " Parallelizing RSA Algorithm on Multicore CPU and GPU" , International Journal of Computer Applications , vol.87,No. 6, pp.15-22, February 2014.

[2] C. YAN, and Z. ZHNG, "Efficient acceleration about encryption and decryption of RSA algorithm on GPU ", Journal of Computer Applications , 2013.

[3] W. R. Joseph . "PARIS: A PArallel RSA-prime InSpection tool." (2013).

[4] S. Neves and F. Araujo,"On the performance of GPU public-key cryptography", Application-Specific Systems, Architectures and Processors (ASAP), IEEE International Conference , pp. 133-140 , 2011 .

[5] J. Fang, A. Varbanescu, and H. Sips,"A comprehensive performance comparison of CUDA and OpenCL." Parallel Processing (ICPP), IEEE International Conference , pp. 216-225, 2011.

[6] Karimi, Kamran, N. Dickson, and F.Hamze, "A performance comparison of CUDA and OpenCL", arXiv preprint arXiv:1005.2581, 2010.

[7] W.Fan, X. Chen, and X. Li, "Parallelization of RSA algorithm based on compute unified device architecture",Grid and Cooperative Computing (GCC), IEEE 9th International Conference , pp. 174-178. , 2010.

[8] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu,"Optimization principles and application performance evaluation of a multithreaded GPU using CUDA", Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73-82, 2008.

[9] CudaVsOpenCL. Available: http://wiki.tiker.net/CudaVsOpenCL

[10] S.Mahajan, M.Singh, "Efficient Algorithm for RSA Text Encryption using CUDA-C", Fifth International Conference on Communications Security & Information Assurance (CSIA-2014), pp. 233-241,May 14-15, 2014.

[11] Porting CUDA Applications to OpenCL™ | AMD. Available : http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/porting-cuda-applications-to-opencl/

[12] M. Scarpino, “Foundations of OpenCL Programming,” OpenCL in Action , 2nd ed. Shelter Island NY, USA: Manning, 2012, ch.1,pp.3-14 .

[13] Khronos OpenCL Working Group. "The OpenCL Specification, version 1.0. 29, 8 December 2008." URL http://khronos.org/registry/cl/specs/opencl-1.0 29

[14] H.T. David, and T. S. Abdelrahman, "hi CUDA: a high-level directive-based language for GPU programming," Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, pp. 52-61, 2009

[15] M.Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing in CUDA", IEEE micro 28, vol no. 4 , pp.13-27,2008

[16] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture", Ieee Micro 28, vol no. 2, pp. 39-55 , 2008.

[17] R.Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa,"The OpenCL Programming Book", Fixstars Corporation 63, 2010.

[18] W.W.Hwu, “GPU Computing Gems Jade Edition”, Morgan Kaufmann Publishers Inc., 2011.

[19] H. Wu, "Implementation of public key algorithms in CUDA", 2010.

[20] R.Szerwinski, and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography", Cryptographic Hardware and Embedded Systems–CHES , Springer, pp. 79-99, 2008.

[21] Nvidia, C. U. D. A, "Programming guide", 2008.

[22] Nvidia, C. U. D. A, "Compute unified device architecture programming guide", 2007.

[23] Udacity CS344: Intro to Parallel Programming. Available: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming

[24] Coursera: Heterogeneous Parallel Programming. Available: https://www.coursera.org/#course/hetero

[25] V. Mohan, "PROGRAMMING MODEL FOR GPGPU FOR VARIOUS ALGORITHMIC PROBLEMS",PhD diss., VIT UNIVERSITY, 2012.

[26] J. D.Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing", Proceedings of the IEEE 96, vol no. 5 ,pp. 879-899,2008.

[27] W. Stallings, " Network and internetwork security: principles and practice", vol. 1, Upper Saddle River, NJ: Prentice Hall, 1995.

[28] D.B. Kirk, and W. H. Wen-mei, "Programming massively parallel processors: a hands-on approach", Newnes, 2012.

[29] W.H. Wen-Mei, "GPU Computing Gems Emerald Edition", Elsevier, 2011.

[30] P.M.C.Saraiva, "OpenSSL acceleration using Graphics Processing Units",PhD diss.,TECNICO LISBOA ,2013.

PUBLICATIONS

[1] Sonam Mahajan,Maninder Singh, "Performance Analysis of Efficient RSA Text Encryption Using NVIDIA CUDA-C and OpenCL",Second International Conference on Emerging Research in Computing, Information, Communication and Applications- (ERCICA-14) , Elsevier, Banglore, to be held on Aug 1-2, 2014.
Status-Accepted.

[2] Sonam Mahajan, Maninder Singh, "Efficient Algorithm for RSA Text Encryption using CUDA-C",Fifth International Conference on Communications Security & Information Assurance (CSIA-2014),Delhi, pp. 233-241, May 14-15, 2014.
Status-Published.