Spring 2013

Analysis of Parallel Montgomery Multiplication in CUDA

Yuheng Liu

Analysis of Parallel Montgomery Multiplication in CUDA

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Yuheng Liu

May 2013

The Designated Project Committee Approves the Project Titled

Analysis of Parallel Montgomery Multiplication in CUDA

by

Yuheng Liu

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2013

Mark Stamp	Department of Computer Science
Richard Low	Department of Mathematics
Soon Tee Teoh	Department of Computer Science

ABSTRACT

Analysis of Parallel Montgomery Multiplication in CUDA

by Yuheng Liu

For a given level of security, elliptic curve cryptography (ECC) offers improved efficiency over classic public key implementations. Point multiplication is the most common operation in ECC and, consequently, any significant improvement in performance will likely require accelerating point multiplication.

In ECC, the Montgomery algorithm is widely used for point multiplication. The primary purpose of this project is to implement and analyze a parallel implementation of the Montgomery algorithm as it is used in ECC. Specifically, the performance of CPU-based Montgomery multiplication and a GPU-based implementation in CUDA are compared.

ACKNOWLEDGMENTS

I would like to thank to my advisor, Dr. Mark Stamp, who helped me with my research and provided continuous support, and immense patience during my studying of the project. Without Dr.Stamp's persistent help and guidance, this paper would not have been finished.

I also want to thank my committee members, Dr. Richard Low and Dr. Soon Tee Teoh, for their patience and support of my project.

I also would like to thank my family for their understanding, encouragement, and support.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

For the past 20 years, throughout many types of communication, especially over the Internet, public key cryptography has become the major secure form of communication in the area of security. Public key cryptography is a crypto system with two different keys. One key is called a public key, which is used for encryption, and the other key is called a private key, which is used for decryption.The private key is exclusive to users themselves and should not be revealed to anyone else. The public key, as its named, is an open to everyone [1].

With higher security demands in massive computation or better performance in mobile devices, new techniques have been developed over the past 20 years in public key cryptography. In all new public key cryptography algorithms, the elliptic curve is the most promising and best crypto when compared to the first generation public key cryptography, such as RSA and Diffie-Hellman [2]. Similar to RSA, ECC offers vast abilities in key generation, digital signatures, secure key distribution, and encryption, etc [3]. Elliptic curves have more advantages in many aspects due to their rich and complex mathematical structures. The strength of an elliptic curve cryptography is increasing with the current booming mobile device industry. Wireless devices and cellular phones have more limitations regarding computing power, battery supply, storage, and processing capacity than computers, so efficiency of resource distribution is crucial. One outstanding feature of ECC is that it provides an equivalent security level to RSA with fewer key size, achieving a promising high security level, and, at the same time, saves precious resources on mobile devices [3].

According to [4], we realize that a core feature for changing the execution time of ECC is multiplication. As the most crucial mathematic operation, multiplication has been studied in many papers in order to improve the performance of ECC.

1.1 Previous work

Research in both software and hardware have been done in terms of accelerating multiplication of elliptic curve cryptography. In paper [5], an algorithm is presented that multiplies polynomials with integer coefficients, efficiently using the Number Theoretic transform (NTT) on GPU. Another article [7] utilizes residue number system to modular exponentiation in order to attain the best performance on a GPU in terms of throughput and latency. In the hardware research area, we have seen the development of [6] a cryptographic processor architecture for elliptic curve in GF(2^m) in an attemp to achieve higher performance [6]. By using FPGA and an applied, widely used, Montgomery algorithm, this study [8] perform an extreme fast point multiplication of elliptic curve. They claim that their result is more than 3.4 times faster than other work reported within the literature.

1.2 Organization

This research project introduces a parallel method for multiplication based on the Montgomery algorithm and implementation in CUDA programming language in an attempt to achieve a higher execute time.

The project is organized as follows. Chapter 2 provides background knowledge and presents an overview of elliptic curve cryptography. In Chapter 3, we present the mathematics of elliptic curve cryptography which is the basis for this project. Chapter 4 describes the Montgomery algorithm. In Chapter 5, we briefly give an

introduction for the parallel technique CUDA. In Chapter 6, we present the design and implementation of parallel Montgomery multiplication in CUDA architecture. Our experiments, results, and an analysis are discussed in chapter 7. In the end, we present our conclusion and future work.

CHAPTER 2

Elliptic Curve Cryptography

2.1 Background

Elliptic Curve Cryptography(ECC) was first discovered by Neal Koblitz and Victor Miller in 1985 independently. It solves a discrete logarithm problem which is the core and strength leading to why elliptic curve cryptography is more secure than RSA [8].

There are many advantages to using various aspects of elliptic curve cryptography. One is that it can reach the same level of security as RSA while using a smaller key size; this not only saves computer power comsumption, especially when using wireless devices, but also requires less memory. This is a very attractive feature for mobile devices, tablets, and especially on smart cards. Another advantage is from an internal structure, mathematics, is that ECC is aimed at solving a discrete logarithm problem, which is more difficult than the problem that RSA can solve, which is integer factorization [9]. We will discuss a discrete logarithm problem in the following chapter.

An simple elliptic curve E can be displayed as follows [10]:

$$E : y^2 = x^3 + ax + b$$

We denote a special point at infinity ∞. A typical elliptic curve, with points, is displayed in Figure 1 [9].

Figure 1 shows a geometric figure of a point addition operation in ECC. It can be defined as drawing a line between two points P and Q [9] in an attempt to gain the

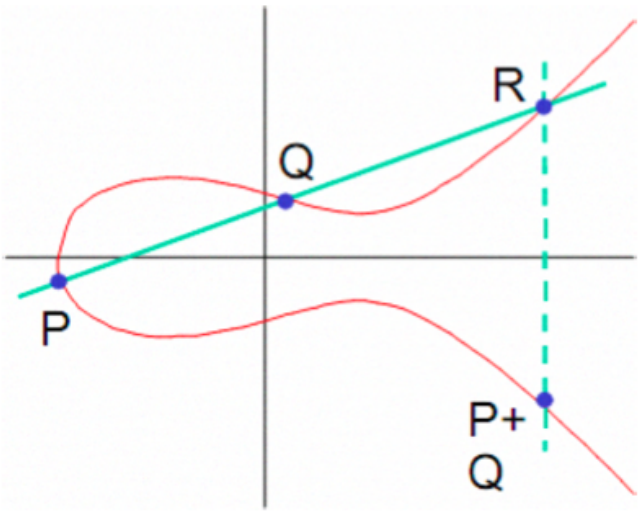

Figure 1: An ellitpic curve

third point R. Discussed in a previous study [8], if geometric elliptic curves are over an infinite field of real numbers, it is not practical for a cryptography background, though better results are produced [8]. Therefore, an elliptic curve cryptographic system should be designed in a finite field with large key size in order to satisfy security and real world requirements. The specific mathematical illustration will be explained in the next chapter.

2.2 Elliptic Curve Discrete Logarithm Problem (ECDLP)

ECDLP has been considered as a difficult problem to solve in modern cryptography, especially in public key cryptography. The problem is described as a point multiplication of ECC that can be interpreted as follows: we have a point P on the elliptic curve, in order to get point Q on the same elliptic curve, point P is added k times to itself.

So ECDLP involves a scalar multiplication. When we have determined k and P

then it is quite easier to find kP, which is Q. If we know point P and point Q, its very hard to find the scalar K. In the case that scalar K becomes larger, it is considered that K is computationally infeasible to obtain [11].

2.3 Fundamental of ECC

When it comes to choose an elliptic curve, for the level of cryptosystem is in consideration, an important aspect when selecting an elliptic curve is to avoid supersingular curves [12]. The math can be much faster since several terms go to 0. The curves presented in this paper are nonsupersingular and, to date, require fully exponential algorithms to crack.

2.4 History of ECC

Nearly 20 years ago, elliptic curve cryptography was seen as one of the Discrete Logarithm Problem-based crypto systems [11]. In 1985, Miller and Koblitz discovered that elliptic curves could be very useful for public key cryptography. During that time it was not very efficient to perform the needed calculations. By the late 1990s, the way to implement the scheme was ten times faster, which allowed the performance of elliptic curve mathematics to take the same amount of time as the implementation of integer factorization for the same number of bits.

2.5 ECC vs RSA

As we mentioned in the previous chapter, one reason why we favor an elliptic curve cryptosystem is that it requires fewer bits for the same security compared to a cryptographical system, such as RSA and PGP. It's known that security increases sub exponentially in RSA cryptography, while in elliptic curve cryptography the increase

of the security is exponentially [2]. This significant feature leads to a net reduction in computing power, cost, memory storage, and execute timing, which security companies favor and is also endorsed by National Security Agency.

Although ECC is a more promising alternative as the foundation for future Internet security, currently most applications use RSA. One reason is that the most widespread public key algorithm, RSA, came first before ECC, its mathematics are simpler than those involved in elliptic curve cryptography, and RSA is easier to implement. Users like to use something that they are familiar with and find easy to understand; ECC still has a long way to go before it becomes the mainstay of public key cryptosystem. One other reason for its widespread use is that a company called Certicomm owns many of the patents, including the curves themselves, and is making it difficult to deal with ECC.

CHAPTER 3

Mathematics of Elliptic Curve Cryptography

The ability to add any two points on a particular curve in order to obtain the third point on the same curve is where the magic of Elliptic curves comes in. In geometric meaning, refer to Figure 1, point addition is described as finding the Summation $P + Q$, a line is drawn between point P and point Q, and arrive at the third point $P + Q = R$, which is also on the elliptic curve. The difficulty to crack ECC is exponential with key size if the elliptic curve parameters are base point and the form of curves are chosen correctly.

The mathematics of Elliptic curves appear in different shapes when it comes to different fields. This paper focuses on Elliptic curves in a Galois field. The mathematics involved in this field are illustrated in detail.

3.1 Finite fields

A field satisfies the following properties [5]: a field has an additive identity element for the addition rules; a field has a multiplicative identity element for the multiplication rules; and every element in a field has an inverse.

In honor of Evariste Galois, a finite field is also named as the Galois field, which consists of an addition operation, a multiplication operation, and an inversion operation. Noted here is that these two operations, addition and multiplication, are not traditional mathematical addition and multiplication in different fields [8].

Prime fields GF(Z_p) and binary fields GF(2^m) are two main types of finite fields that are widely used in elliptic curve cryptography. We count all integers between 1

and a prime p in prime fields. Binary fields have a set number of bits in binary form. For elliptic curves, this corresponds to

$$y^2 = x^3 + ax + b \bmod p \text{ for prime fields}$$

and

$$y^2 + xy = x^3 + ax + b \text{ for binary fields.}$$

In binary fields, there are a polynomial basis and a normal basis that can be selected as the representation of elements. We focus on a polynomial basis in this project since it is faster and easier to realize. Figure 2 shows the overall picture of elliptic curves taxnomy.

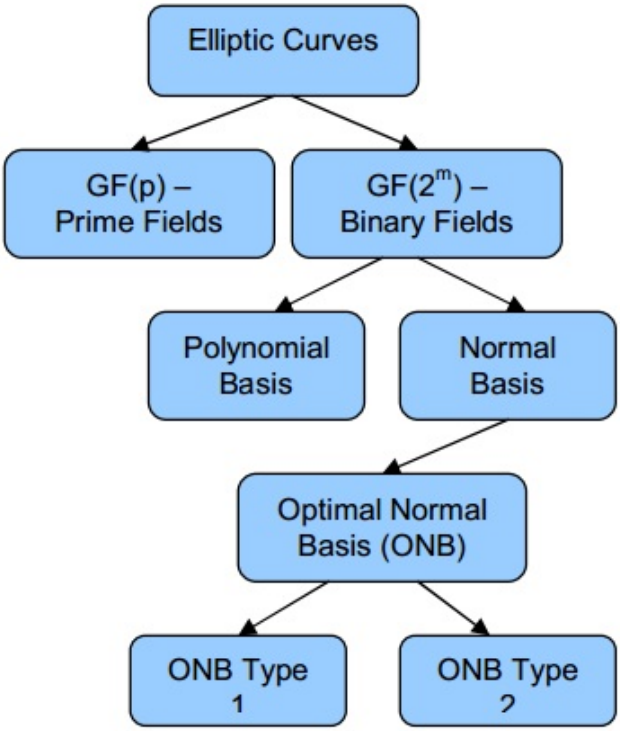

Figure 2: Structure of Elliptic Curves

3.2 Characteristic of a field

The meaning of the characteristic of a field can be interpreted as the number of values in a field. Therefore, if a field has characteristic p, we write the characteristic of that field as F_p. This kind of finite field is also known as Galois Fields in honor of the mathematician who was the first one to describe them.

3.3 Polynomial basis

3.3.1 Introduction to polynomials

The definition of a polynomial is the sum of different powers of a variable. For example:

$$x^5 + 3x^2 + 6$$

In this case, the arithmetic above has not set the polynomial equal to any specific constant, thus there is no way to know the value of x. Furthermore, the arithmetic assumes that the coefficients of the polynomial are real and that x is real or possibly complex. For computer arithmetic, to prevent numbers from exceeding storage capacity, we select coefficients modulo 10 in this example. Suppose we add another polynomial to the above one:

$$x^3 + 7x^2 + 6$$

Since the coefficients are modulo 10, we have:

$$x^5 + x^3 + 2$$

This type of arithmetic is called a polynomial basis. Suppose a prime number is chosen for the modulus of the coefficients. Then we have a polynomial basis over a finite field. This is an important concept for cryptography, error correction, and so on. The related deep knowledge can be read [19].

For our project, and for a computer and cryptography perspective, we choose the prime number 2 as the modulus, which means the coefficients can only take on the value of zero or one. For computer storage, the polynomials can be stored as continuous bits in the memory with each bit position representing the coefficient of a power of x. For example, a polynomial as such:

$$x^8 + x^5 + x^3 + x + 2$$

would be stored on a computer as:
100100011 represents each position of bits representing the coefficient of a power of x and 876543210 represents the corresponging index.

3.3.2 Polynomial basis represenation

The standard binary field element representation is the coefficients of a polynomial. We take the coefficients modulo 2 to facilitate an efficient implementation and make them binary. This allows an element to be defined as a string of bits. The polynomial representation of an element in GF (2^m) is given as [21]:

$$a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_2x^2 + a_1x + a_0 : a_i i \in (0, 1)$$

3.3.3 Addition and subtraction

Addition modulo 2 is just exclusive-or. For example, $1 + 1 = 0 \bmod 2$, $0 + 1 = 1 \bmod 2$, and $0 + 0 = 0 \bmod 2$. In a computer operation, we can use the XOR instruction to add two polynomials rapidly. There is no carry to propagate, which is a very fast operation, even in high-level languages.

Addition and subtraction are true and only true for mod 2 coefficients. For example, $1 - 1 = 0$, since $1 + 1 = 0 \bmod 2$, addition equals to subtraction. Therefore,

in binary modular mathematics, we recognize addition and subtraction as the same type of operation.

3.3.4 Multiplication

Modular multiplication is an important operation in popular Public Key Cryptography, such as ECC and RSA, with a different implementation approach.

In a polynomial basis for ECC, multiplication is simply a shift and exclusive-or operation. Note that the highest exponent keeps increasing in the polynomial. The high exponent is called the degree of the polynomial. We never want is the degree of polynomials to overflow our storage capacity. So we use modular math to solve this problem. But there is a difference from an integer basis, instead of modulo being a prime number, we want modulo to be a prime polynomial. A prime polynomial, also known as an irreducible polynomial, is an identical concept using different terms. An irreducible polynomial is like a prime number and has no polynomial factors.

3.3.5 Inversion

An inversion is defined as follows: Given $a(x) \in GF(2^m)$, find $a(x)^{-1}$ such that $a(x)a(x)^{-1} \equiv 1 \bmod f(x)$, where $f(x)$ is the irreducible polynomial. The most popular method is the Extended Euclidean Algorithm, seen in Algorithm 3.1 . For an implementation optimized for GF(2^m), refer to Algorithm 3.2 [21, 5].

3.4 Mathematics of elliptic curves over Galois field

The Galois field, usually in the form of F_2, could be represented either as a polynomial basis or normal basis. An Elliptic curve equation could be based on one

curve, which is called a Weierstrass form elliptic curve:

$$y^2 + xy = x^3 + a_2x^2 + a_6$$

The variable x and y cover a plane, and x, y can be integers, real, complex, polynomial basis, or optimal normal basis. This is the part that makes the math deep. According to previous research:

$$y^2 + y = x^3 + a_4x + a_6$$

$$y^2 + xy = x^3 + a_2x^2 + a_6$$

The first equation is called a supersingular curve, which is not what we are looking for though it can be calculated quickly. What we need is the second curve of the equation, which is called nonsupersingular. Until now, no known attack has taken full exponential time to conquer this method. Therefore, this form is perfect for cryptographic applications. Another point we need to consider is to carefully choose the correct coefficients in order to achieve maximum security benefits; incorrectly selecting the wrong coefficients would let the curve become an easy target for a cryptanalyst to attack [5].

3.5 Point Addition

Suppose we have two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ that are distinct. The sum of P and Q is R which finds a line across P and Q. Arithmatic notation [5]:

Given:

$$P = (x_1, y_1)$$

$$Q = (x_2, y_2)$$

Then:

$$R = P + Q = (x_3, y_3)$$

if $P \neq Q$:

$$\theta = \frac{y_2 - y_1}{x_2 - x_1}$$

$$x_3 = \theta^2 + \theta + x_1 + x_2 + a_2$$

$$y_3 = \theta(x_1 + x_3) - y_1$$

if $P = Q$:

$$\theta = x + \frac{y}{x}$$

$$x_3 = \theta^2 + \theta + a_2$$

$$y_3 = x^2 + (\theta + 1)x_3$$

The geometric meaning as follow in Figure A.13.

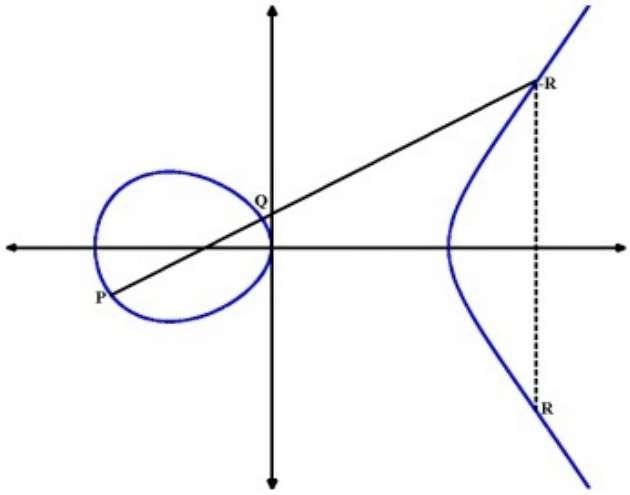

Figure 3: The geometric meaning of point addition P + Q = R

3.6 Point Doubling

Arithmetic notation [5]:

When y_P is not 0,

$2P = R$ where

$$s = \frac{3x_P^2 + a}{2y_P}$$

$$x_R = s^2 - 2x_P \text{ and } y_R = -y_P + s(x_P - x_R)$$

Figure 4 The geometric meaning of point doubling.

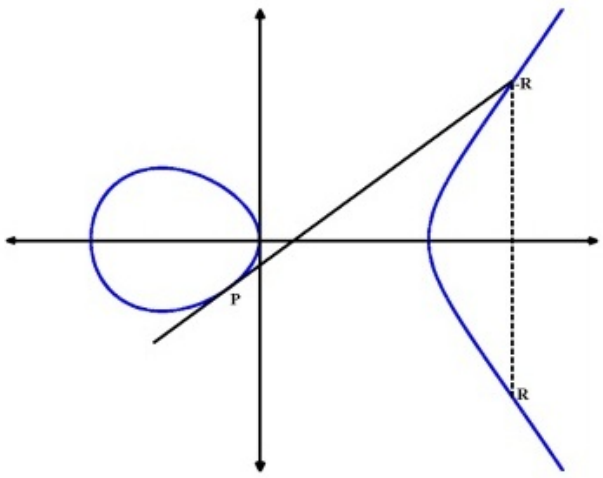

Figure 4: Point doubling $2P = R$

CHAPTER 4

Montgomery Algorithm

Montgomery's algorithm is used for an efficient modular multiplication. As we know, division operation is very time consuming. Montgomery's algorithm can efficiently calculate modular multiplication by not performing a division operation. From an elliptic curve cryptography perspective, the Montgomery multiplication algorithm could be in different forms and different coordinate systems in order to achieve the best performance.

According to many cryptographic systems, modular multiplications usually operate in chains. For example, RSA cryptographic system [4], Paillier's probabilistic public-key scheme [16], Diffie-Hellman key exchange scheme [15], and elliptic curve cryptography are required for a chain of modular multiplication with a large integer or prime polynomial. Therefore, efficient design for implementing modular multiplication is essential.

For mathematic representation, Montgomery's algorithm and its other improvements for modular multiplication are displayed as follows. $x = a \times b \bmod n$. For modular exponentiation is given as: $x = a^b \bmod n$ [7].

First, the basic procedures for Montgomery's algorithm are convert all the operands to Montgomery representations; second, we perform Montgomery's algorithm for each operations as required; finally, we convert all Montgomery representation of operands back to their original representations [4].

4.1 General Montgomery multiplication algorithm

The Montgomery multiplication algorithm is a way to speed-up modular exponentiation [22]. One good feature of Montgomery's algorithm is that it works for any modular N [23]. Suppose we want to calculate the following operation:

$$ab \bmod N$$

we select $R = 2^k$, where k is large enough so that $R > N$. Because R is represented as a power of two, it is very easy for a computer to determine the result of modulo R. We know that the numbers used in computer computing are represeted as binary, so the operation is a k bits shift.

N' and R' are obtained as

$$RR' - NN' = 1$$

R' and N' can be found via Euclidean Algorithm.

Next, we need to find a Montgomery form, which are $a' = aR \bmod N$ and $b' = bR \bmod N$. The advantage and the power of a Montgomery form is that, when repeated, multiplication is required, as it performs modular exponentiation.

Then, two numbers are multiplied under a Montgomery form, so that we get a result which is also in a Montgomery form, refer to:

$a'b' = abR^2$

In order to have a result demonstrates a Montgomery form, we want to have $abR(\bmod N)$ not abR^2. The following is a mathematical derivation in terms of how to convert abR^2 to $abR(\bmod N)$.

Let $X = abR^2$, calculate

$$m = (X(\text{mod } R))N(\text{mod } R)$$

note that $R = 2^K$, so that all modular operations are efficient.

Then let

$$x = \frac{X + mN}{R(\text{mod } R)}$$

Verify x = abR (mod N)

$$x = xRR' = XR' = abR^2R' = abR(\text{mod } N)$$

CHAPTER 5

CUDA

The activities involved in cryptographic systems are computationally intensive, with some showing a significant features of parallelism. With the emergence of general-purpose computing for graphics processing units, much research regarding cryptographic computation has been done in these two cross areas. In this chapter, we briefly introduce a popular parallel technique, CUDA, which is presented by NVIDIA.

5.1 GPGPU

GPGPU is the short term for General-purpose computing on graphics processing units, and is a GPU technique generally dealing with graphics data computation, but also performs traditional application computation by CPU, such as in Cryptography [24].

With GPGPU, engineers who want to explore the new usage of GPU perform graphics data, as usual, in addition to taking advantage of applied stream processing applied on non-graphics data.

The process is shown as the figure 5.

5.2 GPGPU Programming Concepts

There are some critical concepts involved in GPGPU programming that include stream processing, GPU programming concepts, and GPU techniques. In conclusion, an application that is perfect for GPGPU programming has the feature of arithmetic

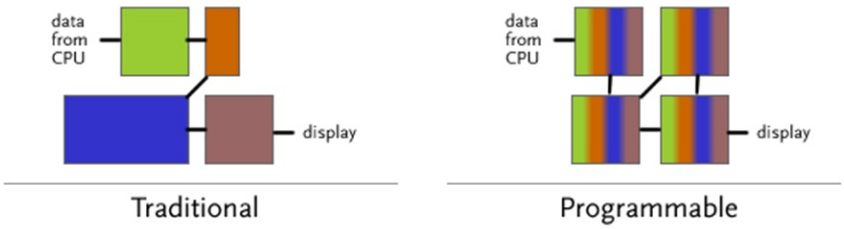

Figure 5: Traditional GC vs Programmable GC.

intensity, large data sets, minimal dependency between data elements, and high parallelism [25].

5.3 CUDA Architecture

NVIDIA introduces a general purpose parallel computing architecture, CUDA, whose purpose is for normal use in a traditional computation area using NVIDIAs CUDA-enabled graphics hardware [24]. A programmer can then use C, Fortran, OpenCL, and DirectCompute programming languages to compute in parallel using multi-core.

5.4 CUDA Programming Model

There are basically three core abstract concepts of a CUDA programming model, they are hierarchy of thread groups, shared memories, and barrier synchronization.

Introducing these abstract concepts, we can perform parallelism such as data parallelism, thread parallelism, or task parallelism, that can direct a developer whether to divide a problem they have into sub problems. Any available processor core can schedule and execute any threads for each block in any order, no matter if it is in sequential or in parallel. Therefore, a compiled CUDA program can run on any of cores in an NVIDA's GPU.

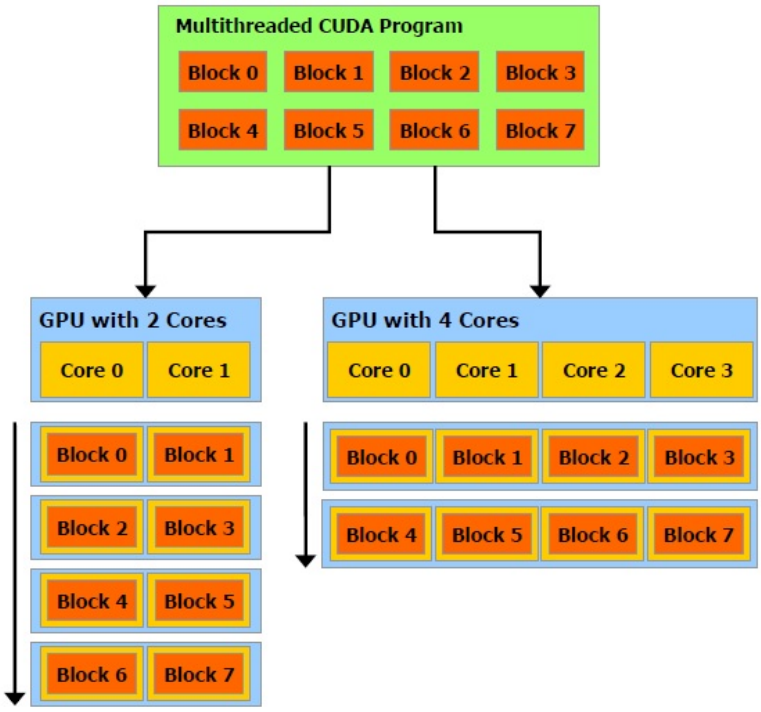

Figure 6: Automatically scaling

For Figure 6, a multithreaded program can execute independently in any blocks of threads [24]. Obviously, a multi-cores GPU can execute a multi-program in less time than a GPU with fewer cores.

5.5 CUDA Threading model

The threading model of CUDA is single instruction, multiple data, and is short for SIMD. In general, SIMD means the same instruction is performed on many pieces of different data. Several Processors share a common Control Unit and Memory. All processors receive the same instruction but operate on different data

Threads are grouped by a grid consisting of one dimension or two dimension blocks. A block contains a set of threads. A parallel code, called a kernel, is executed in multiple threads.

5.6 CUDA Memory architecture

Another important concept in parallel processing with multiple cores is how to deal with memory allocation. When we use GPU to do calculations, we need to write memory between the host (CPU) and the device (GPU) memory. Getting to know the various types of memory is essential for a programmer who wants to program in GPU with CUDA. We briefly explain the main four types of device memory: global memory, texture memory, constant memory, and shared memory.

Global memory is the largest available chunk of memory and the slowest memory store to access. Global memory is not cached.

Texture memory is a cached, read-only segment of memory. As it is named, texture memory is optimized for 2D spatial locality and most of its usage, in reality, is dealing with 2D textures.

Constant memory has the feature similar to texture memory. It's cached and has a read-only segment of memory that exists in the main memory of NVIDIA GPU. The difference between constant memory and texture memory is that it is not particular built for 2D spatial locality in an alternative way.

Shared memory shares memory across a unit block, so threads can access this same block of memory. It is also where Kernel function parameters are stored. By using a device code, other data can be moved into shared memory for manipulating.

CHAPTER 6

Implementation and Experiment

6.1 Purpose and design

There are three main goals of this project, which are as follows:

1. Explore normal method for implementing scalar multiplication of elliptic curve cryptography in finite field characteristic 2.

2. Improve scalar multiplication of elliptic curve cryptography using Montgomery's algorithm in a projective coordinate

3. Improve scalar multiplication of elliptic curve cryptography using Montgomery's algorithm in a projective coordinate

From a cryptographic perspective, elliptic curves can be defined over real numbers, complex numbers, and any other fields. This paper is interested on elliptic curves defined over finite fields with characteristics of 2(Binary field).

The paper focus on a scalar multiplication operation of an elliptic curve cryptography since scalar multiplication is the most important operation and is key to the speed of elliptic curve cryptography. An improved scalar multiplication is used in order to create an elliptic curve cryptography that is faster. Our study is using the most popular research available for elliptic curve cryptography.

In this chapter, basic concepts of scalar multiplication will be introduced for the preparation of the improved method. What follows are the implementations of regular scalar multiplication, applying Montgomery's algorithm to scalar multiplication, par-

allel Montgomery multiplication in CUDA programming language, and comparison to the other public key cryptography RSA.

6.2 Brief introduction of scalar multiplication

P is a point that belongs to an elliptic curve, and scalar K represents a positive integer. We calculate the summation kP which means we add point P to itself $K-1$ times. As known from previous research, scalar multiplication is also described as a discrete logarithm problem.

6.3 Hierarchy of scalar multiplication

The three layer hierarchy of scalar multiplication is depicted in Figure 7. In this way, the main implementation of scalar multiplication relies on the first and second layer of the model. The first layer presents basic arithmetic operations that are the basic foundation for scalar multiplication in elliptic curve cryptography. The second layer includes point doubling and point addition for performing point mathematics operations of elliptic curve crypto system. The third layer is the scalar multiplication which consists of the first two layers. This paper focuses on the second layer used to accelerate the speed of calculating scalar multiplication. There are also many algorithms and strategies for the efficiency of computing arithmetic operations like addition, squaring, division, multiplication, and inversion of the first layer; however, this is not the scope of this paper. For those who are interested in those topics, refer to [11].

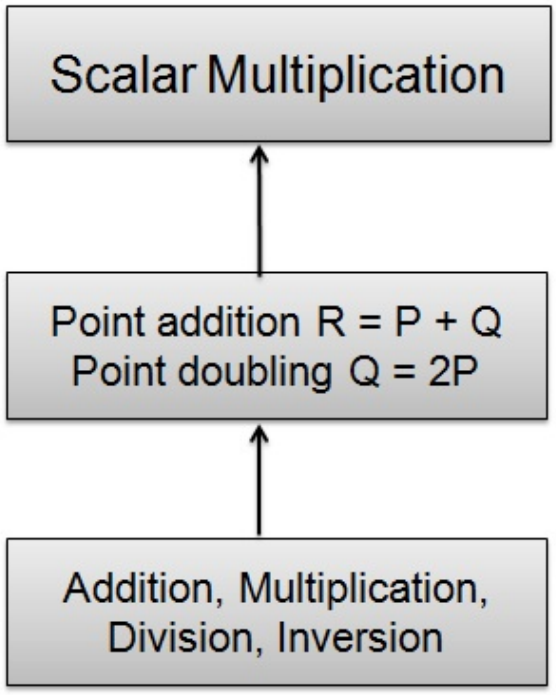

Figure 7: The three layer hierarchy of scalar multiplication

6.4 General scalar multiplication

In this section, we are implementing general scalar multiplication over a finite field. In order to perform a scalar multiplication, we need to define our data storage structure in order to storage the numbers we will use to manipulate, define elliptic curve parameters, and arithmetic operations that consist of scalar multiplication. The following section illustrate this information in detail.

6.4.1 Define data storage structure

```
define WORDSIZE (sizeof(int)*8)
define NUMBITS 163
define NUMWORD (NUMBITS/WORDSIZE)
define MAXLONG (NUMWORD+1)
typedef unsigned long ELEMENT;
typedef struct {
   ELEMENT e[MAXLONG];
} FIELD2N;
```

`WORDSIZE` represents the number of bits in a machine word. `NUMBITS` is the number of bits the polynomial math will be expected to work on, which is the degree of irreducible polynomial mathematically. `NUMWORD` is the maximum index of machine words into a polynomial array. `MAXLONG` represents the number of machine words needed to hold a polynomial. Once the above variables are defined, we can define field storage structure `FIELD2N`, which is a field of characteristic 2 GF(2^m).

6.4.2 Define EC parameters

Elliptic curve parameters in this project include the curve you choose and base point. We use the curve equation

$$y^2 + xy = x^3 + ax + b$$

where $a = 5$ and $b = 9$. We choose (007, 308338392 3736842520 1350130208) as our base point. This point is chosen by pre-calculated [11] and assured that it is on the curve we have selected.

```
typedef struct {
   INDEX form;
   FIELD2N a2;
   FIELD2N a6;
} CURVE;
```

```
typedef struct {
   FIELD2N x;
   FIELD2N y;
} POINT;

typedef struct {
   CURVE crv;
   POINT pnt;
} EC PARAMETER;
```

6.4.3 Implement polynomial arithmetic operations

Polynomial addition is the exclusive-or of two sets of coefficients. For example:

```
INDEX i;
FOR LOOP (i) c.e[i] = a.e[i] XOR b.e[i];
```

Polynomial multiplication is slightly different from integer multiplication. Two steps are needed to perform a complete polynomial modular multiplication. The first routine will be a multiplication of two polynomials; in the second routine, we need the product from the first routine modulo, a prime polynomial, in order to complete the polynomial multiplication.

A prime polynomial is also called an irreducible polynomial. In mathematics, a polynomial is defined as irreducible when it cannot be factored into the product of two, and for polynomials case whose coefficients are of a specified type [26]. Identifying whether a polynomial is a prime polynomial can be a large research topic in academy. In this paper we use the existing prime polynomials from [11] that were previously tested by mathematicians and engineers.

6.4.4 Point addition in affine coordinates

Affine coordinates are shown in the standard representation $P = (x, y)$ for elliptic curve points. Addition and doubling in affine coordinates are given in Chapter 3

respectively.

6.4.5 Implement point doubling

The elliptic curve doubling routine we used here is for Schroeppel's algorithm [11] over polymomial basis. Using an input with P_1, P_3 as our source and destination, we operate the equation $P_3 = 2 \cdot P_1$.

6.4.6 Scalar multiplication

Scalar multiplication will be calculated based on Koblizs balance expansion [11], which consists of point addition and point doubling.

6.5 Montgomery multiplication

In this section, Montgomery point multiplication is discussed in detail and is used to compute point addition and point doubling. Basic arithmetic operations are the same as the operations that are used in general scalar multiplication.

Lets define $P(x)$ as a irreducible polynomial over GF(2^m). We will make sure an elliptic curve $E(F_q)$ is non-supersingular and also set base points $(x, y) \in GF(2^m)$that satisfies the following equation,

$$y^2 + xy = x^3 + ax + b$$

First we define point $P = (x_1, y_1)$ and point $Q = (x_2, y_2)$ which belong to the curve. Then the sum of P and Q, $P + Q = (x_3, y_3)$ and the subtraction of P and Q, $P - Q = (x_4, y_4)$, also on the curve [27],

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + (\frac{x_1}{x_1 + x_2})^2$$

From the above equation, the x coordinates of P, Q and $P - Q$ which are x_1, x_2

and x_4 is all that we need to determine for the value of x_4, which are the summation of point P and Q. We use X/Z to represent the x coordinate of P. We then convert $2P = (X_{2P}, Y_{2P}, Z_{2P})$ and $P + Q = (X_3, Y_3, Z_3)$ to a projective coordinate. The process is computed as [11],

$X_{2P} = X^4 + b \times Z^4$

$Z_{2P} = X^2 \times Z^2$

$Z_3 = (X_1 \times Z_2 + X_2 \times Z_1)^2$

$X_3 = x \times Z_3 + (X_1 \times Z_2) \times (X_2 \times Z_1)$

Montgomery point multiplication

Algorithm 1 Algorithm for Montgomery point multiplication

Input : $k = (k_{n-1}, k_{n-2}....., k_1, k_0)_2$ with $k_{n-1} = 1, P(x, y) \in E(F_{2^m})$
Output : $Q = kP$
Set $X_1 = x, Z_1 = 1, X_2 = x^4 + b, Z_2 = x^2$
for i from $n - 2$ down to 0 **do**
  if $(k_i = 1$ **then**
    $\text{Madd}(X_1, Z_1, X_2, Z_2), \text{Mdouble}(X_2, Z_2)$
  else
    $\text{Madd}(X_2, Z_2, X_1, Z_1), \text{Mdouble}(X_1, Z_1)$
  end if
end for
Return$(Q = \text{Mxy}(X_1, Z_1, X_2, Z_2))$

Point addition: $P(x, y) \in E(F_{2^m})$ is a point defined on the curve E. Computing point addition using the following algorithm $Madd(X_1, Z_1, X_2, Z_2)$.

Point doubling - $Mdouble(X_2, Z_2)$.

Algorithm 2 Algorithm for Montgomery point addition

$T_1 = x$
$X_1 = X_1 \times Z_2$
$Z_1 = Z_1 \times X_2$
$T_2 = X_1 \times Z_1$
$Z_1 = Z_1 + X_1$
$Z_1 = Z_1^2$
$X_1 = Z_1 \times T_1$
$X_1 = X_1 + T_2$

Algorithm 3 Algorithm for Montgomery point doubling

$T_1 = c$
$X = X^2$
$Z = Z_2$
$T_1 = Z \times T_1$
$Z = Z \times X$
$T_1 = T_1^2$
$X = X^2$
$X = X + T_1$

6.6 Parallel Montgomery multiplication in CUDA

6.6.1 Design

Parallel scalar multiplication in CUDA is based on Montgomery's algorithm over a binary field from the previous chapter. In [23], it is stated that because of hardware resource limitations, a trade off must be made for a fully parallel implementation of the second and third layers. According to this study, the parallel on the second layer demonstrates a better timing performance compared to the parallel on the third layer, which is a more attractive approach.

Therefore, my investigation and design focuses on the point addition and point doubling in the second layer. The design for each are as follows:

Parallel point addition: As algorithm 2 demonstrates, the first three operations can be paralleled at the same time since they are calculated independently. T_1 gains

from x, X_1 gains fromX_1 and Z_2 , Z_1 gains from Z_1 andX_2 . In total, the point addition computation is composed of one squaring, four multiplications, and two additions. Without time-consuming inversion, this is more efficient.

Parallel point doubling: As algorithm 3 above demonstrates, the first three steps can be paralleled at the same time since they are calculated independently. T_1 gets from c. X gets from X_2. Z gets from Z_2. Point doubling consists of one addition, two multiplications, and four squarings, whose computational complexity is simpler than point addition.

6.6.2 Implementation in CUDA

This approach achieves task parallelism using CUDA on NVIDIA GPUs. To realize task parallelism in CUDA, we schedule independent tasks on devices that are designed for data parallel or SPMD applications.

First, according to Montgomery multiplication in figure 4, in order to parallel operations in point addition and in point doubling, we need to allocate memory for the parameters that are being calculated for a GPU device. In figure 4, we need to allocate x, y, X_1, Z_1, X_2, Z_2 using cudaMalloc.

Secondly, using cudaMemcpy we copy the data from a host memory to device memory.

In our third step, figure 8, we enable two different kernels to be executed by the GPU, this approach is used block partition to achieve parallelism. The two kernels in the project refer to a Montgomery point addition and point doubling. All CUDA programs define an execution configuration using two instructions that process a kernel launch by setting a block dimension and a grid dimension. The block dimension

specifies how many threads in each block, and the grid dimension refers to the total number of threads to be launched. Once we have two kernels in different blocks,

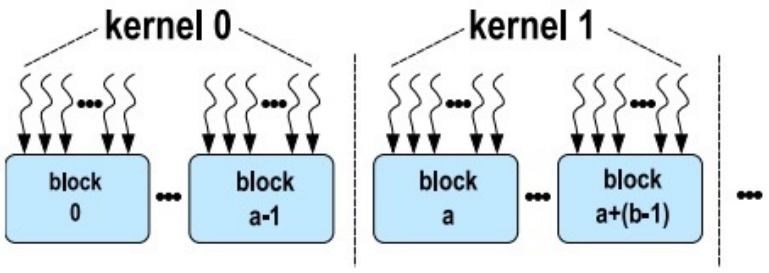

Figure 8: Block Partitioning. Each kernel contains n blocks.

these two kernels will be issued as one along with the current CUDA scheduler to conduct thread interleaving and block interleaving. Under each kernel(point addition or point doubling) we can execute the multiplications in parallel as analyzed above in the pre-fixed block by the following algorithm shown in Algorithm 4.

Algorithm 4 Algorithm for thread interleaving

```
merged_kerne<<< dimGrid, dimBlock >>>(ptr1_1,ptr2_1,...,int dimGridKernel1)
int index = blockID × dimBlock + threadID
if blockID ≤ dimGridKernel1 then then
  kerne_1()
else
  if index ¡ dimGrid then then
    index = index - dimGridKernel1
    kernel_2()
  end if
end if
```

6.7 ECC vs RSA

In this section we will briefly compare the performance of ECC and RSA under an equivalent key size.

Here the comparison from [24] shows, with different key length, the timing of key generation are different between ECC and RSA. As we show in Table 1, ECC provides fewer key size than RSA but can achieve higher security level. The results

ECC Key length	RSA Key length	ECC times	RSA times
163	1024	0.08	0.16
233	2240	0.18	7.47
283	3072	0.27	9.80
409	7680	0.64	133.90
571	153605	1.445	679.06

Table 1: Timing of key generation between ECC and RSA.

show that ECC outperforms RSA with the increasing key size.

CHAPTER 7

Results and analysis

In this chapter we apply three approaches for implementing a scalar multiplication for ECC, described in Chapter 6, by calculating each executing time for performance estimation, and comparing and analyzing the advantages and disadvantages. Three algorithms are presented as follows and in this sequence: general scalar multiplication, Montgomery scalar multiplication, and parallel Montgomery scalar Multiplication in CUDA.

7.1 Hardware and software usage

In this section, we describe the hardware and software used for our experiments. In particular, graphics cards were required to be CUDA enabled NVIDIA's graphics card. So that we can perform the necessary functions.

Hardware:

Processor 2.7 GHz Intel Core i7

Memory 8 GB 1600 MHz DDR3

Graphics NVIDIA GeForce GT 650M

Software:

CUDA kit 5

Nsight IDE

Language:

C, CUDA

7.2 General scalar multiplication

The approach, in terms of how to implemenat general scalar multiplication, is described in Chapter 6. We apply this method for a general scalar multiplication that does not have a particular optimization or improvement.

7.2.1 Experiment 1

In our first experiment, we test the execution time(ms) of scalar multiplication with a different key size and constant scalar k = 9 to evaluate performance.

Key size	1	2	3	4	5	6	7	8
75	0.005	0.005	0.005	0.004	0.005	0.005	0.004	0.005
93	0.007	0.008	0.008	0.008	0.008	0.007	0.008	0.007
131	0.015	0.015	0.016	0.015	0.016	0.015	0.015	0.015
163	0.03	0.031	0.03	0.034	0.032	0.03	0.031	0.031
194	0.041	0.04	0.04	0.041	0.04	0.042	0.04	0.04
294	0.105	0.103	0.103	0.105	0.103	0.103	0.105	0.103

Table 2: Test cases for key size 75, 93, 131, 163, 194 and 294 for general scalar multiplication

Table 2 shows eight test cases for each key size, we then calculate average time, minimum time, maximum time, and standard deviation time depending on the test case table. The following experiments, experiments 2 to 8, will follow the same format as experiment 1.

Key size	Min	Max	Average	Standard Deviation
75	0.004	0.005	0.00475	0.00046291
93	0.007	0.008	0.007625	0.000517549
131	0.015	0.016	0.01525	0.00046291
163	0.03	0.034	0.031125	0.001356203
194	0.04	0.042	0.0405	0.000755929
294	0.103	0.105	0.10375	0.001035098

Table 3: Timings in key size 75, 93, 131, 163, 194 and 294 for general scalar multiplication

7.2.1.1 Analysis for experiment 1

As the results demonstrate, in Tables 2 and 3, in general scalar multiplication the timing grows with increasing key size.

7.2.2 Experiment 2

In our second experiment, we test the execution time of scalar multiplication with a different scalar k and constant key size = 163.

Table 4 demonstrates eight test cases for different scalar Ks, then we calculate an average time, minimum time, maximum time, and standard deviation time depending on the test case table.

7.2.2.1 Analysis for experiment 2

As our results demonstrate, in Tables 4 and 5, in general scalar multiplication the timing grows with each increasing scalar K.

Scalar K	1	2	3	4	5	6	7	8
9	0.03	0.031	0.03	0.034	0.032	0.03	0.031	0.031
30	0.205	0.202	0.206	0.204	0.206	0.206	0.204	0.212
90	0.884	0.874	0.879	0.87	0.874	0.878	0.882	0.875
160	1.806	1.81	1.802	1.822	1.813	1.801	1.814	1.808
240	3.053	3.041	3.058	3.049	3.051	3.069	3.042	3.07
360	4.965	4.937	4.968	4.977	4.978	4.961	4.968	4.982

Table 4: Test cases for different scalar K 9, 30, 90, 160, 240 and 360 for general scalar multiplication

Scalar K	Min	Max	Average	Standard Deviation
9	0.03	0.034	0.031125	0.001356203
30	0.202	0.212	0.205625	0.002924649
90	0.874	0.884	0.877	0.0046291
160	1.801	1.814	1.8095	0.00688684
240	3.041	3.07	3.054125	0.010986193
360	4.937	4.982	4.967	0.014081396

Table 5: Timings of different scalar K 9, 30, 90, 160, 240 and 360 for general scalar multiplication

7.3 Montgomery scalar multiplication

7.3.1 Experiment 3

In this experiment, we test the execution time of a Montgomery scalar multiplication using a projective coordinate with a different key size and a constant scalar k

= 9. The algorithm is decribed in Chapter 6.

Key size	1	2	3	4	5	6	7	8
75	0.000175	0.000125	0.000127	0.000127	0.000128	0.000155	0.000129	0.000166
93	0.000155	0.000155	0.000153	0.000157	0.00015	0.000155	0.000153	0.000154
131	0.000463	0.000442	0.000466	0.000467	0.000514	0.000456	0.000486	0.000479
163	0.000686	0.000891	0.000726	0.000804	0.000686	0.000687	0.000783	0.0007
194	0.000942	0.000935	0.000932	0.000945	0.000963	0.000937	0.000929	0.000951
294	0.002006	0.001989	0.002017	0.00203	0.001987	0.001965	0.00216	0.002013

Table 6: Test cases for key size 75, 93, 131, 163, 194 and 294 for Montgomery scalar multiplication

Key size	Min	Max	Average	Standard Deviation
75	0.000125	0.000175	0.0001415	2.05E-05
93	0.00015	0.000155	0.000154	2.07E-06
131	0.000442	0.000514	0.000471625	2.17E-05
163	0.000686	0.000891	0.000745375	7.46E-05
194	0.000929	0.000963	0.00094175	1.12E-05
294	0.001965	0.00216	0.002020875	5.98E-05

Table 7: Timings in key size 75, 93, 131, 163, 194 and 294 Montgomery scalar multiplication

7.3.1.1 Analysis for experiment 3

Our results demonstrate(Tables 6 and 7) in a Montgomery scalar multiplication the timing grows with each increasing key size.

7.3.2 Experiment 4

In this experiment, we test the execution time of a Montgomery scalar multiplication with a different scalar k and a constant key size = 163.

Scalar K	1	2	3	4	5	6	7	8
9	0.000686	0.000891	0.000726	0.000804	0.000686	0.000687	0.000783	0.0007
30	0.005166	0.005113	0.005179	0.0052	0.005098	0.005378	0.005279	0.005235
90	0.015125	0.015299	0.015225	0.01702	0.015749	0.015294	0.015248	0.015304
160	0.026006	0.027987	0.027135	0.027018	0.027046	0.027062	0.026988	0.029314
240	0.039941	0.042505	0.040353	0.040271	0.040626	0.041077	0.042141	0.042034
360	0.061651	0.062912	0.061381	0.060826	0.062725	0.061193	0.064061	0.06085

Table 8: Test cases for different scalar K 9, 30, 90, 160, 240 and 360 Montgomery scalar multiplication

7.3.2.1 Analysis for experiment 4

Our results demonstrate (Tables 8 and 9) in a Montgomery scalar multiplication the timing grows with each increasing scalar K.

Scalar K	Min	Max	Average	Standard Deviation
9	0.000686	0.000891	0.000745375	7.46E-05
30	0.005098	0.005378	0.005206	9.14E-05
90	0.015125	0.01702	0.015533	0.000628456
160	0.026006	0.029314	0.0273195	0.000965462
240	0.039941	0.042505	0.0411185	0.00098133
360	0.060826	0.064061	0.061949875	0.001161402

Table 9: Timings of different scalar K 9, 30, 90, 160, 240 and 360 Montgomery scalar multiplication

7.4 Parallel Montgomery scalar Multiplication in CUDA

7.5 Experiment 5

In this experiment, we test an execution time for a parallel Montgomery scalar multiplication without computing an allocation time from the main memory to GPU memory while using a different key size and a constant scalar $k = 9$. The algorithm is described in Chapter 6.

7.5.0.2 Analysis for experiment 5

Our results demonstrate (Tables 10 and 11) in a CUDA Montgomery scalar multiplication, the timing grows with each increasing key size.

Key size	1	2	3	4	5	6	7	8
75	0.000062	0.000065	0.000074	0.000067	0.000066	0.000064	0.000063	0.000066
93	0.000065	0.000066	0.000065	0.000061	0.000066	0.000066	0.000067	0.000066
131	0.000066	0.000066	0.000063	0.000065	0.000064	0.000063	0.000064	0.000065
163	0.000066	0.000064	0.000071	0.000065	0.000066	0.000107	0.000063	0.000067
194	0.000067	0.000068	0.000065	0.000066	0.000066	0.000063	0.000066	0.000067
294	0.000065	0.000062	0.000063	0.000064	0.000075	0.000064	0.000065	0.000063

Table 10: Test cases for key size 75, 93, 131, 163, 194 and 294 for CUDA Montgomery multiplication

Key size	Min	Max	Average	Standard Deviation
75	0.000062	0.000074	0.000065875	3.68E-06
93	0.000061	0.000067	0.00006525	1.83E-06
131	0.000063	0.000066	0.0000645	1.20E-06
163	0.000063	0.000107	0.000071125	1.47E-05
194	0.000063	0.000068	0.000066	1.51E-06
294	0.000062	0.000075	0.000065125	4.12E-06

Table 11: Timings in key size 75, 93, 131, 163, 194 and 294 for CUDA Montgomery multiplication

7.5.1 Experiment 6

In experiment 6, we test the execution time of a parallel Montgomery scalar multiplication with computing allocation time along with a different key size and a constant scalar $k = 9$.

Scalar K	1	2	3	4	5	6	7	8
75	0.275412	0.273803	0.255312	0.282646	0.272105	0.275174	0.284079	0.268895
93	0.376531	0.314122	0.3549	0.344457	0.323721	0.354259	0.336767	0.383522
131	0.563428	0.514985	0.526557	0.554458	0.530579	0.537951	0.568609	0.543341
163	0.690852	0.759843	0.668226	0.669494	0.690764	0.715209	0.729618	0.703363
194	0.875926	0.854781	0.833686	0.936573	0.915386	0.855753	0.907927	0.94181
294	1.618554	1.633568	1.676067	1.642039	1.609354	1.663714	1.660483	1.645004

Table 12: Test cases for key size 75, 93, 131, 163, 194 and 294 for CUDA Montgomery multiplication with allocation overhead

Scalar K	Min	Max	Average	Standard Deviation
75	0.255312	0.284079	0.27342825	0.0089125
93	0.314122	0.383522	0.348534875	0.024012641
131	0.514985	0.568609	0.5424885	0.018660191
163	0.668226	0.759843	0.703421125	0.030976014
194	0.833686	0.94181	0.89023025	0.040710303
294	1.609354	1.676067	1.643597875	0.022831227

Table 13: Timings in key size 75, 93, 131, 163, 194 and 294 for CUDA Montgomery multiplication with allocation overhead

7.5.1.1 Analysis for experiment 6

Our results demonstrate (Tables 12 and 13) in a CUDA Montgomery scalar multiplication, along with memory allocation overhead, the timing grows with each increasing key size.

7.5.2 Experiment 7

In experiment 7, we test the execution time of a parallel Montgomery scalar multiplication without a computing allocation time while using a different scalar k and constant key size = 163.

Scalar K	1	2	3	4	5	6	7	8
9	0.000064	0.000064	0.000067	0.000066	0.000065	0.000065	0.000065	0.000066
30	0.000072	0.00008	0.000073	0.000073	0.00007	0.000075	0.000074	0.000071
90	0.00009	0.000096	0.000085	0.000088	0.000085	0.000086	0.000083	0.00009
160	0.000097	0.000092	0.000094	0.000092	0.000094	0.000097	0.000092	0.000092
240	0.000096	0.000096	0.000091	0.000096	0.000095	0.000093	0.000095	0.000096
360	0.000102	0.000098	0.000098	0.0001	0.000101	0.000098	0.000098	0.000104

Table 14: Test cases for different scalar K 9, 30, 90, 160, 240 and 360 for CUDA Montgomery multiplication

Scalar K	Min	Max	Average	Standard Deviation
9	0.000064	0.000067	0.00006525	1.04E-06
30	0.00007	0.00008	0.0000735	3.07E-06
90	0.000083	0.000096	0.000087875	4.12E-06
160	0.000092	0.000097	0.00009375	2.19E-06
240	0.000091	0.000096	0.00009475	1.83E-06
360	0.000098	0.000104	0.000099875	2.30E-06

Table 15: Timings of different scalar K 9, 30, 90, 160, 240 and 360 for CUDA Montgomery multiplication

7.5.2.1 Analysis for experiment 7

Our results demonstrate (Tables 14 and 15) in a CUDA Montgomery scalar multiplication the timing grows with each increasing scalar K.

7.5.3 Experiment 8

In experiment 8, we test the execution time of a parallel Montgomery scalar multiplication along with computing allocation time while using a different scalar k and a constant key size = 163.

Scalar K	1	2	3	4	5	6	7	8
9	0.700126	0.668529	0.665548	0.76269	0.723339	0.674247	0.710953	0.701169
30	0.837697	0.892199	0.822166	0.881156	0.777102	0.88086	0.855339	0.804022
90	1.101502	1.105301	1.02507	1.119897	1.112071	1.048786	1.032163	1.105014
160	1.139519	1.192315	1.226287	1.213454	1.22521	1.136232	1.130745	1.209691
240	1.209115	1.202208	1.189265	1.175106	1.233138	1.17945	1.211495	1.148747
360	1.260058	1.347244	1.359554	1.294778	1.264083	1.329231	1.294484	1.091362

Table 16: Test cases for different scalar K 9, 30, 90, 160, 240 and 360 for CUDA Montgomery multiplication with allocation overhead

7.5.3.1 Analysis for experiment 8

Our results demonstrate (Tables 16 and 17) in a CUDA Montgomery scalar multiplication along with memory allocation overhead the timing grows with each increasing scalar K.

Scalar K	Min	Max	Average	Standard Deviation
9	0.665548	0.76269	0.700825125	0.03258227
30	0.777102	0.892199	0.843817625	0.041023883
90	1.02507	1.119897	1.0812255	0.03894406
160	1.130745	1.226287	1.184181625	0.041712163
240	1.148747	1.233138	1.1935655	0.026107257
360	1.091362	1.359554	1.28009925	0.084462766

Table 17: Timings of different scalar K 9, 30, 90, 160, 240 and 360 for CUDA Montgomery multiplication with allocation overhead

7.6 Three methods comparison

In this section, we take the average timing, which are the results from the last section, and use them as the estimation timing among our next three approaches. For representation convenience we will use Regular scalar multiplication(RSM), Montgomery scalar Multiplication(MSM), Parallel Montgomery Multiplication in CUDA(CUDAMSM) and Parallel Montgomery Multiplication in CUDA with overhead(CUDAMSMO). CUDAMSN and CUDAMSMO are based on the same algorithm, and the only different is CUDAMSMO calculate the time of CUDA overhead.

7.6.1 Comparison in different key size

1. Timing of an ECC scalar multiplication comparison, with different key size among RSM, MSN, CUDAMSM and CUDAMSMO, is demonstrated in table 18. Runtime (Multiplication K = 9)

Key size	RSM	MSM	CUDAMSM	CUDAMSMO
75	0.00475	0.0001415	0.000065875	0.27342825
93	0.007625	0.000154	0.00006725	0.348534875
131	0.01525	0.000471625	0.0000705	0.5424885
163	0.031125	0.000745375	0.000071125	0.703421125
194	0.0405	0.00094175	0.000074	0.89023025
294	0.10375	0.002020875	0.00008125	1.643597875

Table 18: Timing of scalar multiplication comparison with different key size among RSM, MSN, CUDAMSM and CUDAMSMO

7.6.1.1 Analysis

Since the CUDAMSMO method produces a larger execution time, we extract RSM, MSM, and CUDAMSN then compare them in one graph in order to illustrate our results clearly and precisely. In figure 9, it is demonstrated that the Montgomery algorithm can accerelate scalar multiplication rapidly with increasing key size more than a general scalar multiplication, which is used in an elliptic curve cryptosystem. CUDAMSM also demonstrates good results when the overhead of copying data from local memory to device(GPU) memory is ignored. The reason is that using Montgomery multiplication reduces divison and inversion operations. The timing calculated for a Montgomery scalar multiplication, using projective coordinates, is required up to $(m-1)(6M+3A+5S)+(10M+7A+4S+I)$clock cycles, where M, A, S and I are represented by the number of clock cycles for multiplication, addition, squaring, and inversion, respectively [25]. Therefore, the scalar multiplication needs a

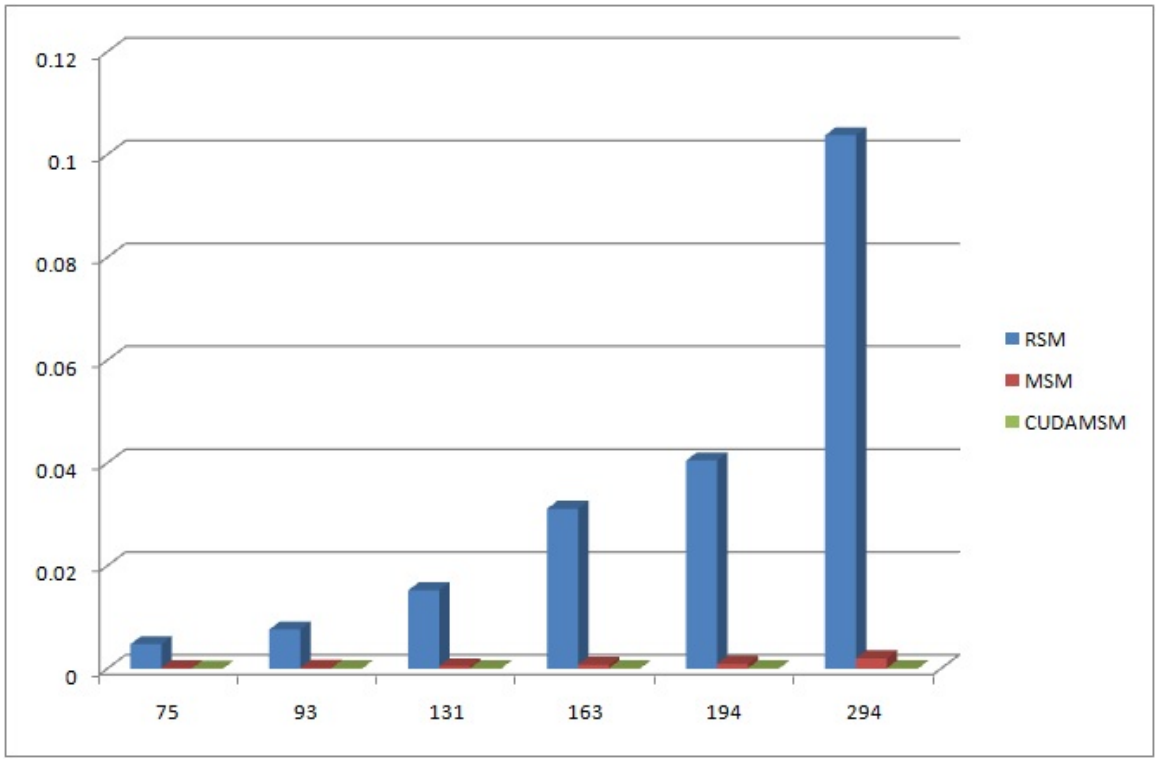

Figure 9: Run time with different key size for Regular scalar multiplication VS Montgomery scalar Multiplication VS Parallel Montgomery Multiplication in CUDA

$991M + 976S + 493A$ clock time in total which is overwhelming time consuming [19].

As our results demonstrate in figure 10, the Montgomery algorithm can largely accelerate the speed of scalar multiplication for an ECC. The parallel Montgomery method in CUDA can attain good results in the end; however, increased overhead of transfer data from Host memory to device(GPU) memory can be expected and its results are not optimal. For my project, CUDA overhead is created from: 1.allocated new memory in GPU for the data I needed to calculate; 2. copied data from the host memory to GPU memory; 3. copied results that were computed in the GPU back to the host memory. The reason why CUDA overhead is so high is that the size of an element depends on the key size; when the key size grows, the time for CUDA to allocate memory(CUDA overhead) increases as well. For realistic applicaitons or

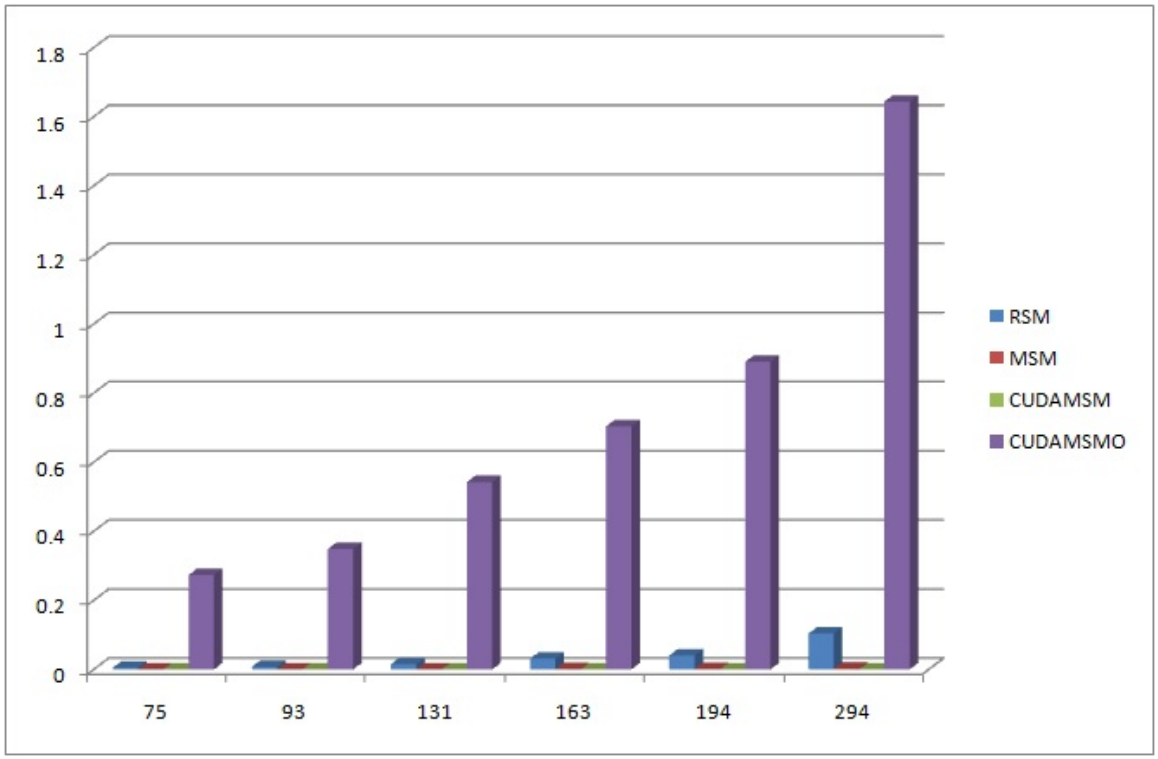

Figure 10: Run time with different key size for RSM, MSN, CUDAMSM and CUDAMSMO

when using an efficient allocating method, we can overcome this disadvantage in the future.

7.6.2 Comparison in different scalar K

2. Timing of an ECC scalar multiplication in comparison with different K (Q = KP) among RSM, MSN, CUDAMSM, and CUDAMSMO is demonstrated in table 18. Runtime (key size: 163)

7.6.2.1 Analysis

Our results demonstrate (figure 11) with an increase in K the runtime of regular scalar multiplication (without any optimization) goes up extremely quick. Under

Scalar K	RSM	MSM	CUDAMSM	CUDAMSMO
9	0.031125	0.000745375	0.00006525	0.700825125
30	0.205625	0.005206	0.0000735	0.843817625
90	0.877	0.015533	0.000087875	1.0812255
160	1.8095	0.0273195	0.00009375	1.184181625
240	3.054125	0.0411185	0.00009475	1.1935655
360	4.967	0.061949875	0.000099875	1.28009925

Table 19: Run time comparison of scalar multiplication with different K

these circumstances, the overhead produced from cuda allocation can be ignored when the value of K becomes a large number. The time lost from the Montgomery multiplication over a projective coordinate is due to the algorithm presented in Chapter 6. The times we perform a point addition and a point doubling declines significantly since the scalar k is represented in its binary form. For example, if $K = 360$, we need to compute $P = kQ$. In general when using a scalar multiplication we need to run a loop 360 times in order to gain our final results. However in a Montgomery point multiplication over a projective coordinate system $k = 360$ converts to an array: $k[9] = 1, 0, 1, 1, 0, 1, 0, 0, 0$; 101101000 is the binary form of 360. The times needed for loop shrink becomes 9 times; this is a large reduction for scalar multiplication. Other than the efficient computation gained from the Montgomery algorithm, we can parallel two or more arithmetic operations at the same time; this improved process

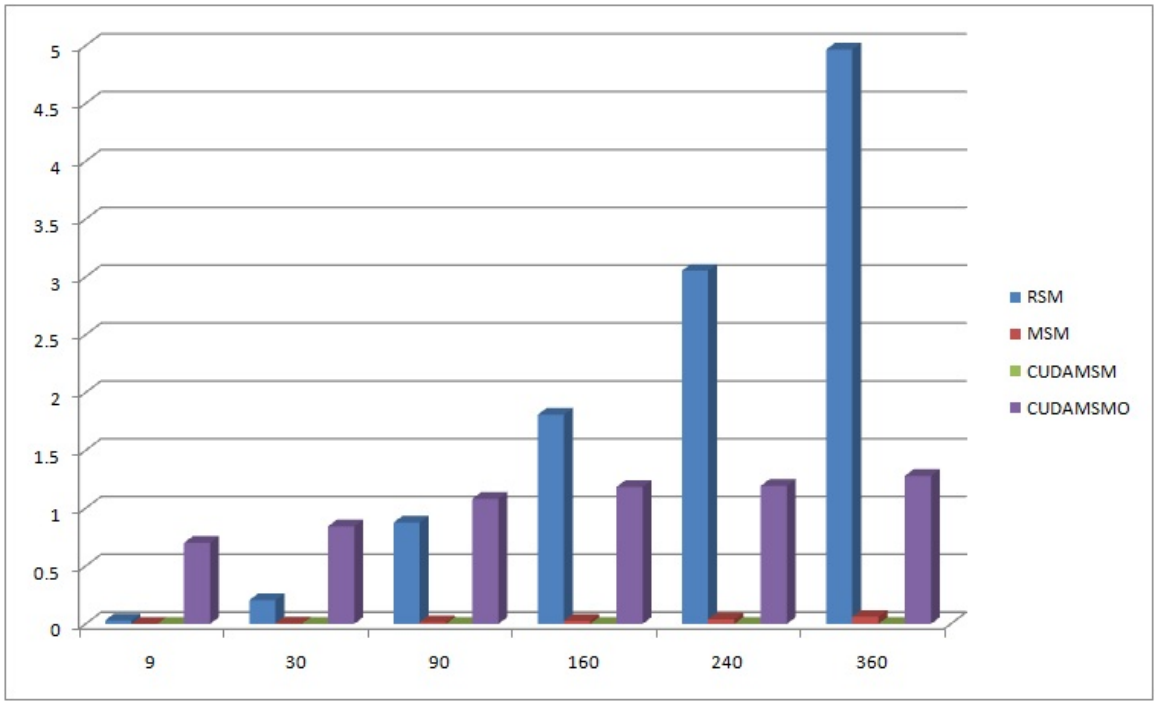

Figure 11: Run time with different K for Regular scalar multiplication VS Montgomery scalar Multiplication VS Parallel Montgomery Multiplication in CUDA

can increase speed as well.

CHAPTER 8

Conclusion and Future Work

In this project, a parallel design strategy in CUDA for elliptic curve point multiplication is presented in order to efficiently accelerate the speed and work along with a scalar parameter K increasing. Our algorithm demonstrated good results without CUDA allocation overhead. Therefore, in the future, we could study and apply this new approach to access global memory in CUDA Kernels more efficiently. As a whole, the architecture provided demonstrates a promising parallel solution in GPU for elliptic curve scalar multiplication.

Future work should further explore better improvements in the areas of hardware and software with in all three layers of scalar multiplication, implementing additional design strategies in both software and hardware comparisons. Since inversion operation is an important operation within the arithmetic level of elliptic cuvres, this is probably another interesting research topic for future study.

LIST OF REFERENCES

[1] Elliptic curve cryptography,
http://en.wikipedia.org/wiki/Elliptic_curve_cryptography

[2] Elliptic curve from NSA,
http://www.nsa.gov/business/programs/elliptic_curve.shtml

[3] K. Gupta, S. Silakari, ECC over RSA for Asymmetric Encryption: A Review, *IJCSI InternationalJournal of Computer Science Issues*, Vol. 8, Issue 3,No. 2, May 2011,
http://www.ijcsi.org/papers/IJCSI-8-3-2-370-375.pdf

[4] S. Baktri, E. Savas, Highly-Parallel Montgomery Multiplication for Multi-core General-Purpose Microprocessors,
Department of Computer Engineering, Bahcesehir University, 2011,
http://eprint.iacr.org/2012/140.pdf

[5] P. Emeliyanenko, Efficient Multiplication of Polynomials on Graphics Hardware, Max-Planck-Institut fur Informatik, Saarbrucken, Germany, 2009,
https://domino.mpi-inf.mpg.de/intranet/ag1/ag1publ.nsf/0/ca00677497561c7ec125763c0044a41a/$FILE/gpgpu_mul.pdf

[6] T. F. Al-Somani, G. Adnan, M. K. Ibrahim, Highly Efficient Elliptic Curve Crypto-Processor with Parallel GF(2m) Field Multipliers,
Journal of Computer Science, Jan 2006

[7] S. Fleissner, GPU-Accelerated Montgomery Exponentiation,
Department of Computer Science and Engineering, The Chinese University of Hong Kong, 2007

[8] M. Estes, P. Hines, Efficient Implementation of an Elliptic Curve Cryptosystem Over Binary Galois Fields in Normal and Polynomial Bases,
George Mason University, 2006, http://teal.gmu.edu/courses/ECE746/project/reports_2006/ECC_IN_SW_report.pdf

[9] M. Y. Malik, Efficient implementation of elliptic curve cryptography using low-power digital signal processor,
National UniversityofScience and Technology(NUST), Pakistan, 2010,
http://arxiv.org/ftp/arxiv/papers/1109/1109.1877.pdf

[10] M. Stamp, *Information security principles and practice*,Wiley, 2 edition, May 3, 2011

[11] M. Rosing, *Implementing Elliptic Curve Cryptography*, Manning Publications, January 1, 1998

[12] R. Lercier, F. Morain, Counting the Number of Points on Elliptic Curves Over Finite Fields: Strategies and Performances,
Advances in Cryptology - EUROCRYPT 95, Lecture Notes in Computer Science Volume 921, 1995, pp 79-94,
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.2386`

[13] W. Diffie, M. E. Hellman, New Directions in Cryptography, *IEEE Transactions on Information Theory*,
`http://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf`

[14] I. Damgaard, M. Jurik, A generalisation, A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System,
In Proceedings of the4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography, PKC 01,
pages 119-136, London, UK, 2001, Springer-Verlag

[15] A. E. Cohen, K. K. Parhi, GPU Accelerated Elliptic Curve Cryptography in GF(2^m),
it 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2010,
`http://www.ece.umn.edu/users/aecohen/papers/aecohen_parhi_mwscas2010.pdf`

[16] J. Bajard, S. Duquesne, N. Meloni, Combining Montgomery ladder for elliptic curves defined over F_p and RNS representation,
Computer Science Department, UCLA, Los Angeles, California, 2010,
`http://eprint.iacr.org/2010/311.pdf`

[17] A. Hariri, Bit-Serial and Bit-Parallel Montgomery Multiplication and Squaring over GF(2^m),
IEEE Transactions on Computers, Volume: 58, Issue: 10, 2009

[18] Ramsey, Glenn Jr., Hardware/Software Optimizations for Elliptic Curve Scalar Multiplication on Hybrid FPGAs,
Department of Computer Engineering, Rochester Institute of Technology, 2008,
`https://ritdml.rit.edu/bitstream/handle/1850/7765/GRamseyThesis06-2008.pdf?sequence=1`

[19] M. Stamp, R. M. Low, *Applied Cryptanalysis: Breaking Ciphers in the Real World*,
Wiley-IEEE Press, 1 edition, April 25, 2007

[20] CUDA C Programming Guide 4.0, 2011,
http://www.shodor.org/media/content//petascale/materials/UPModules/matrixMultiplication/cudaCguide.pdf

[21] R. Szerwinski, Exploiting the Power of GPUs for Asymmetric Cryptography, *Cryptographic Hardware and Embedded Systems*, CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008

[22] Irreducible polynomial,
http://en.wikipedia.org/wiki/Irreduciblepolynomial

[23] D. Hankerson, J. Lopez-Hernandez, and A. Menezes, Software implementation of elliptic curve cryptography over bi- nary fields,
Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000,
Proceedings, 1965:124, August 2000

[24] N. A. Saqib, F. Rodriguez-Henriquez, A. Diaz-Perez, A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over GF (2^m),
Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International

[25] N. Jansma, B. Arrendondo, Performance Comparison of Elliptic Curve and RSA Digital Signatures, 2004,
http://nicj.net/files/performance_comparison_of_elliptic_curve_and_rsa_digital_signatures.pdf

[26] B. Ansari and M. Hasan, High Performance Architecture of Elliptic Curve Scalar Multiplication,
IEEE Trans. Computers, vol. 57, no. 11, pp. 1443-1453, Nov. 2008

[27] M. Guevara, C. Gregg, K. Hazelwood, K. Skadron, Enabling Task Parallelism in the CUDA Scheduler,
Department of Computer Science, University of Virginia, 2009,
http://www.cs.virginia.edu/kim/docs/pmea09.pdf

APPENDIX

Algorithm

A.1 Inversion from Algorithm 3.1

Algorithm 2.3 Extended Euclidean Algorithm for Polynomial Representation

INPUT: $a(x) \in GF(2^m)$, irreducible polynomial $f(x)$ of degree m
OUTPUT: $u(x) = a(x)^{-1} \bmod f$

1: $u(x) \leftarrow 1$
2: $v(x) \leftarrow 0$
3: $s(x) \leftarrow f(x)$
4: $g(x) \leftarrow a(x)$
5: **while** $s(x) \neq 0$ **do**
6: compute $q(x)$ and $r(x)$ such that $g(x) = q(x)s(x) + r(x)$
7: $t(x) \leftarrow u(x) - v(x)q(x)$
8: $u(x) \leftarrow v(x)$
9: $g(x) \leftarrow s(x)$
10: $v(x) \leftarrow t(x)$
11: $s(x) \leftarrow r(x)$
12: **end while**

Figure A.12: Extended Euclidean Algorithm for Polynomial Representation

A.2 Inversion for Algorithm 3.2

Algorithm 2.4 Inversion in $GF(2^m)$ for Polynomial Representation

INPUT: $a(x) \in GF(2^m)$, irreducible polynomial $f(x)$ of degree m
OUTPUT: $u(x) = a(x)^{-1} \bmod f$

1: $u(x) \leftarrow 1$
2: $v(x) \leftarrow 0$
3: $s(x) \leftarrow m$
4: $\delta \leftarrow 0$
5: **for** $i = 1$ to $2m$ **do**
6: **if** $a(x)_m = 0$ **then**
7: $a(x) \leftarrow xa(x)$
8: $u(x) \leftarrow xu(x) \bmod f(x)$
9: $\delta \leftarrow \delta + 1$
10: **else**
11: **if** $s(x)_m = 1$ **then**
12: $s(x) \leftarrow s(x) - a(x)$
13: $v(x) \leftarrow (v(x) - u(x)) \bmod f(x)$
14: **end if**
15: $s(x) \leftarrow xs(x)$
16: **if** $\delta = 0$ **then**
17: $t(x) \leftarrow f(x)$
18: $f(x) \leftarrow s(x)$
19: $s(x) \leftarrow t(x)$
20: $t(x) \leftarrow u(x)$
21: $u(x) \leftarrow v(x)$
22: $v(x) \leftarrow t(x)$
23: $u(x) \leftarrow xu(x) \bmod f(x)$
24: $\delta \leftarrow 1$
25: **else**
26: $u(x) \leftarrow (u(x)/x) \bmod f(x)$
27: $\delta \leftarrow \delta - 1$
28: **end if**
29: **end if**
30: **end for**

Figure A.13: Inversion in GF(2m) for polynomial representation