Performance Analysis of Efficient RSA Text Encryption Using NVIDIA CUDA-C and OpenCL

Sonam Mahajan
PG Student
Department of CSE
Thapar University,Patiala
+917837068177
sonam_mahajan@yahoo.in

Maninder Singh
Associate Professor
Department of CSE
Thapar University , Patiala
+919815608309
msingh@thapar.edu

ABSTRACT
Computer security relies on cryptography as a means to protect the data that we people have become increasingly reliant on. The main research in computer security domain is how to enhance the speed of RSA algorithm. The computing capability of Graphic Processing Unit as a co-processor of the CPU can leverage massive-parallelism. CUDA and OpenCL are the two programming interfaces used for this purpose. 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. This paper presents RSA algorithm as a source for comparing the performance of CUDA and OpenCL. First the traditional algorithm is studied. Secondly, the parallelized RSA algorithm is designed for both Nvidia CUDA and OpenCL framework. Thirdly, the designed algorithm is realized for small and large prime numbers on CUDA, OpenCL and C and performance gap is concluded. So, the main fundamental problem of RSA algorithm such as speed and use of poor or small prime numbers resulting in significant security holes, despite the RSA algorithm's mathematical soundness can be alleviated by GPU programming. Decision depends which programming interface is to use depending upon the various factors such as prior familiarity with the existing system, availability of development tools to target the GPU hardware, performance and portable programming language.

Keywords
CPU, GPU, CUDA, RSA, Cryptographic Algorithm.

1. INTRODUCTION
RSA (named for its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman [1]) is a public key encryption scheme. This algorithm relies on the difficulty of factoring large numbers which has seriously affected its performance and so restricts its use in wider applications. Therefore, the rapid realization and parallelism of RSA encryption algorithm has been a prevalent research focus. With the advent of CUDA and OpenCL technology, it is now possible to perform general-purpose computation on GPU [2]. The primary goal of our work is to speed up the most computationally intensive part of RSA process. For that GCD and modulo calculation of RSA keys is implemented using NVIDIA's CUDA and OpenCL platform and finally a performance analysis is done between the two.

The reminder of this paper is organized as follows. In section 2, we study the traditional RSA algorithm. In section 3 and 4, we explained architecture of programming interfaces. In section 5, we explained the design and implementation of parallelized algorithm on both frameworks. Section 6 gives the result of our parallelized algorithm and section 7 concludes the paper.

2. TRADITIONAL RSA ALGORITHM
RSA is an algorithm for public- key cryptography [1] and is considered as one of the great advances in the field of public-key cryptography. It is suitable for both signing and encryption. Electronic commerce protocols mostly reply on RSA for security. Sufficiently long keys and up-to-date implementation of RSA is considered more secure to use. RSA is an asymmetric key encryption scheme which makes use of two different keys for encryption and decryption. The public key that is known to everyone is used for encryption. The messages encrypted using the public key can only be decrypted by using private key. The key generation process of RSA algorithm is as follows:

The public key is comprised of a modulus n of specified length (the product of primes p and q), and an exponent e. The length of n is given in terms of bits, thus the term "8-bit RSA key" refers to the number of bits which make up this value. The associated private key uses the same n, and another value d such that $d*e = 1 \bmod \varphi(n)$ where $\varphi(n) = (p - 1)*(q - 1)$ [3]. For a plaintext M and cipher text C, the encryption and decryption is done as follows:

$C = M^e \bmod n$, $M = C^d \bmod n$.

For example, the public key (e, n) is (131, 17947), the private key (d, n) is (137, 17947), and let suppose the plaintext M to be sent is: parallel encryption

ICONIAAC '14, October 10 - 11 2014, Amritapuri, India

978-1-4503-2908-8/14/08…$15.00.
http://dx.doi.org/10.1145/2660859.2660941

- Firstly, the sender will partition the plaintext into packets as: pa ra ll el en cr yp ti on. We suppose a is 00, b is 01, c is 02..... z is 25;
- Then further digitalize the plaintext packets as: 1500 1700 1111 0411 0413 0217 2415 1908 1413;
- After that using the encryption and decryption transformation given above calculate the cipher text and the plaintext in digitalized form;
- Convert the plaintext into alphabets, which is the original: parallel encryption.

3. CUDA ARCHITECTURE OVERVIEW[3]

NVIDIA's Compute Unified Device Architecture (CUDA) platform provides a set of tools to write programs that make use of NVIDIA's GPUs [3]. These massively-parallel hardware devices process large amounts of data simultaneously and allow significant speedups in programs with sections of parallelizable code making use of the Simultaneous Program, Multiple Data (SPMD)[4] model.

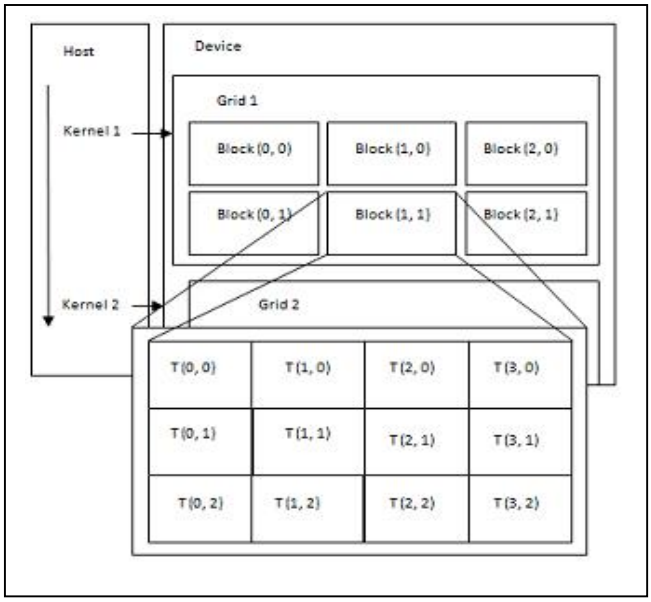

Figure1. CUDA system model

The platform allows various arrangements of threads to perform work, according to the developer's problem decomposition. In general, individual threads are grouped into up-to 3-dimensional blocks to allow sharing of common memory between threads. These blocks can then further be organized into a 2-dimensional grid. The GPU breaks the total number of threads into groups called warps, which, on current GPU hardware, consist of 32 threads that will be executed simultaneously on a single streaming multiprocessor (SM). The GPU consists of several SMs which are each capable of executing a warp. Blocks are scheduled to SMs until all allocated threads have been executed. There is also a memory hierarchy on the GPU. Three of the various types of memory are relevant to this work: global memory is the slowest and largest; shared memory is much faster, but also significantly smaller; and a limited number of registers that each SM has access to. Each thread in a block can access the same section of shared memory.

4. OpenCL ARCHITECTURE OVERVIEW

OpenCL is an open standard framework for programming on heterogeneous platforms. It is a framework for parallel programming. The main aim of OpenCL is to write a portable yet effective code. The following hierarchy of models describe the OpenCL in detail:

4.1 Platform Model[5]

This model consists of a host device which is connected with the OpenCL compliant devices. OpenCL device consists of many compute units which are divided into processing elements. Processing elements are responsible for performing computations on a device. Host device executes the host application and it sends commands to the processing elements which present in the GPU devices for execution of the parallel code. Figure 2 explains the platform model for OpenCL.

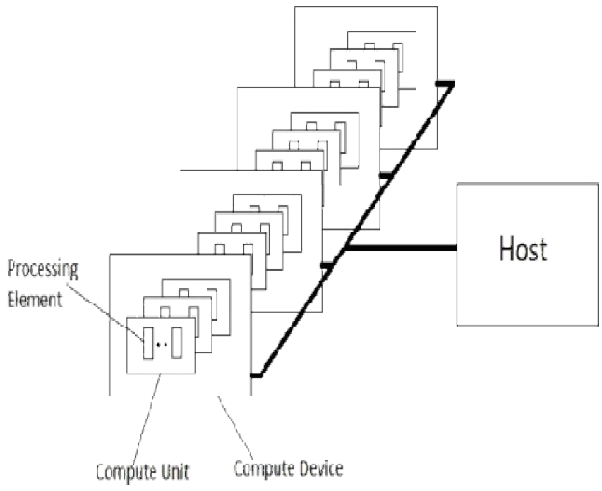

Figure 2. OpenCL Platform model

4.2 Execution Model[5]

Execution of OpenCL program consists of two parts: Kernels and host program. Kernels execute on OpenCL compliant devices and host program executes on host. Host program is responsible for defining the context for kernels and its management. The main task of execution model is execution of kernels.

In OpenCL, tasks are known as kernels. Kernels are functions that are sent to OpenCL complaint devices by host application and host application is a regular C/C++ application program. Host application manages devices with the help of context and context act as a device container. Program is a kernel container from which host selects a function to create a kernel. Kernel is dispatched to a command queue. Through command queue, host tells devices what to do. Figure 3 explains the Kernel distribution among devices.

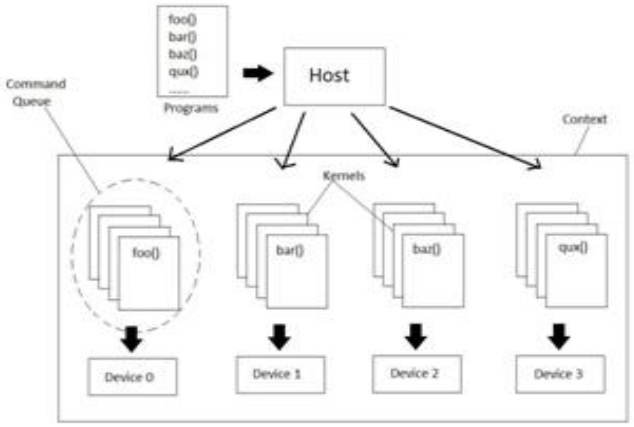

Figure 3. OpenCL execution model

4.3 Memory Model[5]

In OpenCL, every kernel argument that references memory has an address space modifier. There are four address spaces. Global memory stores data for entire GPU device. It is a read and write memory. Constant is similar to global memory but it is a read only memory. In local memory, data for all the work items in a work group is stored in this memory. Private memory stores data for a particular work item.

4.4 Programming Model[5]

OpenCL supports data parallel and task parallel parallel programming models. Data parallel programming model is the primary model for the design of the OpenCL. In a data parallel system, each device receives the same instructions. But it operates on different sets of data. Task parallel programming model allows different devices to perform different tasks. Each task operates on different data

5. RSA PARALLELIZED ALGORITHM

The algorithm used to parallelize the RSA modulo function works as follows in CUDA:

- CPU accepts the values of the message and the key parameters;
- CPU allocates memory on the CUDA enabled device and copies the values on the device;
- CPU invokes the CUDA kernel on the GPU;
- GPU encrypts each message character with RSA algorithm with the number of threads equal to the message length;
- The control is transferred back to CPU;
- CPU copies and displays the results from the GPU.

As per the flow given above the kernel is so built to calculate the cipher text $C = M^e \bmod n$. The kernel so designed works efficiently and uses the novel algorithm for calculating modulo value. The algorithm for calculating modulo value is implemented such that it can hold for very large power of numbers which are not supported by built in data types. The modulus value is calculated using the following principle:

- $C = M^e \bmod n$;
- $C = (M^{e-x} \bmod n * M^x \bmod n) \bmod n$.

Hence iterating over a suitable value of x gives the desired result.

5.1 OpenCL and CUDA Kernel Code

Kernel execution in two different GPU frameworks CUDA and OpenCL differ substantially. Their APIs are different for different purposes such as context creation and data copying APIs differ in CUDA and OpenCL framework. The conventions for kernel mapping onto different GPU's processing elements also differ. These differences lead to run-time performance gap between the two frameworks. The kernel code used in our experiment is shown below. First user assign the block and thread index as in CUDA and work-groups and work-items in OpenCL, so as to let each thread/work-item know which elements they are supposed to process. It is shown in below Figure4 and Figure5. Then it makes a call for another device / kernel function to calculate the most intense part of the RSA algorithm.

```
__global__ void rsa(int * num,int *key,int *den,unsigned int * result)
{
int i=threadIdx.x + blockDim.x * blockIdx.x;
int temp;

if(i<3)
{
        temp=mod(num[i],*key,*den);
        atomicExch(&result[i],temp);
}
}

__device__ long long int mod(int base,int exponent,int den)
{
        unsigned int a=(base%den)*(base%den);
        unsigned long long int ret=1;
        float size=(float)exponent/2;
        if(exponent==0)
        {
                return base%den;
        }
        else
        {
                while(1)
                {
                        if(size>0.5)
                        {
                                ret=(ret*a)%den;
                                size=size-1.0;
                        }
                        else if(size==0.5)
                        {
                        ret=(ret*(base%den))%den;
                        break;
                        }
                        else
                        {
                                break;
                        }
                }
        return ret;
        }
}
```

Figure 4. CUDA kernel code

```
__kernel void RSA( __global const int * num,__global int *key,
                   global int *den,__global int * result)
{
 int i = get_global_id(0);   //current thread
 int n = get_global_size(0); //input size
 int iData = num[i];
 unsigned long int r=1;
 unsigned long int *ret=&r;
 if(i<3)          // no. of elements to process
 {
        mod(num[i],*key,*den,*ret);
        result[i] = *ret;
 }
}

__kernel void mod(int base,int exponent,int den, int ret)
{
        unsigned int a=(base%den)*(base%den);
        float size=(float)exponent/2;
        if(exponent==0)
        {
                ret=base%den;
        }
        else
        {
                while(1)
                {
                        if(size>0.5f)
                        {
                                ret=(ret*a)%den;
                                size=size-1.0f;
                        }
                        else if(size==0.5f)
                        {
                        ret=(ret*(base%den))%den;
                        break;
                        }
                        else
                        {
                                break;
                        }
                }
        }
}
```

Figure 5. OpenCL kernel code

6. VERIFICATION

In this section we setup the test environment and design two tests. At the first test, we develop a program running in traditional mode (only use CPU for computing) , NVIDIA CUDA-C and OpenCL framework for small prime numbers. Comparison is done between the (CPU-CUDA) , (CPU-OpenCL) cases and speed up is calculated. And at the second test, we use NVIDIA CUDA and OpenCL framework to run the RSA algorithm for large prime numbers in multiple-thread mode.

6.1 Test Environment

The code has been tested for:

- Values of messages between 0 and 800 which can accommodate the complete ASCII table;
- 8 bit key value.

The computer we used for testing has an Intel(R) Core(TM) i3-2370M 2.4GHZ CPU, 4 GB RAM, Windows 7OS and a Nvidia GeForce GT 630M with 512MB memory, and a 2GHZ DDR2 memory. At the first stage, we use Visual Studio 2010 for developing and testing the traditional RSA algorithm using C language, NVIDIA CUDA-C and OpenCL for small prime numbers. Series of input data used for testing and the result will be showed later.

At the second stage, we again use Visual Studio 2010 for developing and testing parallelized RSA developed using NVIDIA CUDA v5.5 and OpenCL for large prime numbers. After that the results of stage one are analysed for calculating the respective speedup and results of stage two are analysed for performance check between the NVIDIA CUDA-C and OpenCL.

6.2 Results

In this part, we show the experimental results for CUDA RSA ,OpenCL RSA and CPU RSA for small value of n.

Table 1. Comparison of CPU RSA and CUDA RSA for small prime numbers i.e (n=131*137)

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

Table 2. Comparison of CPU RSA and OpenCL RSA for small prime numbers i.e (n=131*137)

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

Table 1 and Table 2 shows the relationship between the amount of data inputting to the RSA algorithm and the execution times (in seconds) in traditional mode and multiple thread mode. The first column shows the number of the data input to the algorithm, and the second column shows the number of blocks/work-groups used to process the data input. In the above table 64 threads/work-items per block/work-group are used to execute RSA. The execution time is calculated in seconds. In the last two columns speed up is calculated between (CPU-CUDA) and (CPU-OpenCL). Above results are calculated by making average of the results so taken 20 times to have final more accurate and precise results.

From Table 1 and Table 2, we can see the relationship between the execution time in seconds and the input data amount (data in bytes) is linear for certain amount of input. When we use 256 data size to execute the RSA algorithm, the execution time is very short as compared to traditional mode which is clearly proved in the above section where the comparison is made for CPU RSA and CUDA RSA and OpenCL RSA for small prime numbers and hence for small value of n. So we can say when the data size increases, the running time will be significantly reduced depending upon the number of threads used.

The enhancement of the execution performance and speedup using C, CUDA and OpenCL framework can be visually demonstrated by Figure 5 and Figure 6

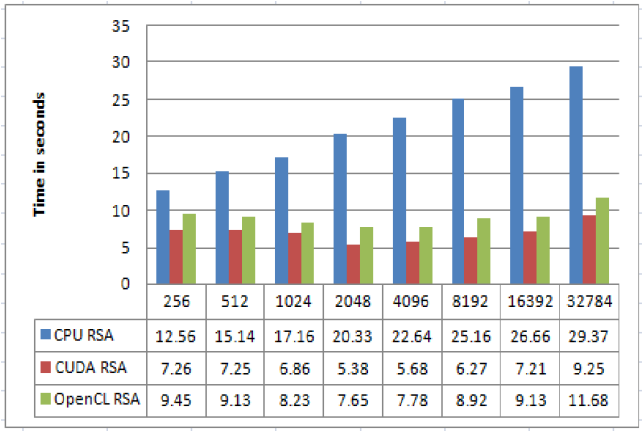

Figure 5. Graph showing effect of data input on CPU RSA and GPU RSA

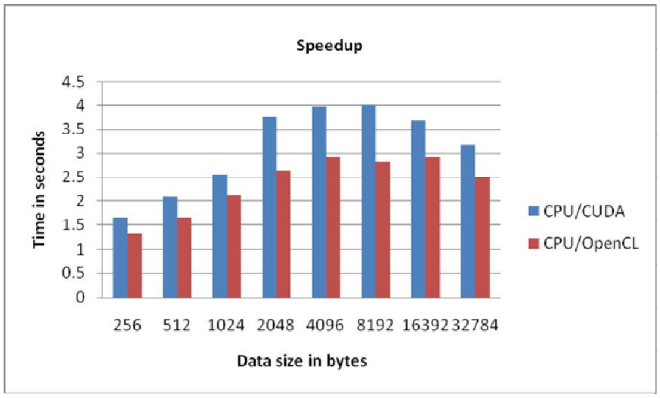

Figure 6. Graph showing speedup of data input on CPU RSA and GPU RSA

6.3 OpenCL and CUDA RSA for Large Prime Numbers

In this part, we show the experimental results for CUDA RSA and OpenCL RSA for large prime numbers or large value of n.

Table 3 CUDA RSA and OpenCL RSA for large values of n(1009*509)

Data size(bytes)	No. of work-groups/blocks	No. of work-items /threads	*CUDA RSA(T_1)*	*OpenCL RSA(T_2)*
256	8	32	6.08	8.57
512	16	32	6.52	8.83
1024	32	32	6.69	9.01
2048	64	32	5.53	7.62
4096	128	32	6.58	8.88
8192	256	32	6.66	8.97
16392	512	32	7.81	9.86
32784	1024	32	8.76	11.05

From the above table we find that execution time of CUDA is less than that of OpenCL. GPU setting for kernel execution along with API's for CUDA and OpenCL vary. Such as, the context creation, data copying, kernel mapping conventions are different in both. All such factors combined up to give more execution time in OpenCL as compared to CUDA. Furthermore, we also find that when the data size increases from 1024 to 8192, the execution time of 7168 threads almost no increase, which just proves our point of view, the more the data size is, the higher the parallelism of the algorithm, and the shorter the time spent. Execution time varies according the number of threads/work-items and number of blocks /work-groups used for data input. In the above table threads/work-items per block/work-group are constant i.e we use 32 threads/work-items per block/work-group and number of blocks/work-groups used are adjusted according to the data input.

The comparison of the execution performance of data input in bytes using the large value of prime numbers (n=1009 * 509) and hence large value of n on NVIDIA CUDA and OpenCL framework can be visually demonstrated by Figure 6.

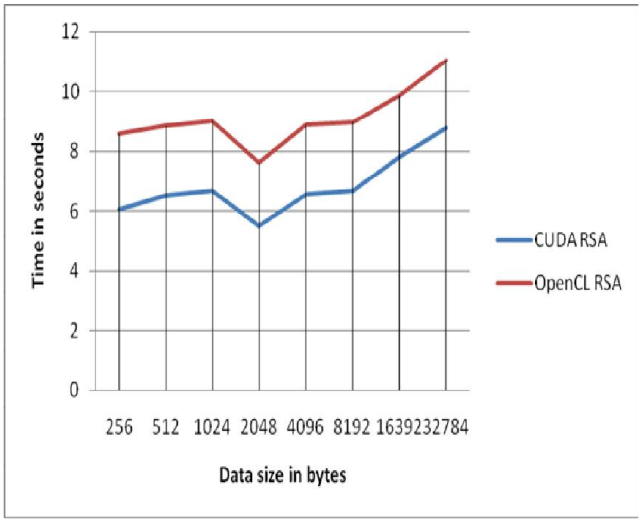

Figure 6. Performance Comparison of CUDA RSA and OpenCL RSA for large values of n (1009*509)

7. CONCLUSIONS

In this paper, we presented our experience of porting RSA algorithm from NVIDIA CUDA to OpenCL architecture. We analyzed the parallel RSA algorithm on two different frameworks. The bottleneck for RSA algorithm lies in the data size and key size i.e the use of large prime numbers. The use of small prime numbers make RSA vulnerable and the use of large prime numbers for calculating n makes it slower as computation expense increases. This paper design a method to computer the data bits parallel using the threads/work-items respectively based on CUDA and OpenCL.

The second issue we deal in this paper is the performance comparison of CUDA and OpenCL using RSA algorithm. Both programming interfaces offer same functionality. Minimal changes are needed to port one kernel code to another. Code-writing is involved in porting the GPU-related sources, such as GPU setup and data transfer code. In our tests, CUDA performed better than OpenCL. So, CUDA is a better choice for applications demanding high performance. Otherwise the choice between the two can be made depending upon the various factors including prior familiarity with the system, target GPU hardware, availability of development tools and portability.

8. REFERENCES

[1] Rivest,R.L.,Shamir,A., and Adleman,L. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120{126, 1978.

[2] Owens,J., Luebke,D., Govindaraju,N. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26(1): 80{113 , March 2007.

[3] Heninger,N., Durumeric,Z., Wustrow,E., and Halderman,J.A. Mining your Ps and Qs: detection of widespread weak keys in network devices. In Proceedings of the 21st USENIX conference on Security symposium, pages 205{220. USENIX Association, 2012 .

[4] NVIDIA|CUDA documents |Programming Guide |CUDA.DOI=http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[5] The OpenCL specification,version:1.0. DOI= http://www.khronos.orf/registry/cl/specs/opencl-1.0.pdf, 2008