

PROGRAMMING MODEL FOR GPGPU FOR VARIOUS ALGORITHMIC PROBLEMS

A PROJECT REPORT

Submitted in partial fulfillment for the award of the degree of

**B.TECH
in
Information Technology**

by
Vaibhav Mohan 08BIT230

**Under the Guidance of
Prof. Balaraman S.**



VIT
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)

School of Information Technology & Engineering

MAY 2012

DECLARATION BY THE CANDIDATE

I hereby declare that the project report entitled "**PROGRAMMING MODEL FOR GPGPU FOR VARIOUS ALGORITHMIC PROBLEMS**" submitted by me to Vellore Institute of Technology University, Vellore, in partial fulfillment of the requirement for the award of the degree of **B.Tech (Information Technology)** is a record of bonafide project work carried out by me under the guidance of **Prof. Balaraman S.** I further declare that the work reported in this project has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore

Signature of the Candidate

Date : 1st May, 2012

Vaibhav Mohan



VIT
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)

School of Information Technology & Engineering [SITE]

CERTIFICATE

This is to certify that the project report entitled "**PROGRAMMING MODEL FOR GPGPU FOR VARIOUS ALGORITHMIC PROBLEMS**" submitted by **Vaibhav Mohan (08BIT230)** to Vellore Institute of Technology University, Vellore in partial fulfillment of the requirement for the award of the degree of **B.Tech(Information Technology)** is a record of bonafide work carried out by him under my supervision. The project fulfills the requirements as per the regulations of this Institute and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Prof Balaraman, S.
Supervisor
Associate Professor, SITE

Prof. C. Ranichandra
Program Manager,
B.Tech (IT)

Internal Examiner (s)

External Examiner (s)

ACKNOWLEDGEMENT

I owe great many thanks to a great many people who helped and supported in completing this project.

I would like to deeply thank **Prof. Balaraman, S.** the Guide of my project, for guiding in my work, and correcting various documents of mine with attention and care. He has taken initiative to suggest ideas throughout my project work.

I express my thanks to the Chancellor **Dr. G. Viswanathan** for extending his support and providing us the infrastructure to carry this out.

My deep sense of gratitude to Vice Chancellor **Dr. V. Raju** and Pro-Vice Chancellor **Dr. S. Narayanan** for their support. My thanks are due to **Dr. R. Saravanan** (Director of SITE), **Prof. C. Ranichandra** (Program Manager), and **Prof. M. Asha Jerlin** (Year Coordinator) for their support.

I would also thank my Institution, VIT, and my faculty members without whom this project would have been a distant reality. I also extend my heartfelt thanks to my family and well wishers.

Place: Vellore

Date: 1st May, 2012

Vaibhav Mohan
(08BIT230)

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	SYNOPSIS	IV
	LIST OF TABLES	V
	LIST OF FIGURES	VI
	LIST OF ACRONYMS	VIII
1.	INTRODUCTION	1
1.1	Background	1
1.2	Problem Statement	2
1.3	Importance	3
2.	OVERVIEW AND PLANNING	4
2.1	Proposed System Overview	4
2.2	Challenges	4
2.3	Architecture Design	5
2.4	Modules design and description	7
2.5	Architecture Specification	8
2.6	Hardware Requirements	9
2.7	Software Requirements	9
3.	LITERATURE SURVEY AND REVIEW	10
3.1	Literature Survey	10
3.1	Literature Summary	16

4.	SYSTEM DESIGN	17
4.1	High-Level Design	17
4.1.1	For RSA Algorithm	17
4.1.2	For Dense Matrix-Matrix Multiplication	19
4.2	Low-Level Design	21
4.2.1	For RSA Algorithm	21
4.2.2	For Dense Matrix-Matrix Multiplication	21
4.3	Test Cases Generation	21
4.3.1	For RSA Algorithm	21
4.3.2	For Dense Matrix-Matrix Multiplication	23
5.	System Implementation	25
5.1	Description on the software used	25
5.2	Description on Methods/functions used	26
5.3	Module wise implementation description	28
5.4	Code	29
5.4.1	Code for CPU implementation of RSA Algorithm	29
5.4.2	Code for GPU implementation of RSA Algorithm	31
5.4.3	Code for CPU implementation of Dense Matrix-Matrix Multiplication	34
5.4.4	Code for GPU implementation of Dense Matrix-Matrix Multiplication	35
6.	Results and Discussion	37
6.1	Output / Results	37
6.2	Results Analysis	40
6.3	Discussion	44

SYNOPSIS

Demand in the consumer market for graphics hardware that accelerates rendering of 3D images has resulted in Graphic Cards that are capable of delivering astonishing levels of performance. These results were achieved by specifically tailoring the hardware for the target domain. As graphics accelerators become increasingly programmable however, this performance has made them an attractive target for other domains. Graphic processing units provide a low-cost parallel computing architecture. It is possible to achieve massive parallelism by SIMD (Single Instruction Multiple Data) on General Purpose Graphics Processing Unit (GPGPU) integrated with Central Processing Unit (CPU).

In this project, two application of different algorithmic requirements - RSA Algorithm and Dense Matrix-Matrix Multiplication - are taken up for implementing on a parallel infrastructure with acceleration programming model, Compute Unified Device Architecture (CUDA), which uses multithreading technique. RSA Algorithm is one of the algorithms for public-key cryptography that is based on the presumed difficulty of factoring large. RSA Algorithm using CUDA can achieve high performance. The Dense Matrix-Matrix Multiplication algorithm uses block algorithm for processing the elements using Accelerator Unit (GPU) and the CPU. The performance enhancement with the GPU programming is recorded. This comparison is presented for both the applications.

LIST OF FIGURES

Figure No.	Title	Page No.
Fig. 1.1.1	Ping-Pong operation in the GPU to produce intermediate results	2
Fig. 2.3.1	Evolution of GPU and CPU w.r.t. time	5
Fig. 2.3.2	Floating-Point operations per second and memory bandwidth for CPU and GPU	6
Fig. 2.3.3	Architecture of CPU and GPU	6
Fig. 2.3.4	Figure Showing architecture of CUDA	7
Fig. 2.5.1	Architecture of system deploying RSA Algorithm	8
Fig. 2.5.2	Architecture of system deploying GPGPU for Dense Matrix-Matrix Multiplication	9
Fig. 4.1.1.1	Use case for RSA Algorithm	17
Fig. 4.1.1.2	Flow chart for RSA Algorithm	18
Fig. 4.1.2.1	Use case for Dense Matrix-Matrix Multiplication	19
Fig. 4.1.2.2	Flow chart for Dense Matrix-Matrix Multiplication	20
Fig. 6.1.1	Output of RSA Algorithm implemented on CPU using message size 4500	37
Fig. 6.1.2	Output of RSA Algorithm implemented on GPU using message size 4500	38
Fig. 6.1.3	Output of Dense Matrix-Matrix Multiplication implemented over CPU with Dimension 1000x1000	39
Fig. 6.1.4	Output of Dense Matrix-Matrix Multiplication implemented over GPU with Dimension 3000x3000	40
Fig. 6.2.1	Graph of Time taken by CPU and GPU in performing RSA encryption	41

LIST OF ACRONYMS

GPU	-	Graphics Processing Unit
CPU	-	Central Processing Unit
RSA	-	Rivest Shamir Adleman
CUDA	-	Compute Unified Device Architecture
w.r.t.	-	With Respect To
SIMD	-	Single Instruction Multiple Data
TFLOPs	-	Tera Flops
GFLOPs	-	Giga Flops
BSP	-	Bulk Synchronous Parallel
GPGPU	-	General Purpose GPU
ALU	-	Arithmetic and Logical Unit
API	-	Application Programming Interface
CUFFT	-	CUDA Fast Fourier Transforms Library
CUBLAS	-	CUDA Basic Linear Algebra Subroutines Library
CUSPARSE	-	CUDA Sparse Matrix library
NVCC	-	Nvidia CUDA C/C++ Compiler

1. INTRODUCTION

1.1 Background:

In today's era, there is a great importance to parallel programming to gain high performance in terms of time required for data computation. There are some constraints to achieve parallelism on CPU (Central Processing Unit). It is possible to achieve data parallelism by SIMD (Single Instruction Multiple Data) on General Purpose Graphics Processing Unit (GPGPU) integrated with Central Processing Unit (CPU). In implementing security algorithms on GPGPU, most of research is going on. In this project, RSA algorithm and Dense Matrix-Matrix Multiplication is implemented to utilize the parallel architecture of Graphic Card (GPU) using a programming model Compute Unified Device Architecture (CUDA) which uses multithreading technique. RSA algorithm is one of the security algorithms. RSA is an algorithm for public-key cryptography that is based on the presumed difficulty of factoring large integers, the factoring problem. RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described it in 1978. A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message. GPUs provide high computation power at low costs and have been described as desktop supercomputers. The GPUs have been used for many general purpose computations due to their low cost, high computing power, and high availability. The latest GPUs, for instance, can deliver close to 1 Tera Flops (TFLOPs) of compute power at a higher cost. The stages of were exploited for parallelism with the flow of execution handled serially using the pipeline in the earlier, GPGPU model. The GPUs expose a general, data-parallel programming model today in the form of CUDA. The recently adopted OpenCL standard will provide a common computing model to not only all GPUs, but also to other platforms like multi-core, many-core, and Cell/B.E. CUDA from NVIDIA presents a heterogeneous programming model where the parallel hardware can be used in conjunction with the CPU. In conjunction with a CPU, it can be used as Bulk Synchronous Parallel (BSP) hardware with the CPU deciding the barrier for synchronization. GPU programming models are constrained in such a way that the compiler and runtime can reason about the application and extract the parallelism automatically. Examples of this include DirectX, CUDA, and Cg. Intel architecture is more general purpose than GPU and other coprocessor architecture. Unlike GPUs, Intel architectures have:

- 1) Inter core communication through substantial, coherent cache hierarchies.
- 2) Efficient, low latency thread synchronizations across the entire processor array.
- 3) Narrower effective SIMD width.

At a high level, the goal is to define a constrained programming model that efficiently and portably targets highly parallel general purpose cores, such as Intel multi-core and Tera-scale systems. There are different ways to classify parallel

computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy. Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction and Data. Each of these dimensions can have only one of two possible states: Single or Multiple. GPU based processors can efficiently perform floating point operations and use parallelism at massive levels due to which they can be a suitable choice for processing large amounts of data.

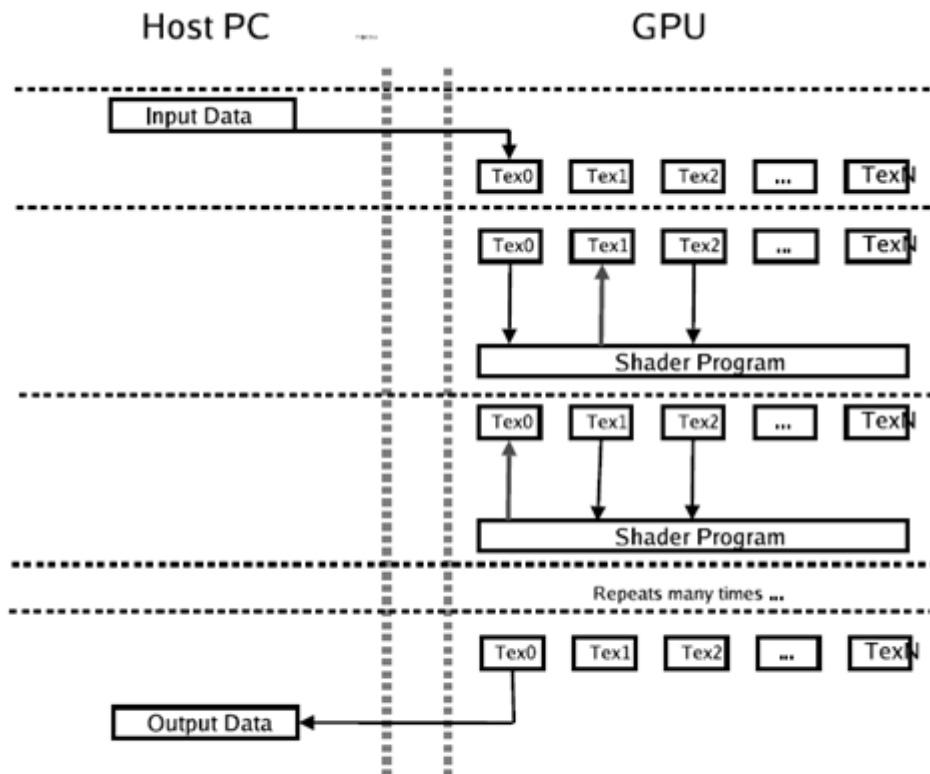


Fig 1.1.1 Ping-pong operation in the GPU to produce intermediate results

1.2 Problem Statement:

When we implement RSA algorithm on CPU, it takes a lot of time as CPU processes the text for encryption one by one. On the other hand GPU processes the text in parallel manner as it has more number of core as compared to CPU and hence can show massive parallel computation thus giving us an increase in performance boost of RSA algorithm. Implementing a public key cryptosystem is always a tradeoff between security and efficiency. The problem with the number theoretic cryptosystems (i.e. RSA) is that they require a lot of computational power for providing a high level of security and most likely a low level of efficiency. Public key algorithms are known to be slower than symmetric key alternatives because of their basis in modular arithmetic. Therefore, how to make a more efficient and faster implementation of public key algorithms is concerned. Running the public key algorithms by use of the parallel properties of the GPU in modular multiplication and

modular exponentiation may be a solution to this problem. Multiplication of big integers is one of the building blocks in doing modular arithmetic. The field of General-purpose GPU which is about solving problems other than graphics rendering using the GPU was until recently without a good solution. CUDA is a framework which makes these kinds of implementations more available to the general public of programmers. Using the unique properties of the GPU through CUDA has greatly increased the efficiency of many computational problems. The target in this project is to study and analyse the majority of algorithms related to the modular multiplication and modular exponentiation, and then to design and make an implementation of a public key algorithm in CUDA. Finally, this project will compare the performance between the GPU implementation and the CPU implementation in order to look into the possibility of improving the performance of public key algorithms.

1.3 Importance:

The necessity for information security has become more and more widespread during these days. Fast modular exponentiation algorithms are often considered of practical significance in public-key cryptosystems. Parallelization of public key algorithms could be very useful for a high level of security system and save a lot of computation time. With the combination of them, the public key cryptosystem will be more efficient and effective for those kinds of system.

Furthermore, in this research the performance of public key algorithm will be compared between the GPU implementation and the CPU implementation. It could be used to determine the direction of parallelization of public key algorithms in the future. The Dense Matrix-Matrix Multiplication is common in various scientific domains. With the development of the GPGPU field, modern graphics processing units (GPUs) have been at the leading edge of increasing chip-level parallelism. Current NVIDIA GPUs are many core processor chips with parallelism architecture. This degree of hardware parallelism reflects the fact that GPU architectures evolved not only to fit the needs of real-time computer graphics but also parallel computing. On the other hand, the GPU is easy use and cheaper compared to a computer cluster for the purpose of parallel computations. So the research in this field will have a different angle for parallel computation.

2. OVERVIEW AND PLANNING

2.1 Proposed system overview:

This project focuses on how to make a more efficient and faster implementation of public key algorithms and Dense Matrix-Matrix Multiplication. Two experiments implementing a public key algorithm are performed on different hardware platforms. One is to implement the selected algorithms normally on a CPU with different data sizes, and then record the execution time and other related data. Another is to execute designed parallel algorithms on a CUDA-enabled GPU, and record related data as well. Finally the performance comparison is performed between those experiments. The parallelization of public key algorithms is mainly performed in the part of modular multiplication and modular exponentiation. Therefore, this project implements a representative public-key algorithm RSA respectively on the CPU and the CUDA-enabled GPU, and compares their performances to find out whether the public-key algorithm could be implemented faster and more efficient on a GPU. The project also compares the CPU as well as GPU implementation of dense Matrix-Matrix Multiplication and compares the results. Theoretically, the performance that RSA as well as Dense Matrix-Matrix Multiplication implemented on a GPU should be better than that on the CPU since parallelization is performed on the CUDA-enabled GPU with massive parallel processors. In addition, there are still other related issue concerned in this project, such as time consumption in data transfer between host and device. The CUDA driver API and C runtime for CUDA are two of the programming interfaces to CUDA . The C runtime for CUDA handles kernel loading and kernels' setting before they are launched. The implicit code initialization, CUDA context management, CUDA module management (cubin and function mapping), kernel configuration, and parameter passing are all performed by the C runtime for CUDA. In addition, CUDA supports C++ code and can be compiled with any C++ compiler. However, the current version of CUDA does not support all features of C++. Therefore, all functions in this project are mostly performed in C.

2.2 Challenges:

The challenges faced in this project are how to parallelize the working of RSA algorithm on GPU so that we can gain a performance boost in the RSA encryption of texts. Since CUDA doesn't support recursive calls to the functions, hence we cannot use that while programming for GPU. There is also a challenge to generate good prime numbers so that the encryption done must be secure and hard to crack. Also the challenge faced in implementing Dense Matrix-Matrix Multiplication is memory constraint.

2.3 Architecture Design:

Driven by the insatiable market demand for real time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many core processor with tremendous computational horsepower and very high memory bandwidth.

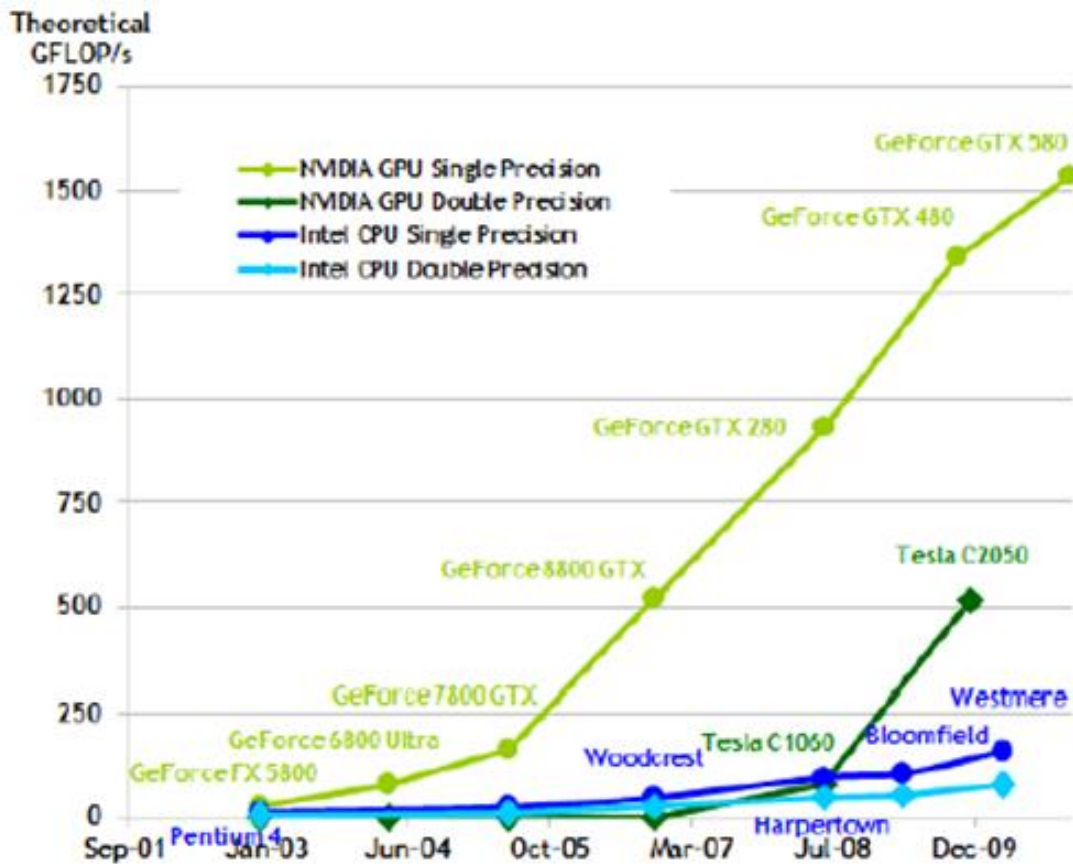


Fig 2.3.1 Evolution of GPU and CPU w.r.t time

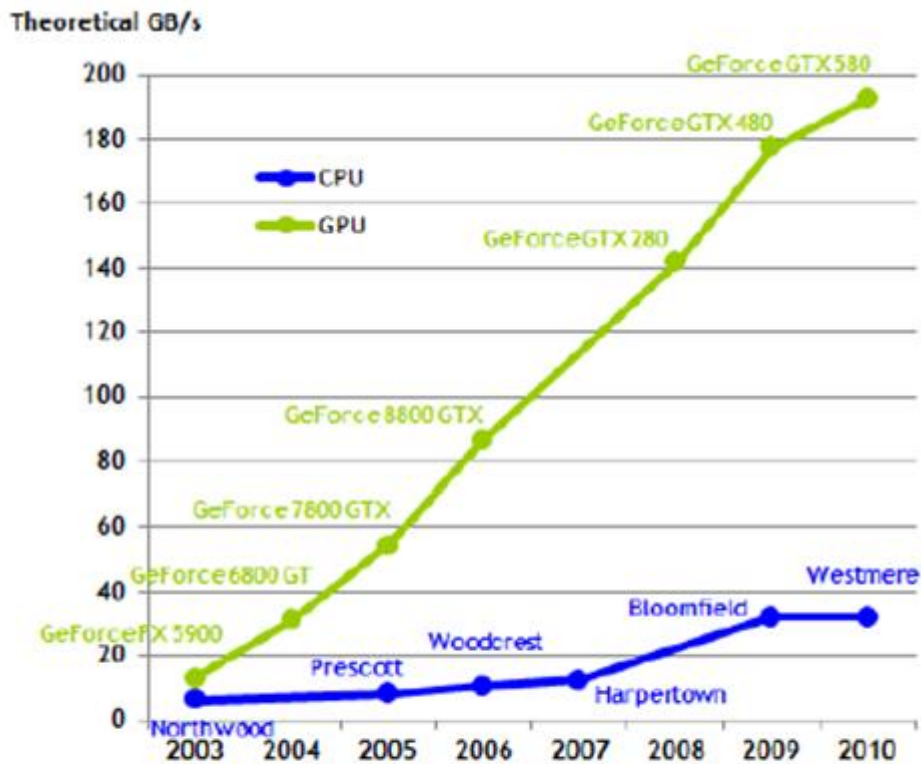


Fig 2.3.2 Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

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 – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

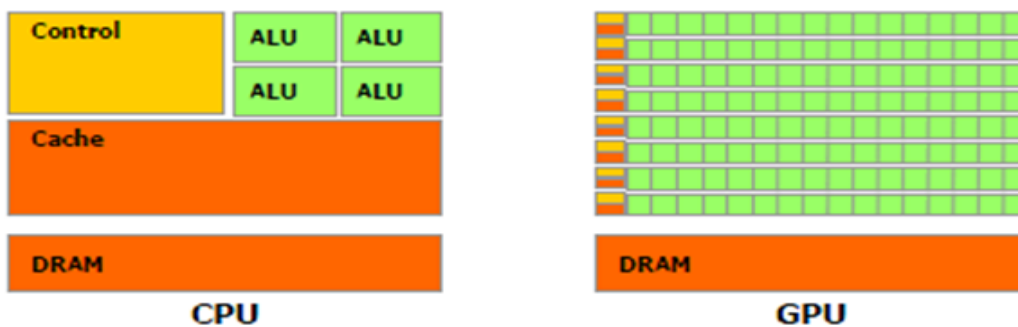


Fig 2.3.3 Architecture of CPU and GPU

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that 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 as a high-level programming language. As illustrated by Figure 1-3, other languages or application programming interfaces are supported, such as CUDA FORTRAN, OpenCL, and DirectCompute.

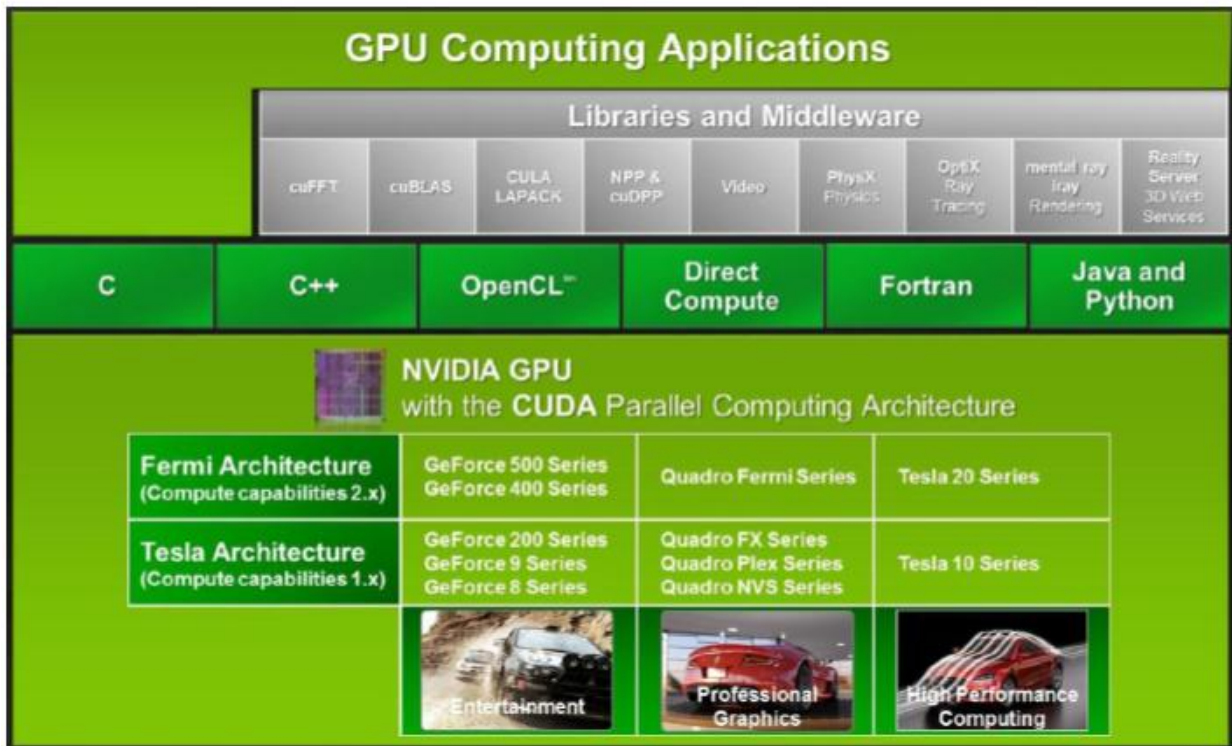


Fig 2.3.4 A figure showing architecture of CUDA

2.4 Modules design and description:

- I. Module 1 – CPU Implementation of RSA Algorithm: In this module, the RSA algorithm is implemented using C++ and utilizes CPU in encryption or decryption. The time taken by CPU in computing prime numbers, encrypting text, and decrypting text is calculated and stored for comparing with GPU implementation.
- II. Module 2 – GPU Implementation of RSA Algorithm: In this module, the RSA algorithm is implemented using C on GPU. The algorithm is parallelized in order to perform operations efficiently as compared to CPU implementation. Here also the time taken by GPU in computing prime numbers, encrypting text and decrypting text is calculated and then the comparison of time taken on CPU and GPU is done.
- III. Module 3 – CPU Implementation of Dense Matrix-Matrix Multiplication: In this module, the Dense Matrix-Matrix Multiplication is implemented

using C++ and utilizes CPU in performing Multiplication. The time taken by CPU is calculated and stored for comparing with GPU implementation.

- IV. Module 4 – GPU Implementation of Dense Matrix-Matrix Multiplication: In this module, the Dense Matrix-Matrix Multiplication is implemented using C on GPU. The algorithm is parallelized in order to perform operations efficiently as compared to CPU implementation. Here also the time taken by is calculated and then the comparison of time taken on CPU and GPU is done.

2.5 Architecture Specification:

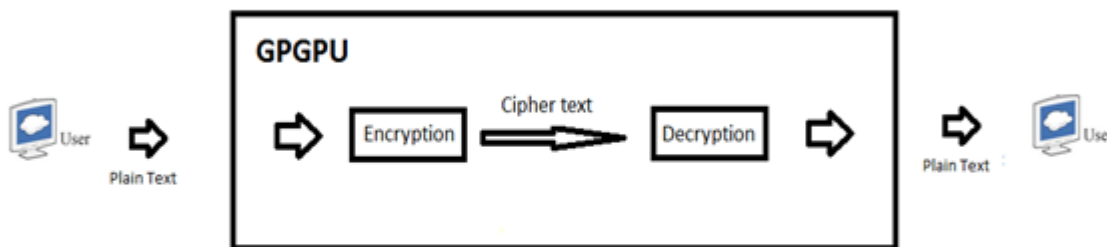


Fig 2.5.1 Architecture of system deploying RSA Algorithm

The User enters plain text as input to the system. This text is then encrypted using RSA algorithm and is then converted into cipher text. It is then decrypted on the other user's end with the help of his/her private key and message is then read by the intended user.

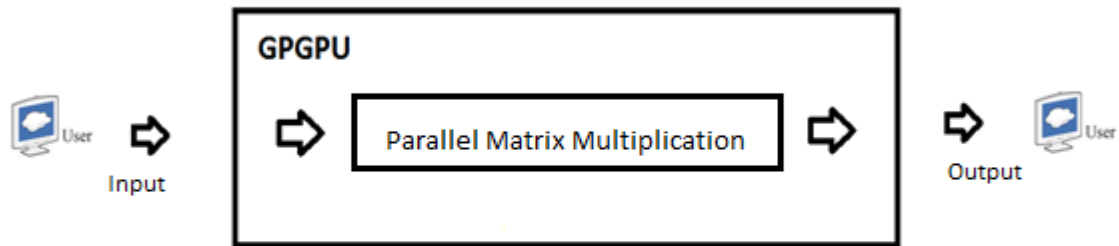


Fig 2.5.2 Architecture of System Deploying GPU for Dense Matrix-Matrix Multiplication

The User just runs the program and the program computes the time taken in performing $N \times N$ Matrix-Matrix Multiplication where 'N' is the dimension of the Matrix which is supplied by the user of the program.

2.6 Hardware Requirements:

- I) Intel Core 2 Duo processor E7400 @ 2.8 GHz
- II) NVIDIA Graphics Card with CUDA

2.7 Software Requirements:

- I) NVIDIA latest graphics drivers. (Here we used NVIDIA Geforce GTS 250 with driver version 296.10)
- II) CUDA toolkit v4.0.17
- III) CUDA Tools
- IV) GPU Computing SDK v4.0.19
- V) NVIDIA Parallel NSight
- VI) Microsoft Visual Studio 2008/2010
- VII) .NET Framework 3.5 or later

3. LITERATURE SURVEY AND REVIEW

3.1 Literature Survey:

Title:

Toward Acceleration of RSA Using 3D Graphics Hardware.

Author:

A. Moss, D. Page and N.P. Smart

Description:

Efficient arithmetic operations modulo a large prime (or composite) number are core to the performance of public key cryptosystems. RSA is based on arithmetic in the ring \mathbb{Z}_N , where $N = pq$ for large prime p and q , while Elliptic Curve Cryptography (ECC) can be parameterised over the finite field \mathbb{F}_p for large prime p . With a general modulus m taking the value N or p respectively, on processors with a w -bit word size, one commonly represents $0 \leq x < m$ using a vector of $n = dm/2w$ radix-2 w digits. Unless specialist coprocessor hardware is used, modular operations on such numbers are performed in software using well known techniques that operate using native integer machine operations. Given the significant computational load, it is desirable to accelerate said operations using instruction sets that harness Single Instruction Multiple Data (SIMD) parallelism; in the context of ECC, a good overview is given by Hankerson et al. Although dedicated vector processors have been proposed for cryptography these are not commodity items. In an alternative approach, researchers have investigated cryptosystems based on arithmetic in fields modulo a small prime m or extension thereof. Since ideally we have $m < 2w$, the representation of $0 \leq x < m$ is simply one word; low-weight primes offer an efficient method for modular reduction. Examples that use such arithmetic include Optimal Extension Fields (OEF) which can provide an efficient underpinning for ECC; torus based constructions such as T30; and the use of Residue Number Systems (RNS) to implement RSA. Issues of security aside, the use of such systems is attractive as operations modulo m may be more efficiently realised by integer based machine operations. This fact is reinforced by the aforementioned potential for parallelism; for example, addition operations in an OEF can be computed in a component-wise manner which directly maps onto SIMD instruction sets. However, the focus on use of integer operations in implementation of operations modulo large and small numbers ignores the capability for efficient floating point computation within commodity desktop class processors. This feature is often ignored and the related resources are left idle: from the perspective of efficiency we would like to utilise the potential for floating point arithmetic to accelerate our implementations. Examples of this approach are provided in work by Bernstein which outline high-performance floating point based implementations of primitives such as Poly1305 and Curve25519. Beyond algorithmic optimisation, use of floating point hardware in general purpose processors such as the Intel Pentium 4 offered Bernstein some

significant advantages. Specifically, floating point operations can often be executed in parallel with integer operations; there is often a larger and more orthogonally accessible floating point register file available; good scheduling of floating point operations can often yield a throughput close to one operation per-cycle. Further motivation for use of this type of approach is provided by the recent availability of programmable, highly SIMD-parallel floating point co-processors in the form of Graphics Processing Units (GPU). Driven by market forces these devices have developed at a rate that has outpaced Moore's Law: for example, the Nvidia 7800-GTX uses 300 million transistors to deliver roughly 185 Gflop/s in contrast with the 55 million transistor Intel Pentium 4 which delivers roughly 7 Gflop/s. Although general purpose use of the GPU is an emerging research area, until recently the only published prior usage for cryptography was by Cook et al. who implemented block and stream ciphers using the OpenGL command-set; we are aware of no previous work accelerating computationally expensive public key primitives. Further, quoted performance results in previous work are somewhat underwhelming, with the GPU executing AES at only 75% the speed of a general purpose processor. This was recently improved, using modern GPU hardware, by Harrison and Waldron who also highlight the problems of overhead in communication with the card and miss reporting of host processor utilisation while performing GPU computation. This paper seeks to gather together all three strands of work described above. Our overall aim is arithmetic modulo a large number so we can execute operations required in the RSA public key cryptosystem; we implement this arithmetic with an RNS based approach which performs arithmetic modulo small floating point values. The end result is an implementation which firstly fits the GPU programming model, and secondly makes effective use of SIMD-parallel floating point operations on which GPU performance relies. We demonstrate that with some caveats, this implementation makes it possible to improve performance using the GPU versus that achieved using a general purpose processor (or CPU). An alternative approach is recent work implementing a similar primitive on the IBM Cell, another media-biased vector processor. However, the radically different special purpose architecture of the GPU makes the task much more difficult than on the general purpose IBM Cell, hence our differing approach. We organise the paper as follows. In Section 2 we give an overview of GPU architecture and capabilities. We use Section 3 to describe the algorithms used to implement modular exponentiation in RNS before describing the GPU implementation in Section 4. The experimental results in Section 4.3 compare the GPU implementation with one on a standard CPU, with conclusions in Section 5.

Title:

GPU Cluster for High Performance Computing

Author:

Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover

Description:

The GPU, which refers to the commodity off-the-shelf 3D graphics card, is specifically designed to be extremely fast at processing large graphics data sets (e.g., polygons and pixels) for rendering tasks. Recently, the use of the GPU to accelerate non-graphics computation has drawn much attention. This kind of research is propelled by two essential considerations:

Price/Performance Ratio: The computational power of today's commodity GPUs has exceeded that of PC-based CPUs. For example, the nVIDIA GeForce 6800 Ultra, recently released, has been observed to reach 40 GFlops in fragment processing. In comparison, the theoretical peak performance of the Intel 3GHz Pentium4 using SSE instructions is only 6 GFlops. This high GPU performance results from the following:

(1) A current GPU has up to 16 pixel processors and 6 vertex processors that execute 4-dimensional vector floating point instructions in parallel;

(2) pipeline constraint is enforced to ensure that data elements stream through the processors without stalls; and

(3) unlike the CPU, which has long been recognized to have a memory bottleneck for massive computation, the GPU uses fast on-board texture memory which has one order of magnitude higher bandwidth (e.g., 35.2GB/sec on the GeForce 6800 Ultra). At the same time, the booming market for computer games drives high volume sales of graphics cards which keeps prices low compared to other specialty hardware. As a result, the GPU has become a commodity SIMD machine on the desktop that is ready to be exploited for computation exhibiting high compute parallelism and requiring high memory bandwidth.

Evolution Speed: Driven by the game industry, GPU performance has approximately doubled every 6 months since the mid-1990s, which is much faster than the growth rate of CPU performance that doubles every 18 months on average (Moore's law), and this trend is expected to continue. This is made possible by the explicit parallelism exposed in the graphics hardware. As the semiconductor fabrication technology advances, GPUs can use additional transistors much more efficiently for computation than CPUs by increasing the number of pipelines. Recently, the development of GPUs has reached a new high-point with the addition of single-precision 32bit floating point capabilities and the high level language programming interface, called Cg. The developments mentioned above have facilitated the abstraction of the modern GPU as a stream processor. Consequently, mapping scientific computation onto the GPU has turned from initially hardware hacking techniques to more of a high level designing task. Many kinds of computations can be accelerated on GPUs including sparse linear system solvers, physical simulation, linear algebra operations, partial difference equations, fast Fourier transform, level-set computation, computational geometry problems, and also non-traditional graphics, such as volume rendering, ray-tracing, and flow visualization. (We refer the reader to the web site of General-Purpose Computation Using Graphics Hardware (GPGPU) for more information.) Whereas all of this work has been limited to computing small-scale problems on a single GPU, in this paper we address the large scale computation on a GPU cluster. Inspired by the attractive Flops/\$ ratio and the projected development of the GPU, we believe that a GPU

cluster is promising for data-intensive scientific computing and can substantially outperform a CPU cluster at the equivalent cost. Although there have been some efforts to exploit the parallelism of a graphics PC cluster for interactive graphics tasks, to the best of our knowledge we are the first to develop a scalable GPU cluster for high performance scientific computing and large-scale simulation. We have built a cluster with 32 computation nodes connected by a 1 Gigabit Ethernet switch. Each node consists of a dual-CPU HP PC with an nVIDIA GeForce FX 5800 Ultra — the GPU that cost \$399 in April 2003. By adding 32 GPUs to this cluster, we have increased the theoretical peak performance of the cluster by 512 Gflops at a cost of only \$12,768. As an example application, we have simulated airborne contaminant dispersion in the Times Square area of New York City. To model transport and dispersion, we use the computational fluid dynamics (CFD) model known as the Lattice Boltzmann Method (LBM), which is second order accurate and can easily accommodate complex-shaped boundaries. Beyond enhancing our understanding of the fluid dynamics processes governing dispersion, this work will support the prediction of airborne contaminant propagation so that emergency responders can more effectively engage their resources in response to urban accidents or attacks. For large scale simulations of this kind, the combined computational speed of the GPU cluster and the linear nature of the LBM model create a powerful tool that can meet the requirements of both speed and accuracy. In the context of modeling contaminant transport, Brown et al. have presented an approach for computing wind fields and simulating contaminant transport on three different scales: mesoscale, urban scale and building scale. The system they developed, called HIGRAD, computes the flow field by using a second-order accurate finite difference approximation of the Navier-Stokes equations and doing large eddy simulation with a small time step to resolve turbulent eddies. These simulations required a few hours on a supercomputer or cluster to solve a 1.6 km \times 1.5 km area in Salt Lake City at a grid spacing of 10 meters (grid resolution: 160 \times 150 \times 36). In comparison, our method is also second order accurate, incorporates a more detailed city model, and can simulate the Times Square area in New York City at a grid spacing of 3.8 meters (grid resolution: 480 \times 400 \times 80) with small vortices in less than 20 minutes. This paper is organized as follows: Section 2 illustrates how the GPU can be used for non-graphics computing. Section 3 presents our GPU cluster, called the Stony Brook Visual Computing Cluster. In Section 4, we detail our LBM implementation on the GPU cluster, followed by the performance results and a comparison with our CPU cluster. Section 5 presents our dispersion simulation in the Times Square area of New York City. In Section 6, we discuss other potential usage of the GPU cluster for scientific computations. Finally, we conclude in Section 7.

Title:

Implementation of public key algorithms in CUDA

Author:

Hao Wu

Description:

In the field of cryptography, public key algorithms are widely known to be slower than symmetric key alternatives for the reason of their basis in modular arithmetic. The modular arithmetic in e.g. RSA and Diffie Hellman is computationally heavy when compared to symmetric algorithms relying on simple operations like shifting of bits and XOR. Therefore, how to make a more efficient and faster implementation of public key 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). CUDA (Compute Unified Device Architecture) is a framework which makes the GPGPU more accessible and easier to learn for the general population of programmers. This is because it builds on C and hides many of the complicated details of how the GPU works from a CUDA developer. Using the unique properties of the GPU through CUDA has greatly increased the efficiency of many computational problems. Multiplication of big integers is one of the building blocks in doing modular arithmetic. Running the public key algorithms by use of the parallel properties of the GPU in modular multiplication and modular exponentiation may be a solution to this problem.

The target in this research is to study and analyse the majority of algorithms related to the modular multiplication and modular exponentiation, and then to design and make an implementation of a public key algorithm in CUDA. Finally, this project will compare the performance between the GPU implementation and the CPU implementation in order to look into the possibility of improving the performance of public key algorithms. The research questions are divided into four groups, the first one regarding modular multiplication and modular exponentiation of big integers and their parallelism, the second one about integrating parallel modular multiplication and modular exponentiation into the public key algorithm, the third one concerning optimization of the algorithm, and final one regarding performance comparison of public key algorithm between the GPU implementation and the CPU implementation.

Title:

RSA Algorithm

Author:

En.wikipedia.org

Description:

RSA is an algorithm for public-key cryptography that is based on the presumed difficulty of factoring large integers, the factoring problem. RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described it in 1978. A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept

secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message.

The RSA algorithm involves three steps: key generation, encryption and decryption.

Key generation

RSA involves a **public key** and a **private key**. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. The keys for the RSA algorithm are generated the following way:

1. Choose two distinct prime numbers p and q .
 - For security purposes, the integers p and q should be chosen at random, and should be of similar bit-length. Prime integers can be efficiently found using a primality test.
2. Compute $n = pq$.
 - n is used as the modulus for both the public and private keys
3. Compute $\phi(n) = (p-1)(q-1)$, where ϕ is Euler's totient function.
4. Choose an integer e such that $1 < e < \phi(n)$ and greatest common denominator of $(e, \phi(n)) = 1$, i.e. e and $\phi(n)$ are coprime.
 - e is released as the public key exponent.
 - e having a short bit-length and small Hamming weight results in more efficient encryption - most commonly $0x10001 = 65537$. However, small values of e (such as 3) have been shown to be less secure in some settings.
5. Determine $d = e^{-1} \bmod \phi(n)$; i.e. d is the multiplicative inverse of $e \bmod \phi(n)$.
 - This is more clearly stated as solve for d given $(d * e) \bmod \phi(n) = 1$
 - This is often computed using the extended Euclidean algorithm.
 - d is kept as the private key exponent.

The **public key** consists of the modulus n and the public (or encryption) exponent e . The **private key** consists of the modulus n and the private (or decryption) exponent d which must be kept secret.

Encryption

Alice transmits her public key (n, e) to Bob and keeps the private key secret. Bob then wishes to send message **M** to Alice.

He first turns **M** into an integer m , such that $0 < m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the cipher text c corresponding to

$$c = m^e \pmod{n}.$$

This can be done quickly using the method of exponentiation by squaring. Bob then transmits c to Alice.

Note that at least nine values of m will yield a cipher text c equal to m , But this is very unlikely to occur in practice.

Decryption

Alice can recover m from c by using her private key exponent d via computing

$$m = c^d \pmod{n}.$$

Given m , she can recover the original message **M** by reversing the padding scheme.

3.2 Literature Summary:

From the above literature survey, we come to conclusion that we are having massively parallel processors (Graphics Cards) at our disposal thus giving us great computing power. We could harness this power and utilize it in implementing many tasks in parallel. One of the applications of this could be implementation of RSA algorithm on the GPU. RSA algorithm performs a lot of modular multiplications that are slow on CPU as compared to that on GPU as GPUs have a lot of processors and hence this process could be parallelized on GPUs and we could get a performance boost in encryption and decryption of text using RSA algorithm. We can also conclude that if we perform the tasks on GPU clusters, flops per dollar ratio is low and hence GPU is far cheaper in case of computation than CPU.

4. SYSTEM DESIGN

4.1 High-Level Design:

4.1.1 For RSA Algorithm

- **USE CASE**

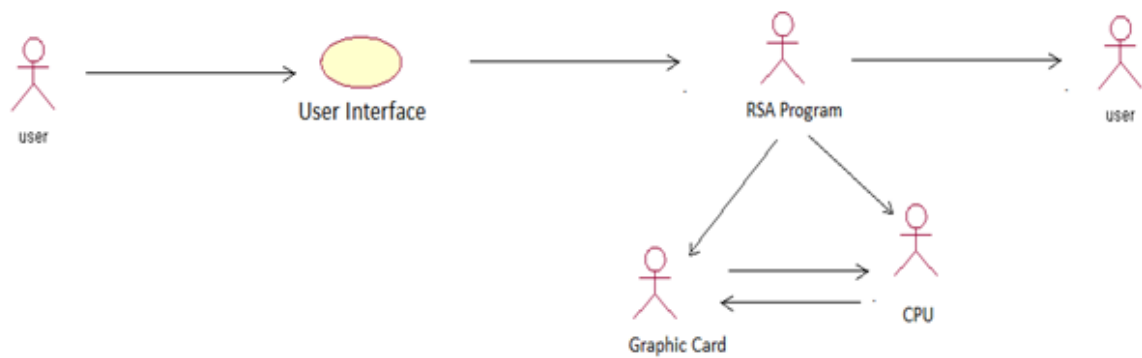


Fig 4.1.1.1 Use Case for RSA Algorithm

Here in the use case diagram, there are five characters and they are users, RSA program, CPU and GPU. Here the user can do encryption/decryption only by using the UI provided.

- **FLOW CHART**

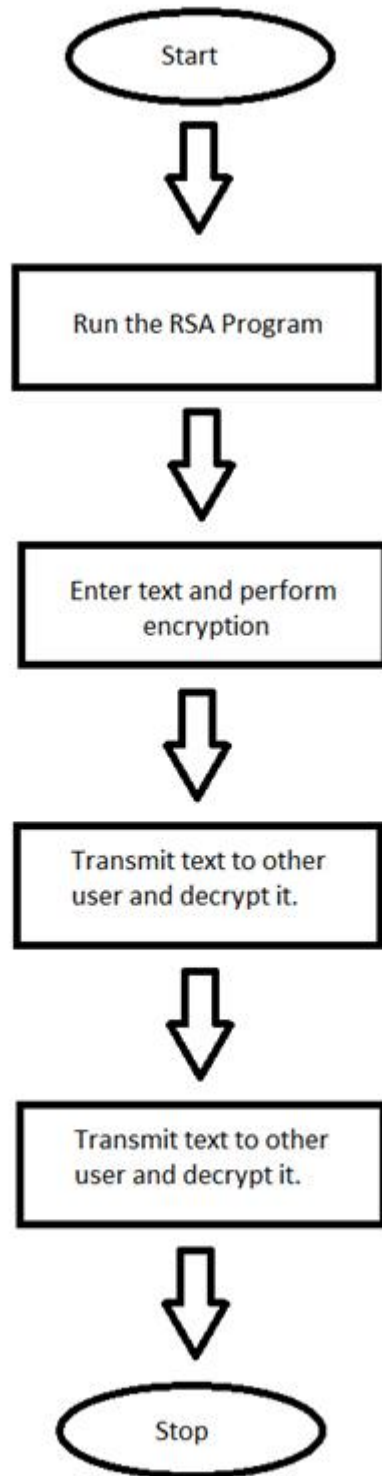


Fig 4.1.1.2 Flow Chart for RSA Algorithm

4.1.2 For Dense Matrix-Matrix Multiplication

- **USE CASE**

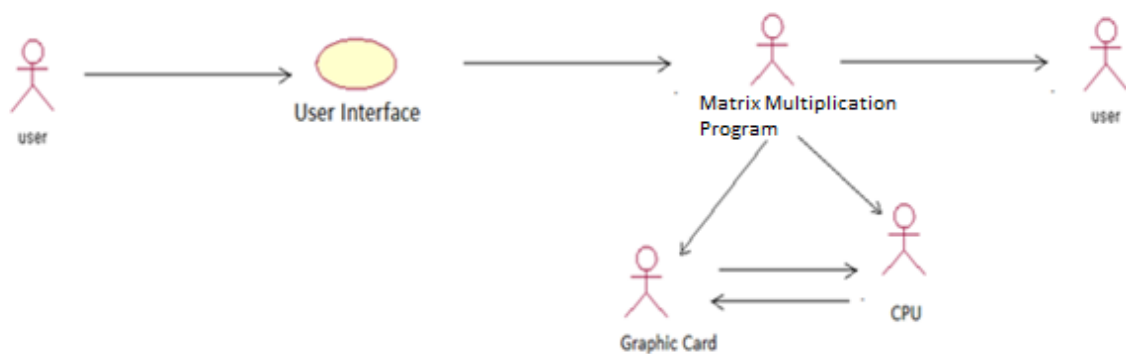


Fig 4.1.2.1 Use Case for Dense Matrix-Matrix Multiplication

Here in the use case diagram, there are five characters and they are users, Matrix Multiplication Program, CPU and GPU. Here the user can do Multiplication only by using the UI provided.

- **FLOW CHART**

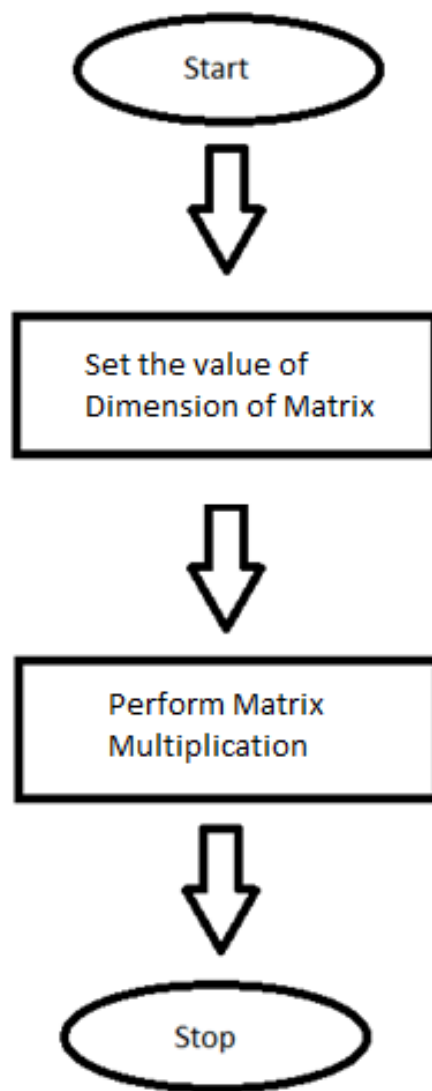


Fig 4.1.2.2 Flow chart for Dense Matrix-Matrix Multiplication

4.2 Low-Level Design:

4.2.1 For RSA Algorithm

- Start the program.
- Import the necessary packages.
- Generate two prime numbers and test for their primality.
- Calculate p, q, e, and d in the program.
- Perform Encryption and show the encrypted cipher text.
- After that in order to demonstrate the decryption using RSA, apply decryption algorithm and decrypt the cipher text and show the result to the user.
- End the program.

4.2.2 For Dense Matrix-Matrix Multiplication Algorithm

- Start the program.
- Import the necessary packages.
- Get input for the Dimension of the Matrix.
- Perform Matrix Multiplication
- End the program.

4.3 Test Cases Generation

4.3.1 For RSA Algorithm

Table 4.3.1.1 Test Cases for RSA Encryption and Decryption

Msg Size	CPU(Encryption) (Time taken in ms)	GPU(Encryption) (Time taken in ms)	CPU(Decryption) (Time taken in ms)	GPU(Decryption) (Time taken in ms)
100	16	30.56	65	408.5
200	31	33.09	126	427.67
300	47	34.45	189	440.82
400	63	34.45	254	440.83
500	79	34.46	314	442.23
600	93	34.46	382	452.07
700	109	34.46	438	452.1
800	127	34.41	504	462.68
900	142	34.44	565	462.89
1000	160	34.46	681	463.18
1100	171	34.46	706	475.1
1200	188	34.45	758	475.31
1300	209	34.48	819	488.3

1400	221	34.46	889	489.94
1500	237	34.47	944	490.9
1600	249	34.46	1006	508.36
1700	291	34.46	1071	508.962
1800	284	34.44	1129	527.77
1900	299	34.47	1197	532.64
2000	314	34.44	1257	532.96
2100	326	34.45	1320	533.23
2200	348	34.45	1379	532.88
2300	361	34.44	1467	532.86
2400	374	34.45	1507	533.08
2500	393	34.46	1568	533.36
2600	413	34.46	1637	532.9
2700	420	34.48	1695	532.85
2800	435	34.48	1758	532.92
2900	452	34.52	1823	532.97
3000	478	34.45	1917	532.9
3100	487	34.45	1948	533.13
3200	498	33.42	2020	530.71
3300	513	33.42	2070	531.1
3400	530	33.39	2136	530.78
3500	547	33.4	2202	530.74
3600	564	33.38	2259	531.06
3700	581	33.42	2387	530.74
3800	594	33.4	2406	531.02
3900	608	33.36	2448	531.09
4000	624	35.2	2517	534.75
4500	701	68.72	2827	671.2
5000	780	69.18	3140	671.64
5500	859	70.19	3454	671.12
6000	938	68.75	3772	670.27
6500	1016	68.77	4085	672.17
7000	1096	68.77	4399	671.32
8000	1252	69.26	5034	670.32
9000	1411	103.47	5658	808.79
10000	1570	103.44	6289	808.01
11000	1738	104.42	6929	807.74
12000	1892	103.03	7555	808.01
13000	2052	137.28	8188	945.57
14000	2217	137.28	8823	944.59
15000	2373	137.36	9484	944.57
16000	2534	137.36	10086	947.2
17000	2705	172.99	10731	1081.64
18000	2864	172.09	11354	1083.07
19000	3030	171.65	11991	1081.54

20000	3195	173.04	12638	1083.01
21000	3352	205.85	13264	1218.1
22000	3521	205.88	13900	1219.18
23000	3683	208.52	14533	1220.01
24000	3849	205.94	15179	1220.93
25000	4020	240.07	15819	1356.27
26000	4179	240.51	16455	1357.09
27000	4352	240.21	17089	1356.57
28000	4514	240.23	17723	1357.14
29000	4688	274.37	18363	1493.41
30000	4857	274.84	19002	1493.89
31000	5023	275.03	19659	1495.64
32000	5194	274.51	20287	1493.88
33000	5366	308.7	20933	1925.55
34000	5557	309.151	21631	1965.71
34500	5622	308.68	21904	1999

4.3.2 For Dense Matrix-Matrix Multiplication Algorithm

Table 4.3.2.1 Test Cases for Dense Matrix-Matrix Multiplication

Matrix dimension	CPU (Time taken in ms)	GPU (Time taken in ms)
50	4	0.292
100	50	1.109
150	109	3.742
200	271	6.917
250	512	17.132
300	893	26.009
350	1410	42.822
400	2106	38.749
450	3152	96.954
500	4265	138.614
550	6499	179.884
600	8555	189.32
650	11369	288.29
700	14424	332.938
750	17665	455.066
800	21347	316.723
850	26511	662.75
900	31266	754.819
950	37201	908.077

1000		43749		906.041
1050		51895		1264.69

Table 4.3.2.2 Test Cases for Dense Matrix-Matrix Multiplication (GPU only)

Matrix Dimension	GPU (Time Taken in ms)
50	0.292
100	1.109
150	3.742
200	6.917
250	17.132
300	26.009
350	42.822
400	38.749
450	96.954
500	138.614
550	179.884
600	189.32
650	288.29
700	332.938
750	455.066
800	316.723
850	662.75
900	754.819
950	908.077
1000	906.041
1050	1264.69
1100	1308.25
1200	1025.76
1300	2593.94
1400	2562.16
1500	2701.15
2000	2698.79
2500	2815.94
3000	2595.73
3500	2917.93
4000	2655.18
4500	2602.11
5000	2766.64
5500	2659.58

5. System Implementation

5.1 Description on the software used:

The various softwares and SDKs used in implementing the project are as follows

- a) Nvidia CUDA toolkit v4.0.17 (32 Bit)
- b) Nvidia GPU computing SDK v4.0.17 (32 Bit)
- c) Nvidia Parallel NSight (32 Bit)
- d) Microsoft Visual Studio 2010 Ultimate

The description of above mentioned softwares and SDKs are as follows

- a) **Nvidia CUDA toolkit v4.0.17**: The NVIDIA CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU accelerated applications. The CUDA Toolkit includes a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of the applications. We also find programming guides, user manuals, API reference, and other documentation. It comprises of CUDA C/C++ compiler, GPU Debugging and Profiling tools, GPU-accelerated math libraries, and GPU accelerated performance primitives.

Key Features

- CUDA Libraries
 - cuFFT, cuBLAS, cuSPARSE, cuRAND, NPP, Thrust
 - Development Tools
 - NVIDIA CUDA C/C++ Compiler (NVCC)
 - Visual Profiler, CUDA-GDB Debugger, CUDA-MEMCHECK
 - Support for Windows, Linux and MacOS
- b) **Nvidia GPU computing SDK v4.0.17**: The NVIDIA GPU computing SDK provides various math libraries such as CUBLAS for performing various algebraic operations. Matrix-Matrix Multiplication, Matrix-vector multiplication, and Vector-vector multiplications are provided in this library by Nvidia.
 - c) **Nvidia Parallel NSight**: NVIDIA Parallel Nsight brings GPU Computing into Microsoft Visual Studio. We can build, Debug, Profile and Trace heterogeneous compute and graphics applications using CUDA C/C++, OpenCL, DirectCompute, Direct3D, and OpenGL.

- d) **Microsoft Visual Studio 2010**: Microsoft Visual Studio is a powerful IDE that ensures quality code throughout the entire application lifecycle, from design to deployment. This IDE is used in doing coding for the project.

5.2 Description on Methods/functions used:

There are various programs written in this project and a comparison of performance of them over CPU and GPU is done. Hence each algorithm contains an implementation that uses only CPU as well as implementation that uses only GPU. All the functions used in every implementation is described below:

- a) CPU implementation of RSA Algorithm
- **The *display()* function**: This function is used by the program to display the computed values of n , p , q , ϕ , e and d used by the RSA Algorithm.
 - **The *generatePandQ()* function**: This function is used by the program to compute two randomly generated prime numbers p and q that are used in various computations in RSA algorithm.
 - **The *generateEandD()* function**: This function is written to calculate the values of *public key* e and *private key* d that is used in performing encryption and decryption of the text.
 - **The *extEuclidean()* function**: This function is used to find the greatest common divisor of two numbers. Used in calculating e from ϕ and d from e and ϕ .
 - **The *gcd()* function**: This function is used by *extEuclidean()* function in calculating greatest common divisor.
 - **The *isPrime()* function**: This function is used to test the primality of the randomly generated number.
 - **The *encryption()* function**: This function is used to perform the encryption of the text.
 - **The *decryption()* function**: This function is used to perform the decryption of the text.
 - **The *double diffclock()* function**: This function is used to calculate the difference between two times.
 - **The *chargen()* function**: This function is used to generate random characters in message to be used in RSA encryption.
 - **The *main()* function**: This function is used to perform the execution of program and call all the user defined functions.

b) GPU implementation of RSA Algorithm

- **The *display()* function:** This function is used by the program to display the computed values of n , p , q , ϕ , e and d used by the RSA Algorithm.
- **The *generatePandQ()* function:** This function is used by the program to compute two randomly generated prime numbers p and q that are used in various computations in RSA algorithm.
- **The *generateEandD()* function:** This function is written to calculate the values of *public key* e and *private key* d that is used in performing encryption and decryption of the text.
- **The *extEuclidean()* function:** This function is used to find the greatest common divisor of two numbers. Used in calculating e from ϕ and d from e and ϕ .
- **The *gcd()* function:** This function is used by *extEuclidean()* function in calculating greatest common divisor.
- **The *isPrime()* function:** This function is used to test the primality of the randomly generated number.
- **The *modular_mult()* function:** This function is used to perform the encryption/decryption of the text using GPU.
- **The *chargen()* function:** This function is used to generate random characters in message to be used in RSA encryption.
- **The *main()* function:** This function is used to perform the execution of program and call all the user defined functions.

c) CPU implementation of matrix multiplication:

- **The *main()* function:** This function first generates two matrices of given dimensions, stores it in an array, and then reads it and finally perform the matrix multiplication over CPU.
- **The *double diffclock()* function:** This function is used to calculate the difference between two times.

d) GPU implementation of matrix multiplication:

- **The *MatMul()* function:** This function is used to transfer the matrices from CPU to the memory of GPU, and then after the completion of computation again transfer result from GPU to CPU and free the memories allocated in GPU.
- **The *MatMulKernel()* function :** This function is used to perform matrix multiplication of two matrices on GPU in a parallel manner. It creates a thread for each element for multiplication on GPU thereby gaining massive parallelism in matrix multiplication.
- **The *main()* function:** This function first generates two matrices of given dimensions, stores it in an array, and then reads it and finally perform the matrix multiplication over GPU.

5.3 Module wise implementation description:

- a) **Module 1:** This module is the CPU implementation of RSA Algorithm. In this module, the RSA algorithm is implemented using C++ that utilizes CPU for encryption and decryption of text. It involves generation of pseudo prime numbers for calculating phi for the RSA Algorithm. Using the value of p , q , and ϕ , the public key 'e' and the private key 'd' is calculated for encryption and decryption of the text respectively.
- b) **Module 2:** This module is the GPU implementation of RSA Algorithm. In this module, the RSA algorithm is implemented using C++ that utilizes GPU for encryption and decryption of text. It involves generation of pseudo prime numbers for calculating phi for the RSA Algorithm. Using the value of p , q , and ϕ , the public key 'e' and the private key 'd' is calculated for encryption and decryption of the text respectively. The encryption and decryption of text requires modular multiplication. In this module, the modular multiplication is performed over Graphics Processing Unit (GPU).
- c) **Module 3:** In this module, the implementation of multiplication of matrices of given dimension is done that utilizes CPU for performing computations. Here two matrices that contain randomly generated floating point numbers are generated and are used by CPU in order to calculate the Matrix Multiplication of both the matrices.
- d) **Module 4:** The module 4 comprises of implementation of matrices of given dimension that utilizes GPU for performing the matrix computations in a parallel manner. Each thread on GPU is used to compute the value of single element. Here two matrices that contain randomly generated floating point numbers are generated. The matrices are then transferred to the GPU memory for computation, the calculations are carried out, and the results are then sent again to CPU. Here it gets stored in a matrix.

5.4 Code

5.4.1 Code for CPU implementation of RSA Algorithm:

```
/*
This algorithm can encrypt a message < n where n is product of two prime
numbers....
*/
#include<.....> //Contains all the include files used
#define CHAR_GEN 4500
using namespace std;
long long int n,p,q,m,phi,e,d;
char message[100000];

void generatePandQ();
int isPrime(long long int&);
void display();
void generateEandD();
double diffclock(clock_t clock1,clock_t clock2)
{
    double diffticks=clock1-clock2;
    double diffms=(diffticks*1000)/CLOCKS_PER_SEC;
    return diffms;
}
long long int gcd(long long int , long long int );
void extEuclidean(long long int , long long int , long long int &, long long int
&);
long long int encryption(long long int );
long long int decryption(long long int );

void display()
{
    cout << "n =\t" << n << endl;
    cout << "p =\t" << p << endl;
    cout << "q =\t" << q << endl;
    cout << "phi =\t" << phi << endl;
    cout << "e =\t" << e << endl;
    cout << "d =\t" << d << endl;
}

void generatePandQ()
{
    //algorithm for generating p and q used in the RSA Algorithm
}
```

```

void generateEandD()
{
    //algorithm for generating public and private keys e and d used in the RSA
    Algorithm
}

void extEuclidean(long long int a,long long int b, long long int &lastx, long
long int &lasty)
{
    //algorithm for finding gcd
}

long long int gcd( long long int a, long long int b)
{
    //algorithm for finding gcd
}

int isPrime( long long int &x)
{
    long long int lim,i;
    if(x % 2 == 0)
        return 0;
    lim = (long long int)sqrt((double)x);
    for(i = 3; i <= lim ; i += 2)
        if( x % i == 0)
            return 0;
    return 1;
}

long long int encryption( long long int msg)
{
    //encryption algorithm
}

long long int decryption(long long int e_msg)
{
    //decryption algorithm
}

void chargen()
{
    int i;
    int ch=65;
    for(i=0;i<CHAR_GEN;i++)
    {
        message[i]=(char)ch;
        ch++;
        if(ch==123)
        {

```

```

        ch=65;
    }
}
message[i]='\0';
}

int main()
{
    int i;
    long long int e_msg=0,d_msg=0,*emsg,*dmsg;
    generatePandQ();
    generateEandD();
    display();
    chargen();
    emsg = new long long int[strlen(message)];
    dmsg = new long long int[strlen(message)];
    cout<<"Message size used : "<<CHAR_GEN<<endl;
    clock_t begin=clock();
    for(i=0;i<strlen(message);i++)
    {
        e_msg = encryption((long long int)message[i]);
        emsg[i]=e_msg;
    }
    clock_t end=clock();
    cout << "Encryption Time : " << double(diffclock(end,begin)) << "
ms"<< endl;
    begin=clock();
    for(i=0;i<strlen(message);i++)
    {
        d_msg = decryption((long long int)emsg[i]);
        dmsg[i]=d_msg;
    }
    end=clock();
    cout << "Decryption Time : " << double(diffclock(end,begin)) << "
ms"<< endl;
    getch();
    return 0;
}

```

5.4.2 Code for GPU implementation of RSA Algorithm:

```

#include<.....> //Contains all the include files used
#define CHAR_GEN 34500
using namespace std;
char message[100000];
long long int n,p,q,phi,e,d;
long long int *emsg,*dmsg,*emsg_d,*dmsg_d;
long long int size;

```

```

void generatePandQ();
int isPrime(long long int&);
void display();
void generateEandD();
long long int gcd(long long int , long long int );
void extEuclidean(long long int , long long int , long long int &, long long int &);
long long int encryption(long long int );
long long int decryption(long long int );
__global__ void modular_mult(long long int size,long long int* emsg_d,int n_d,int
e_d)
{
    //algorithm for performing modular multiplication
}

```

```

void display()
{
    cout << "n =\t" << n << endl;
    cout << "p =\t" << p << endl;
    cout << "q =\t" << q << endl;
    cout << "phi =\t" << phi << endl;
    cout << "e =\t" << e << endl;
    cout << "d =\t" << d << endl;
}

```

```

void generatePandQ()
{
    //algorithm for generating two random prime numbers
}

```

```

void generateEandD()
{
    //algorithm for generating public key and private key for RSA encryption
}

```

```

void extEuclidean(long long int a,long long int b, long long int &lastx, long long int
&lasty)
{
    //algorithm for finding GCD
}

```

```

long long int gcd( long long int a, long long int b)
{
    long long int temp = 0;
    while(b != 0)
    {
        temp = a;

```

```

        a = b;
        b = temp % b;
    }
    return a;
}
int isPrime( long long int &x)
{
    long long int lim,i;
    if(x % 2 == 0)
        return 0;
    lim = (long long int)sqrt((double)x);
    for(i = 3; i <= lim ; i += 2)
        if( x % i == 0)
            return 0;
    return 1;
}
void chargen()
{
    long long int i;
    int ch=65;
    for(i=0;i<CHAR_GEN;i++)
    {
        message[i]=(char)ch;
        ch++;
        if(ch==123)
        {
            ch=65;
        }
    }
    message[i]='\0';
}
int main()
{
    long long int m=1;
    long long int i,N=50000;

    generatePandQ();
    generateEandD();
    display();
    chargen();
    cout<<"Message Size : "<<CHAR_GEN<<endl;
    size=strlen(message);
    emsg = new long long int[size];
    dmsg = new long long int[size];
    //Algorithm for kernel invocation
    unsigned int timer_rsa=0;
    unsigned int timer_rsa_dec=0;
    cutilCheckError(cutCreateTimer(&timer_rsa));
}

```

```

        cutilCheckError(cutStartTimer(timer_rsa));
modular_mult<<<blocksPerGrid, threadsPerBlock>>>(size,msg_d,(int)n,(int)e);
        cudaDeviceSynchronize();
        cutilSafeCall( cudaMemcpy(msg, msg_d, size, cudaMemcpyDeviceToHost)
);
        cutilCheckError(cutStopTimer(timer_rsa));
        double dSeconds = cutGetTimerValue(timer_rsa)/((double)1 * 1000.0);
        cout<<"Encryption Time = "<<dSeconds*1000<<"ms"<<endl;
        for(i=0;i<size;i++)
        {
                dmsg[i]=msg[i];
        }
        cutilCheckError(cutCreateTimer(&timer_rsa_dec));
        cutilCheckError(cutStartTimer(timer_rsa_dec));
        cutilSafeCall( cudaMemcpy(dmsg_d, dmsg, size, cudaMemcpyHostToDevice)
);
        modular_mult<<<blocksPerGrid,
threadsPerBlock>>>(size,dmsg_d,(int)n,(int)d);
        cudaDeviceSynchronize();
        cutilSafeCall( cudaMemcpy(dmsg, dmsg_d, size, cudaMemcpyDeviceToHost)
);
        cutilCheckError(cutStopTimer(timer_rsa_dec));
        dSeconds = cutGetTimerValue(timer_rsa_dec)/((double)1 * 1000.0);
        cout<<"Decryption Time = "<<dSeconds*1000<<"ms"<<endl;
        getch();
        return 0;
}

```

5.4.3 Code for CPU implementation of Dense Matrix-Matrix Multiplication:

```

#include<.....> //Contains all the include files used
using namespace std;
int max_mul=1050;
double diffclock(clock_t clock1,clock_t clock2)
{
        double diffticks=clock1-clock2;
        double diffms=(diffticks*1000)/CLOCKS_PER_SEC;
        return diffms;
}
int main()
{
        float *a,*b,*c;
        long long int i,j,k;
        a=(float*)malloc(max_mul*max_mul*sizeof(float));
        b=(float*)malloc(max_mul*max_mul*sizeof(float));

```

```

        c=(float*)malloc(max_mul*max_mul*sizeof(float));
for(i=0;i<max_mul;i++)
{
    for(j=0;j<max_mul;j++)
    {
        a[i * max_mul + j]=(rand() / (float)RAND_MAX);
        b[i * max_mul + j]=(rand() / (float)RAND_MAX);
    }
}
clock_t begin=clock();
for(i=0;i<max_mul;i++)
{
    for(j=0;j<max_mul;j++)
    {
        c[i * max_mul + j]=0;
        for(k=0;k<max_mul;k++)
        {
            c[i * max_mul + j]=c[i * max_mul + j]+a[i *
max_mul + k]*b[k * max_mul + j];
        }
    }
}
clock_t end=clock();
cout << "Multiplication Time : " << double(diffclock(end,begin)) << "
ms"<< endl;
    getch();
    return 0;
}

```

5.4.4 Code for GPU implementation of Dense Matrix-Matrix Multiplication:

```

#include<.....> //Contains all the include files used
#define MIN 0
#define MAX 3000
#define BLOCK_SIZE 16
using namespace std;
typedef struct
{
    int width;
    int height;
    float* elements;
} Matrix;

```

```

__global__ void MatMulKernel(const Matrix A, const Matrix B, Matrix C)
{

    //algorithm for performing matrix multiplication on parallel architecture
}

void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    //contains the code for copying data from host to device
    //Then the matrixmulkernel is invoked and result thus obtained
    //are copied to host from device
}

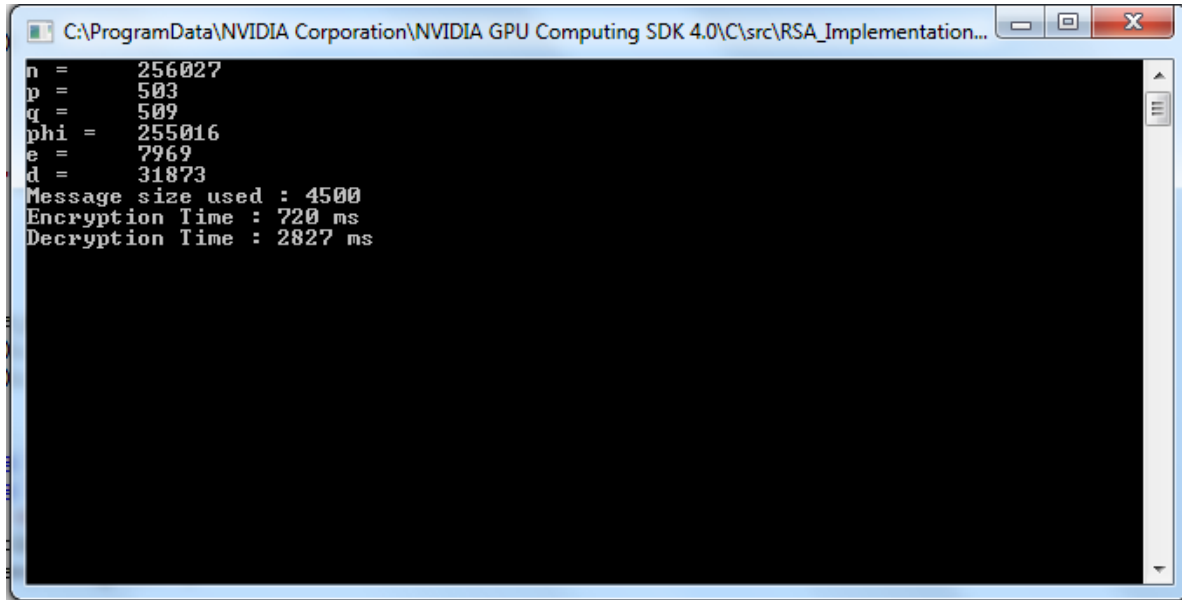
int main()
{
    Matrix l_a,l_b,l_c;
    l_a.width=l_a.height=l_b.width=l_b.height=l_c.width=l_c.height=MAX
;
    int i,j;
    l_a.elements=(float*)malloc(l_a.width*l_a.height*sizeof(float));
    l_b.elements=(float*)malloc(l_b.width*l_b.height*sizeof(float));
    l_c.elements=(float*)malloc(l_c.width*l_c.height*sizeof(float));
    for(i=MIN;i<MAX;i++)
    {
        for(j=MIN;j<MAX;j++)
        {
            l_a.elements[i*MAX+j]=(rand() /
(float)RAND_MAX);
            l_b.elements[i*MAX+j]=(rand() /
(float)RAND_MAX);
        }
    }
    MatMul(l_a,l_b,l_c);
    return 0;
}

```


6. Results and Discussion:

6.1 Output / Results:

a) Output for CPU Implementation of RSA Algorithm

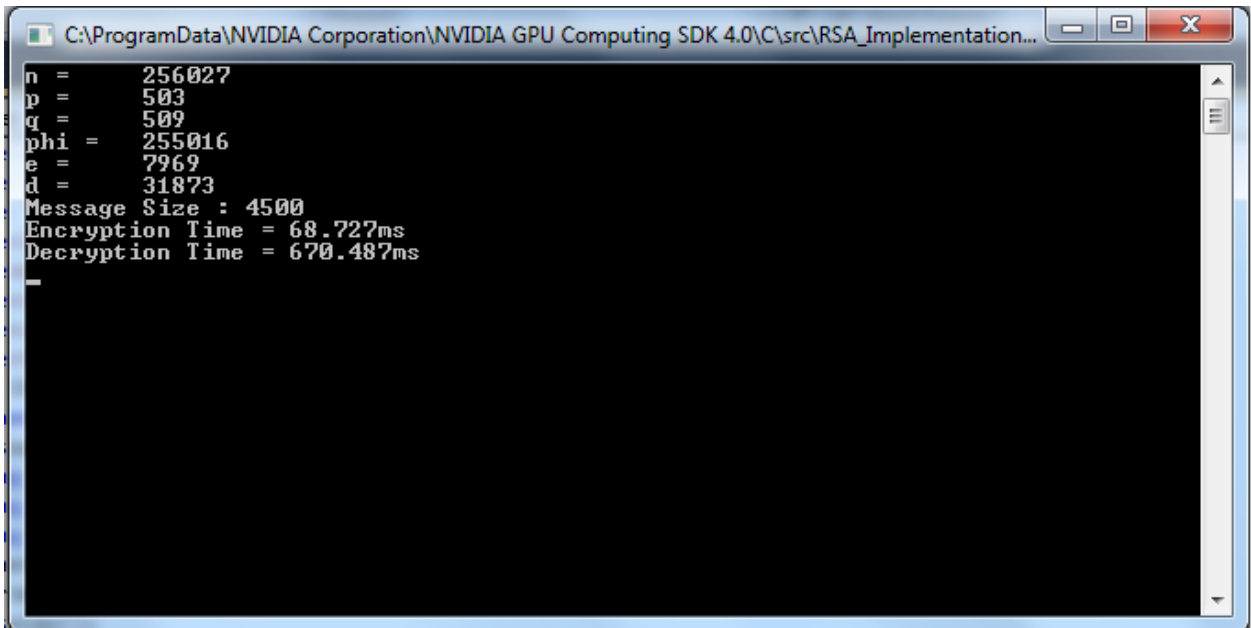
A screenshot of a terminal window with a black background and white text. The window title is "C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0\C\src\RSA_Implementation...". The output text is as follows:

```
n = 256027
p = 503
q = 509
phi = 255016
e = 7969
d = 31873
Message size used : 4500
Encryption Time : 720 ms
Decryption Time : 2827 ms
```

Fig 6.1.1 Output of RSA Algorithm implemented on CPU using message size 4500

Fig 6.1.1 shows the output of the RSA Algorithm program that is implemented on CPU. The CPU implementation of RSA Algorithm implemented using C++ utilizes CPU for encryption and decryption of text. It involves generation of pseudo prime numbers for calculating phi for the RSA Algorithm. Using the value of p, q, and phi, the public key 'e' and the private key 'd' is calculated for encryption and decryption of the text respectively.

b) Output for GPU Implementation of RSA Algorithm



```
C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0\src\RSA_Implementation...
n = 256027
p = 503
q = 509
phi = 255016
e = 7969
d = 31873
Message Size : 4500
Encryption Time = 68.727ms
Decryption Time = 670.487ms
```

Fig 6.1.2 Output of RSA Algorithm implemented on GPU using message size 4500

Fig 6.1.2 shows the output of the RSA Algorithm program that is implemented on GPU. The GPU implementation of RSA Algorithm is implemented using C++ that utilizes GPU for encryption and decryption of text. It involves generation of pseudo prime numbers for calculating phi for the RSA Algorithm. Using the value of p, q, and phi, the public key 'e' and the private key 'd' is calculated for encryption and decryption of the text respectively. The encryption and decryption of text requires modular multiplication. In this module, the modular multiplication is performed over Graphics Processing Unit (GPU).

c) Output for CPU Implementation of n-Dimension Dense Matrix-Matrix multiplication:

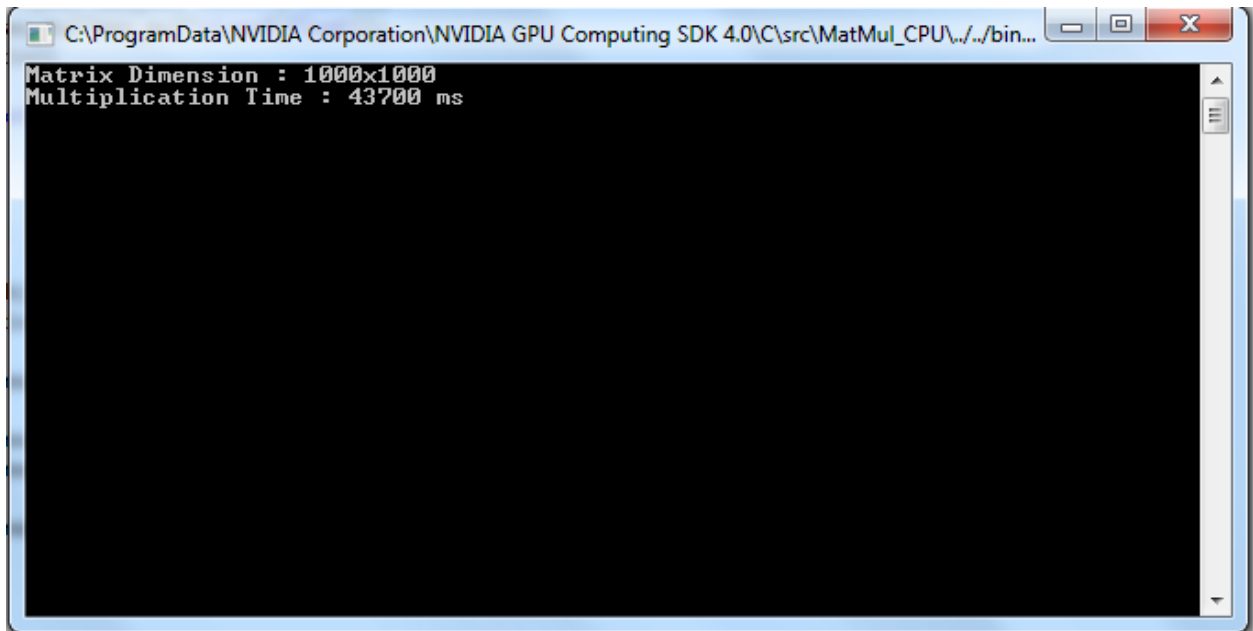
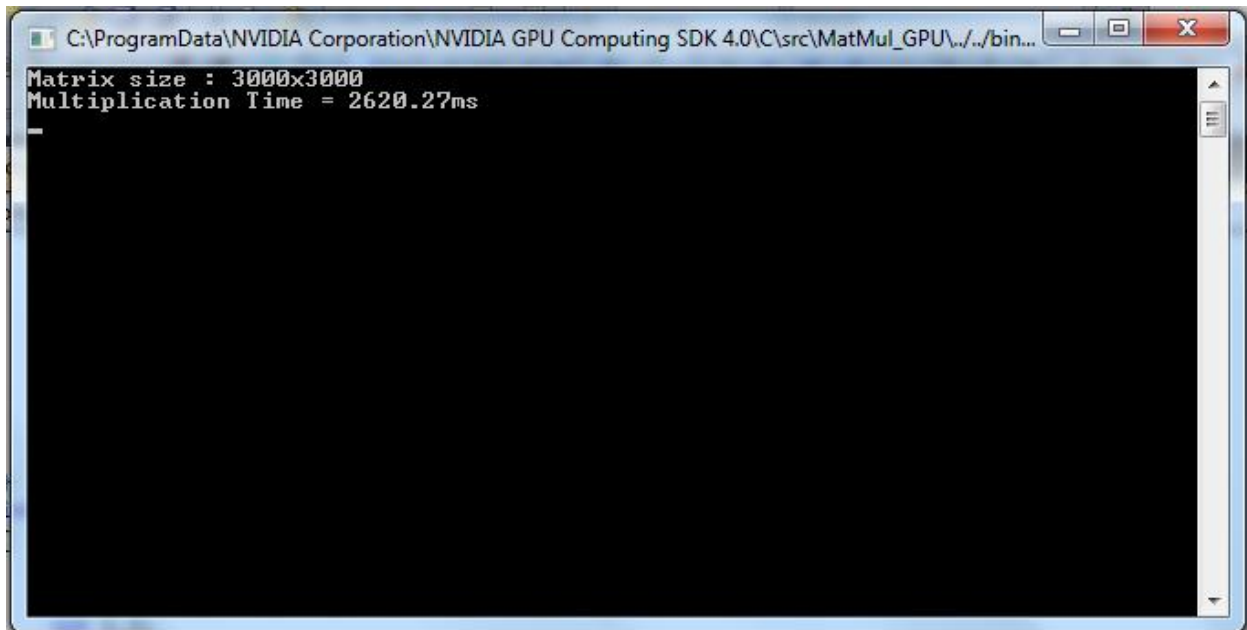


Fig 6.1.3 Output of the Dense Matrix-Matrix Multiplication implemented over CPU with Dimension 1000x1000

Fig 6.1.3 shows the output of the Dense Matrix-Matrix Multiplication program that is implemented on CPU. The Matrices are randomly generated and the time taken in performing the computation is displayed.

d) Output for GPU Implementation of 320 X 320 Matrix multiplication:



```
C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0\C\src\MatMul_GPU\..\bin...  
Matrix size : 3000x3000  
Multiplication Time = 2620.27ms
```

Fig 6.1.4 Output of the Dense Matrix-Matrix Multiplication program implemented over GPU

Fig 6.1.4 shows the output of the Dense Matrix-Matrix Multiplication program that is implemented on GPU. The Matrices are randomly generated and the time taken in performing the computation is displayed.

6.2 Result Analysis:

The results that are generated after implementing the above mentioned four modules needs to be compared on the basis of time taken by each implementation i.e. the time taken by the CPU implementation vs. the time take by GPU implementation of the same algorithm. The time taken to perform Encryption/Decryption of RSA algorithm, and also time taken for performing matrix multiplication on both CPU as well as GPU is calculated. The results thus obtained is compared to show the findings whether the CPU implementation take less time or GPU implementation of the same algorithm takes less time.

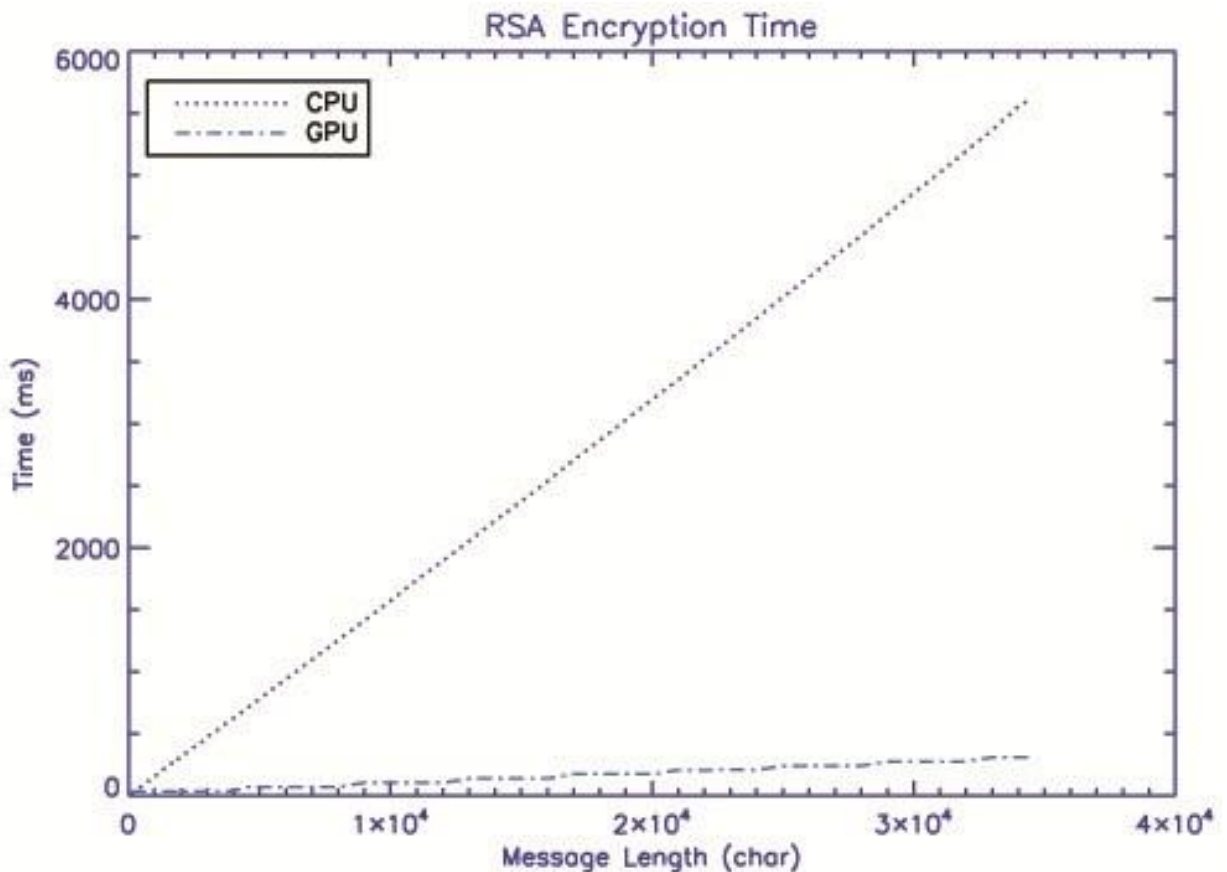


Fig 6.2.1 Graph of time taken by CPU and GPU in performing RSA Encryption

Figure 6.2.1 shows the time taken by CPU in performing encryption using RSA Algorithm with respect to that of time taken by GPU in doing the same. The number of characters in a message taken for performing the encryption is shown on x-axis and time taken in performing the encryption corresponding to the given message length is shown on y-axis. It can be inferred from the graph that initially the performance of CPU is better as compared to that of GPU when there are less number of characters in message. As the number of characters in message increases, the GPU takes less time in comparison to that with CPU in performing the Encryption using RSA Algorithm.

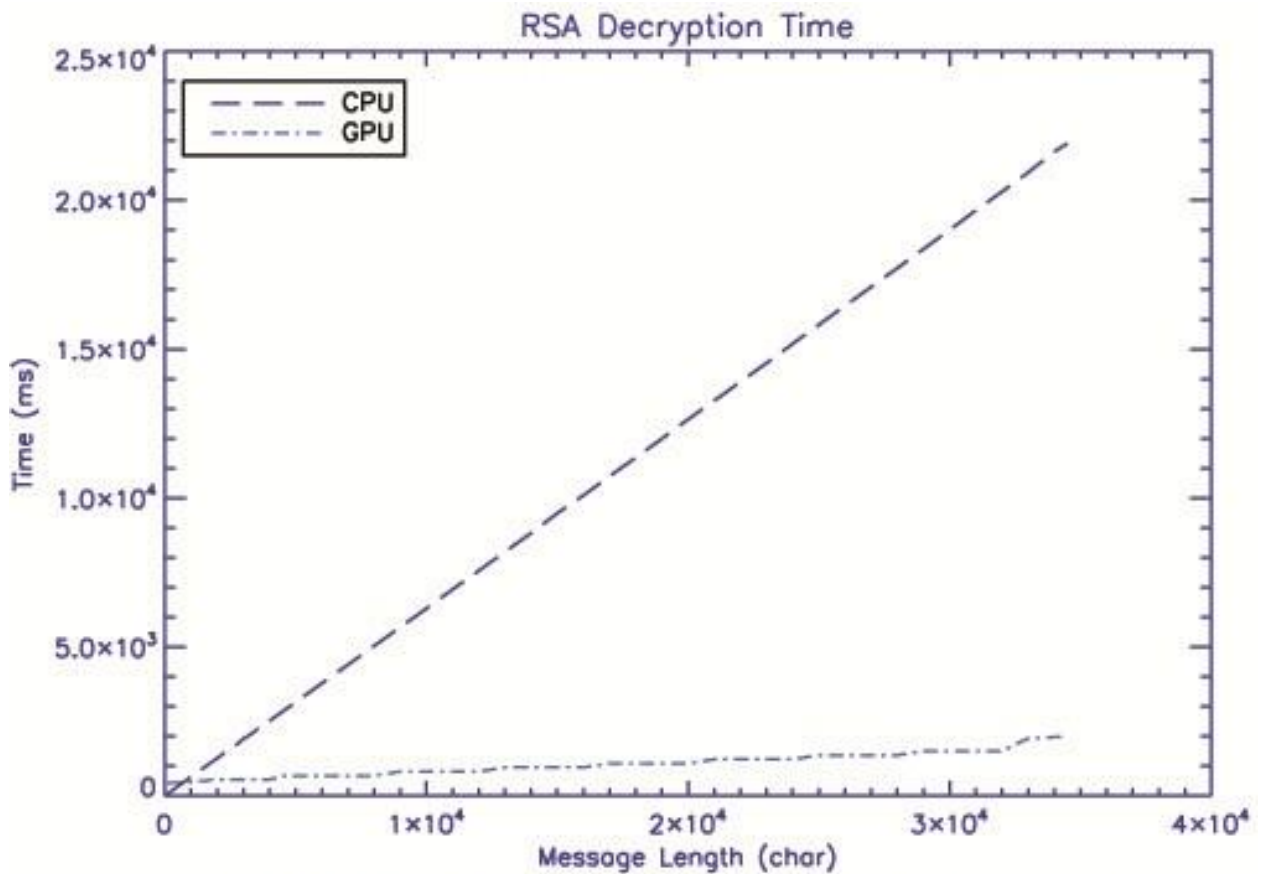


Fig 6.2.2 Graph of time taken by CPU and GPU in performing RSA Decryption

Figure 6.2.2 shows the time taken by CPU in performing decryption using RSA Algorithm with respect to that of time taken by GPU in doing the same. The number of characters in a message taken for performing the decryption is shown on x-axis and time taken in performing the decryption corresponding to the given message length is shown on y-axis. It can be inferred from the graph that initially the performance of CPU is better as compared to that of GPU when there are less number of characters in message. As the number of characters in message increases, the GPU takes less time in comparison to that with CPU in performing the Encryption using RSA Algorithm.

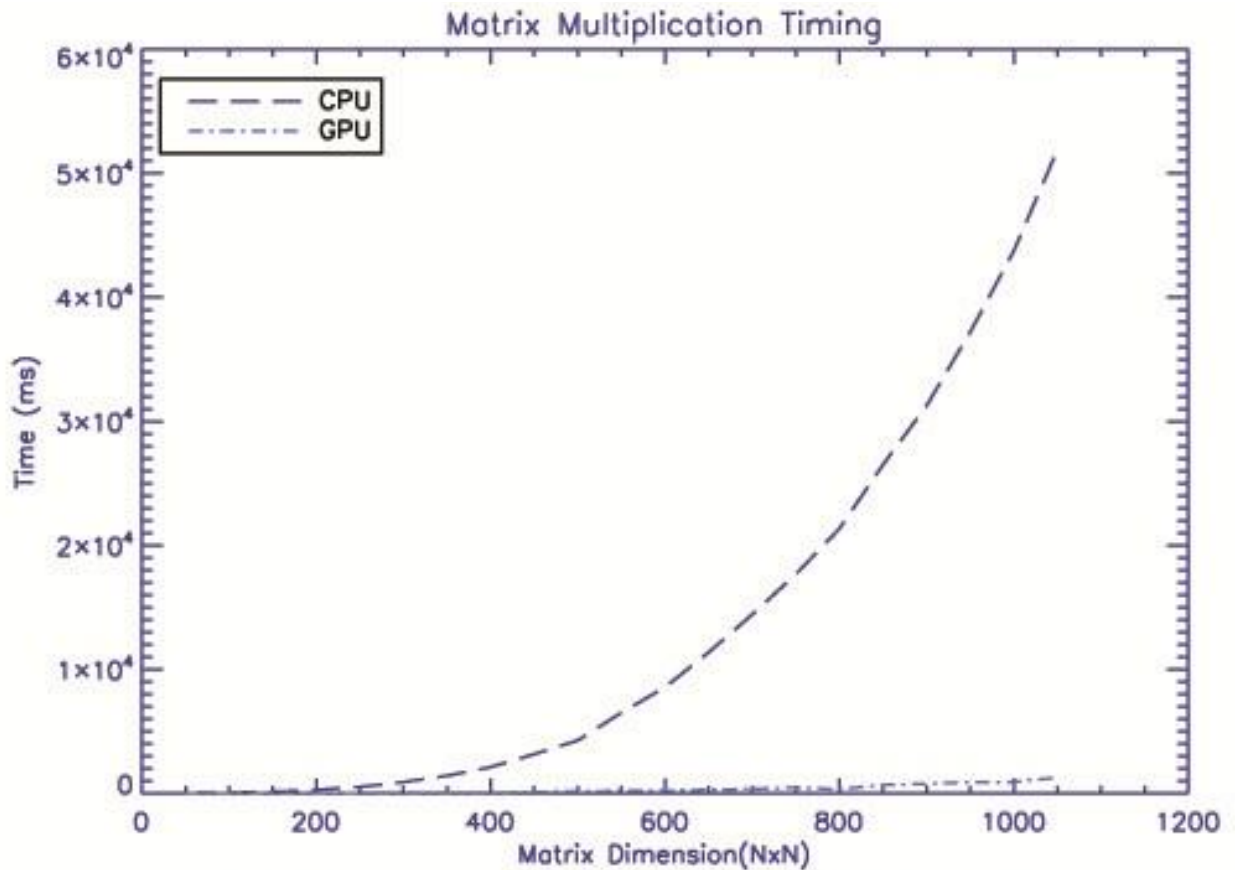


Fig 6.2.3 Graph of time taken by CPU and GPU in performing Dense Matrix-Matrix Multiplication

Figure 6.2.3 shows the time taken by CPU in performing dense Matrix-Matrix multiplication with respect to that of time taken by GPU in doing the same. The dimension of matrix taken for performing the multiplication is shown on x-axis and time taken in performing the multiplication corresponding to the given matrix dimension is shown on y-axis. It can be inferred from the graph that initially the performance of CPU is better as compared to that of GPU when there are less number of elements in matrix i.e. the dimension of the matrix is small. As the matrix dimension increases, the GPU takes less time in comparison to that with CPU in performing the multiplication.

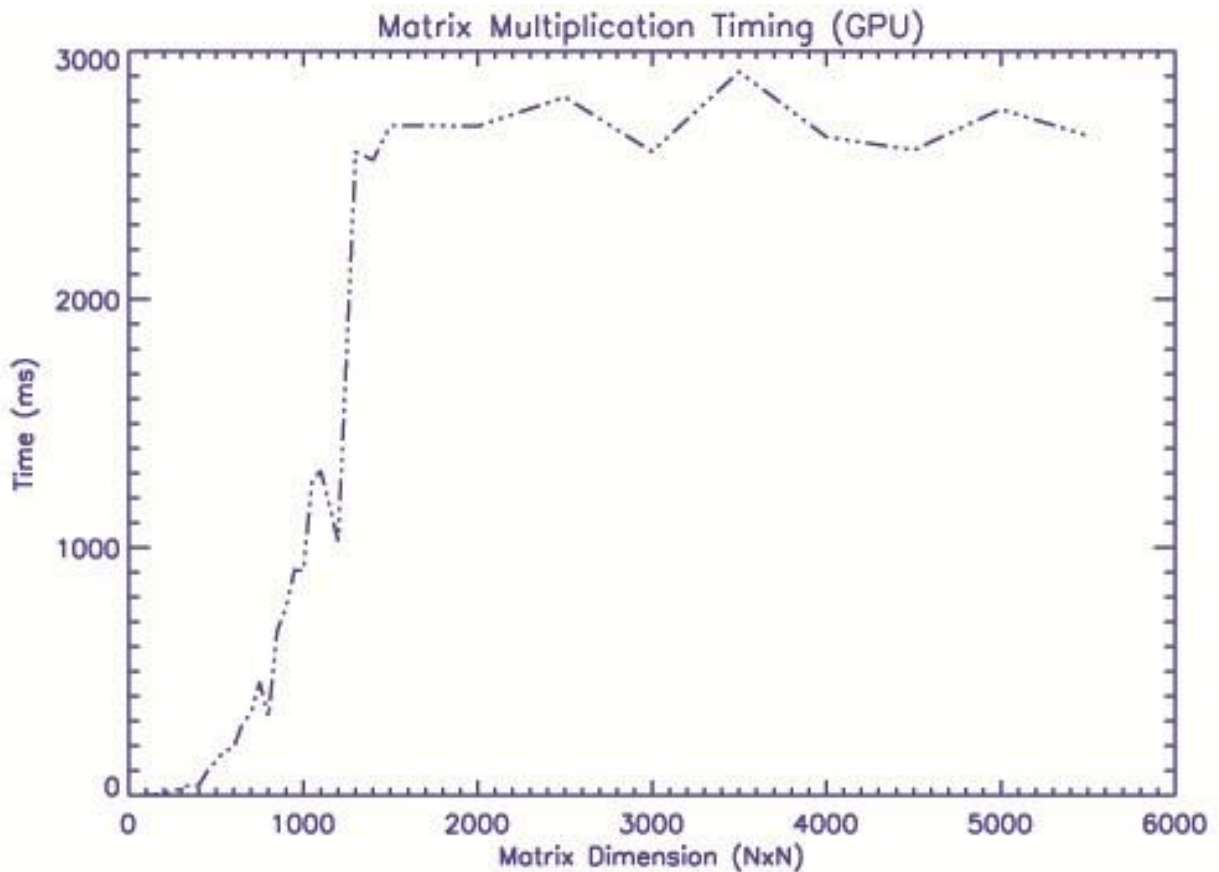


Fig 6.2.4 Graph of time taken by GPU in performing Dense Matrix-Matrix Multiplication

Figure 6.2.4 shows the time taken by GPU in performing dense Matrix-Matrix multiplication. The dimension of matrix taken for performing the multiplication is shown on x-axis and time taken in performing the multiplication corresponding to the given matrix dimension is shown on y-axis. Here the time taken by CPU is not taken into consideration as beyond the matrix dimension of 1000, the CPU starts taking time in terms of minutes which is too large as compared to time taken by GPU in performing multiplication for same matrix dimension.

6.3 Discussion:

Driven by the insatiable market demand for real time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many core processor with tremendous computational horsepower and very high memory bandwidth. It is proposed that the GPU implementation will take less time as compared to CPU implementation as GPUs available nowadays are capable of performing massively parallel floating point operations in a less time and are able to achieve speed of computation in terms of few hundred Gflops where as CPUs can perform operations in terms of Mflops.

7. Conclusion and Future Work

7.1 Conclusion:

RSA Algorithm and Dense Matrix-Matrix Multiplication on GPU using CUDA is implemented on NVIDIA Graphics Processing Unit integrated with Central Processing Unit. It is observed that the multithreading architecture and SIMD approach of CUDA helps for performance improvement in a great sense. There is tremendous difference in the results obtained on CPU and on GPU. So, CUDA programming is one of the best approaches to optimize the time for various algorithms which require huge amount of data, and is further suitable for operations which require floating point arithmetic. Therefore, GPUs can be used to run various algorithms efficiently, with their capabilities to handle floating point arithmetic and big data as well.

7.2 Scope of Future Work:

The RSA algorithm implemented in this project can further be improved by making it to work for even millions of characters. Also we can use large size of prime numbers in order to achieve higher level of security in RSA Encryption and Decryption. The Dense Matrix-Matrix multiplication implemented on GPU in this project just utilizes basic CUDA functions. They are not highly optimized. We can use CUBLAS libraries in implementing the dense Matrix-Matrix multiplication that would give a high gain in performance over the current implementation.

8. References:

Books

1. A. Menezes, P. van Oorschot, and S. Vanstone (1996) *Hand Book of applied cryptography*.
2. David B. Kirk, and Wen-mei W. Hwu (2010) *Programming Massively Parallel Processors : A hands-on approach*.
3. Gene H. Golub, and Charles F. Van Loan (2007) *Matrix Computations*.
4. Marcelo E. Kaihara (2011) *An Implementation of RSA 2048 on GPUs using CUDA*.

Conference papers

1. Jyoti B. Kulkarni, A. A. Sawant, Vandana S. Inamdar, "Database Processing by Linear Regression on GPU using CUDA", *2011, Proceedings of the ICSCCN 2011, IEEE International Conference*, Pg 20-23.

Web sites

1. RSA Algorithm, http://en.wikipedia.org/wiki/RSA_%28algorithm%29
2. CUDA C Programming Guide, Nvidia.com.
3. <http://developer.nvidia.com/cuda-downloads>
4. <http://www.microsoft.com/visualstudio/en-in>
5. Getting Started With CUDA, www.nvidia.com
6. Programming with CUDA, [www.nvidia .com](http://www.nvidia.com)

