Multiple Precision Integer GCD Performance Analysis on Parallel Architectures

*Jitendra V. Tembhurne, S. R. Sathe

***Abstract*—The multiple precision Greatest Common Divisor (GCD) computation is a key component in the field of computer algebra and in many cryptography applications such as public-key cryptography; in the setup phase of RSA, and in the implementation of point operations (addition, subtraction, multiplication etc.) on Elliptic Curve Cryptography (ECC), for factorization attacks and in the statistical testing of pseudo random number generators. Multiple precision computations on the CPUs are computationally expensive, but substantial performance is achieved by using OpenMP with the GNU Multiple Precision Library (GMP) and Compute Unified Device Architecture (CUDA). In this paper, we have explored the computational power of NVIDIA Graphics Processing Units (GPUs). We have implemented the Multiple Precision Integer (MPI) operations on the GPU using C CUDA. We implemented the several GCD computation algorithms using MPIs operations. Implementation results based on the CPU and GPU indicate that a significant speedup is achieved by the GPU using CUDA and OpenMP with GMP, as compared with the single core CPU implementation.**

***Index Terms*— CUDA, GCD, GPU, OpenMP**

I. Introduction

In today's High Performance Computing (HPC) era, Graphics Processing Units (GPUs) are becoming increasingly common to use in general purpose computations, which contain a set of streaming processors. General Purpose GPUs (GPGPUs) are widely used for parallelization of different categories of applications such as fluid dynamics, ray tracing, bioinformatics, and gaming etc. GPUs are best suited for high throughput computing applications that handle large amount of data and we can exploit the Single Instruction Multiple Data (SIMD) architecture of the GPU for data-parallel applications. We have to exploit the power of computing facilities available in the form of GPUs having many-cores for the computation of complex problems and the more time consuming problems which are often used in mathematical computation. In this work, we concentrate on the computation of a big number integer GCD. GCD computation is often used in many mathematical and cryptography applications such as primality testing, to find the multiplicative inverse, to find relatively prime numbers, for factorization attacks [1], and in public key cryptography algorithms such as RSA and ECC [2], [3].

The tremendous computation power of the GPUs is provided by NVIDIA [4] and the GPU works on the principle of Single Instruction Multiple Data (SIMD). NVIDIA's GPU such as Quadro FX 3800 has 192 computing cores with 462.3 GFLOPS of computational horsepower. NVIDIA CUDA support the heterogeneous computation model, where the CPU and the GPU both work together to accomplish every computation. On the CPU, sequential part of the program is executed while, on the GPU parallel part of the program is executed. Host (CPU) initiates the program execution by allocating the memory on host and device (GPU) and calling the kernel on a device. At the time of kernel call, GPU generates multiple threads based on the on-device kernel code, which runs concurrently along the threads. Moreover, all the threads have their own local memory and access to the shared and global memory for efficient data handling.

We have been motivated by the GPU's massively parallel computation which is achieved by a large number of cores available on the GPU. It is easy to achieve high performance on the GPU for the computation of multiple precision arithmetic operations as well as MPI GCD computations. The research work addresses the following issues:

1) Design and implementation of parallel Multiple Precision Integer (MPI) arithmetic on the GPU using CUDA architecture.
2) Design and implementation of parallel MPI GCD algorithms on CUDA.
3) Design and implementation of parallel MPI GCD algorithms for CPU using OpenMP and GMP library.
4) Performance analysis of parallel GCD algorithm on the CPU and GPU.

The outline of the rest of the paper is as follows. Section II is dedicated to the review of previous work carried by the different researchers. Section III describes the OpenMP and CUDA architecture for parallel computation. In section IV, we have demonstrated the implementation of multiple precision operations on integers and the computation of GCD for a big number on the CPU and GPU. Implementation results based on the CPU and GPU and comprehensive experimental analysis are discussed in section V and in section VI conclusions drawn from the experimental results are presented.

II. Existing Works

Parallel implementation of any algorithm requires many processors to run concurrently. To execute an algorithm concurrently, we have to form an independent computation

Manuscript revised on 20th August, 2015.

*Jitendra V. Tembhurne is with the Visvesvaraya National Institute of Technology, Nagpur- 440010, Maharashtra, India (Corresponding author, phone: +91-712-2801793; e-mail: dt11cse077@cse.vnit.ac.in).

S. R. Sathe is with the Visvesvaraya National Institute of Technology, Nagpur-440010, Maharashtra, India (e-mail: srsathe@cse.vnit.ac.in).

step which will subsequently be performed. When the algorithmic steps are independent of data to be handled while computing than, the algorithm is suitable for the parallel implementation. In this paper, we are targeting to a parallel computation of GCD of big integers. In the past years, researchers were working on the implementation of the parallel GCD algorithm to achieve better time complexity of different models of computation such as EREW, CREW, and CRCW. The problem of computing GCD of two non-negative integers efficiently in parallel is one of the open problems in the theory of parallel computation.

A simple parallel implementation of GCD of two *n*-bit integers on the Concurrent Read Concurrent Write (CRCW) computation model was proposed in [5]. The authors claim the parallel run-time of the algorithm in terms of bit operation is O (n /log n) by the use of $n^{1+\epsilon}$ processors, where ϵ is any positive constant. This implementation was the improvement of the algorithm proposed by KMR in [6], which is the first sub-linear GCD algorithm, runs on O($n$loglog$n$/ log$n$) times using the same CRCW model. The extended GCD algorithm proposed in [2], [3] and [7] is very useful for data dependence test for any given code block to identify the data dependencies, if any. In 1994 Sorenson at el. [9], suggested the parallel extended GCD algorithm implementation on a CRCW SM MIMD (Multiple Instruction Multiple Data) model using O(n) processors and proposed to exploit the use of CRCW model. The aim of this paper was to speedup the process of checking data dependency in the given code block. Further, the GCD computation of MPIs using GMP for shared memory architecture was proposed in [8]. The accelerated integer GCD algorithm by Sorenson [9] is derived from the right-shift *k*-ary GCD algorithm and has been shown to be very effective for computing the GCD of moderate to large sized integer numbers in a sequential manner. Sedjelmaci [12], illustrated the parallel implementation of Schönhage's algorithm [10] on distributed memory architectures. This algorithm uses the half-GCD algorithm, which has two MPIs. For fast computation author has exploited the parallel Karatsuba's multiplication algorithm [11].

The main difficulty in the Euclid's GCD algorithm is the expensive cost of the multiple precision divisions. In [12], [13] a Lehmer-Euclid GCD algorithm was proposed, where the MPIs reduce to a single precision integer after working with the leading bits of integers. Subsequently, extended Euclidean algorithm is applied to calculate the GCD from reduced single precision integers.

Recently, the researchers exploited the use of parallel computer provided by the vendors like Intel, NVIDIA and AMD. With recent advances in parallel computing hardware, GPUs are coined as the general purpose programming model for performing faster computations. Moreover, it is not difficult to develop non-graphics applications using GPUs. NVIDIA provides a general purpose parallel programming model known as CUDA [4], [14], which uses the C or C++ programming languages for the development of general purpose applications. CUDA as a parallel computing model is gaining its place for speeding up a large number of applications such as [15], [16], [17], [18] and [19]. Due to its extensive popularity, we have chosen the CUDA for MPI GCD implementations.

The MPI implementation of different arithmetic operations on the GPU has been demonstrated in [20], [21]. The author claims that significant speedup can be achieved by the GPUs as compared with the GNU Multiple Precision library on the CPUs. In addition, the performance of the multiple precision modular multiplications has been improved by 20% using MPI based computations. We would like to point out that, while implementing the multiple precision GCD, all the required arithmetic operations are needed to be implemented on the GPU using CUDA.

In [22], the high throughput, multiple precision Binary GCD algorithm on CUDA architecture has been proposed. The fixed bit-length of integers were chosen to be 1024-bits. This algorithm computes the many GCDs at a time and the measure of speedup is calculated not on single GCD computation but on the many GCD computations. The author claims that the proposed GPU algorithm runs 11.3 times faster than the CPU version of the algorithm. Another implementation of polynomial GCD computation on the GPU using Maple 13 has been proposed in [23]. In this work, the author has developed an algorithm to compute a GCD of univariate polynomials with integer coefficients on the GPU and conferred the significant speedup over CPU based GCD algorithm.

The EREW PRAM model based parallel randomized algorithm to compute GCD of two n bits integers has been proposed which requires the computation time of $O(n \log\log n / \log n)$ [24]. On the contrary, a new parallel GCD algorithm has been proposed by Sedjelmaci [25] to compute the GCD of $O(n)$ bits of n integers in time complexity of $O(n/\log n)$ by using $(n^{2+\epsilon})$ processors, where $\epsilon > 0$ for a CRCW PRAM model of computation. The similar Binary GCD [22] based parallel big integers GCD computation on the shared memory model has been implemented for large integers of bits length ranging from 1024-bits to 4096-bits [26]. The speedup achieved was measured on the Intel Xeon Phi machine with 240 threads as compared with the single-core CPU.

III. Overview of OpenMP and CUDA

A. OpenMP

OpenMP (**Open M**ulti-**P**rocessing) is a directive based language for expressing parallelism on the shared memory multiprocessor systems [27], [28]. OpenMP is a multithreaded platform implementation and a method for parallel program design whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided amongst them. A team of threads runs in parallel, with the set of directives and runtime environment variables allocated to different processors. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program [27], [28], [29].

By default, each thread performs the execution of parallel section of the code independently. Work-sharing constructs like, *#pragma omp for [clause]*, *#pragma omp sections [clause]* and *#pragma omp single [clauses]* can be utilized

to divide a task amongst the threads so that each thread executes its allocated part of the code. In this way, task parallelism and data parallelism both can be achieved using OpenMP. OpenMP is available for various platforms and the list of different compilers can be found at [30].

B. CUDA

NVIDIA introduced CUDA, a general purpose parallel computing architecture, with a new parallel programming model and instruction set architecture [4], [14] which leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C or C++ as a high level programming language. Moreover, it also supports other programming languages, application programming interfaces (API), or directives based approaches such as FORTRAN, OpenCL, OpenACC etc. The GPU as general purpose programming has evolved into a many core processor and massively parallel architecture with huge computational horsepower and very high memory bandwidth [31]. The reason behind the discrepancy in floating point capability between the CPU and the GPU is that the GPU is specialized for compute intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

The architecture of the GPU supports CUDA enabled framework [4], which is organized into an array of highly threaded Streaming Multiprocessors (SMs). The number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. In addition, each SM has a number of Streaming Processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 32 gigabytes of graphics double data rate (GDDR) DRAM, referred to as global memory. For graphics applications, they hold, texture information, and video images for three-dimensional rendering, however, for computing their function as a very high bandwidth, off-chip memory. For massively parallel applications, higher bandwidth makes it up for the long latency time. The GPU follows the SIMD based execution model. The execution of a typical CUDA program is started by the host (CPU). When a kernel function is called or invoked, the complete execution is moved to a device (GPU), where a large number of threads within the thread-blocks are generated to take the advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a *block of threads* or *thread block*. The collection of *thread blocks* is called a *grid*. Threads running on the GPU are grouped into wraps consisting of 32 threads. When all the threads of a kernel complete their execution, the grids terminate and the remaining part of execution continues on the host until another kernel is launched. Each thread contains own memories such as, register memory and local memory. The thread block contains shared memory for read and write, and the grid has global memory for read and write and in addition, it contains read only memory called as constant memory.

IV. Multiple-precision Integer Arithmetic

A. Multiple Precision Integer (MPI) Arithmetic on GPU

In this section, we present an implementation of different arithmetic functions of MPIs on the GPU using CUDA. In multiple precision arithmetic functions, we have chosen the decimal digit (i.e. base of the number is 10) but the same can be used for different radix representation of the number like binary etc. Non-negative integers can be represented in various ways, the most common form is base 10. For example, $a = 243$ base 10 means $a = 2\times10^2 + 4\times10^1 + 3\times10^0$. The algorithms implemented on the GPU for multiple precision arithmetic operations are Addition (Add), Subtraction (Sub), Multiplication (Mul) and Division (Div) as mentioned in pseudo code from Algorithm 1 to Algorithm 4 [2], [7], [32] which are the sequential MPI algorithms. Apart from these basic arithmetic integer operations, we have implemented the MPI comparison, left-shift, and right-shift etc. and other supporting operations.

Algorithm 1: Multiple-precision Addition

Input: positive integers x and y, each having n base b digits.
Output: $x + y = (w_n w_{n-1} \ldots w_1 w_0)_b$ in radix b representation.
1. $c \leftarrow 0$ //c is the carry
2. For i from 0 to $n-1$ do
   2.1 $w_i \leftarrow (x_i + y_i + c) \% b$
   2.2 If ($(x_i + y_i + c) < b$) then $c \leftarrow 0$; Else $c \leftarrow 1$
3. $w_n \leftarrow c$
4. *return* $(w_n w_{n-1} \ldots w_1 w_0)_b$

Algorithm 2: Multiple-precision Subtraction

Input: positive integers x and y, each having n base b digits, with $x \geq y$.
Output: $x - y = (w_n w_{n-1} \ldots w_1 w_0)_b$ in radix b representation.
1. $c \leftarrow 0$
2. For i from 0 to $n-1$ do
   2.1 $w_i \leftarrow (x_i - y_i + c) \% b$
   2.2 If ($(x_i - y_i + c) \geq 0$) then $c \leftarrow 0$; Else $c \leftarrow -1$
3. *return* $(w_n w_{n-1} \ldots w_1 w_0)_b$

Algorithm 3: Multiple-precision Multiplication

Input: positive integers x and y having n and m base b digits, respectively.
Output: $x \times y = (w_{n+m+1} \ldots w_1 w_0)\ b$ in radix b representation.
1. For i from 0 to $(n + m + 1)$ do: $w_i \leftarrow 0$
2. For i from 0 to m do
   2.1 $c \leftarrow 0$
   2.2 For j from 0 to n do
      $temp \leftarrow w_{i+j} + x_j \times y_i + c$
      $w_{i+j} \leftarrow temp \% b$, $c \leftarrow temp / b$
   2.3 $w_{i+n+1} \leftarrow c$
3. *return* $(w_{n+m+1} \ldots w_1 w_0)_b$

Algorithm 4: Multiple-precision Division

Input: positive integers $x = (x_n \ldots x_1 x_0)_b$, $y = (y_m \ldots y_1 y_0)_b$ with $n \geq m \geq 1$, $y_m \neq 0$.
Output: the quotient $q = (q_{n-m} \ldots q_1 q_0)_b$ and remainder $r = (r_m \ldots r_1 r_0)_b$ such that $x = q.y + r$, $0 \leq r < y$.
1. For j from 0 to $(n - m)$ do: $q_j \leftarrow 0$
2. While $(x \geq y.b^{n-m})$
   2.1 $q_{n-m} \leftarrow q_{n-m} + 1$
   2.2 $x \leftarrow x - y.b^{n-m}$
3. For i from n down to $(m + 1)$ do
   3.1 If $(x_i = y_m)$ then $q_{i-m-1} \leftarrow b - 1$
      Else $q_{i-m-1} \leftarrow$ floor $(x_i.b + x_{i-1}) / y_m)$
   3.2 While $(q_{i-m-1} (y_m b + y_{m-1}) > x_i b^2 + x_{i-1} b + x_{i-2})$
      $q_{i-m-1} \leftarrow q_{i-m-1} - 1$
   3.3 $x \leftarrow x - q_{i-m-1}\, y\, b^{i-m-1}$
   3.4 If $(x < 0)$ then
      $x \leftarrow x + y\, b^{i-m-1}$
      $q_{i-m-1} \leftarrow q_{i-m-1} - 1$
4. $r \leftarrow x$
5. *return* (q, r)

A data structure used to represent the multiple precision integers is shown in Listing I. The MPI is stored in a 1D unsigned character array in the form of individual decimal digit.

LISTING I
DATA STRUCTURE OF MULTIPLE-PRECISION INTEGER

```
struct MPInteger {
 unsigned char Digit[MAX_DIGITS_CAPACITY];
 int Length;  char Sign;
};
```

The complexity of *n*-digit MPI operations such as addition and subtraction is *O(n)*. But, due to data dependency at carries, borrows generation of these MPI operations is the biggest hurdle for parallel implementation. In Algorithm 1, *x* and *y* arrays stored input data ranging from 0 to *b-1*, where *b* is the base of the MPI number and *c* is to store the carries. At step 2.1 and 2.2, the value of w_i depends on the updated value of *c*. Since, we cannot parallelize the algorithm directly. Algorithm 5 shows the parallelization of Algorithm 1 by avoiding the propagation of carries. So, the MPI addition is performed in Step 2 without releasing the carries. In step 3.1 carries are computed, and in step 3.2 carries are used to calculate the sum of the corresponding digits of MPIs. Here, we have to store all carries in the separate array. A digit normalization is used until all carries are generated. This process of normalization has been applied fewer times. Since, the carry propagation iterates, for example, 9…99 + 1…11, the carry-skip approach [33] has been adopted.

In Algorithm 6, carry skipping approach is applied from steps 2.1 to 2.4 and from steps 2.5 to 2.7 respectively. Hence, MPI Add, Sub and Mul can be implemented using carry skip approach.

The MPI multiplication algorithm implementation on the GPU is presented in Algorithm 7. This algorithm uses the MPI addition and Carry-Skip algorithm to compute the MPI multiplication. The normalization of MPI multiplication is similar to carry computation in MPI Add, Sub and Mul. In addition, parallel implementation of the MPI division comprises of the implementation of MPI Add, Sub and Mul. Since, the time consumed by the MPI division is significantly longer than the MPI Add, Sub and Mul operations. The steps 1 to 3 in Algorithm 4 can be easily parallelized with the help of MPI Add, Sub and Mul.

To implement "divide by 2" operation for the GCD computation algorithms, we have used the binary shift right method in parallel. The steps of the algorithm are as follows;

- Partitioned the MPI into *n* 32-bit integer such as MPI_1, MPI_2,… MPI_n, where *n* is the number of partitions of MPI. Represents MPI_1, MPI_2,… MPI_n in the vector after (MPI_i & *1*) operation.
- Copy all Least Significant Bit (LSB) of MPI_i into *LSB[1..n]* array.
- Perform $MPI_i \leftarrow (MPI_i >> 1)$
- Copy *LSB[2..n]* at the Most Significant Bit (MSB) of MPI_i from MPI_1, MPI_2,… MPI_{n-1} i.e. copy at 31st, 63rd and so on position of MPI.
- Combined all MPI_1, MPI_2,… MPI_n to get required MPI.

However, the parallel left-shift is also implemented in a similar way, the way a right-shift is implemented, except the bit shift is towards the left direction.

Algorithm 5: Parallel MPI Addition (ParMPIAdd)

Input: positive integers *x* and *y*, each having *n* base *b* digits.
Output: $x + y = (w_n w_{n-1} \ldots w_1 w_0)_b$ in radix *b* representation.
1. For$_{parallel}$ *i* from 0 to *n-1* do
   $w_i \leftarrow x_i + y_i$
3. While ($\max(w_i)_{0 \le i < n} \ge b$)
   3.1 $c_0 \leftarrow 0$
   3.2 For$_{parallel}$ *i* from 0 to *n-1* do
      $c_{i+1} \leftarrow w_i / b$
   3.3 For$_{parallel}$ *i* from *0* to *n-1*
      $w_i \leftarrow (w_i \% b) + c_i$
   3.4 $w_n \leftarrow c_n$
5. *return* $(w_n w_{n-1} \ldots w_1 w_0)_b$

Algorithm 6: Parallel Carry-Skip Algorithm (ParCK)

Input: positive integers *x* and *y*, each having *n* base *b* digits.
Output: $x + y = (w_n w_{n-1} \ldots w_1 w_0)_b$ in radix *b* representation.
1. Call **ParMPIAdd(x_i, y_i)**
2. While ($\max(w_i)_{0 \le i < n-1} = b$)
   2.1 For$_{parallel}$ *i* from 0 to *n*-1 do
      2.1.1 If ($w_i = b$) then $h_i \leftarrow i$; Else $h_i \leftarrow n - 1$
   2.2 $k \leftarrow \min(h_i)_{0 \le i < n}$
   2.3 For$_{parallel}$ *i* from *k*+1 to *n*-1 do
      2.3.1 If ($w_i < b$-1) then $h_i \leftarrow i$; Else $h_i \leftarrow n - 1$
   2.4 $z \leftarrow \min(h_i)_{k+1 \le i < n}$
   2.5 $w_k \leftarrow w_k - b$
   2.6 For$_{parallel}$ *i* from *k*+1 to *z* -1 do
      2.6.1 $w_i \leftarrow w_i - (b - 1)$
   2.7 $w_z \leftarrow w_z + 1$
3. *return* $(w_n w_{n-1} \ldots w_1 w_0)_b$

Algorithm 7: Parallel MPI Multiplication (ParMPIMul)

Input: positive integers *x* and *y* having *n* and *m* base *b* digits, respectively.
Output: $x \times y = (w_{n+m+1} \ldots w_1 w_0)_b$ in radix *b* representation.
1. For$_{parallel}$ *i* from 0 to ($n + m + 1$) do
   $w_i \leftarrow 0$; $c_i \leftarrow 0$
2. For$_{parallel}$ *i* from 0 to *m* do
   2.1 For *j* from 0 to *n* do
      $w_{i+j} \leftarrow w_{i+j} + x_j \times y_i$
3. While ($\max(w_i)_{0 \le i < n+m+1} \ge b$)
   3.1 For$_{parallel}$ *i* from 0 to ($n + m + 1 - 1$) do
      $c_{i+1} \leftarrow w_i / b$
   3.2 For$_{parallel}$ *i* from 0 to ($n + m + 1 - 1$) do
      $w_i \leftarrow (w_i \% b) + c_i$
4. $w_{n+m+1} \leftarrow c_{n+m+1}$
5. *return* $(w_{n+m+1} \ldots w_1 w_0)_b$

V. GCD'S IMPLEMENTATION

A. GCD Algorithms

The GCD algorithms are considered for parallel implementation and performance measurement uses the simplest arithmetic operations as mentioned in section IV. The pseudo codes for MIPs GCD are indicated in Algorithm 8, 9, 10 and, 11 respectively.

Algorithm 8: Extended Euclidean Algorithm (EEA)

Input: Non-negative integers *a, b* such that $a \ge b > 0$.
Output: GCD $(a, b) = d$, and integers x, y satisfying $ax + by = d$.
1. If ($b = 0$) then set $d \leftarrow a$; $x \leftarrow 1$; $y \leftarrow 0$; *return* (d, x, y);
2. Set $x_2 \leftarrow 1$; $x_1 \leftarrow 0$; $y_2 \leftarrow 0$; $y_1 \leftarrow 1$;
3. While ($b > 0$)
   3.1 $q \leftarrow a/ b$; $r \leftarrow a - qb$; $x \leftarrow x_2 - qx_1$; $y \leftarrow y_2 - qy_1$;
   3.2 $a \leftarrow b$; $b \leftarrow r$; $x_2 \leftarrow x_1$; $x_1 \leftarrow x$; $y_2 \leftarrow y_1$; $y_1 \leftarrow y$;
4. Set $d \leftarrow a$; $x \leftarrow x_2$; $y \leftarrow y_2$; *return* (d, x, y);

Algorithm 9: Binary GCD (Bin)

Input: Non-negative integers *a, b* such that $a \ge b > 0$.

Output: GCD (a, b).
1. $g \leftarrow 1$;
2. While (a and b even) { $a \leftarrow a/2$; $b \leftarrow b/2$; $g \leftarrow 2 \times g$; }
3. While (a is even) $a \leftarrow a/2$;
4. While (b is even) $b \leftarrow b/2$; // **now a and b are both odd**
5. While ($a \neq b$)
   5.1 (a, b) $\leftarrow$ ($|a - b|$, min(a, b));
   5.2 $a \leftarrow a/2^{v(a)}$; // **$v(a)$ is the 2-valuation of a**
6. *return* ($g \times a$);

Algorithm 10: Extended Binary GCD (EBin)

Input: Non-negative integers a, b such that $a \geq b > 0$.
Output: GCD (a, b) = v, and integers x, y satisfying $ax + by = v$.
1. $g \leftarrow 1$;
2. While (a and b is both even) $a \leftarrow a/2$; $b \leftarrow b/2$; g $\leftarrow$ 2g;
   2.1 $u \leftarrow a$; $v \leftarrow b$; $A \leftarrow 1$; $B \leftarrow 0$; $C \leftarrow 0$; $D \leftarrow 1$;
3. While (u is even) $u \leftarrow u/2$;
4. If ($A \equiv B \equiv 0$ (mod 2)) then $A \leftarrow A/2$; $B \leftarrow B/2$;
   Else $A \leftarrow (A + y)/2$; $B \leftarrow (B - x)/2$;
5. While (v is even)
   5.1 $v \leftarrow v/2$;
   5.2 If ($C \equiv D \equiv 0$ (mod 2)) then $C \leftarrow C/2$; $D \leftarrow D/2$;
      Else $C \leftarrow (C + y)/2$; $D \leftarrow (D - x)/2$;
6. If ($u \geq v$) then $u \leftarrow u - v$; $A \leftarrow A - C$; $B \leftarrow B - D$;
   Else $v \leftarrow v - u$; $C \leftarrow C - A$; $D \leftarrow D - B$;
7. If ($u = 0$) then $x \leftarrow C$; $y \leftarrow D$; *return*$(x, y, g \times v)$;
   Else go to step 4

Algorithm 11: Mixed Binary Euclid (MBE)

Input: Integers a, b such that $a \geq b > 1$, with b is odd.
Output: GCD (a, b).
1. While ($b > 1$)
1.1 $r \leftarrow a \bmod b$; $s \leftarrow b$ r;
1.2 While ($r > 0$ *and* $r \bmod 2 = 0$) $r \leftarrow r/2$;
1.3 While ($s > 0$ *and* $s \bmod 2 = 0$) $s \leftarrow s/2$;
      If ($s < r$) then $u \leftarrow r$; $v \leftarrow s$;
      Else $u \leftarrow s$; $v \leftarrow r$;
2. If ($b = 1$) then *return* 1; Else *return* u;

B. GCD Implementation on CPU

To implement GCD algorithms on the CPU is straight forward for small integers, like 32-bit or 64-bit computation, which is the maximum precision required and is supported by all the available CPU architectures. But, the computation of the GCDs for bigger integers that are usually used in cryptographic applications is really challenging. In public-key cryptography algorithms such as RSA, we need the large key values such as 1024-bits and more for secure encryption of confidential data. So, in order to handle big integers we have used the GMP [34, 35]. This is one of the fastest libraries to carry operations on big integers on the CPU.

The steps to implement GCD computation are as follows;

1. Generating two sets of MPIs before starting the GCD computation. We have randomly generated the integers by implementing two random number generation algorithms such as Multiply-with carry algorithm and Linear Congruential Generator (LCG) algorithm [36].

2. Implementing the GCD algorithm by using random numbers generated in step 1.

The steps shown above are utilized to implement the sequential GCD computation. But, our target is to implement a parallel variant of GCD. To develop parallel implementation on the CPU, we used the OpenMP, which work on a shared memory architecture. In OpenMP, all the created threads shared global memory for the successive computation. Here, we have to decide the shared data, private data, number of threads, and scheduling scheme to schedule the parallel tasks with synchronization amongst the running threads. All these can be achieved by using the OpenMP directives. OpenMP eases the implementation by providing the large set of compile time directives.

The Listing II shows the OpenMP implementation of parallel GCD computation on the CPU. The implementation uses the GMP library for MPIs to handle and perform many GCD computations in parallel by all the available cores on the CPU. The *for-loop* at line 9 specifies N, the number of GCD computing integers. Line 17 shows the application of OpenMP directives to compute the GCDs in parallel by all threads. Each thread will perform its own GCD computation without intervening the work of other threads. The proposed OpenMP implementation is a direct application of OpenMP directives to compute GCD without any parallel algorithm optimization on arithmetic operations such as Add, Sub, Mul and Div. But, the same has been considered for the GPU implementation. All arithmetic operations on the CPU used the default GMP's multiple precision library functions.

LISTING II
GCD IMPLEMENTATION USING OPENMP AND GMP

```
1  // MAIN PROGRAM FOR GCD COMPUTATION
2  main(){
3     mpz_t *gcd, *u, *v;
4     //MEMORY ALLOCATION FOR MPZ
5     u = malloc(sizeof(mpz_t) * N);
6     v = malloc(sizeof(mpz_t) * N);
7     gcd = malloc(sizeof(mpz_t) * N);
8     //INITIALIZATION OF MPZ
9     for(i=0; i<N; i++){
10       mpz_init(u[i]); mpz_init(v[i]);
11       mpz_init(gcd[i]);
12    }
13    //CALLING LCG ALGORITHM
14    RandLCG(); RandLCG();
15
16 //GCD COMPUTATION PARALLEL SEGMENT
17 #pragma omp parallel for shared(u, v, gcd,
18 N) private(i, result) num_threads(threads)
19 default(none) schedule(dynamic,chunck_size)
20    for(i=0; i<N; i++){
21       EEAGcd(result, u[i], v[i]);
22       mpz_set (gcd[i], result);
23    }
24    free(u);free(v);free(gcd);
   }
```

C. GCD Implementation on GPU

For the implementation of parallel GCD computation on the GPU, we have selected the N MPIs with variable bit length. We have chosen N as the multiple of 8, as we know the *block* dimension is also multiple of 8. Particularly , we can efficiently use the *block* dimension to carry the group of threads. Moreover, it is known from the specification of the GPU CUDA device, the computation is performed by the *warp*. However, for handling MPIs, we used the more optimized algorithm implementation discussed in section IV.

The optimization applied to all these implementation's performance improvements are listed as; 1) The algorithm listed in section IV and V uses some constant data frequently at the time of computation. Since, the constant data are stored in constant memory and access to these constant data via cache memory, this improves the computational efficiency and reading efficiency. 2) During implementation, we have observed that the use of temporary variables is frequent in the listed algorithms. So, rather than storing the temporary variables in global memory we have stored them in a shared memory to reduce reading latency on the GPU.

The parallel algorithm for Algorithm 5 is proposed in the form of Algorithm 12, with two 1D arrays handling *n* MPIs. The Algorithm 8 has the data dependencies which we have identified, but it is not possible to remove completely. The proposed algorithm is parallelized to handle many pairs of MPIs and compute the GCDs for these pairs in parallel. The same approach holds true for all the GCD algorithms. Hence, they have been also parallelized in a similar way as proposed in the Algorithm 12.

Algorithm 12: Parallel Extended Euclidean Algorithm (ParEEA)

Input: Non-negative integers $a[1..n]$, $b[1..n]$ such that $a[i] \geq b[i] > 0$. Where a, b are the arrays of n MPIs and i is the index referred to the individual MPI in the respective MPI array, a and b.

Output: GCD ($a[1..n]$, $b[1..n]$) = $gcd[1..n]$

1. If ($b_i = 0$) then /* do in parallel */
   $d_i \leftarrow a_i$; $x_i \leftarrow 1$; $y_i \leftarrow 0$; *return*;
2. Set $x2_i \leftarrow 1$; $x1_i \leftarrow 0$; $y2_i \leftarrow 0$; $y1_i \leftarrow 1$; /* do in parallel */
3. While ($b_i > 0$) /* do in parallel */
   3.1 $q_i \leftarrow a_i / b_i$; $r_i \leftarrow a_i - q_i b_i$; $x_i \leftarrow x2_i - q_i x1_i$; $y_i \leftarrow y2_i - q_i y1_i$;
   3.2 $a_i \leftarrow b_i$; $b_i \leftarrow r_i$; $x2_i \leftarrow x1_i$; $x1_i \leftarrow x_i$; $y2_i \leftarrow y1_i$; $y1_i \leftarrow y_i$;
4. Set $d_i \leftarrow a_i$; $x_i \leftarrow x2_i$; $y_i \leftarrow y2_i$;
5. *return*;

Listing III demonstrates the high level implementation of parallel GCD computation on the GPU wherein, we see the device function at line 1 and the kernel function at line 15, which will be launched on the GPU device with specified number of threads/block. The same kernel code will be executed by all the launched thread simultaneously on the GPU. For optimal and efficient memory usage, we have used the shared memory for fast access to data in the implementation. Moreover, we have investigated the data size, which can properly fit into the shared memory available on the GPU. The data type *ullint* in Listing III is corresponding to the *MPInteger* data structure specified in the Listing I. In device function *__device__ ullint EEA( )* all the temporary variables are stored in shared memory. Kernel stored the pairs of MPIs into the shared memory and performs the computation of GCDs. The GCD of a pair of MPIs is computed by the single thread, i.e. first GCD is computed by $thread_0$, second GCD is computed by $thread_1$, and so on. Hence, the kernel computes many GCD in parallel for many pairs of MPIs.

LISTING III
GCD IMPLEMENTATION ON GPU USING CUDA

```
1  // DEVICE FUNCTION TO COMPUTE GCD(a, b)
2  __device__ ullint EEAGCD(ullint *a,
3                           ullint *b){
4      ullint x, x1, y, y1, temp, quotient;
5      x = 0; x1 = 1; y = 1; y1 = 0;
6      while(*b != 0){
7        temp = *b;       quotient = *a / *b;
8        *b = *a % *b;    *a = temp; temp = x;
9        x = x1-quotient * x; x1=temp; temp=y;
10       y = y1-quotient * y; y1=temp;
11     }
12     return *a;
13   }
14 // GCD KERNEL
15 __global__ void kernel(ullint *a, ullint *b,
16                        ullint *c, int N){
17   //blockSize is the size of shared memory
18   __shared__ ullint aShared[blockSize];
19   __shared__ ullint bShared[blockSize];
20   int id=threadIdx.x+blockSize * blockDim.x;
21   int tid = threadIdx.x;
22   if(id >= N ) {return;}
23   aShared[tid] = a[id];
24   bShared[tid] = b[id];
25   __syncthreads();
26
27   if(id > 0)
28     c[id]=EEAGCD(aShared[tid],bShared[tid]);
   }
```

VI. RESULTS AND DISCUSSIONS

Experimental setup for the GCD computation based on the CPU and GPU is shown in Table I and Table II. The implementation of MPIs GCD on the CPU is straight forward. For the sequential implementation, we have used the GMP library, GMP 5.1.1. We have explored the shared memory architecture by using OpenMP for the parallel implementation of GCD algorithms and integrated with the GMP library for handling the multiple precision integers. The GMP library provides many functions for the implementation of multiple precision algorithms to handle integer or floating point operations. The OS used to be Linux OpenSUSE 12.1 with GCC compiler 4.5.1 and OpenMP 3.1 on the CPU side, and for compilation on the GPU we used NVIDIA Nsight Eclipse Edition 2.0 with CUDA 5.0 SDK.

NVIDIA CUDA device Quadro FX 3800 supports the 512 threads per thread block. So, to use the full power of the GPU CUDA device, we have selected the block size of 512 threads per block for the data values starting from the range 1048576 to 8388608.

The figures from Fig. 1 to Fig. 4, the *x-axis* represents the number of MPIs and the *y-axis* represents the computation time in milliseconds required to compute the GCDs. Here we have chosen the different group of data values of MPIs for the GCD computation. The groups of data values are 1048576, 2097152, 4194304, and 8388608 MPIs respectively. We have also selected the different MPI bit-length of each group as the 2048-bit, 4096-bit, and 8192-bit respectively. All the GCD algorithms presented in section V are tested on all the chosen group of data values with varying bit-length of MPIs both on the CPU and GPU.

TABLE I
THE GPU SPECIFICATION

NVIDIA CUDA specification	
CUDA Device	Quadro FX 3800
CUDA driver / runtime version	5.0
No. of CUDA cores	192
Memory Bandwidth	51.2 GB/sec
GPU clock rate	1.2 GHz
Global Memory	1 GB
Max threads/ block	512
Warp size	32

TABLE II
THE CPU SPECIFICATION

HP Z600 Workstation	
Processor	2 Quad Core Intel Xeon 5650 Series Processors with 12M cache, DDR3 1333 MHz, HT, Turbo
CPU clock rate	2.66 GHz
CPU cores	12
Memory	4GB

Moreover, Fig. 1 to Fig. 8 also shows the performance evaluation of proposed parallel GCD algorithms on the GPU, OpenMP implementation on the CPU, and corresponding sequential CPU algorithms in case of randomly generated different bit-length MPIs. The execution time of the CPU is only the computation time to compute GCDs, and execution time of the GPU does not include the memory data transfer between CPU and GPU.

The speedup ratio shown in Fig. 9 and Fig. 10 is the ratio of execution time of CPU to that of the GPU. The speedup ratio clearly indicates that the GPU algorithms and OpenMP algorithms are faster than the CPU algorithms. Moreover, we have observed that, for larger group of data values, we achieved more performance as compared to the small group of data values. Also, Fig. 9 and Fig. 10 show the speedup for GCD algorithms on varying bit-length of integers. The OpenMP implementation corresponding to GCD algorithms for maximum data values (8388608) achieves the speedup as, 12.12x for Extended Euclidean algorithm, 11.44x for Binary GCD, 12.71x for Extended Binary GCD, and 11.56x for MEB GCD compared with a single core of the CPU. We have achieved significant speedup on the GPU for various GCD algorithms as 15.75x for Extended Euclidean algorithm, 15.87x for Binary GCD, 15.24x for Extended Binary GCD, and 17.32x for MEB GCD compared to CPU implementation.

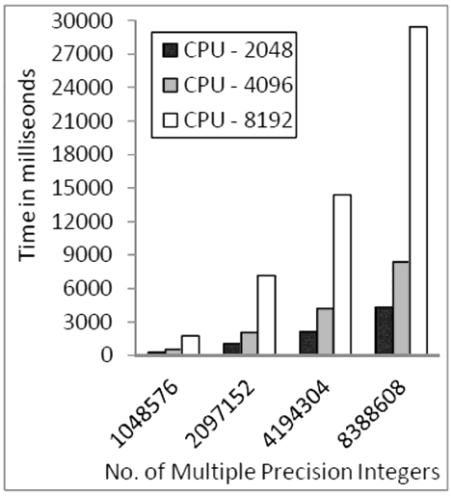

Fig. 1. Binary GCD on CPU

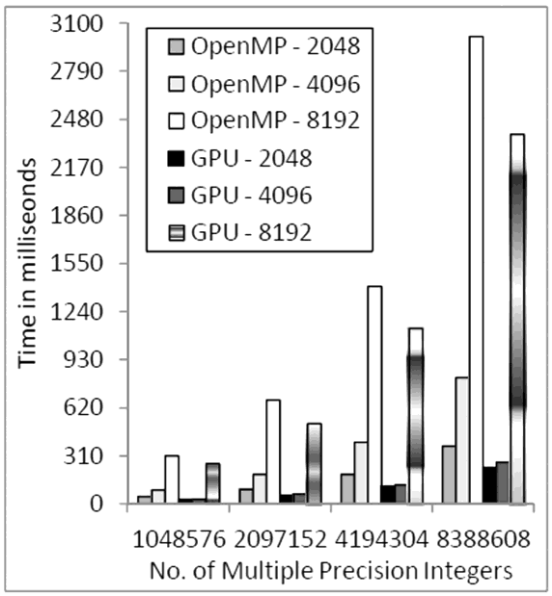

Fig. 2. Binary GCD Using OpenMP and GPU

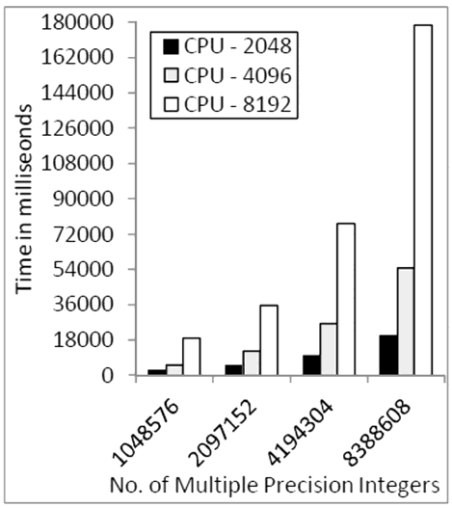

Fig. 3. EBin GCD on CPU

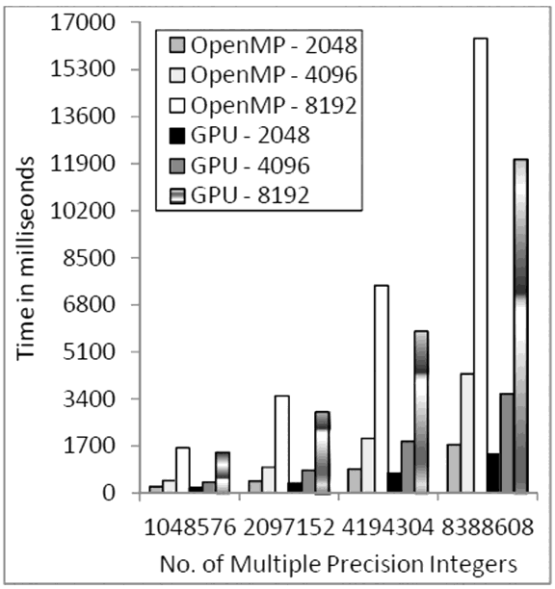

Fig. 4. EBin GCD Using OpenMP and GPU

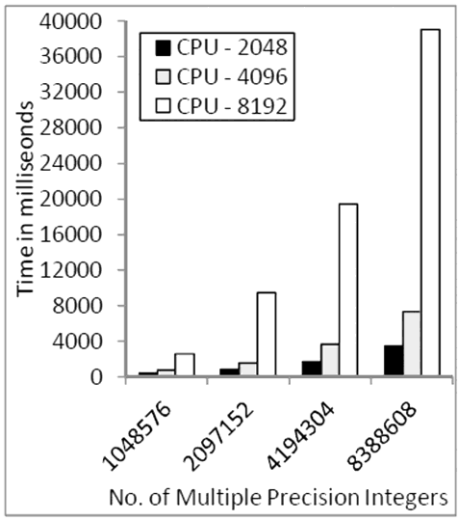

Fig. 5. EEA GCD on CPU

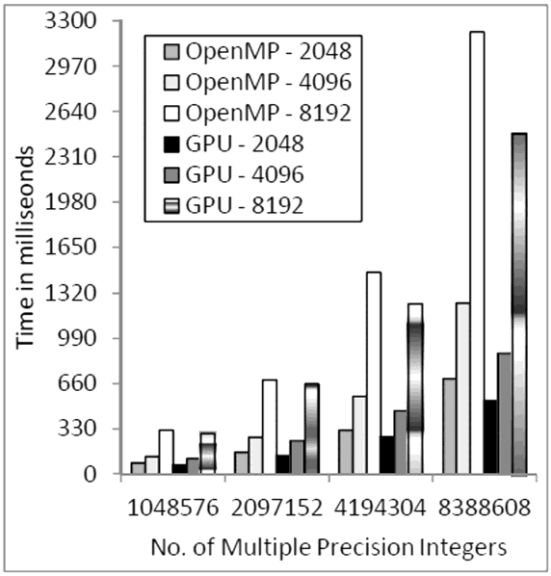

Fig. 6. EEA GCD Using OpenMP and GPU

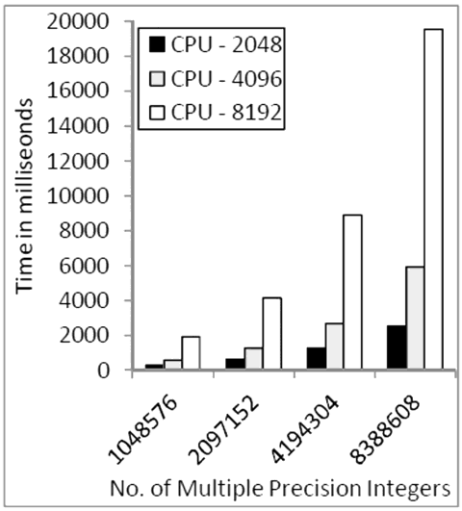

Fig. 7. MBE GCD on CPU

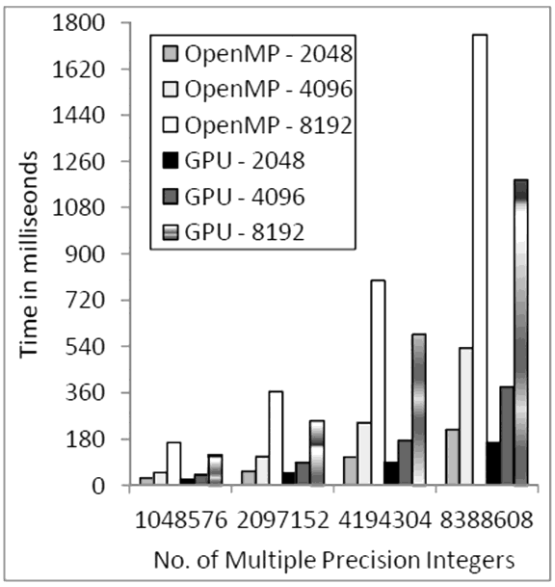

Fig. 8. MBE GCD Using OpenMP and GPU

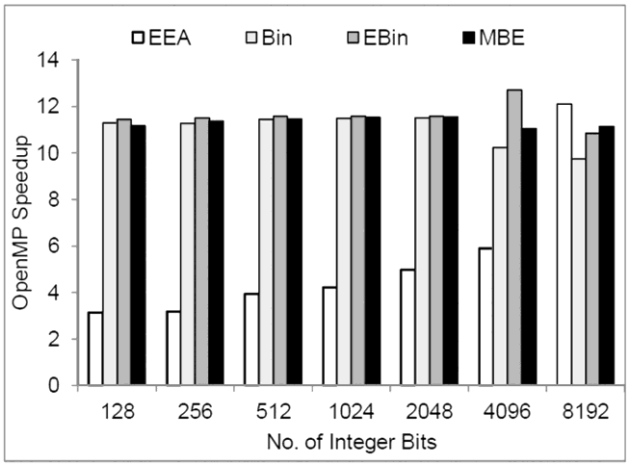

Fig. 9. OpenMP speedup

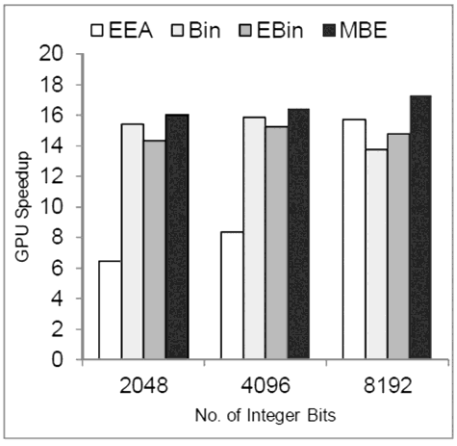

Fig. 10. GPU speedup

VII. Conclusion

In this paper, we have proposed the efficient parallel algorithm for the computation of GCD based on Extended Euclidean GCD, Binary GCD, Extended Binary GCD, and Mixed Binary-Euclid GCD algorithms. The parallel algorithms for multiple precision integers GCD computation on the CPU using OpenMP and on the GPU using CUDA C programming language has been implemented on NVIDIA CUDA and Intel Xeon architectures. Moreover, the performance evaluation on the GPU has achieved significant high speedup in comparison with the single-core CPU. In addition, the speedup achieved by OpenMP compared with the single-core CPU is slightly lower than the speedup achieved by the GPU. Therefore, both OpenMP and GPU perform computation efficiently. This parallel implementation can be used in various cryptography and number theory applications where we need to handle big numbers and also to compute the GCD for big numbers.

Acknowledgment

The research has been supported by Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology (VNIT), Nagpur, India under TEQIP-II scheme of Ph.D enrollment 2011-2012. I would like to thank Mr. Hemprasad Patil for editing the paper and providing valuable suggestions to restructure the paper.

References

[1] J. M. Smiljanic and P. N. Ivanis, "Attack on the RSA Cryptosystem using Integer Factorization," *19th Telecommunication Forum (TELFOR)*, Belgrade, Serbia, 22-24 Dec, 2011, pp. 550-553.

[2] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography*. 1st ed., CRC Press, Boca Raton, Florida, 1997.

[3] H. Cohen and G. Frey, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. 2nd ed., Chapman and Hall/CRC, Taylor Francis Group, 2006.

[4] D. B. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers, 2010.

[5] B. Chor and O. Goldreich, "An Improved Parallel Algorithm for Integer GCD," *Algorithmica*, vol. 5, pp. 1-10, 1990.

[6] P. Wu and J. Chen, "Parallel Extended GCD Algorithm," *In Proc. 8th Int. Parallel Processing Symposium*, Cancun, 26-29 April, 1994, pp. 357-361.

[7] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms.* 3rd ed., Vol. 2, Addison-Wesley, Reading, Mass, 1998.

[8] K. Weber, "Parallel Implementation of the Accelerated Integer GCD Algorithm," *Journal of Symbolic Computation*, vol. 21, pp. 457-466, 1996.

[9] J. Sorenson, "Two fast GCD algorithms," *Journal of Algorithms*, vol. 16, No. 1, pp. 110-144, 1994.

[10] G. Cesari, "Parallel Implementation of Schönhage's Integer GCD Algorithm," *In Lecture Notes in Computer Science 1423*, Ito J. P. Buhler (Ed.), 1998, pp. 64-76.

[11] T. Jebelean, "Using the Parallel Karatsuba Algorithm for Long Integer Multiplication and Division," *European Conf. on Parallel Processing, Lecture Notes in Computer Science*, Vol.1300, 1997, pp. 1169-1172.

[12] S. M. Sedjelmaci, "On a Parallel Lehmer-Euclid GCD Algorithm", *In Proc. Int. Symposium on Symbolic and Algebric Computation*, Ontario, Canada, 2001, pp. 303-308.

[13] D. H. Lehmer, "Euclid's algorithm for large numbers," *American Mathematical Monthly*, vol. 45, pp. 227-233, 1938.

[14] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General Purpose GPU Programming.* Addison-Wesley Publishers, 2011.

[15] Y. Lin, C. Lin and D. Lou, "Efficient Parallel RSA Decryption Algorithm for Many-core GPUs with CUDA," *Int. Conf. on Telecommunication Systems, Modelling and Analysis,* Prague, Czech Republic, 24-26 May, 2012.

[16] N. Nishikawa, K. Twai and T. Kurokawa, "High-Performance Symmetric Block Cipher on CUDA," *2nd Int. Conf. on Networking and Computing*, Osaka, 30 Nov, 2011 to 2 Dec, 2011, pp. 221-227.

[17] Q. Li, C. Zhong, K. Zhao, X. Mei and X. Chu, "Implementation and Analysis of AES Encryption on GPU", *9th Int. Conf. on High Performance Computing and Communication*, Liverpool, 25-27 June, 2012, pp. 843-848.

[18] N. Nishikawa, K. Iwai and T. Kurokawa, "High-Performance Symmetric Block Cipher on Multicore CPU and GPUs," *International Journal of Networking and Computing*, vol. 2 No. 2, pp. 251-268, 2012.

[19] S. Lee, D. Kim, J. Yi and W. Ro, "An Efficient Block Cipher Implementation on Many-Core Graphics Processing Units," *Journal of Information Processing Systems*, vol. 8, No. 1, pp. 159-174, 2012.

[20] K. Zhao, "Implementation of Multiple-Precision Modular Multiplication on GPU," Technical report, 2010.

[21] K. Zhao and X. Chu, "GPUMP: a Multiple-Precision Integer Library for GPUs," *IEEE 10th Int.* Conf. *on Computer and Information Technology*, Bradford, 29 June, 2010 to 1 July, 2010, pp. 1164-1168.

[22] N. Fujimoto, "High Throughput Multiple-Precision GCD on the CUDA Architecture," *IEEE Int. Symposium on Signal Processing and Information Technology*, Ajman, 14-17 Dec, 2009, pp. 507-512.

[23] P. Emeliyanenko, "High-performance Polynomial GCD Computation on Graphics Processors," *IEEE Int. Conf. on High Performance Computing and Simulation*, Istanbul, 4-8 July, 2011, pp. 215-224.

[24] J. P. Sorenson, "A randomized sublinear time parallel GCD algorithm for the EREW PRAM," *Information Processing Letters, vol.* 110, pp. 198-201, 2010.

[25] S. M. Sedjelmaci, "Fast parallel GCD algorithm of many integers," *ACM Communications in Computer Algebra*, vol. 47(3/4), pp. 92-93, December 2013.

[26] J. Chen, W. Waston III, and M. F. Chen, "Efficient GCD Computation for Big Integers on Xeon Phi Coprocessor*," 9th IEEE Int. Conf. on Networking, Architecture, and Storage*, Tianjin, 6-8 Aug. 2014, pp. 113-117.

[27] OpenMP Introduction, http://en.wikipedia.org/wiki/OpenMP (Accessed on 22 July 2014).

[28] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. 1st ed., McGraw-Hill Science/Engineering/Math, 2003.

[29] B. Chapman, G. Jost, R. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, 2007.

[30] OpenMP compilers, http://openmp.org/wp/openmp-compilers/ (Accessed on 10 May 2014).

[31] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, Version 4.2, 2012.

[32] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge Monographs on Computational and Applied Mathematics (No. 18). Cambridge University Press, November 2010.

[33] M. Lehman and N. Burla, "Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units," *IRE Transaction on Electronic Computers*, vol. Ec-10, No. 4, pp. 691-698, 1961.

[34] GMP Library, http://gmplib.org/ (Accessed on 10 May 2014).

[35] T. Granlund, *The GNU Multiple Precision Arithmetic Library*, ed. 5.1.1, 2013.

[36] M. Manssen, M. Weigel, and A. K. Hartmann, "Random Number Generator for Massively Parallel Simulations on GPU," *The European Physical Journal Special Topics*, vol. 210 No. 1, pp. 53-71, 2012.

Jitendra V. Tembhurne, completed his M. E. in Computer Science and Engineering in 2011 and currently pursuing Ph. D in the Department of Computer Science & Engineering, VNIT, Nagpur, Maharashtra, India. His area of research is the parallelization of cryptography applications on multi-core and many-core architecture using OpenMP and CUDA.

Shailesh R. Sathe, completed his M. Tech. from Indian Institute of Technology (IIT), Bombay and Ph.D. from R.T.M. Nagpur University (at VRCE/VNIT). At present, he is Professor and Dean Planning & Finance, VNIT, Nagpur, India. He has handled many national research projects on Image processing, Security and Parallel processing. His current research interest includes Parallel processing, Algorithms and Theoretical Computer Science etc. He has published more than 30 papers in various reputed International journals/conferences.