Large Integer Arithmetic in GPU for Cryptography

BY

LEE WEN DICK

SUPERVISOR: MR LEE WAI KONG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman
in partial fulfilment of the requirements
for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology
(Perak Campus)

APRIL 2017

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: _____
_____
_____

Academic Session: _____

I _____

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

_____ _____

(Author's signature) (Supervisor's signature)

Address:

_____

_____ _____

_____ Supervisor's name

Date: _____ **Date**: _____

DECLARATION OF ORIGINALITY

I declare that this report entitled "**Large Integer Arithmetic in GPU for Cryptography**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : _____

Name : Lee Wen Dick

Date : 3-4-2017

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Mr Lee Wai Kong who has given me this oppurtunity to engage in a research and development project. It brings me passion to develop my final year project with huge interest and motivation.

ABSTRACT

Most computer nowadays support 32 bits or 64 bits of data type on various type of programming languages and they are sufficient for most use cases. However, in cryptography, the required range and precision are more than 64 bits which are computationally expensive on CPUs. In this report, we present our design and implementation of a multiple-precision integer library including basic arithmetic, Montgomery multiplication and exponentiation with parallel techniques for GPUs which is implemented using CUDA, a parallel computing platform and application programming interface model created by NVIDIA. Experimental results will be shown that a significant speedup can be achieved comparing the performance of N. Emmart and C. Weems, "Pushing the Performance Envelope of Modular Exponentiation Across Multiple Generations of GPUs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

1.1 Problem Statement and Motivation

Cryptography plays an important role in our daily life. Nowadays cryptography uses public key-encryption which involved a set of multiple-precision integer operations. A server such as SSL server that relies on public-key encryption needs to compute a large number of multiple-precision integer operation requires large computing power (Kaiyong Zhao & Xiaowen Chu 2010). Modern PCs are very good at computing numbers whose length does not exceed 32 bits or 64 bits. However, when numbers exceeded the limit, computer arithmetic will fail. Different constraints imposed by the underlying hardware architecture and programing language cause the failure (Youssef Bassil & Aziz Barbar 2004). Therefore, different algorithms and technique were developed to solve the problem of arithmetic computation on big number on the CPU as well as GPU.

Recent improvement in Graphics Processing Units (GPU) ushered a new era of GPU computing (Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E & Phillips, J. C 2008). For example, commodity GPUs like NVIDIA's GTX 1070 has 1920 processing cores and can achieve 5783 GFLOPS of computational horsepower.

We are inspired by the point that end users and application servers can employ GPUs to increase the computation speed of multiple-precision integer operations. However, there is difficulty to achieve high performance on GPUs due to the complicated memory architecture and the relatively slow integer operations. (Kaiyong Zhao & Xiaowen Chu 2010)

GPU is a type of processor that is responsible to compute computer graphic. GPUs were originally created for a high-performance workstation and therefore costly. In early 1990, 3D games with rendering processor were appearing since the 3D accelerator hardware was made. API OpenGL was basically from a graphic application of professional workstation and adopted to make a graphic 3D game programming, the same as the emergence of the DirectX and Direct3D (Khoirudin, & Shun-Liang, J 2015).

In addition, GPU became more affordable and more powerful. Recently, due to the promotion of gaming on PC, the development of GPU overtook the CPU development.

The rapid transformation on GPU allows improvement in parallel using the multiple cores. This is more effective when the programmer wants to process a lot of vertices or fragments in the similar way. With multi-core that control by very high memory bandwidth, GPU serves with incredible resources for both non-graphics processing and graphics processing at the same time.

On a modern GPU, shader number or often called the Stream Processor (for stream input and output/Stream), has reached the hundreds or even thousands. GPU calculation abilities can reach Terra FLOPS, which hundreds of times faster than the CPU. Therefore, other non-graphic calculation can be performed in GPU. The comparison between GPU and the CPU is shown in Table 1 by (Khoirudin & Shun-Liang, J 2015).

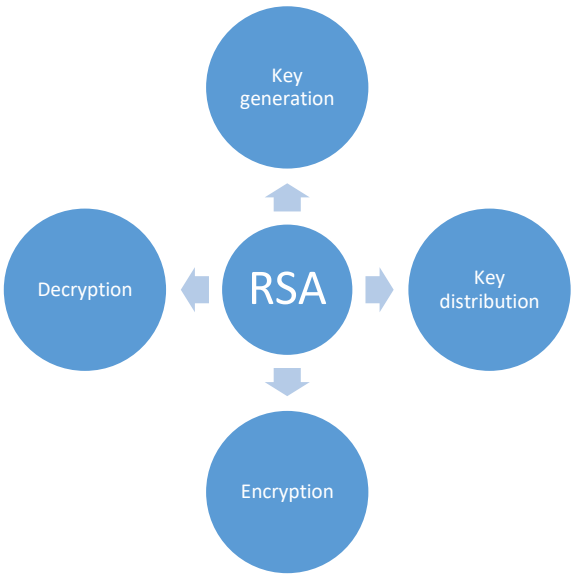
CPU	GPU
Parallelism through time multiplexing	Parallelism through space multiplexing
Emphasis on low memory latency	Emphasis on high memory throughput
Allows wide range of control flows + control flow optimisation	Very control flow restricted
Optimised for low latency access to caches data set	Optimised for data parallel, throughput computation
Very high clock speed	Mid-tempo clock speed
Peak computation capability low	Higher peak computation capability
Off-chip bandwidth lower	Higher off-chip bandwidth
Handle sequential code well	Requires massively parallel computing
Great for task parallelism	Great for data parallelism

Table 1. Comparison between CPU and GPU

As mentioned above, modern field of cryptography can be divided into Symmetric-key cryptography and Public-key cryptography. One of the practical public-key encryption system is the RSA algorithm.

RSA is one of the first practical public-key cryptosystems and is widely used for secure data transmission. Basically, RSA produces public key based on two large prime

numbers along with an auxiliary value. The RSA algorithm includes modular arithmetic as well as Montgomery modular exponentiation on the large numbers which have the key size of 1024 to 4096 bit typically. Therefore large integer arithmetic is needed in cryptography.



RSA involves a public key and a private key. The public key is distributed to the public for message encryption, and only can be decrypted by the private key. In practical, three very large positive integers *e*, *d* and *n* such that with modular exponentiation for all message *m*:

$$(m^e)^d \equiv m \ (mod \ n)$$

To encrypt the message *m* such that $0 \leq m < n$, we computes the ciphertext *c* by using the public key *e*, corresponding to

$$c \equiv m^e \ (mod \ n)$$

To recover the message, we can decrypt the encrypted message *c* using private key *d* by computing

$$c^d \equiv (m^e)^d \equiv m \ (mod \ n)$$

As we can see, the RSA encryption scheme requires heavy load of computation of modular exponentiation that are computational expensive on CPU. Therefore, we implemented Montgomery Multiplication and exponentiation on GPU to compute multiple messages in parallel to save time.

Besides, another public-key cryptosystem is Diffie-Hellman Key Exchange. It was one of the first public-key protocols as originally conceptualised by Ralph Merkle and named after Whitfield Diffie and Martin Hellman by (Diffie, W. & Hellman, M. 1976). Diffie-Hellman Key Exchange involved arithmetic exponential to compute the key which should be at least 2048 bits. Elliptic curve cryptography (ECC) is another approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields also implement Montgomery modular exponentiation as well.

Most algorithms in cryptography require expensive computation to perform encryption and decryption multiple times. Therefore, a GPU large integer arithmetic library is needed to support the operations.

1.2 Project Scope

The scope of project is to develop a library for multiple-precision integer arithmetic for GPU which provide the operation for non-negative addition, subtraction, multiplication, division, Montgomery exponentiation and multiplication in parallel technique to optimise the usage of GPU. The performance of the library will then be analysed against GMP library for CPU to compare the computational time between GPU and CPU. At last, an optimised implementation of public key cryptography algorithm will be designed based on the developed library.

1.3 Project Objective

Although there is existing libraries supporting the GPU, they are not open source. Besides, the libraries are designed to support only either host side function call or device side function call. Host side function call is when the CPU calls the function and it is passed to GPU to compute and then the result is returned to the CPU whereas device side function call is when the GPU calls the function in GPU, which is good for developing algorithms that use multiple large integer arithmetic operations. The diagram below illustrates the communication between host side and device side, from (Nigerianewsday.com 2016).

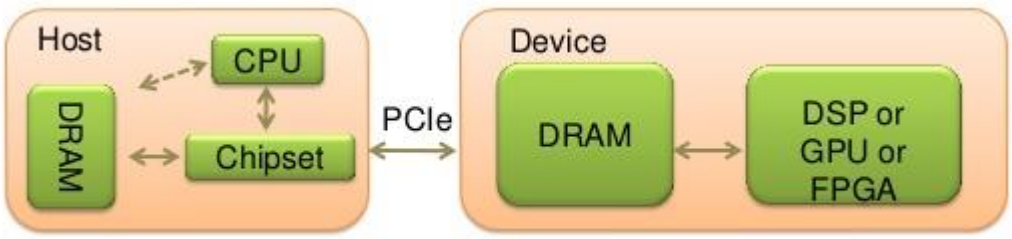


Figure 1: Host side and device side

As the libraries implements such principle, there is overheads from the communication between CPU and GPU which will slow down the computation speed. Therefore, a further improvement will be added in this project to overcome this weakness.

1.4 Impact, significance and contribution

By developing the open source library, computation time can be reduced in large number arithmetic by using GPUs, which will be useful and convenient in the field of cryptography. The challenges we need to overcome which are to override the need of overheads from communication between CPU and GPU, algorithms and code optimisation to achieve the speed. Besides, a good manual memory management for dynamic memory allocation is needed in the C programming language via a group of functions in the C standard library, namely malloc and free.

1.5 Background Information

In this library, large integer is stored and represented in a system of radix 2^{32}. In short, the large integer is represented in an array of unsigned integer, $d_0, d_1, d_2 \ldots d_n$ as $d_0b^n + d_1b^{n-1} + \ldots + d_nb^0$, where $0 \leq d_i < 2^{32}$. In such way, memory optimisation can be achieved as we can fully optimise 4 bytes of memory given in each array index.

Therefore, different parallel algorithms can be applied to perform arithmetic operations on GPU. This project will support arithmetic operations such as (modular) addition, subtraction, multiplication, division and exponentiation, Montgomery multiplication and exponentiation. By having the library equips with arithmetic operations, it enables user to perform large integer calculation for the purpose of cryptography in a fast manner.

Chapter 2: Literature Review

General Purpose Computing on Graphics Processing Units is a relatively new field. Therefore, the work on arbitrary-precision arithmetic mostly relies on CPUs. Several libraries such as Library for Efficient Data types and Algorithms (LEDA and GNU Multi-Precision library (GMP)) which explicitly target CPU hardware architectures is established. Another established library is ARPREC which itself is based on MPFUN, a multiple precision library for Fortran. Although most of them provide a huge set of different data-type and operations, our goal is to accelerate the computation speed.

In 2010, Kaiyong Zhao and Xiaowen Chu created a multiple-precision library for CUDA, the GPU Multiple-Precision library (GPUMP) in the paper (K. Zhao & X. Chu 2010). GPUMP performs its operation on integer types with arbitrary but fixed length. The functionality of CPUMP includes operations such as multiple-precision comparison, (modular) addition and subtraction, multiplication, division, Montgomery reduction, Montgomery multiplication and exponentiation. GPUMP applies sequential arithmetic algorithms in parallel. The weakness of the algorithms is when the number grows beyond the predefined length limit, it fails and become inefficient for small numbers in term of both computation time and memory usage. There is several optimisation techniques they used to improve the performance. Firstly, constant value with cache memory is used as most algorithms use the same data multiple times during calculations, since GPU can adopt cache mechanism, which achieve high reading and calculation efficiency. Besides, for temporary value, GPUMP uses shared memory on algorithms that required temporary variables as local or global memory will cause long reading latency. Their result comparing performance of GPU and CPU is as shown below.

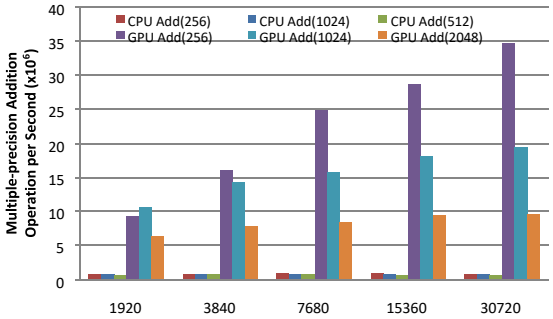


Figure 1: Multiple-precision Addition running on CPU & GPU(K. Zhao & X. Chu 2010)

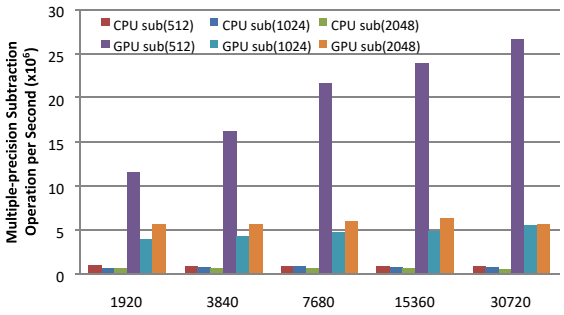


Figure 2: Multiple-precision Subtraction running on CPU & GPU(K. Zhao & X. Chu 2010)

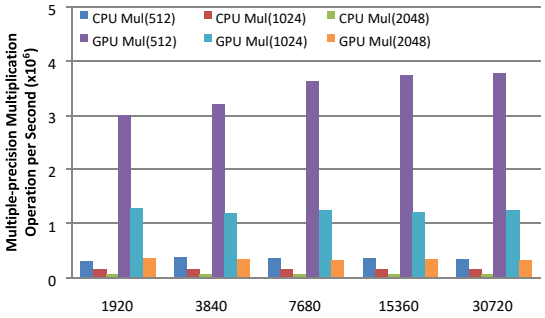


Figure 3: Multiple-precision Multiplication running on CPU & GPU(K. Zhao & X. Chu 2010)

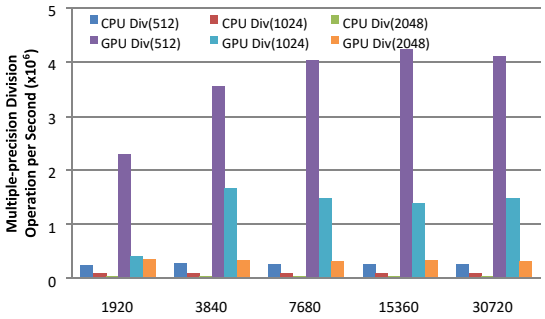


Figure 4: Multiple-precision Division running on CPU & GPU (K. Zhao & X. Chu 2010)

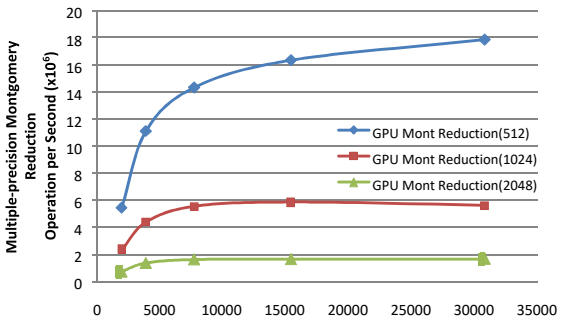


Figure 5: Multiple-precision Montgomery Reduction running on GPU (K. Zhao & X. Chu 2010)

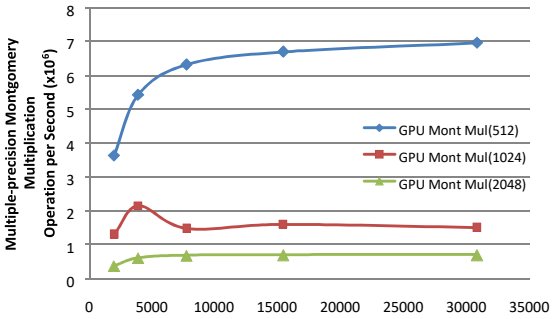


Figure 6: Multiple-precision Montgomery Multiplication running on GPU(K. Zhao & X. Chu 2010)

In 2011, Takatoshi Nakayama and Daisuke Takahashi created a multiple-precision library for floating-point number types, the CUDA Multi-Precision library (CUMP) in the paper (Takatoshi, N. & Daisuke, T. 2011). In this library, improvement is done by supporting floating-point number types which was absent in GPU Multiple-Precision library (GPUMP) by Kaiyong Zhao and Xiaowen Chu.

In 2015, Bernhard Langer implements Arbitrary-Precision Arithmetic on the GPU using CUDA in the paper (Langer, B. 2016). He represented methods to perform arbitrary-precision integer arithmetic in GPU by employing a two-level parallelisation scheme. He minimises the code divergence within SIMD units while providing effective load balancing across all units. Bernhard Langer implements school-method multiplication which has a high complexity of $O(n^2)$ slows down the computation on multiplication. To improve the complexity, Karatsuba Algorithm which has a complexity of $O(n^{\lg 3})$ can be used. Results below show the comparison between timings of code execution on the CPU against GPU with a test system of Intel Core i7 4700MQ CPU and a NVIDIA K2100M GPU, focus on multiplication.

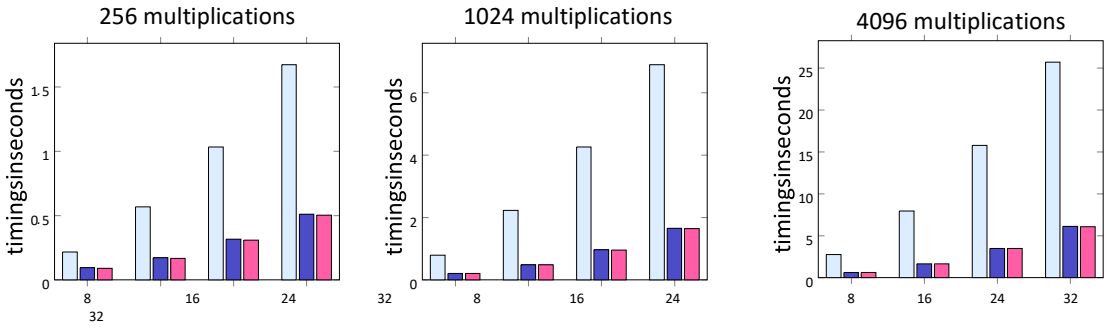


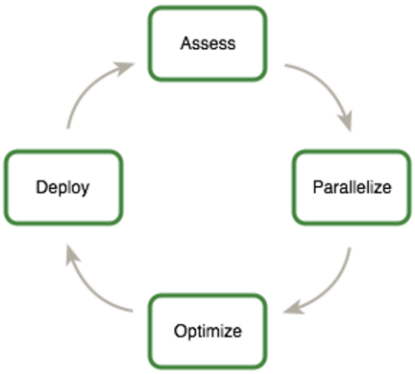
Figure 7:Integer length with σ=8 *Figure 8: Integer length with σ=8* *Figure 9: Integer length with σ=8*

(Langer, B. 2016)

Chapter 3: Proposed Method / Approach

3.1 Design Specification

3.1.1 Methodologies and General Work Procedures



We followed the Assess, Parallelize, Optimize, Deploy(APOD) design cycle, which is a cyclical process for our project with the goal to quickly tackle the sections of the code that would be optimised by GPU acceleration.

Assess

After converting the code from CPU based to GPU based, we assessed and tracked the sections of code that are responsible for the mass of the execution time. The bottlenecks for parallelisation is evaluated and start to work on GPU acceleration.

Parallelise

We implemented the code with the aids of existing GPU-optimized library such as Thrust, a C++ template library for CUDA based on the Standard Template Library (STL). Thrust allows us to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C. In most of the time, we make use of the host_vector, which is stored in host memory and device_vector, which lives in GPU device memory. Like std::vector, host_vector and device_vector are generic containers.

Optimise

After we completed code parallelization, we optimise the implementation on CUDA to improve performance. We considered many way of possible optimisation such as

Coalesced Access to Global Memory, making use of parallel library, division modulo operations with shift operations and so on.

Deploy

We took the partially parallelised implementation and carry it through to production after each round of optimisation.

3.1.2 Tools to use

The core tool to be used is NVIDIA GTX 1070 graphic card which is supported by CUDA. C/C++ compiler are needed which are readily provided by Microsoft Visual Studio 2013 Professional running on the 64-bits Windows 10 operating system. Besides, CUDA toolkit is needed to be installed on Microsoft Visual Studio to implement CUDA programming.

3.1.3 Verification Plan

To prove the correctness of calculation, we compute the calculation with Wolfram Language on Wolfram Development Platform which provided us free access in Open Cloud.

3.2 System Design / Overview

3.2.1 Algorithms

There are different type of algorithms available to implement on arithmetic operation. Unfortunately, not every algorithms can be parallelised and sometimes only some parts of the algorithm can be parallelised. Parallel addition and subtraction have been implemented. Before going into algorithms, the user-defined structure to represent the large integer is introduced below.

```
class bigInt{
        vector<unsigned int> radixForm;
}
```

bigInt class consist of a vector of unsigned integer to store the radix form of the large integer. Each index of the vector can store 32 bits of *unsigned int* from 0 up to 4,294,967,295 therefore the memory space of the vector can be fully optimised. When the *bigInt* data type is declared with an integer value, the integer value will then push into the vector in index zero, else if the value is in term of string (intended for value

that larger than 2^{32}) will be feed into an algorithm to convert the large number into radix representation of 2^{32}, which will be discussed below.

Algorithm 1: Radix Representation of large integer in base 2^{32}

Input: An integer as a string S

Output: $d_0b^0 + d_1b^1 + \ldots + d_nb^n$

1: $bigInt\ val = 0, pow = 0, base = 10$

2: **for** $i = S - 1\ to\ 0$ **do**

3: $t = s_i - \text{'0'}$

4: $val = val + t$

5: $pow = pow * base$

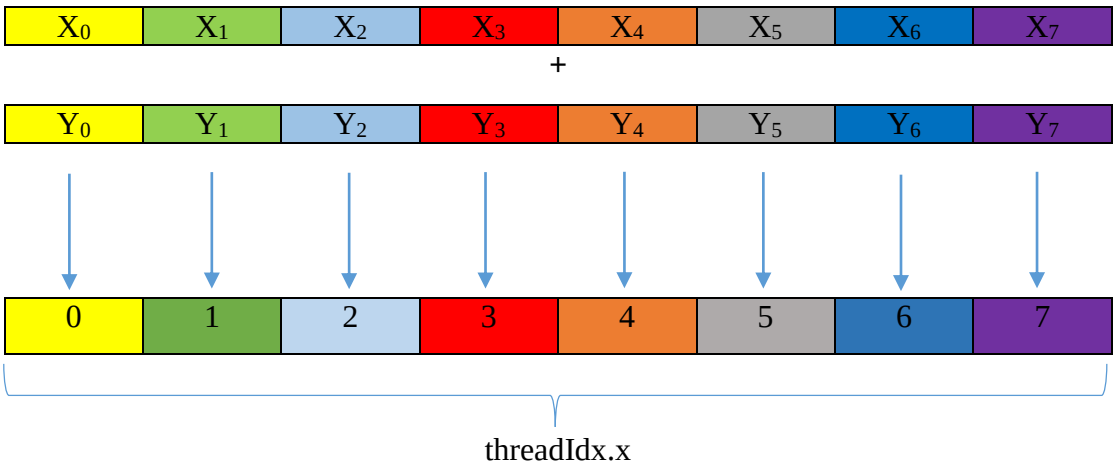
6: **end for**

Parallel Addition

Given two array representing the large integer, parallel operation can be done by adding the values in corresponding index from two array.

To illustrate the parallelism on addition on GPU architecture, we first assume that all the values in the vector had been transfer into array of memory block of GPU. Each thread in a block can be assigned some computation by using their thread ID. The diagram below illustrates the parallel technique to perform addition.

Assume both arrays X and Y consisted of radix representation of $X_0b^0 + X_1b^1 + \ldots + X_nb^n$



Each thread performs a sub-addition in parallel and store the values into the memories reserved along with their carries to transfer them back into CPU platform.

Synchronisation issues arise from carry propagation due to its sequential nature as the addition of carry to the result may propagate a new carry in some condition. Therefore, a carry propagation function is used to determine the number of carry to be added in final output. We denote three distinctive values in the array with G for generation, P for propagation and N for no carry. In each parallel operation, when two values are added and they exceeded the radix, a generation G will be marked. If the value added is equal to radix b-1, then P will be marked as it stands the chance to generate carry. Else N will be marked. A generalized associative operation $\oplus$ are determined to perform the propagation and is shown as table below.

C_{i-1}	C_i	$C_i = C_i \oplus C_{i-1}$
N	N	N
P	N	N
G	N	N
N	G	G
P	G	G
G	G	G
N	P	N
P	P	P
G	P	G

Table 1. Result for carry propagation function

It computes the resulting behaviour between pairs iteratively and give us the correct carry output to add from the addition result from parallel algorithm. We can see that C_i = P inherit the value from C_{i-1}, therefore P is the identity element.

Although the GPU threads enable addition to be performed in parallel, the carry propagation function requires O(n) time to produce correct resulting behaviour. Therefore not much improvement can be done on addition using parallel technique.

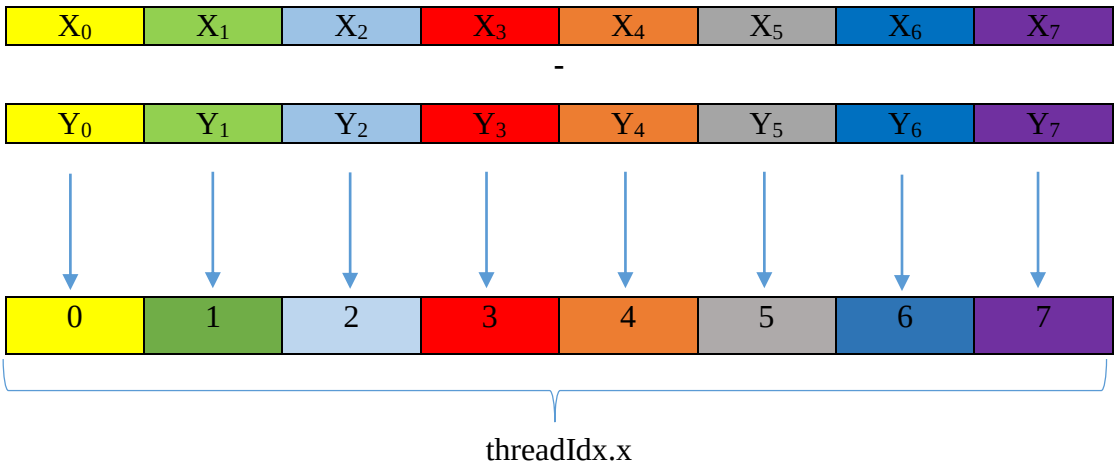
Algorithm 2: Carry propagation (GPU kernel)

1: **function** ADD(*d_lhs*, *d_rhs*, *d_carry*)
2: $idx = threadIdx.x$
3: $tempAdd = d\_lhs[idx] + d\_rhs[idx]$
4: $d\_carry[0] = -1$
5: **if** $tempAdd = 2^{32} - 1$ **then**
6: $d\_carry[idx + 1] = 0$ //P
7: **else if** $tempAdd < 2^{32} - 1$ **then**
8: $d\_carry[idx + 1] = -1$ //N
9: **else**
10: $d\_carry[idx + 1] = 1$ //G
11: $d\_carry[idx] = tempAdd \ \boldsymbol{mod}\ 2^{32}$
12: **end if**
13: **end function**

Parallel Subtraction

Given two array representing the large integer, parallel operation can be done by subtracting the values in corresponding index from two array. To illustrate the parallelism on subtraction on GPU architecture, we first assume that all the values in the vector had been transfer into array of memory block of GPU. Each thread in a block can be assigned some computation by using their thread ID. The diagram below illustrates the parallel technique to perform subtraction

Assume both arrays X and Y consisted of radix representation of $X_0b^0 + X_1b^1 + \ldots + X_nb^n$



In each thread, e.g., threadIdx.x = 0 it performs a sub-addition X_0-Y_0, while threadIdx.x = 1 performs X_0 - Y_0 and threadIdx.x = *n* performs X_n - Y_n in parallel and store the values into the memories reserved along with their carries to transfer them back into CPU platform.

Similar to parallel addition, the synchronisation issues arise from "borrowed" propagation due to its sequential nature as the subtraction of "borrowed" from result may propagate a new "borrowed" operation in some condition. Therefore, a "borrowed" propagation function is used to determine the number of borrows to be added in final output. We denote three distinctive values in the array with G for generation, P for propagation and N for no borrow. In each parallel operation, when two values are subtracted and they exceeded below 0, a generation G will be marked. If the value subtracted is equal to 0, then P will be marked as it stands the chance to generate borrow. Else N will be marked. A generalized associative operation $\oplus$ are determined to perform the propagation and is shown as table below.

C_{i-1}	C_i	$C_i = C_i \oplus C_{i-1}$
N	N	N
P	N	N
G	N	N
N	G	G
P	G	G
G	G	G
N	P	N
P	P	P
G	P	G

Table 2.. Result for borrow propagation function

It computes the resulting behaviour between pairs iteratively and give us the correct "borrowed" output to subtract from the subtraction result from parallel algorithm. We can see that C_i = P inherit the value from C_{i-1}, therefore P is the identity element.

Although the GPU threads enable subtraction to be performed in parallel, the "borrowed" propagation function requires O(n) time to produce correct resulting behaviour. Therefore not much improvement can be done on subtraction using parallel technique.

Algorithm 3: Borrow propagation (GPU kernel)

1: **function** SUB(*d_lhs, d_rhs, d_carry, d_N*)
2: $idx = threadIdx.x$
3: $tempSub = d\_lhs[idx] - d\_rhs[idx]$
4: **if** $d\_N - 1 \neq idx$ **then**
5: **if** $tempSub = 0$ **then**
6: $d\_borrow[idx + 1] = 0$ //P
7: $d\_lhs[idx] = 0$
8: **else if** $tempSub < 0$ **then**
9: $d\_borrow[idx + 1] = 1$ //G
10: $d\_lhs[idx] = 2^{32} + tempSub$
11: **else**
12: $d\_borrow[idx + 1] = -1$
13: $d\_lhs[idx] = tempSub$
14: **end if**
15: **else**
16: $d\_lhs[idx] = tempSub$
17: **end if**
18: **end function**

In final year project II, after the further study of parallel arithmetic algorithms, we realised that parallel multiplication proposed by other researches did not well optimised on the GPU as well as our implementation on parallel addition and subtraction. The reason behind it is because of insufficient data elements to be run simultaneously in parallel. This is a requirement to achieve satisfactory performance on CUDA. Besides, for two large integer to perform arithmetic in GPU, we must transfer them between the host and the device, which the process is very slow. Therefore, the parallel process is unable to compensate the time taken for data transfer in such small amount of data elements.

In the end, we moved our focus on bulk parallel computations on Montgomery multiplication and exponentiation, which on each thread of GPU will handle one sequential Montgomery multiplication. We launched as many threads as possible to

maximize the workload on GPU and determine the throughput of our result, the amount of Montgomery multiplication and exponentiation computed per second.

Therefore, the implementation of *bigInt* class become the foundation to convert a large integer in the form of string to an array of base 2^{32} for Montgomery multiplication and exponentiation. In order to perform the conversion, addition, subtraction and multiplication are needed. However, for the sake of completeness, we implemented division and some operator such as $=, <, \leq, >, \geq, \neq$ and so on to support daily operation.

The algorithms of minor operations that are not being used in radix form conversion and Montgomery multiplication and exponentiation are not being mentioned here.

Algorithm 4: Multiplication

Input: Non-negative integers *x* and *y* with *n* + 1 radix *b* digits

Output: $x \cdot y = (z_{n+s+1} z_{n+s} \dots z_1 z_0)_b$

1: **for** $i = 0$ *to* $n + s + 1$ **do**

2: $z_i = 0$

3: **end for**

4: **for** $i = 0$ *to* s **do**

5: $c = 0$

6: **for** $j = 0$ *to* n **do**

7: $(uv)_b = z_{i+j} + x_j \cdot y_i + c$

8: $z_{i+j} = v$

9: $c = u$

10: **end for**

11: $z_{i+j+1} = u$

12: **end for**

Montgomery Multiplication & Exponentiation

RSA and Diffie-Hellman key exchange scheme required the computation of modular exponentiation which is computational expensive. The Montgomery multiplication algorithm is used to speed up the modular multiplications needed during exponentiation computation.

$$MonPro(a, b) = a \cdot b \cdot r^{-1} \ (mod\ n)$$

Let the modulus *n* be a *k*-bit integer such that $2^{k-1} \leq n \leq 2^k$ and let *r* be 2^k where $\gcd(r, n) = 1$. Given two *n*-residues $\bar{a}$ and $\bar{b}$ where $\bar{a} = a \cdot r \ mod\ n$ and $\bar{b} = b \cdot r \ mod\ n$, the Montgomery product is defined as the *n*-residue

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \ (mod\ n)$$

$$= a \cdot r \cdot b \cdot r \cdot r^{-1} \ (mod\ n)$$

$$= c \cdot r \ (mod\ n)$$

Where r^{-1} is the inverse of *r* modulo *n* with the property $r^{-1} \cdot r = 1 \ (mod\ n)$

To describe the Montgomery reduction algorithm, an additional quantity n' is introduced with the property $r^{-1} \cdot r - n \cdot n' = 1$

Algorithm 5: General Montgomery Multiplication

1: **function** $MonPro(\bar{a}, \bar{b})$
2: $t = \bar{a} \cdot \bar{b}$
3: $u = (t + (t \cdot n' mod\ r) \cdot n)/r$
4: **if** $u \geq n$ then
5: **return** $u - n$
6: **else**
7: **return** u
8: **end if**
9: **end function**

Multiplication that involving modulo *r* and division by r can be computed quickly since r is a power of 2. Thus, the Montgomery multiplication is faster than common computation of $a \cdot b \, mod \, n$ which requires division by n. However, Montgomery Multiplication is more effective when several modular multiplication with same modular are needed such as to compute modular exponentiation since the conversion to and from between ordinary residue and *n*-residue and computation of n' are time consuming.

We implemented Montgomery Exponentiation with binary method. Let *j* be the number of bits in the exponent *e*. We can compute $x = a^e \, mod \, n$ with the complexity of $O(j)$ to the Montgomery Multiplication.

Algorithm 6: Montgomery Exponentiation

1: **function** $MonExp(a, e, n)$
2: $a = a \cdot r \, mod \, n$
3: $x = 1 \cdot r \, mod \, n$
4: **for** $i = j - 1 \; to \; 0$ **do**
5: $\quad \bar{x} = MonPro(\bar{x}, \bar{x})$
6: $\quad$ **if** $e_i = 1$ **then**
7: $\quad\quad \bar{x} = MonPro(\bar{x}, \bar{a})$
8: $\quad$ **end if**
9: **end for**
10: **return** $x = MonPro(\bar{x}, 1)$
11: **end function**

In line 10, we converted $\bar{x}$ back to *x* with the property of Montgomery algorithm which $\bar{x} \cdot 1 \cdot r^{-1} = x \cdot r \cdot r^{-1} = x \, mod \, n$.

There are different ways to perform Montgomery Multiplication. In this project, we implemented Separated Operand Scanning (SOS) method and Coarsely Integrated Operand Scanning (CIOS) method. In short, SOS method separated multiplication and reduction steps where CIOS integrated them.

Separated Operand Scanning (SOS) Method

Algorithm 7: SOS Montgomery Multiplication

1: **function** $sosMonPro(\bar{a}, \bar{b})$
2: **for** $i = 0\ to\ s - 1$ **do**
3: $c = 0$
4: **for** $j = 0\ to\ s - 1$ **do**
5: $(C, S) = t[i + j] + a[j] \cdot b[i] + c$
6: $t[i + j] = S$
7: $t[i + s] = C$
8: **for** $i = 0\ to\ s - 1$ **do**
9: $C = 0$
10: $m = t[i] \cdot n'[0] mod\ 2^{32}$
11: **for** $j = 0\ to\ s - 1$ **do**
12: $(C, S) = t[i + j] + m \cdot n[j] + c$
13: $t[i + j] = S$
14: $ADD(t[i + s], C)$
15: **for** $j = 0\ to\ s$ **do**
16: $u[j] = t[j + s]$
17: $B = 0$
18: **for** $i = 0\ to\ s - 1$ **do**
19: $(B, D) = u[i] - n[i] - B$
20: $t[i] = D$
21: $(B, D) = u[s] - B$
22: $t[s] = D$
23: **if** $B = 0$ **then**
24: **return** $t[0], t[1], \dots, t[s - 1]$
25: **else**
26: **return** $u[0], u[1], \dots, u[s - 1]$
27: **end function**

In algorithm 7 above, from line 2 to line 7 performed multiplication on $t = \bar{a} \cdot \bar{b}$. From line 8 to line 16, this section performed $u = (t + (t \cdot n' mod\ r) \cdot n)/r$. In the last step line 17 to line 26, the subtraction is then performed to reduce *u* if necessary.

As SOS method separated the multiplication from reduction, it is obviously slower than CIOS method that integrated both multiplication and reduction together. However, there is one optimisation that we can perform in the part of multiplication $a \cdot b$ in SOS method. When *a* is equal to *b*, we can optimise the Montgomery multiplication algorithm for squaring, which is useful on line 5 in Algorithm 6. The optimisation of squaring is achieved because almost half of the single-precision multiplication can be skipped since $a_i \cdot a_j = a_j \cdot a_i$. Diagram below illustration a simple example.

		1	2	3
X		1	2	3
		$3 \cdot 1$	$3 \cdot 2$	$3 \cdot 3$
	$2 \cdot 1$	$2 \cdot 2$	$2 \cdot 3$	
$1 \cdot 1$	$1 \cdot 2$	$1 \cdot 3$		

The following pseudocode replaces the first part of the Algorithm 7 in order to perform the optimised Montgomery squaring.

1: **for** $i = 0\ to\ s - 1$ **do**
2: $(C, S) = t[i + i] + a[i] \cdot a[i]$
3: **for** $i = j + 1\ to\ s - 1$ **do**
4: $(C, S) = t[i + j] + 2 \cdot a[j] \cdot a[i] + C$
5: $t[i + j] = S$
6: $t[i + s] = C$

However, the value $2 \cdot a[j] \cdot a[i]$ may cause overflow to occur which may exceed 2^{64}, the memory limit of variable in computer nowadays. Therefore, we rewrite the snippet above with 3 steps, $a[j] \cdot a[i]$ terms are added first, the result is then doubled and then $a[i] \cdot a[i]$ are added in.

1: **for** $i = 0\ to\ s - 1$ **do**

2: $C = 0$

3: **for** $j = 0\ to\ s - 1$ **do**

4: $(C, S) = a[j] \cdot a[i]$

5: $t[i + j] = S$

6: $t[s + i] = C$

7: $C = 0$

8: **for** $i = 0\ to\ 2 \cdot s$ **do**

9: $(C, S) = t[i] \cdot 2 + C$

10: $t[i] = S$

11: **for** $i = 0\ to\ s - 1$ **do**

12: $(C, S) = t[i + i] + a[i] \cdot a[i]$

13: $t[i + i] = S$

14: **for** $j = i + 1\ to\ s$ **do**

15: $(C, S) = t[i + j] + C$

16: $t[i + j] = S$

17: $t[s + i] = t[s + i] + C$

18: $C = 0$

Coarsely Integrated Operand Scanning (CIOS) Method

Algorithm 8: CIOS Montgomery Multiplication

1: **function** $ciosMonPro(\bar{a}, \bar{b})$
2: **for** $i = 0\ to\ s - 1$ **do**
3: $C = 0$
4: **for** $j = 0\ to\ s - 1$ **do**
5: $(C, S) = t[j] + a[j] \cdot b[i] + C$
6: $t[j] = S$
7: $(C, S) = t[s] + C$
8: $t[s] = S$
9: $t[s + 1] = C$
10: $m = t[0] \cdot n'[0]\ mod\ 2^{32}$
11: $(C, S) = t[0] + m \cdot n[0]$
12: **for** $j = 1\ to\ s - 1$ **do**
13: $(C, S) = t[j] + m \cdot n[j] + C$
14: $t[j - 1] = S$
15: $(C, S) = t[s] + C$
16: $t[s - 1] = S$
17: $t[s] = t[s + 1] + C$
18: **for** $j = 0\ to\ s - 1$ **do**
19: $u[j] = t[j]$
20: $B = 0$
21: **for** $i = 0\ to\ s - 1$ **do**
22: $(B, D) = u[i] - n[i] - B$
23: $t[i] = D$
24: $(B, D) = u[s] - B$
25: $t[s] = D$
26: **if** $B = 0$ **then return** $t[0], t[1], \dots, t[s - 1]$
27: **else return** $u[0], u[1], \dots, u[s - 1]$
28: **end function**

Method	Multiplications	Additions	Reads	Writes	Space
SOS	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 3$	$2s^2 + 6s + 2$	$2s + 2$
CIOS	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 2$	$6s^2 + 5s + 1$	$s + 3$

From the table above, we can see that the memory reduction for CIOS method is a significant improvement over the SOS method. Besides, the integration in this method is "coarse" because of the alternation between iterations of the outer loop.

Therefore, we can obtain the optimum computation speed on Montgomery Exponentiation by integrating squaring optimised' SOS method and CIOS method.

Algorithm 9: Montgomery Exponentiation

1: **function** $MonExp(a, e, n)$
2: $a = a \cdot r \ mod \ n$
3: $x = 1 \cdot r \ mod \ n$
4: **for** $i = j - 1 \ to \ 0$ **do**
5: $\bar{x} = MonProSOS(\bar{x}, \bar{x})$
6: **if** $e_i = 1$ **then**
7: $\bar{x} = MonProCIOS(\bar{x}, \bar{a})$
8: **end if**
9: **end for**
10: **return** $x = MonPro(\bar{x}, 1)$
11: **end function**

3.2.2 CUDA

Compute Unified Device Architecture (CUDA) is NVIDIA's GPU architecture featured in the GPU cards, for general purpose computing with GPUs. CUDE C/C++ is an extension of C/C++ programming language for general purpose computation. CUDA contains few remarkable parts to be explored which is the memory hierarchy and thread hierarchy (NVIDIA CUDA Programming Guide 2012).

3.2.2.1 Thread Hierarchy

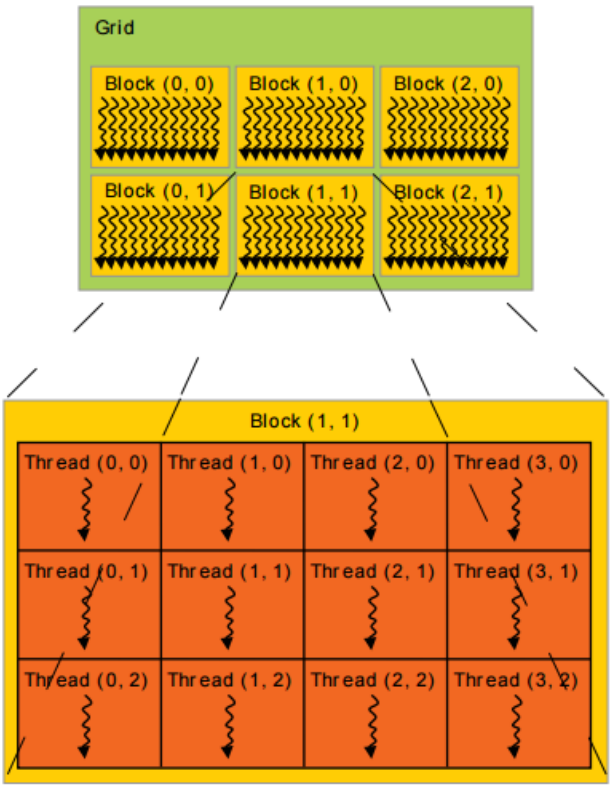


Figure 1: Grid of Thread Blocks

To effectively utilise the full computational capability of the graphics card on the system, CUDA architecture separates it into grids, blocks and threads in a hierarchical structure as shown in Figure 7 above from (NVIDIA CUDA Programming Guide 2012). Since there are a number of threads in one block and a number of blocks in one grid and a number of grids in one GPU, the parallelism that is achieved using such a hierarchical architecture is large.

- **The Grid**. A grid is a group of threads running the same kernel which are not synchronised.
- **The Block**. Grids are composed of blocks. A built in variable "blockIdx" can be used to identify the current block. Block IDs can be organised into a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockDim variable. Each block is a logical unit containing a number of coordinating threads, a certain amount of shared memory.
- **The Thread**. Blocks are composed of threads. Threads within a block execute same instruction of codes but possibly of different data at the same time. Threads within a block can cooperate by sharing data through some shared memory which is expected to be a low-latency memory near each processor core and by synchronising their execution to coordinate memory accesses. Thread ID can be determined by “threadIdx.x”.

3.2.2.2 Memory Hierarchy

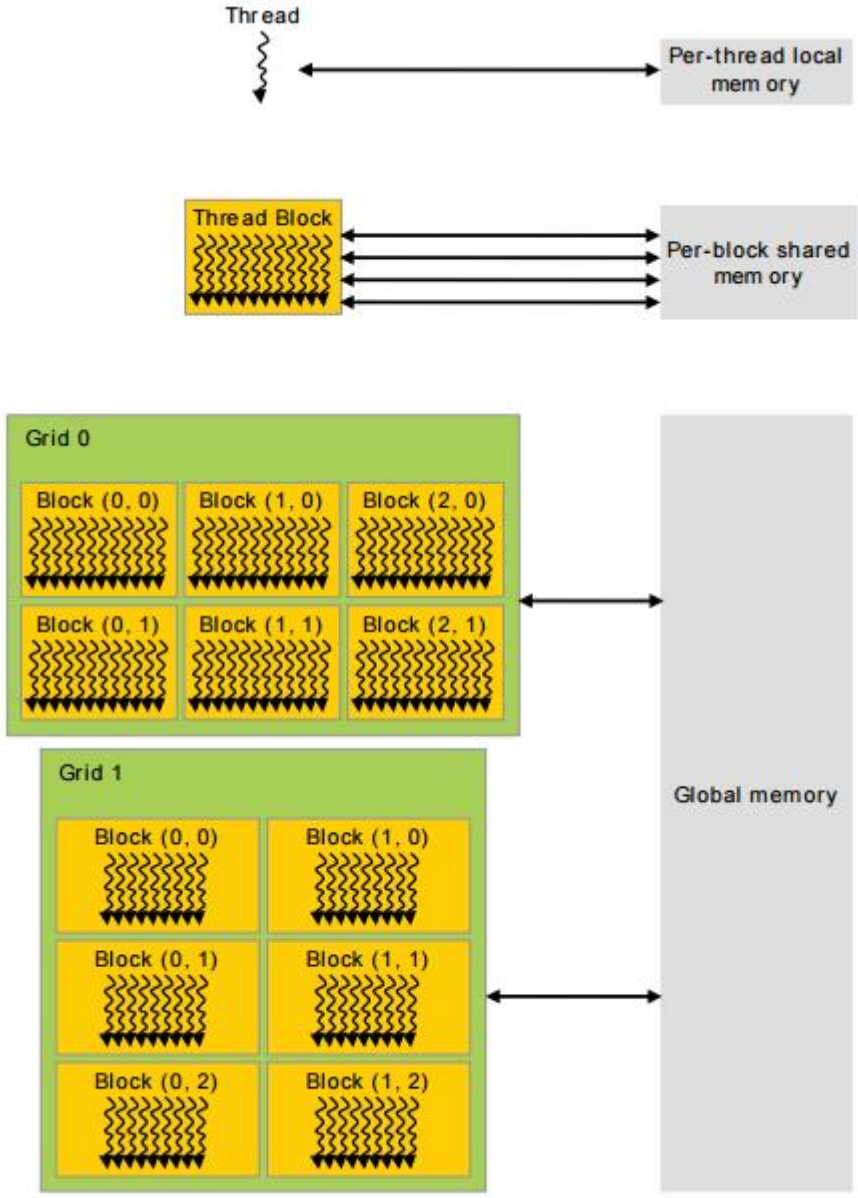


Figure 2: Memory Hierarchy

CUDA threads which has private local memory may access data from various memory spaces during executions as illustrated by Figure 8 from (NVIDIA CUDA Programming Guide 2012). From Figure 8, each thread has per-thread local memory and each thread block has per-block shared memory which with the same lifetime as the block. The global memory is accessible by all threads in the grids.

- **Global memory**. A read and write memory which is slow and uncached.
- **Texture memory**. A read-only memory. Offers different addressing modes, as well as data filtering for some specific data formats

- **Constant memory**. Store constants and kernel arguments. Slow but with cache.
- **Local memory**. It is generally used for whatever does not fit into registers. Slow and uncached but allows automatic fuse reads and writes.
- **Shared memory.** An extremely fast on-chip memory with lower capacity(16-64Kbytes) which shares memory across a unit block that accessible by threads in same block of memory. Besides, kernel function parameters are stored here.

3.3 Implementation issues and challenges

As Computer Science student, we often think on the algorithm analysis and design in order to speed up the computation. However in this library development, parallel techniques are required and we have to realise that not every algorithm can be done in parallel manner. Moreover, a low level C programming language must be used to achieve good manual memory management in order to optimise the code.

3.4 Timeline

At the starting of this semester, the prototype of library coded in C++ which computes using CPU sequentially had been done which included addition ,subtraction, multiplication and division. In this semester, GPU programming have been learnt and part of the arithmetic operation have been replaced by CUDA codes which computes in parallel using GPU such as addition and subtraction. At the end of semester, parallel multiplication and division are expected to be done. In the next semester, Montgomery exponentiation and multiplication, greatest common divisor and code optimization will be explored to ensure that the library achieve an acceptable improvement in term of memory and speed.

No	Task	Duration (Days)	Week						
			1 - 4	5	6	7	8	9	10 +
1	Study GPU programming concepts and practices	28							
2	Explore implementation on GPU programming with C language on the library	3							
3	Study and implement parallel technique on addition	3							
4	Study and implement parallel technique on subtraction	1							
5	Study and implement sequential/parallel Montgomery Multiplication/Exponentiation/ Reduction	14							
6	Study and identify possibility of implementing parallel technique on division	14							

Table 1: Gantt chart for FYP1

No	Task	Duration (Days)	Week						
			1 - 4	5	6	7	8	9	10 +
1	Study Separated Operand Scanning (SOS) method& Coarsely Integrated Operand Scanning (CIOS) method for Montgomery Multiplication	28							
2	Implement SOS & CIOS method on CPU	7							
3	Implement SOS & CIOS method on GPU	7							
4	Implement Coalesced Access structure to improve performance	7							
5	Implement Montgomery Exponentiation with binary exponentiation method	7							
6	Improve SOS multiplication on squaring to reduce computation on multiplication	7							
7	Continue to find way to optimise the performance and calculate the throughput on Montgomery Multiplication and Exponentiation	-							

Table 2: Gantt chart for FYP2

Chapter 4: Implementation & Optimisation

To obtain the best performance from NVIDIA CUDA GPU, we explored on CUDA C Best Practice Guide from CUDA Toolkit Documentation to determine any possible optimisation that can be considered.

4.1 Parallel library

The most straight forward approach to parallelising our system is to make use of existing libraries that take edge of parallel architecture for the sake of us. We used Thrust library from CUDA toolkit which is a parallel C++ template library similar to the C++ Standard Template Library. Thrust provides two vector containers, *host_vector* and *device_vector* that are like vector in C++ STL. We make use of " = " operator to copy a *host_vector* to a *device_vector* (or vice versa) so we can avoid low level declaration like cudaMemcpy to allow maintainable and clean code. By using *host_vector* and *device_vector*, we used syntax like *H.begin()* and *H.end()* as iterator. Besides, we also used *insert* and *push_back* function during data processing.

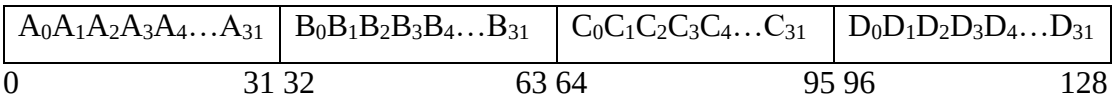
4.2 Performance Metrics

When we attempt to optimise and implement CUDA code, it is compulsory to know how to measure performance accurately. We timed CUDA calls and kernel executions with CPU timer. Since CUDA API functions are asynchronous which they return control back to the calling CPU thread prior to complete their work, by calling *cudaDeviceSynchronize()* immediately before starting and stopping the CPU timer, we are able to synchronize the CPU thread with the GPU.

4.3 Coalesced Access to Global Memory

Global memory loads and stores by threads of a warp are coalesced by the device into as few as one transaction when certain access requirements are met.

For example, suppose the threads of a warp access adjacent 128-byte words (e.g., adjacent *bigInt* values) by a single 128B L1 cache line. Such a pattern is shown below. This way of organising data is called array of structure (AoS)



From the pattern above, 4 coalesced transactions will service that memory access. In our actual implementation, our algorithm accesses the word by using one index so that we can access the other neighbour data with *index+i*. However, the pattern above does not optimise the data access well as we will only need 4 starting word, A_0, B_0, C_0 and D_0 to reach other data. Therefore, we restructured our data to allign the access pattern properly below. This way of organising data is called structure of arrays (SoA).

$A_0B_0C_0D_0\ldots$	$A_1B_1C_1D_1\ldots$	$A_2B_2C_2D_2\ldots$	$A_3B_3C_3D_3\ldots$
0 31	32 63	64 95	96 128

With the structure above, one single coalesced transaction is sufficient to service that memory access hence speed up global memory read operation.

To perform the conversion, suppose we have an array of multiple *bigInt* data which depends on thread size and block size (Concatenation of data depends on $thread\ size * number\ of\ block$) $A_0A_1A_2A_3A_4\ldots A_{31}$ $B_0B_1B_2B_3B_4\ldots B_{31}$ $C_0C_1C_2C_3C_4\ldots C_{31}$,

```
1: thrust::device_vector<unsigned int> deviceA
2: for i = 0 to 32 do
3:     for j = 0 to 1024 * 32 * blockSize do
4:         deviceA.push_back(A[j + i])
5:         j = j + 32
```

After the conversion, the structure in the array is reordered to become $A_0B_0C_0D_0\ldots$ $A_1B_1C_1D_1\ldots$ $A_2B_2C_2D_2\ldots$ $A_3B_3C_3D_3\ldots$ and ready to transfer into kernel function as parameter. Storing the data in SoA make full use of GPU memory bandwidth since there is no interleaving of elements of the same field, the SoA layout on the GPU provides coalesced memory access and can achieve more efficient global memory utilisation.

4.4 Instruction Optimization

Division and modulo operations are needed in Montgomery multiplication. In computer, integer division and modulo operations are costly and should be stay away from. Therefore, we replaced them with bitwise operations since our *bigInt* is represented in power of 2^{32}. The shift operations can be used to avoid expensive division and modulo calculations:

$$a\ mod\ 2^{32} = a\ \%\ 2^{32} = a\ \&\ 4294967295$$

$$a \div 2^{32} = a \gg 32$$

In Montgomery exponentiation, to check if *j*-th item of the set is on, we used the bitwise AND operation $T = S \,\&\, (1 \ll j)$, which make it a much more efficient choice.

4.5 Thread and Block size

To perform Montgomery Multiplication in GPU, we performed kernel call providing the number of blocks in each dimension and threads per block in each dimension. For instance, *MontMul<<<B,T>>>(d_a1, d_b1, d_ans, d_n, d_n1, d_s)*. In our implementation, we uses a 1-D structure which each block will contains the maximum threads available by CUDA, which is 1024 threads. Each thread will handle one computation of Montgomery multiplication. In each block, 1024 Montgomery multiplication will be computed. Therefore, to compute more than 1024 Montgomery multiplication, user have to create more threads by declaring the block size to the kernel by passing it through *MonPro()* function in CPU. In short, a kernel call of block size of *N* can compute $N * 1024$ Montgomery multiplication in parallel.

CUDA Built-In Variables

- *blockIdx.x, blockIdx.y, blockIdx.z* are built-in variable that returns the block ID in the *x-axis, y-axis* and *z-axis* of the block that is executing the given block of code. In our implementation, since we are using a 1D dimensional block, we needed only the *x-axis* by using *blockIdx.x*.
- *threadIdx.x, threadIdx.y, threadIdx.z* are built-in variable that returns the thread ID in the the *x-axis, y-axis* and *z-axis* of the thread that is being executed by this stream processor in this particular block. In our implementation, since we are using a 1D dimensional thread, we needed only the *x-axis* by using *threadIdx.x*.
- blockDim.x, blockDim.y, blockDim.z are built-in variables that return the number of threads in a block in the *x-axis, y-axis* and *z-axis*.

In our implementation, the full global thread ID in *x-dimension* can be computed by:

$$idx = threadIdx + \ blockIdx.x * blockDim.x$$

For instance, with a block dimension of 8, the Global Thread ID represents:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2							

Since we rearrange the structure in the array given to kernel call, during first step which is responsible for the multiplication in Montgomery multiplication algorithm, code below will do the access the correct index for it. *($d\_s = 32$ in our implementation)

1: **for** $i = 0$ *to* d_s **do**

2: $C = 0$

3: **for** $j = 0$ *to* d_s **do**

4: $(C, S) = t[i + j] + C + a[j * 1024 * blockSize + idx] + b[i * 1024 * blockSize + idx]$

5: $t[i + j] = S$

6: $t[i + d\_s] = carry$

As shown above, $j * 1024 * blockSize$ will access the correct index as it will "jump" from group 0 to group 1 to so on, and then perform shifting with *idx* which is our global thread ID which tell GPU which thread it is dealing with. Suppose we have a block size of one below:

0	Group 0	1024	Group 1	2048	Group 2

After the computation in Montgomery Multiplication, the GPU will have to store the result in an array, say, *d_ans*. Montgomery exponentiation will perform multiple Montgomery multiplication, hence we decided to keep the answer in similar structure for Coalesced Access. To do so, the code below perform the restructuring during the data copying process.

1: $counter = 0$

2: for $i = 0$ *to* d_s do

3: $\quad d\_ans[i * 1024 * blockSize + idx] = t[counter]$

4: $\quad counter = counter + 1$

4.6 Shared Memory

Shared memory are physically closer to the Streaming multiprocessors than both the L2 cache and global memory. Therefore, shared memory is roughly 20 to 30 times lower than global memory, and a bandwidth with about 10 times higher. Shared memory is useful as an intra-block thread communication channel. In our implementation, we realised the fact that the constant variable *n* and *n'* are constantly used by all multiplication in the kernel. Hence, we replaced global memory accesses by shared memory in both variables.

```
__shared__ unsigned int shared_n[32], shared_n1[32];
for (int i = 0; i < d_s; i++){
	shared_n[i] = d_n._array[i];
	shared_n1[i] = d_n1._array[i];
}
__syncthreads();
```

4.7 Clean Code

Although the usage of Thrust library can avoid low level declaration like cudaMemcpy to allow maintainable and clean code, it is purely a host side abstraction. It is forbidden to be used inside kernel. Therefore, we created a template outside the *bigInt* class to pass these device vector into kernel.

```
template <typename T>
struct KernelArray
{
	T*  _array;
	int _size;

	// constructor allows for implicit conversion
	KernelArray(thrust::device_vector<T>& dVec) {
		_array = thrust::raw_pointer_cast(&dVec[0]);
		_size = (int)dVec.size();
	}
```

```
};
```

With this template, we can declare device vector as *KernelArray* in our parameter in kernel function as below. For instance:

```
__global__void SOS(KernelArray<unsigned int>d_a1, KernelArray<unsigned
int>d_b1, KernelArray<unsigned int>d_ans, KernelArray<unsigned int>d_n,
KernelArray<unsigned int>d_n1, int d_s, int blkSize)
```

To access the element inside array of *KernelArray*, for example *d_a1* using *_array*:

$$d\_a1.\_array[i]$$

To determine the size of the array of *KernelArray*, using *_size*:

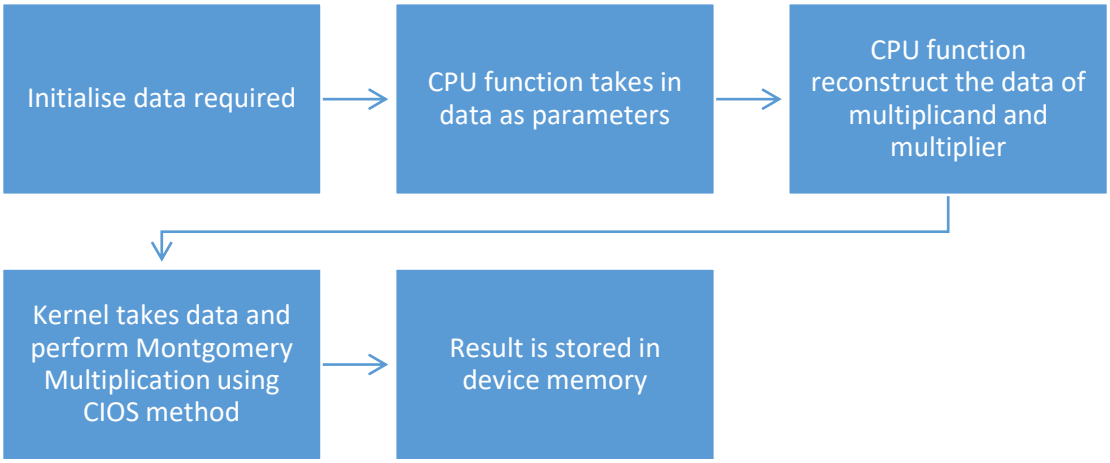
$$d\_a1.\_size$$

4.8 Data usage

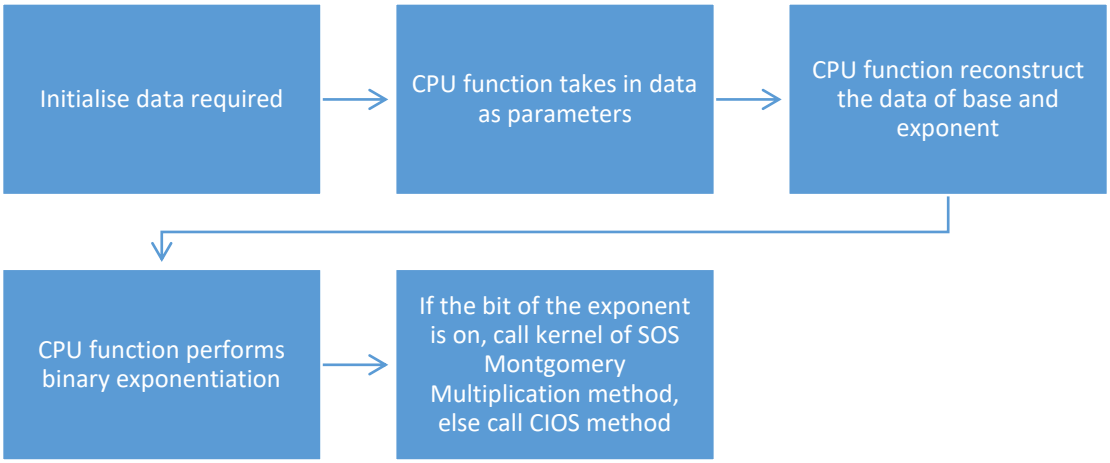
In our implementation, a *bigInt* array stores 32 words of large integer representation, which each word take memory size of 32 bits therefore $32\ bits * 32 = 1024\ bits$. In Montgomery Multiplication, each thread will have a multiplicand and multiplier which occupied memory of $1024\ bits * 2 = 2048\ bits$. Since we launched at least 1024 threads, there is $1024 * 2048\ bits = 2097152\ bits$. At optimum block size of approximately 10, we will have $2097152\ bits * 10 = 20971520\ bits = 262144\ bytes = 2.62144\ Megabytes$ of memory space.

4.9 Flow chart for implementation

The flow chart below shows the process of how user can compute the Montgomery Multiplication. For Montgomery multiplication, our library needed few parameters such as multiplicand, multiplier, *n*, *n'*, *R*, R^{-1} and block size.



The flow chart below shows the process of how user can compute the Montgomery Exponentiation. For Montgomery exponentiation, our library needed few parameters such as base, exponent, *n*, *n'*, *R*, R^{-1} and block size.



4.8 Code Snippet

SOS Method for squaring (Algorithm 7)

```
__global__ void sosV3(KernelArray<unsigned int>d_a1, KernelArray<unsigned int>d_ans, KernelArray<unsigned int>d_n, KernelArray<unsigned int>d_n1, int d_s, int blkSize){
	int idx = threadIdx.x + blockIdx.x * blockDim.x;
	unsigned int t[65] = { 0 };

	unsigned long long tempProduct;
	unsigned long long carry = 0;
	__shared__ unsigned int shared_n[32], shared_n1[32], shared_s;
	shared_s = d_s;
	for (int i = 0; i < shared_s; i++){
		shared_n[i] = d_n._array[i];
		shared_n1[i] = d_n1._array[i];
	}
	__syncthreads();

	////////////Optimization for squaring multiplication///////////
	for (int i = 0; i < shared_s; i++){
		for (int j = i + 1; j < shared_s; j++){
			tempProduct = ((unsigned long long)d_a1._array[j * 1024 * blkSize + idx] * (unsigned long long)d_a1._array[i * 1024 * blkSize + idx]) + (unsigned long long)t[i + j] + carry;
			t[i + j] = tempProduct & 4294967295;
			carry = tempProduct >> 32;
		}
		t[shared_s + i] = carry;
		carry = 0;
	}

	for (int i = 0; i < 65; i++){
		tempProduct = (unsigned long long)t[i] * 2 + carry;
		t[i] = tempProduct & 4294967295;
		carry = tempProduct >> 32;
	}

	for (int i = 0; i < shared_s; i++){
		tempProduct = (unsigned long long)t[i + i] + ((unsigned long long)d_a1._array[i * 1024 * blkSize + idx] * (unsigned long long)d_a1._array[i * 1024 * blkSize + idx]);
		t[i + i] = tempProduct & 4294967295;
		carry = tempProduct >> 32;
		for (int j = i + 1; j < shared_s; j++){
			tempProduct = (unsigned long long)t[i + j] + carry;
			t[i + j] = tempProduct & 4294967295;
			carry = tempProduct >> 32;
		}
		t[shared_s + i] += carry;
		carry = 0;
	}
	/////////////////////////////////////////////////////////////////



	for (int i = 0; i < shared_s; i++){
		unsigned long long c = 0;
		unsigned long long m = ((unsigned long long)t[i] * (unsigned long long)shared_n1[0]) & 4294967295;
```

```
			for (int j = 0; j < shared_s; j++){
				unsigned long long tempProduct = (unsigned long long)t[i +
j] + m*(unsigned long long)shared_n[j] + c;
				c = tempProduct >> 32;
				t[i + j] = tempProduct & 4294967295;
			}
			//ADD(t[i+s], C)
			int counter = i;
			while (c != 0){
				unsigned long long temp = (unsigned long long)t[counter +
shared_s] + c;
				c = temp >> 32;
				t[counter + shared_s] = temp & 4294967295;
				counter++;
			}
		}
		unsigned int u[33];
		for (int j = 0; j < shared_s + 1; j++){
			u[j] = t[j + shared_s];
		}


		int b = 0;
		long long sub;
		for (int i = 0; i < shared_s; i++){
			sub = (long long)u[i] - shared_n[i] - b;
			if (sub < 0){
				t[i] = sub + 4294967296;
				b = 1;
			}
			else{
				t[i] = sub;
				b = 0;
			}
		}
		sub = (long long)u[shared_s] - b;
		u[shared_s] = sub;

		if (sub >= 0){
			int counter = 0;
			for (int i = 0; i < 32; i++){
				d_ans._array[i * 1024 * blkSize + idx] = t[counter++];
			}
		}
		else{
			int counter = 0;
			for (int i = 0; i < 32; i++){
				d_ans._array[i * 1024 * blkSize + idx] = u[counter++];
			}
		}
}
```

CIOS Method for squaring (Algorithm 8)

```
__global__ void ciosV2(KernelArray<unsigned int>d_a1, KernelArray<unsigned
int>d_b1, KernelArray<unsigned int>d_ans, KernelArray<unsigned int>d_n,
KernelArray<unsigned int>d_n1, int d_s, int blkSize){
	int idx = threadIdx.x + blockIdx.x * blockDim.x;
	unsigned int t[33] = { 0 };
	unsigned long long temp;
	__shared__ unsigned int shared_n[32], shared_n1[32], shared_s;
	shared_s = d_s;
	for (int i = 0; i < shared_s; i++){
		shared_n[i] = d_n._array[i];
		shared_n1[i] = d_n1._array[i];
	}
	__syncthreads();

	for (int i = 0; i < shared_s; i++){
		unsigned long long c = 0;
		for (int j = 0; j < shared_s; j++){
			temp = t[j] + (unsigned long long)d_a1._array[j * (1024 *
blkSize) + idx] * (unsigned long long)d_b1._array[i * (1024 * blkSize) + idx] +
c;
			t[j] = temp & 4294967295;
			c = temp >> 32;
		}
		temp = (unsigned long long)t[shared_s] + c;
		t[shared_s] = temp & 4294967295;
		t[shared_s + 1] = temp >> 32;

		unsigned long long m = ((unsigned long long)t[0] * (unsigned long
long)shared_n1[0]) & 4294967295;
		temp = (unsigned long long)t[0] + m*(unsigned long
long)shared_n[0];
		c = temp >> 32;
		for (int j = 1; j < shared_s; j++){
			temp = (unsigned long long)t[j] + m*(unsigned long
long)shared_n[j] + c;
			t[j - 1] = temp & 4294967295;
			c = temp >> 32;
		}
		temp = (unsigned long long)t[shared_s] + c;
		t[shared_s - 1] = temp & 4294967295;
		c = temp >> 32;
		t[shared_s] = t[shared_s + 1] + c;
	}

	unsigned int u[33];
	for (int j = 0; j < shared_s + 1; j++){
		u[j] = t[j];
	}

	int b = 0;
	long long sub;
	for (int i = 0; i < shared_s; i++){
		sub = (long long)u[i] - shared_n[i] - b;
		if (sub < 0){
			t[i] = sub + 4294967296;
			b = 1;
		}
		else{
			t[i] = sub;
			b = 0;
```

```
			}
		}
		sub = (long long)u[shared_s] - b;
		u[shared_s] = sub;

		if (sub >= 0){
			int counter = 0;
			for (int i = 0; i < 32; i++){
				d_ans._array[i * 1024 * blkSize + idx] = t[counter++];
			}
		}
		else{
			int counter = 0;
			for (int i = 0; i < 32; i++){
				d_ans._array[i * 1024 * blkSize + idx] = u[counter++];
			}
		}
}
```

Binary Exponentiation Method (Algorithm 9)

From Chapter 3 Algorithm 9, the binary exponentiation is for exponent that smaller than 64 bits. However, our exponent can grow as large as 2^{2048} which supported by our *bigInt* data type. Therefore, the code snippet below shows how to implement binary exponentiation on *bigInt* data type.

```
for (int i = exp.radixForm.size() - 1; i >= 0; i--){
    if (exp.radixForm.size() - 1 == i){
       for (int j = floor(log2(exp.radixForm[i])); j >= 0; j--){
           sosV3 <<<blockSize, 1024 >>>(d_x1, d_ans, d_n, d_n1, d_s, d_blkSize);
           d_x1 = d_ans;
           if (exp.radixForm[i] & (1 << j)){
               ciosV2 <<<blockSize, 1024 >>>(d_x1, d_a1, d_ans, d_n, d_n1, d_s,
d_blkSize);
               d_x1 = d_ans;
           }
        }
    }
    else{
        for (int j = 31; j >= 0; j--){
            sosV3 <<<blockSize, 1024 >>>(d_x1, d_ans, d_n, d_n1, d_s, d_blkSize);
            d_x1 = d_ans;
            if (exp.radixForm[i] & (1 << j)){
                ciosV2 <<<blockSize, 1024 >>>(d_x1, d_a1, d_ans, d_n, d_n1, d_s,
d_blkSize);
                d_x1 = d_ans;
            }
        }
    }
}
```

Chapter 5: Conclusion

As mentioned in Chapter 2, we followed the (APOD) design cycle which we repeatedly optimise and deploy our library. Therefore, in this first phrase, we identified which method of Montgomery Multiplication is suitable to perform efficient Montgomery Exponentiation. In this phrase, we tested the performance between SOS method and CIOS method with and without the use of coalesced access to global memory with Structure of Array model. The results are benchmarked with integer of size 2^{1024} for multiplicand and multiplier in Montgomery Multiplication. In Montgomery exponentiation, we are using an exponent of size 2^{2048} with base of size 2^{1024}.

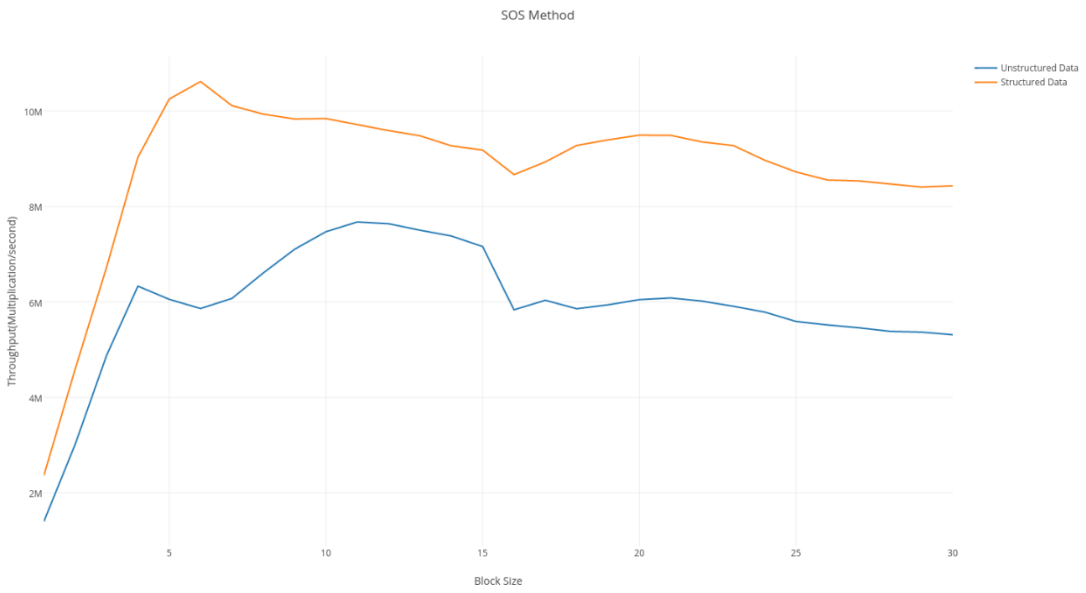


Figure 1: SOS method

From the graph above, we can justify that Coalesced Access to Global Memory is extremely important optimisation to be done. At the optimum block size of 6, the performance is almost doubled with proper structured data, from 5862266 multiplications per second to 10617868 multiplications per second.

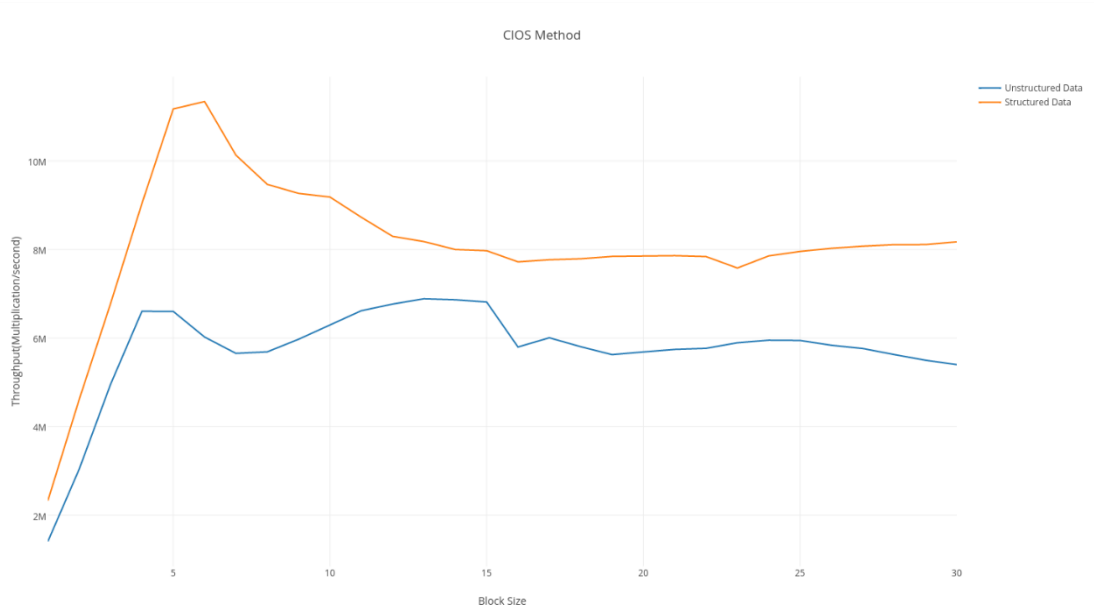


Figure 2: CIOS method

From the graph above, we can justify that Coalesced Access to Global Memory is extremely important optimisation to be done. At the optimum block size of 6, the performance is almost doubled with proper structured data, from 6024097 multiplications per second to 11342885 multiplications per second.

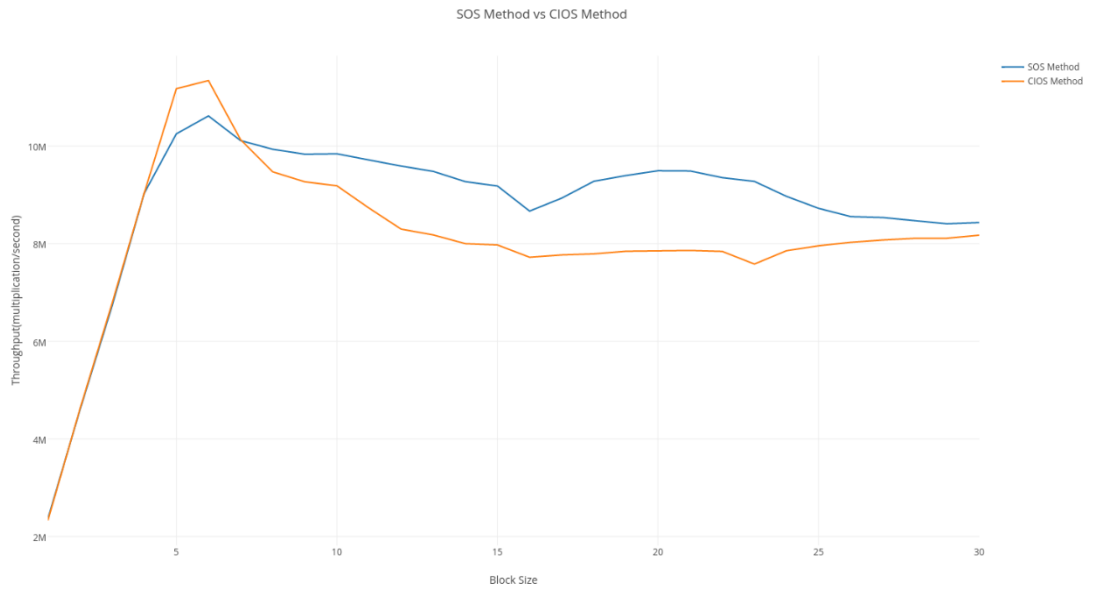


Figure 3: SOS method vs CIOS method

From the figure above, CIOS method performs better than SOS method on optimum block size since CIOS method performs lesser read, write and space than SOS method.

On block size of 6, CIOS method computes 11342885 multiplications per second while SOS method computes 10617868 multiplications per second.

Figure 4: Montgomery Exponentiation

From the analytics of CIOS and SOS method on Montgomery multiplication, we took advantage of both multiplication method on Montgomery exponentiation. As CIOS method is more efficient than SOS method when multiplicand is not equal to multiplier, we compared four exponentiation method. Comparing between purely CIOS multiplication method and purely SOS multiplication method, CIOS method only perform well on the peak of the block size and slow down along with increasing block size. Hence, we introduced SOS method with squaring optimisation. By comparing the result of CIOS method integrate with SOS method without squaring and CIOUS method integrate with SOS squaring optimised method, we can see that the SOS squaring optimised method works nicely with CIOS method, which increase the performance of Montgomery exponentiation overall. From the result, we can see a huge leap of improvement on performance when the block size become gradually large. Our Montgomery exponentiation is able to compute $1.7120393 * 10^7$ exponentiations per second with optimum block size of 8.

Since we identified that CIOS method with SOS squaring method make the most efficient Montgomery exponentiation, we will focus to optimise both of them to increase our efficiency further. In this phrase, we contrast the performance by introducing shared memory to the kernel of CIOS method and SOS squaring method.

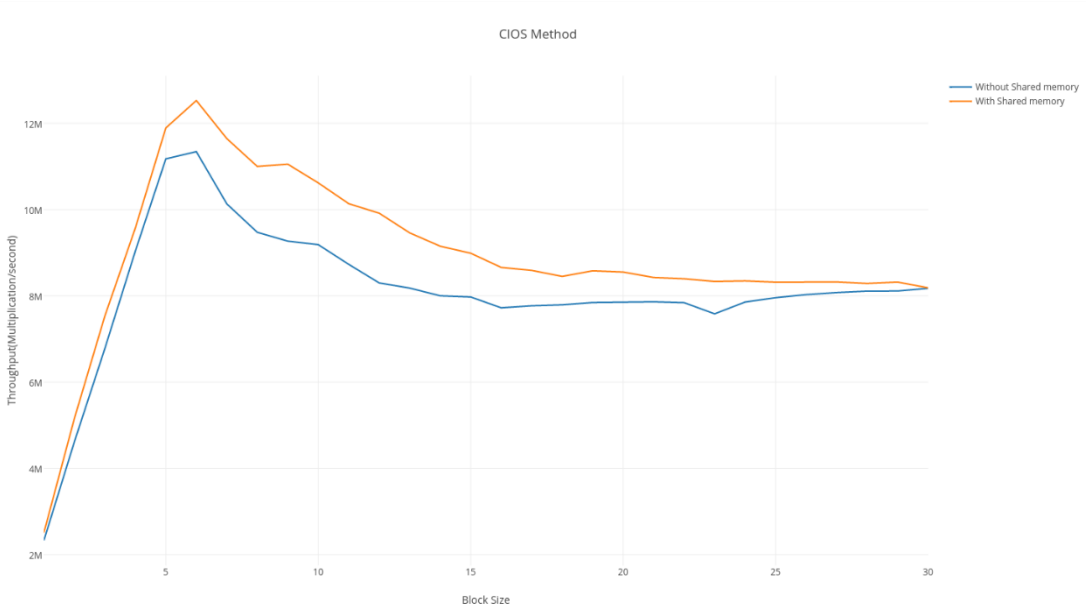


Figure 5: CIOS with shared memory vs without shared memory

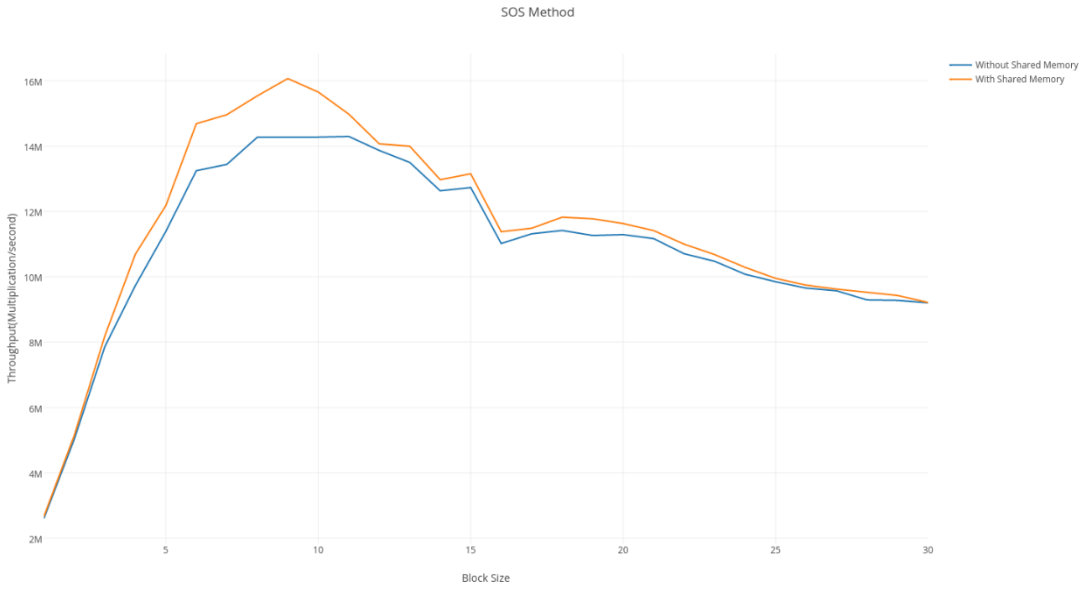


Figure 6: SOS squaring with shared memory vs without shared memory

From the two figures above, we can see an overall small improvement after make use of the Shared memory. Hence, we benchmarked our Montgomery exponentiation again to compare the efficiency.

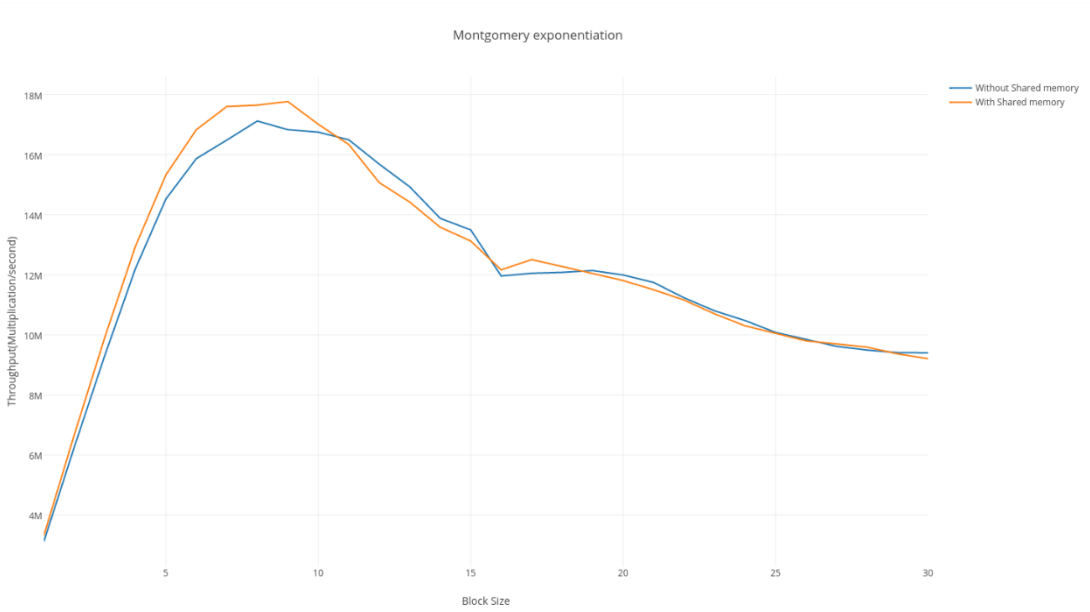


Figure 7: Montgomery exponentiation with shared memory vs without shared memory

As we expected, the efficiency of Montgomery exponentiation increased after the implementation of shared memory. With the small improvement, we are able to compute $1.7768342 * 10^7$ exponentiation per second with optimum block size of 9.

After getting our best performing result, we compared it to the prior work to the paper of Niall Emmart and Charles Weems.

TABLE VII
BEST PERFORMING MODEL BY SIZE AND CARD

	256 bits			512 bits			1024 bits			2048 bits		
Card	Model	Exps/s (1000s)	Utiliz.	Model	Exps/s (1000s)	Utiliz.	Model	Exps/s (1000s)	Utiliz.	Model	Exps/s	Utiliz.
9800	Sampled	545.61	77.3%	Sampled	51.63	50.3%	Distrib	3.67[4]	57.0%	Distrib	478[4]	58.0%
260	Sampled	822.42	78.7%	Sampled	118.72	78.0%	Distrib	5.68[4]	59.5%	Distrib	643[4]	52.7%
580	Three N	5806.28	95.1%	Three N	765.94	94.7%	Local	85.58	88.6%	Distrib	9163	95.6%
680	Three N	3913.66	89.9%	Three N	566.97	94.5%	Local	59.72	86.8%	Distrib	6476	95.3%
780Ti	Three N	6753.25	87.1%	Three N	998.81	93.4%	Three N	127.85	95.1%	Distrib	10778	89.4%
750Ti	Three N	1736.34	90.2%	Three N	204.49	80.6%	Three N	22.75	69.1%	Distrib	2052	61.1%

Corresponding Model Parameters

	256 bits, W=4 Model / Parameters	512 bits, W=5 Model / Parameters	1024 bits, W=6 Model / Parameters	2048 bits, W=6 or 7 Model / Parameters
9800	Sampled: LG=192, S=22	Sampled: LG=128, S=21, RM=60	Distrib: LG=64, T=2	Distrib: LG=64, T=4, W=6
260	Sampled: LG=128, S=22	Sampled: LG=192, S=21	Distrib: LG=64, T=2	Distrib: LG=64, T=4, W=6
580	Three N: LG=128, KS=1, KM=1	Three N: LG=128, KS=1, KM=1, RM=50	Local: LG=128	Distrib: LG=128, T=8, W=6
680	Three N: LG=128, KS=1, KM=1	Three N: LG=128, KS=2, KM=1	Local: LG=128	Distrib: LG=128, T=8, W=6
780Ti	Three N: LG=128, KS=1, KM=1	Three N: LG=128, KS=2, KM=1, RM=63	Three N: LG=128, KS=1, KM=1, RM=126	Distrib: LG=128, T=8, W=7
750Ti	Three N: LG=128, KS=0, KM=1	Three N: LG=512, KS=0, KM=0	Three N: LG=128, KS=0, KM=0, RM=126	Distrib: LG=128, T=8, W=7

TABLE KEY: LG=Launch Geometry KS=Karatsuba Squaring levels KM=Karatsuba Multiplication Levels S=Bits per Sample
T=Threads per exponentiation instance RM=Register max (specified via ptxas option -maxrregcount) W=Window Size

Figure 8: Best performance model table(N. Emmart and C, 2015)

With the comparison of 2048 bits, the optimum performance by 780Ti is able to compute 10778000 modular exponentiation per second while our implementation is

capable to compute 17768342 modular exponentiation per second, which improved the performance about 39%.

With the increasing reliance on technology in daily life such as online banking transaction, email, communication service, social network and infrastructure control system as the internet evolves and computer network become bigger and bigger, it is becoming more and more crucial to secure every aspect of online data and information through protected computer system and network. Therefore, a fast computation of cryptosystem is needed. With the library developed, it can be used to process encryption and decryption in a faster manner with GPU technologies.

Bibliography

Bassil, Y. and Barbar, A. (2016). *Sequential & Parallel Algorithms For the Addition of Big-Integer Numbers*. [online] Available at: https://arxiv.org/ftp/arxiv/papers/1204/1204.0232.pdf [Accessed 7 Apr. 2016].

Diffie, W.; Hellman, M. (1976). "New directions in cryptography" (PDF). IEEE Transactions on Information Theory 22 (6).

Docs.nvidia.com. (2017). Best Practices Guide :: CUDA Toolkit Documentation. [online] Available at: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz4dAQyuGa1 [Accessed 3 Apr. 2017].

K. Zhao and X. Chu (2010) , *GPUMP: A Multiple-Precision Integer Library for GPUs*. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, Bradford, 2010

Khoirudin, and Shun-Liang, J. (2015). GPU Application in Cuda Memory. ACIJ, 6(2), pp.01-10.

Kaya Koc, C., Acar, T. and Kaliski, B. (1996). Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro, 16(3), pp.26-33.

Langer, B. (2016). *Arbitrary-Precision Arithmetic on the GPU*. [online] Available at: http://www.cescg.org/CESCG-2015/papers/Langer-Arbitrary-Precision_Arithmetics_on_the_GPU.pdf [Accessed 7 Apr. 2016].

Menezes, A., Van Oorschot, P. and Vanstone, S. (1997). Handbook of applied cryptography. Boca Raton: CRC Press.

N. Emmart and C. Weems, "Pushing the Performance Envelope of Modular Exponentiation Across Multiple Generations of GPUs," 2015 IEEE International Parallel and Distributed Processing Symposium, Hyderabad, 2015, pp. 166-176.

Nigerianewsday.com. (2016). An Introduction to OpenCL™ Programming with AMD GPUs - AMD & Acceleware Webinar. [online] Available at: http://nigerianewsday.com/opencl-programming-guide.htm [Accessed 7 Apr. 2016].

NVIDIA CUDA Programming Guide. (2012). 4th ed. [ebook] NVIDIA Corporation, pp.8-11. Available at:

https://www.cs.unc.edu/~prins/Classes/633/Readings/CUDA_C_Programming_Guide_4.2.pdf [Accessed 7 Apr. 2016].

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU computing. IEEE Proceedings, May 2008, 879-899.

Takatoshi Nakayama and Daisuke Takahashi: Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing, Proc. 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011), pp. 343--349 (2011).

Appendices

Data used for Figure 1 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)	
	Unstructured Data	Structured Data
1	1400147.464838	2364254.496644
2	3018878.145831	4596557.777053
3	4878346.764303	6726090.988515
4	6331284.344676	9032730.413246
5	6054258.660490	10250882.882406
6	5862266.507188	10617868.199592
7	6073356.797868	10112906.948522
8	6607136.207250	9935453.410894
9	7104087.118730	9833672.093613
10	7471651.195581	9842303.014909
11	7675175.832056	9715354.041118
12	7638904.057538	9590602.775594
13	7503367.464168	9481425.749395
14	7381718.054022	9271217.867328
15	7160496.588766	9182132.901544
16	5835368.795942	8667993.879299
17	6034943.974495	8933808.405549
18	5857927.773255	9278597.926789
19	5939193.494941	9396294.047811
20	6047853.962380	9494846.753692
21	6083109.422340	9489924.884948
22	6017204.639584	9352990.672001
23	5909404.201299	9276312.360612
24	5787734.160624	8969764.749908
25	5590873.013669	8723744.421044
26	5518140.166858	8555247.809025
27	5458831.024447	8535993.664009
28	5384000.695204	8475846.252575
29	5367606.953028	8406918.186094
30	5312784.251133	8433192.104745

Data used for Figure 2 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)	
	Unstructured Data	Structured Data
1	1406254.406892	2328471.473317
2	3041342.989728	4615068.140506
3	4956461.358652	6779585.125786
4	6604690.371033	9037997.349835
5	6603114.734055	11175638.977613
6	6024097.271352	11342885.093304
7	5655228.585650	10132125.017860
8	5689508.505672	9472622.673104
9	5974330.692859	9268179.843272
10	6303166.000431	9186305.031707
11	6618233.162088	8729339.044333
12	6768698.867975	8298635.180740
13	6888647.265362	8179613.235097
14	6865525.834855	8000557.999056
15	6814717.606386	7973797.579403
16	5798710.368375	7722238.248303
17	6007344.259215	7769507.136570
18	5805903.369285	7791788.059614
19	5626489.252057	7844835.202280
20	5687119.581833	7849926.365408
21	5744447.971537	7861813.743998
22	5768707.819068	7841369.281482
23	5894574.398123	7582703.986064
24	5952632.082898	7856207.798527
25	5946680.508957	7956962.976120
26	5838088.684033	8028003.583306
27	5765159.296736	8075545.497103
28	5628573.987322	8109797.114492
29	5500454.253018	8111652.081142
30	5400302.757368	8174306.823765

Data used for Figure 3 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)	
	SOS	CIOS
1	2364254.496644	2328471.473317
2	4596557.777053	4615068.140506
3	6726090.988515	6779585.125786
4	9032730.413246	9037997.349835
5	10250882.882406	11175638.977613
6	10617868.199592	11342885.093304
7	10112906.948522	10132125.017860
8	9935453.410894	9472622.673104
9	9833672.093613	9268179.843272
10	9842303.014909	9186305.031707
11	9715354.041118	8729339.044333
12	9590602.775594	8298635.180740
13	9481425.749395	8179613.235097
14	9271217.867328	8000557.999056
15	9182132.901544	7973797.579403
16	8667993.879299	7722238.248303
17	8933808.405549	7769507.136570
18	9278597.926789	7791788.059614
19	9396294.047811	7844835.202280
20	9494846.753692	7849926.365408
21	9489924.884948	7861813.743998
22	9352990.672001	7841369.281482
23	9276312.360612	7582703.986064
24	8969764.749908	7856207.798527
25	8723744.421044	7956962.976120
26	8555247.809025	8028003.583306
27	8535993.664009	8075545.497103
28	8475846.252575	8109797.114492
29	8406918.186094	8111652.081142
30	8433192.104745	8174306.823765

Data used for Figure 4 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)			
	All CIOS	All SOS	CIOS with normal SOS	CIOS with squaring SOS
1	2843506.512122	2694962.839663	2833087.242743	3122018.288235
2	5702663.347258	5411452.516474	5671978.598818	6273119.666435
3	8533818.245819	8004211.359837	8456577.032458	9321445.804593
4	11307032.517706	10598046.341906	11078826.045543	12201392.599069
5	13886821.144992	12916660.518587	13398090.163634	14520735.143830
6	14904082.450371	13520512.104346	14144555.570443	15868218.367919
7	12313617.721053	13130552.736575	13507147.261474	16485963.655920
8	10861152.098662	11168724.438097	11552507.225155	17120393.314583
9	10347098.648845	10678931.002801	11056594.017876	16832035.789563
10	10078415.862989	10268301.018321	10629432.229169	16748888.408109
11	9430649.666129	10064677.146364	10413048.071073	16496206.837404
12	8863396.164550	9908019.718581	10210939.706359	15684163.842564
13	8551716.090488	9823912.505058	10163337.901528	14929894.109594
14	8357155.143395	9641203.687927	9989057.571279	13879082.082894
15	8197039.457959	9463169.607767	9801226.630490	13495494.106131
16	8014260.943547	8988915.105872	9319487.708615	11959837.274403
17	8016040.796483	9196339.736010	9570517.348923	12051677.743322
18	8031646.116380	9479285.312735	9809089.499316	12081828.748415
19	8048859.308229	9900848.828084	10231030.608586	12141799.286305
20	8061988.114551	9901105.841186	10221610.959840	11993807.737497
21	8085725.072440	9825679.746020	10171099.008071	11744521.864432
22	8151951.664530	9598185.148646	9655095.639168	11233133.152920
23	8217565.639183	8890253.315337	9274535.838157	10800914.041494
24	8266046.857780	8747096.427929	9055728.367258	10476426.262910
25	8239698.328123	8603232.287460	8880617.279262	10082182.369464
26	8239913.648351	8432262.268016	8690660.946633	9857151.749929
27	8234722.173077	8321634.785455	8532032.406308	9618447.454008
28	8258815.573285	8291207.355358	8480263.786038	9494465.318541
29	8286529.769429	8310755.574713	8489234.001692	9409513.192009
30	8334669.528281	8302128.554653	8483125.232379	9402245.453967

Data used for Figure 5 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)	
CIOS	Without shared memory	With shared memory
1	2328471.473317	2508565.082499
2	4615068.140506	5155940.909771
3	6779585.125786	7533376.332310
4	9037997.349835	9573027.219739
5	11175638.977613	11893014.877733
6	11342885.093304	12527315.412592
7	10132125.017860	11645539.988623
8	9472622.673104	10999279.627445
9	9268179.843272	11051989.846461
10	9186305.031707	10618252.624584
11	8729339.044333	10136260.709774
12	8298635.180740	9914953.854027
13	8179613.235097	9459367.828859
14	8000557.999056	9150759.913047
15	7973797.579403	8987636.158406
16	7722238.248303	8659032.696827
17	7769507.136570	8589649.263688
18	7791788.059614	8450243.738207
19	7844835.202280	8580203.530546
20	7849926.365408	8551573.338483
21	7861813.743998	8425399.708542
22	7841369.281482	8395287.375718
23	7582703.986064	8333485.414809
24	7856207.798527	8346521.769098
25	7956962.976120	8317207.315686
26	8028003.583306	8324708.741640
27	8075545.497103	8322454.424907
28	8109797.114492	8287402.538916
29	8111652.081142	8319623.858646
30	8174306.823765	8185729.602412

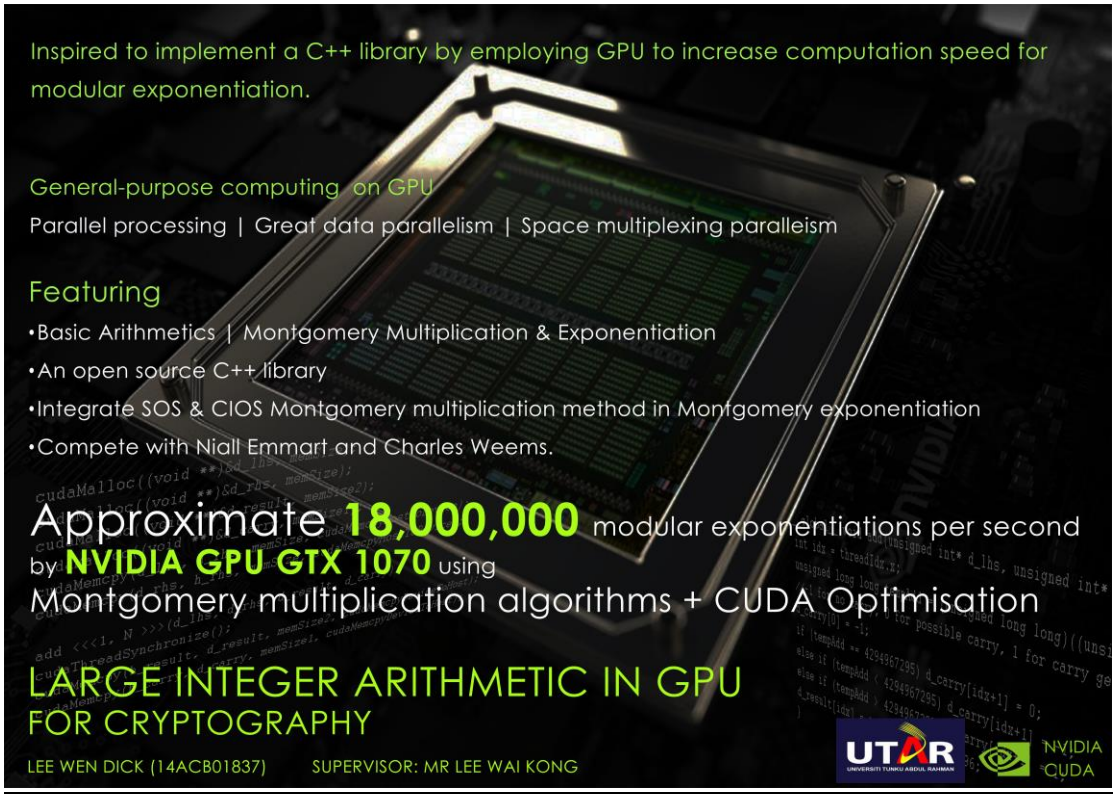
Data used for Figure 6 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)	
SOS with squaring optimisation	Without shared memory	With shared memory
1	2602972.833452	2662941.850137
2	5036818.197475	5155985.490108
3	7864587.058147	8192921.945504
4	9730331.052991	10691811.293619
5	11390146.072617	12180787.803536
6	13247908.102564	14685496.917635
7	13439906.770717	14962162.729861
8	14271047.691608	15538023.524989
9	14274579.803840	16068243.420969
10	14276026.009910	15654483.964308
11	14296303.707803	14979843.373748
12	13867139.074260	14069160.260052
13	13501290.306240	13997817.476025
14	12632663.458253	12972654.974578
15	12732513.373201	13153286.879287
16	11019516.906015	11379813.274415
17	11316190.564863	11486650.558964
18	11418460.380941	11824679.363250
19	11264896.182380	11776247.932966
20	11290892.454613	11630040.277319
21	11169254.015771	11414943.252669
22	10705749.990503	10998830.920480
23	10475439.571463	10679844.132921
24	10078344.218179	10289039.707261
25	9849282.644225	9951854.043393
26	9654033.766965	9745089.736255
27	9571467.676337	9623686.766958
28	9292000.544672	9528054.855065
29	9279016.116327	9428769.962285
30	9204734.867913	9215150.734584

Data used for Figure 7 in Chapter 5:

Number of block(1024 threads per block)	Throughput(Multiplication per second)	
Montgomery Exponentiation	Without shared memory	With shared memory
1	3122018.288235	3323180.304031
2	6273119.666435	6681806.072885
3	9321445.804593	9924198.060072
4	12201392.599069	12941967.254239
5	14520735.143830	15324437.599108
6	15868218.367919	16831728.490093
7	16485963.655920	17604226.643155
8	17120393.314583	17648710.675979
9	16832035.789563	17768342.281259
10	16748888.408109	17011541.510669
11	16496206.837404	16338366.709305
12	15684163.842564	15069433.190728
13	14929894.109594	14421983.661440
14	13879082.082894	13584407.532982
15	13495494.106131	13124976.378879
16	11959837.274403	12165866.856457
17	12051677.743322	12505796.262106
18	12081828.748415	12282810.027172
19	12141799.286305	12044294.836269
20	11993807.737497	11807159.773226
21	11744521.864432	11503425.368358
22	11233133.152920	11157682.921541
23	10800914.041494	10699926.517346
24	10476426.262910	10302270.505581
25	10082182.369464	10047853.428199
26	9857151.749929	9800888.467021
27	9618447.454008	9697965.845483
28	9494465.318541	9592178.895553
29	9409513.192009	9368167.323048
30	9402245.453967	9206738.751390

POSTER


Inspired to implement a C++ library by employing GPU to increase computation speed for modular exponentiation.
General-purpose computing on GPU
Parallel processing | Great data parallelism | Space multiplexing paralleism
Featuring
•Basic Arithmetics | Montgomery Multiplication & Exponentiation
•An open source C++ library
•Integrate SOS & CIOS Montgomery multiplication method in Montgomery exponentiation
•Compete with Niall Emmart and Charles Weems.
Approximate 18,000,000 modular exponentiations per second by NVIDIA GPU GTX 1070 using Montgomery multiplication algorithms + CUDA Optimisation
LARGE INTEGER ARITHMETIC IN GPU FOR CRYPTOGRAPHY
LEE WEN DICK (14ACB01837)
SUPERVISOR: MR LEE WAI KONG
UTAR
NVIDIA CUDA