ON THE EFFICIENCY OF LATTICE-BASED CRYPTOGRAPHIC SCHEMES ON GRAPHICAL PROCESSING UNIT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ZALİHA YÜCE TOK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

DECEMBER 2016

Approval of the thesis:

ON THE EFFICIENCY OF LATTICE-BASED CRYPTOGRAPHIC SCHEMES ON GRAPHICAL PROCESSING UNIT

submitted by **ZALİHA YÜCE TOK** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Department of Cryptography, Middle East Technical University** by,

Prof. Dr. Bülent Karasözen
Director, Graduate School of **Applied Mathematics** ______________

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** ______________

Prof. Dr. Ersan Akyıldız
Supervisor, **Mathematics, METU** ______________

Assoc. Prof. Dr. Sedat Akleylek
Co-supervisor, **Computer Engineering, Ondokuz Mayıs University** ______________

Examining Committee Members:

Prof. Dr. Ersan Akyıldız
Mathematics, METU ______________

Assoc. Prof. Ali Doğanaksoy
Mathematics, METU ______________

Assoc. Prof. Dr. Murat Cenk
IAM Cryptography Department, METU ______________

Assist. Prof. Dr. Oğuz Yayla
Mathematics, Hacettepe University ______________

Assist. Prof. Dr. Mert Özarar
Computer Engineering, Konya Food and Agriculture University ______________

Date: ______________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ZALİHA YÜCE TOK

Signature :

ABSTRACT

ON THE EFFICIENCY OF LATTICE-BASED CRYPTOGRAPHIC SCHEMES ON GRAPHICAL PROCESSING UNIT

Yüce Tok, Zaliha
Ph.D., Department of Cryptography
Supervisor : Prof. Dr. Ersan Akyıldız
Co-Supervisor : Assoc. Prof. Dr. Sedat Akleylek

December 2016, 75 pages

Lattice-based cryptography, a quantum-resistant public key alternative, has received a lot of attention due to the asymptotic efficiency. However, there is a bottleneck to get this advantage on practice: scheme-based arithmetic operations and platform-based implementations. In this thesis, we discuss computational aspects of lattice-based cryptographic schemes focused on NTRU and GLP in view of the time complexity on both CPUs and Graphical Processing Units (GPU). We focus on the optimization of polynomial multiplication methods both on theoretical and implementation point of view. We propose a modified version of interleaved Montgomery modular multiplication algorithm for ideal lattices, sparse polynomial multiplication and its sliding window version for efficient implementations. We show that with the proposed algorithms we significantly improve the performance results of lattice-based signature schemes. We also implement parallelized version of well known polynomial multiplication algorithms such as schoolbook method, NTT by using CUDA and provide a library for selected lattice-based signature schemes on a GPU.

Keywords: lattice-based cryptography, polynomial multiplication, sparse polynomial multiplication with sliding window, interleaved Montgomery modular multiplication, NTRUEncrypt, GPU, FFT-based polynomial multiplication, cuFFT, number theoretic transform (NTT)

ÖZ

GRAFİK İŞLEMCİ BİRİMLERİ ÜZERİNDE KAFES TABANLI KRİPTOGRAFİK ŞEMALARIN UYGULAMALARININ İYİLEŞTİRİLMESİ

Yüce Tok, Zaliha
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Prof. Dr. Ersan Akyıldız
Ortak Tez Yöneticisi : Doç. Dr. Sedat Akleylek

Aralık 2016, 75 sayfa

Kuantum hesaplamaya dirençli açık anahtarlı sistemlere alternatif olan kafes tabanlı kriptografi asimptotik verimliliğinden dolayı büyük ilgi görmektedir. Bununla birlikte, pratikte bunu kullanabilmek için bazı engeller bulunmaktadır: şema tabanlı aritmetik işlemler ve platform tabanlı uygulamalar. Bu tez çalışmasında, kafes tabanlı kriptografik protokollerden NTRU ve GLP'nin CPU ve grafik işlem birimleri (GPU) üzerindeki zaman karmaşıklıklarını hesaplama açısından tartıştık. Burada, polinom çarpımının hem uygulama hem de teorik açıdan iyileştirilmesi üzerine durduk. İdeal kafesler için aralanmış Montgomery modüler çarpma algoritmasının güncellenmesi, seyrek polinomların çarpma algoritmasını ve bunun kayan pencere tekniğini verimli uygulamalar için önerdik. Önerilen algoritmalar ile kafes tabanlı kriptografik şemaların performans sonuçlarının iyileştirildiğini gösterdik. Ayrıca, ilkokul yöntemi, NTT gibi bilinen çarpma algoritmalarını paralel bir şekilde uyguladık ve seçilen kafes tabanlı kriptografik şemalar için GPU'da çalışan bir CUDA yazılım kütüphanesi oluşturduk.

Anahtar Kelimeler: kafes tabanlı kriptografi, polinom çarpımı, kayan pencere yöntemi, aralanmış Montgomery yöntemi, NTRUEncrypt, GPU, FFT tabanlı çarpımı, cuFFT, NTT

To my children,

Ali Efe and Derin Mavi

ACKNOWLEDGMENTS

I would like to thank my supervisor Professor Ersan Akyıldız for his support, and guidance. It was a great honor to work with him for the last twelve years. I would like to thank to my co-advisor Associate Professor Sedat Akleylek for his support, guidance and incredible friendship.

I also would like to thank to my friends Hande, Selin and Handan for providing support and friendship that I needed. A special thanks to my mother and father for their motivation, often by asking: When are you going to finish that thesis? I would also thank to my beloved husband and soul mate Ali to his support for everything. I married the best person out there for me. Finally, I would thank to my children Ali Efe and Derin Mavi, without whom this thesis would have been completed many years ago.

TABLE OF CONTENTS

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Main goal of cryptography is to provide the secrecy and integrity of information while transferred from one side to another. The history was started with the famous Roman Empire Caesar with Caesar cipher and incredible developments took place. Modern cryptosystems on symmetric and asymmetric (public-key) cryptography have been developed and widely used in every field of the modern world.

Symmetric systems today enable us to send the plaintext as an unreadable format called ciphertext and provide to receiver to turn the ciphertext to original plaintext.

Plaintext to ciphertext conversion is called encryption and ciphertext into plaintext conversion is called decryption. Encryption and decryption processes are done with the same key called secret key in symmetric systems. These systems enable us to send information in a secure and fast way with one drawback: difficulty in the sharing of secret key. Sharing of the secret key means both sides have to agree on the same key before communication. Modern cryptography solves this problem by using asymmetric cryptography.

Asymmetric cryptography is a bit more complex than the symmetric cryptography. In those systems two keys are generated; one is called as public key which is known by all and the other one is the private key and kept secret by the owner. The sender encrypts the message with the receiver's public key and only the receiver can decrypt the message with his private key. Also asymmetric key is used for authentication purposes, one can sign the message with his private key and anyone can verify that the message is signed by him. The security of asymmetric cryptographic schemes is based on the computationally hard problems,for example integer factorization and discrete log problems [43]. Systems like RSA and ECDSA are today's most commonly used schemes.

Although today's symmetric and asymmetric systems solve the issues, the emergence of quantum computer studies move today's secure cryptosystems into a doubt situation [11]. Shor's thesis showed that problems based on hardness of factorization problem can be efficiently solved with quantum computers while today's computers are not capable of doing it [54]. This novel property of quantum computation has much attention from the industry to academy and researches are done in similar and different fields. These researches showed that while quantum computing has big advantage on search algorithms and integer factorizations, it does not differ the ordinary computing complexities in an exponential way on other fields.

The difference with conventional and post quantum computers can be described to get a better understanding for why quantum computers are so powerful. In classical computers, computing is done in building blocks called bits which hold a binary value namely 0 and 1 only one value for the time. In quantum computers building blocks made up of qubits which holds 0 and 1 at the same time which is called the superposition of states. This “quantum superposition” provides manipulating all combinations of bits simultaneously which makes the quantum computation parallel and rapid than the classical computing. Protocols or algorithms that are resistant to quantum attacks has been called quantum resistant. The vulnerable algorithms against quantum attacks are called the quantum –insecure in this manner.

In cryptography side, algorithms that their security is guaranteed by integer factorization and discrete log problems are vulnerable to quantum attacks. These algorithms are safe according to today’s technology but it has shown that they can be broken by quantum methods [55]. Also quantum search algorithms like Grover’s makes the related attacks more powerful so new security measures should be held according to this new situation [23]. Cryptographic algorithms that are widely used today like RSA, DSA and ECDSA will be insecure after the quantum computing since they depend on the computationally hardness of integer factorization and discrete log problems. Increasing the key sizes of these algorithms is not sufficient to resist the attacks. The increase on the key size will be adequate until the more powerful quantum computer is developed. On the other side, some of symmetric-key cryptographic algorithms are proved to be quantum-safe. These algorithms security does not depend on the computational assumptions but designed as theoretically secure. When come to the widely used ones like AES, they are also considered as quantum-safe because they will still resistant to quantum attacks only by doubling the key size. Since quantum search does not provide exponential speedups, symmetric key encryption like AES is believed to be quantum-safe [13]. Similarly, good hash functions are also believed to be resistant to quantum adversaries [13]. Table 1.1 shows the status of the widely used cryptographic schemes after quantum computers be in use [14]. As a result, the studies are done in public key area and researchers working for the algorithms both secure after quantum computers in use and efficient for today’s ordinary ones.

Table 1.1: Common Cryptographic Algorithms After Post Quantum Computers

Algorithm	**Status After Post Quantum Computers**
AES 256	increase in key size
SHA 3	increase in output size
RSA	no longer secure
ECDSA, ECDH	no longer secure
DSA	no longer secure

Moving from today’s widely used algorithms to new quantum algorithms will

require great effort. The algorithms standardized today are analyzed by many researchers for many years and this has been long history. As a result, there has not been any complete standard or adequate cryptanalysis on post quantum algorithms.
Post-quantum cryptographic schemes can be classified in five groups:

1.1 Hash-based cryptography

Hash-based cryptography based on the one-time signature (OTS) schemes. OTS is a signature scheme where each key pair is used to sign only one message. The security depends on cryptographic hash function's collision resistancy [11].
First study on hash based systems was done by Lamport and Diffie and system was based on OTS. Merkle adds binary tree structure to signatures and with this way usable signature size is increased to the size of the binary tree [44]. The leaves of the Merkle's binary tree are called the one time signature (OTS) public keys. Root of the Merkle's tree is the main public key. Each inner node of the binary tree is calculated as hash of concatenation of its child's. With this way leaf nodes are used to authenticate the public keys. In every sign operation one of the OTS public keys are used and never the same key is used again. So the state of the tree should be kept that which keys have already been used and which ones are not. This drawback is still the main research areas of hash based signature schemes. Merkle's signatures are so inefficient due to large key sizes, signatures and long signature generation process.
XMSS is one of the most efficient hashed based schemes build on Merkle's trees which is also in the standardization period [12]. It has performance improvements on tree traversal, reduced private keys and shortened signature generation time. Although XMSS has a great performance optimization according to basic Merkle's tree, it is still ten times slower than the RSA.

1.2 Code-based cryptosystems

Error correcting codes are widely used in communication technology to add the additional redundancy to transmission for many years. In 1978, McEliece used error correcting codes for a different purpose and introduces a scheme that uses binary Goppa codes for a public key encryption scheme [42]. Goppa codes are one of the widely used and efficient error correcting codes. The scheme's security depends on the syndrome decoding problem which is known as NP-complete problem. With this property it is secure against post-quantum attacks.

In McEliece cryptosystem, private key is composed of a random binary irreducible Goppa code and the public key is a matrix randomly generated from the private key. Ciphertext is a codeword with some errors which can only detected by the private key. It has great performance statistics on encryption and decryption processes. Besides these advantages, the practical implementation of this system does not exist. The main reason for this is the extremely large key sizes.

Courtois, Finiasz, and Sendrier proposed the CFS cryptosystem in 2001 [16]. Similar to McEliece, the security depends on the syndrome decoding problem. In CFS, while encryption and decryption performance results are good, signature generation process results are far from a practical usage. Like McEliece system, it has got very large key sizes. According to the fastest implementations of CFS, it is 100 times slower than the RSA. Besides McEliece and CFS, several schemes have been proposed that uses different codes like QC codes or quary codes on their schemes to decrease the key size. One of the most significant achievements on those schemes is the Cayrel et al.'s study on smart cards. The proposed method is based on the Stern's protocol and has a better performance results than RSA for 80 bit security level. Also the studies done by Misoczki and Barreto have reduced the public key size significantly [45].

1.3 Multivariate Cryptography

Multivariate cryptography is based on the problem of solving multivariate quadratic polynomial equations and the isomorphism. The first multivariate scheme was proposed in early 1980s [41]. From that time several such systems have been proposed with special equations to meet the efficiency requirements. But this specific choices lead to trapdoors and many of them has been broken during last two decades. Currently one of the most popular schemes is Simple Matrix encryption scheme [57]. Main advantage of this scheme is its efficiency due to work on only a single field. Also its decryption process is very efficient. Beside the encryption purpose, there are also schemes for digital signatures. UOV and Rainbow are most promising ones [34], [35]. Rainbow is more efficient with smaller key size. There also exist BigField schemes such as HFE (Hidden Field Equations) and pFLASH [50], [15]. A variant of HFE, namely HFEv has similar secure signature size with RSA and ECC [17].

1.4 Lattice-based cryptography

Lattice-based cryptographic schemes are one of the most widely studied post-quantum cryptographic protocols. Lattice-based systems depend on the difficulty of the Shortest Vector Problem (SVP) which is proved as NP-hard [55]. There is not any quantum algorithm to solve SVP, so lattice-based systems are assumed to quantum resistant. One of the attractive properties of lattice-based systems is the worst-case to average-case reduction which means all the keys in lattice systems are strong and hard to break. The most well known lattice systems are NTRU-based schemes and BLISS [33].

For several years, lattice-based cryptographic schemes have only been considered secure for large system parameters causing inefficient implementations. In 1998, NTRU cryptosystem was proposed as a public key cryptographic scheme over polynomial rings using the computational properties of hard problems over lattices as an alternative to factorization or discrete logarithm problem based schemes [30]. Standardization of the NTRU drafted in IEEE P1363 and it is still in progress and commercialized by Security Innovation [2], [32]. After this draft was published, the progress has shown that the design has robustness against different kind of attacks. Its encryption process is almost 10 times faster and decryption processes is almost 100 times faster than RSA for the 1024-bit security level. Furthermore, there is no evidence that it's vulnerable to practical or quantum attacks [55]. The main drawback of NTRU system when compared to RSA and ECDSA is its large key size.

The recent improvement on lattice-based cryptography is the BLISS signature system proposal. BLISS signature scheme has better performance results than RSA and also public and private key sizes are in acceptable limits. In addition, BLISS has implementations on embedded devices with good results. Addition to these schemes practical key exchange protocols are studied to work within TLS protocol. The common point of the proposed lattice-based schemes is the suitability to parallelize. The researches on parallel implementation of these schemes exist but limited. Graphical processing units (GPU) is one of the parallel computing environment that attracted researchers due to having high performance computing abilities. Graphical processing units (GPU) main application area is to execute commands in parallel with the computer graphics; hence they have been produced for gaming community. A general purpose GPU, having many cores, has a place on high performance computing applications. There are several studies on parallel implementations of cryptographic protocols since they are useful for operations requiring lots of processing units [18].

Arithmetic operations on the GPU have been widely studied for public key cryptographic schemes such as RSA and elliptic curve based protocols. In [53] efficient implementations of computationally expensive operations in RSA-1024 and 2048 and curve-based cryptographic schemes on NVIDIA 880GTS graphic card were presented. Standard radix form and residue number system approaches were used. This was the first study using the CUDA framework for general purpose GPU in public key cryptography. One year later, [22] explained arithmetic operations over finite fields of large prime characteristic and elliptic curve scalar multiplication operations on the GPU. They also presented both serial and parallel version of these operations for large integers. The first implementation of NTRU on a GPU was given in [29]. They implemented schoolbook multiplication in a circular cyclic manner with $O(n^2)$ complexity on NVIDIA GTX280. The required data was generated on the CPU and then sent to the GPU for the other computations. They also showed that although the schoolbook multiplication method was used, for the same security level NTRU had a better performance than RSA and elliptic curve based cryptographic schemes. Also, there has been an interest for the implementation of lattice-based cryptography in FPGA. The

main reason is to run the arithmetic operations in a parallel way. Multiplication algorithms which are the core part of the lattice-based cryptographic schemes for FPGAs were discussed in [52]. They gave the implementation details of how to parallelize NTT algorithm for the recomputed values. Due to the memory requirements of finding the values, the precomputation step was not performed on the FPGA but these values were given to the system directly.
In this thesis, our aim is to discuss computational aspects of lattice-based cryptographic schemes in view of the time complexity on both CPU and GPU side. Polynomial multiplication is the most time-consuming part of cryptographic schemes whose security is based on ideal lattices. Thus, any efficiency improvement on this building blocks has great impact on the practicability of lattice-based cryptography. In this thesis, we investigate several algorithms for polynomial multiplication and implement them in serial and parallel platforms. Compact and efficient implementation architectures of polynomial multiplication for lattice-based cryptographic schemes are presented for the quotient rings $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n-1)$ and $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$, where p is a prime number. We choose the widely studied NTRU and GLP schemes which are also defined on the fields $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n-1)$ and $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$, respectively.

In this thesis, we discuss the computational aspects of lattice-based cryptographic schemes focused on NTRUEncrypt and GLP signature schemes [3], [5], [6], [7], [8]. We give the details of the selected modular multiplication algorithms. We explain the improvements of the multiplication algorithms for CPU and GPU-based implementations. We also modify the polynomial multiplication methods considering the needs of the selected cryptosystems.

For the ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n-1)$ used in NTRU, we implement the fast convolution and fast convolution with sliding window methods. The proposed methods work with the data input {0,1}. Since NTRU uses the inputs from the data set {-1,0,1}, we generalize these methods to handle NTRU data sets. By using fast convolution with sliding window method the number of required additions is drastically reduced when compared to convolution method. We also implement modified version of interleaved Montgomery modular multiplication method for this quotient ring proposed by [49]. With the proposed algorithms, we improve the multiplication complexity and embed the conversion operation into the algorithm with almost free cost. Schoolbook and parallelized schoolbook (implemented on GPU) methods are also implemented for performance comparison purposes.

For the ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ used in signature scheme, we propose a sparse polynomial multiplication algorithm. This technique was initially proposed by [47] for specific set of inputs, but we generalize it to work with inputs used in GLP signature scheme. We enhance the performance by performing polynomial multiplication with only addition and subtraction operations. Moreover, the proposed methods are easily modified to be used in other cryptographic applications having a sparse polynomial multiplication operation with a slight differences in reduction part since they are independent of the choice of reduction polynomial. We also implement the Number Theoretic Transform (NTT) multiplication method which is widely used and efficient in lattice systems. We parallelize this method on GPU to analyze the performance gains. Another study that we did in parallel computing side is using CUDA Fast Fourier Transform (cuFFT) library which is parallelized FFT library developed for NVIDIA GPUs. Finally we proposed modified version of interleaved Montgomery modular multiplication method similar to the one we defined for ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$.

CHAPTER 2

Preliminaries

In this chapter, we recall the basic definitions on lattice-based cryptography used in thesis. We also give the lattice-based signature schemes; NTRU and GLP.

2.1 Lattice Definitions

Definition 1. (Lattice)

A lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$, i.e, it is a subset $\mathcal{L} \subseteq \mathbb{R}^n$ satisfying the following properties:

- $\mathcal{L}$ is closed under addition and subtraction (subgroup).
- There is an $\epsilon > 0$ such that any two distinct lattice points $x \neq y \in \mathcal{L}$ are at distance at least $\parallel x - y \parallel \geq \epsilon$.

Let $B = [b_1, \dots b_k] \in R^{n \times k}$ be linearly independent vectors in R^n. The lattice generated by B is defined as the all integer linear combinations of columns of B.

$$\mathcal{L}(B) = Bx : x \subseteq Z^k$$

The vectors $b_1, ..., b_k$ are called the basis of the lattice B. n is called the dimension and k is called rank of the lattice.

Definition 2. (Full rank lattice) A full rank lattice is defined as the set of vectors

$$\mathcal{L}(b_1, ..., b_n) = \left\{ \sum_{i=1}^{n} x_i b_i : x_i \subseteq \mathbb{Z} \text{ for } 1 \leq i \leq n \right.$$

which is generated by n linearly independent vectors $b_1, ..., b_n$ in $\mathbb{R}^n$. The vectors $b_1, ..., b_n$ are the basis of the lattice and Bx is the usual matrix-vector multiplication.

Lattices can have more than one bases, in fact they have an infinite number of different basis. By using a lattice basis another basis can be obtained by multiplying existing basis with uni-modular matrix (uni-modular matrix is a matrix with integer coefficients that has ±1 as determinant).

The determinant of a lattice is the absolute value of the determinant of the basis matrix $\det(\mathcal{L}(B)) =\mid \det(B) \mid$. The value of the determinant is independent of the choice of the basis, and geometrically corresponds to the inverse of the density of the lattice points in R^n.

Definition 3. (Determinant) Let $\mathcal{L} = \mathcal{L}(B)$ be a lattice of rank n, where $B \subseteq R^{m\times n}$ is any basis. We define the determinant of the lattice, denoted by $\det(\mathcal{L})$, as the n-dimensional volume of $P(B)$, i.e. $\det(\mathcal{L}) = \sqrt{\det(B^T B)}$.

2.2 Number Theory Research Unit (NTRU) and GLP Signature Scheme

In this thesis, we focus on the most promising ones of the lattice-based cryptosystems, namely NTRU and GLP Signature Schemes. These schemes proposed for the quotient rings $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ and $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ where q is a power of 2 and p is an odd prime. Let $\mathbb{R}_q = (\mathbb{Z}/q\mathbb{Z})[x]/(x^n - 1), \mathbb{R}_p = (\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1), \mathbb{Z}_q = (\mathbb{Z}/q\mathbb{Z})$ and $\mathbb{Z}_p = (\mathbb{Z}/p\mathbb{Z})$. Following sections briefly explain these schemes.

2.2.1 NTRU

NTRU was originally introduced as an encryption scheme [30]. In 2003, digital signature scheme using the NTRU lattices proposed [33]. NTRU encryption scheme was based on homomorphic properties under addition and multiplication. This property leads to emergence of fully homomorphic schemes based on NTRU [40].

NTRU has got much attention than the other schemes because of its practical key size. However, it does not have sufficient performance result according to traditional cryptographic schemes like RSA or ECDSA. It still needs improvement on computational intelligence issues. Studies on these systems primary based on the improvement of lattice polynomial multiplications, since the most time consuming part of NTRU cryptosystems are the polynomial multiplications over the polynomial ring. The other arithmetic operations can be eliminated by the usage of special rings as in the case of NTRU where the encryption scheme depends on the arithmetic in the quotient ring $(x^n - 1)$ which the modular reduction is almost free.
In this section, the details of NTRU cryptosystem is briefly explained. Parameter selection, key generation, encryption, decryption, signature generation and verification processes are discussed.

Domain Parameters Domain parameters are defined in [31] as follows:

- n The dimension of the polynomial ring used in NTRU
- p A positive integer specifying the ring $\mathbb{Z}/p\mathbb{Z}$ over which the coefficients of a certain product of polynomials will be reduced during the encryption and decryption process
- q A positive integer specifying the ring $\mathbb{Z}/q\mathbb{Z}$ over which the coefficients of a certain product of polynomials will be reduced during the encryption and decryption processes; also used in the construction of the public key
- k A security parameter which controls resistance to certain types of attacks, including plaintext awareness
- df The distribution of coefficients of the polynomial f (f is the part of the private key)
- dg The distribution of coefficients of the polynomial g (g is the part of the private key)
- dr The number of 1s and -1s used in a certain random polynomial r, below in the encryption process
- f A polynomial in $\mathbb{Z}[x]/(x^n - 1)$
- f_p A polynomial in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ (this is part of the private key). This polynomial is obtained by reducing the coefficients of f modulo p
- f_q A polynomial in $(\mathbb{Z}/q\mathbb{Z})[x]/(x^n - 1)$ This polynomial is obtained by reducing the coefficients of $f \mod q$.
- L_f The set of polynomials in $\mathbb{Z}[x]/(x^n - 1)$ whose coefficients satisfy df
- g A polynomial in $(\mathbb{Z}/q\mathbb{Z})[x]/(x^n - 1)$ (used with f_q to construct the public key)
- L_g The set of polynomials in $\mathbb{Z}[x]/(x^n - 1)$ whose coefficients satisfy dg
- L_r The set of polynomials in $\mathbb{Z}[x]/(x^n - 1)$ whose coefficients satisfy dr
- f_p^{-1} The inverse of f_p in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$
- f_q^{-1} The inverse of f_q in $(\mathbb{Z}/q\mathbb{Z})[x]/(x^n - 1)$
- h The public key, a polynomial in $(\mathbb{Z}/q\mathbb{Z})[x]/(x^n - 1)$
- r A polynomial in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ (used with h to encode message
- m The plaintext message, a polynomial in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$
- e The encrypted message a polynomial in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$
- G A generating function

- H A hashing function

NTRU works on the ring $\mathbb{R} = (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. An element $f \in \mathbb{R}$ is a polynomial written as

$$f = \sum_{i=1}^{n} f_i x^i = [f_0, f_1, ..., f_{n-1}]$$

IEEE recommends $n = 251$, $q = 128$ and $p = 3$ for domain parameters. In general p is chosen from the set $2, 3, 2 + x$ since it allows very efficient arithmetic. Note that reduction with 3 gives one of the following elements $\{-1, 0, 1\}$ which gives possibility to use special algorithms for performance improvement issues. The polynomial sets $f \in L_f, g \in L_g, r \in L_r$ and $m \in L_m$ which are used for key generation and there are several ways to create for them. Since our goal is to obtain an effective scheme, we should consider in which cases we have easy arithmetic operations. Thanks to [21] for the selected parameters d_f, d_g and d_r we have:

$$L_f = \{1 + p * f : f \in (d_f, 0)\}, L_g = (d, 0), L_r = (d_r, 0)$$

By using these sets, it's guaranteed that computing f_p^{-1} is particularly easy and $f_p^{-1} \equiv 1 \mod p$. Now, we are ready to summarize the key generation, encryption and decryption phases for NTRUEncrypt.

Key Generation Key generation process in NTRU starts with selecting two polynomials f and g where $f \in L_f$ and $g \in L_g$. The private key polynomial f must also have inverses modulo q and modulo p. When suitable parameters selected as discussed before, this condition will be satisfied for most choices of f. The inverse of f will be denoted as follows:
$ff_q^{-1} \equiv 1 \mod q$ and $ff_q^{-1} \equiv 1 \mod p$
The next step in key generation process is the computation of public key h as follows: $h \equiv (p * f_q^{-1} g) \mod q$ Algorithm 1 gives the key generation process of the NTRU system.

Encryption

Encryption process starts with choosing random polynomial r from L_r. Polynomial r is used as a blind factor for the encryption. Blinding factor r is multiplied with public key h and added to message m. All these operations are done in modulo q. Algorithm 2 gives the encryption process for NTRU.

```
Algorithm 1: NTRU Key Generation Algorithm
Input   : NTRU domain parameters (n,p,q)
Output: private key $f \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$, public key $h$
Choose random polynomial $f \in L_f$.
Choose random polynomial $g \in L_g$.
Compute $f_q^{-1}$ such that $f f_q^{-1} \equiv 1 \mod q$.
$h \equiv (p * f_q^{-1} * g) \mod q$
```

```
Algorithm 2: NTRU Encryption Algorithm
Input   : message m, public key h
Output: private key $f \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$, public key $h$
Choose random polynomial $r \in L_r$ as a masking (blinding) factor
$e \equiv (r * h + m) \mod q$.
```

Decryption
The encrypted message is decrypted by using private key f. In decryption process, precomputed inverse polynomial of f is used for performance gain. Algorithm 3 gives the decryption process for NTRU.

```
Algorithm 3: NTRU Decryption Algorithm
Input  : Private key $f$, $f_p^{-1}$, $f_q^{-1}$, ciphertext $e$
Output: Message $m$
$a \equiv (e * f) \mod q$. $a \equiv (r * p * f_q^{-1} * g * f + m * f) \mod q$
$a \equiv (p * r * g + m * f) \mod q$
$a * f_p^{-1} \equiv (p * r * g * f_p^{-1} + m * f * f_p^{-1}) \mod p$
$a * f_p^{-1} \equiv (m * f * f_p^{-1}) \mod p$
$m \equiv a \mod p$ since $f_p^{-1} \equiv 1 \mod p$
```

Remark 4. For most of the time with appropriate parameter choices, the original message is recovered with high probability. However some parameter choices can cause decryption failure, so it is suggested to include a few check bits in each message block [31].

2.2.2 Lattice Based Signature Scheme (GLP Signature Scheme)

In this section we recall the GLP signature scheme given in [25]. This scheme's signature size is shortened according to the memory requirements on embedded devices. The signature scheme is based on the proposed methods defined in [20] and [38]. Following paragraphs will give a brief definition of the scheme.
Let n be a power of 2 and $\mathbb{Z}_p$ be the ring of integers modulo p, where p is a prime number and $p \equiv 1 \mod 2n$. The signature scheme is defined over the polynomial ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1) = \mathbb{Z}_p[x]/(x^n+1)$. The polynomials are in the quotient ring of degree at most $(n-1)$ whose coefficients are in $\left[-\frac{p-1}{2}, \frac{p-1}{2}\right]$.
Key generation, signature generation and signature verification processes are given below. Hash, compression and transform algorithms are explained in [25].

Key Generation
Private key pair (s_1, s_2) is generated randomly from the polynomial ring $\mathbb{Z}_p[x]/(x^n+1)$. Also parameter a which is known publicly also randomly generated from the same ring. Public key is the combination of the private key pair and publicly known parameter a. Algorithm 4 defines the key generation process. For details please see [25].

Algorithm 4: Key Generation Algorithm

```
Input  : Prime $p$, polynomial degree $n$
Output: Public key $t$, private key $(s_1, s_2)$
1 Choose random polynomials $s_1$, $s_2$ from the polynomial ring $\mathbb{Z}_p[x]/(x^n + 1)$
   where all integer coefficients are from the set $\{-1, 0, 1\}$
2 $t = as_1 + s_2$
```

Signature Generation

Signature generation process starts with choosing two random polynomials from the polynomial ring $\mathbb{Z}_p[x]/(x^n + 1)$ with integer coefficients from the set $\{-k, \ldots, -1, 0, 1, \ldots, k\}$ where k is the predefined security parameters. These polynomials are used as blinding factor in generation process. Algorithm 5 Step 2 gives a 160-bit hashed value of the message by using blinding polynomials and publicly known polynomial a. In Step 3 and Step 4 signature pair (z_1, z_2) is generated from the polynomial ring $\mathbb{Z}_p[x]/(x^n + 1)$. If the signatures are not from the set $\{-(k-32), \ldots, -1, 0, 1, \ldots, k-32\}$, signature is rejected, key generation process starts from the beginning. In Step 5 signature is compresses and range check is done similar to the previous step. If the range check fails, again it starts from the beginning. For details please see [25].

Algorithm 5: Signature Generation Algorithm

```
Input  : Private key $(s_1, s_2)$, message $m = \{0, 1\}^*$
Output: Signature $(z_1, z_2')$ with integer coefficients in the range
            $[-(k-32), k-32]$ and 160-bit hash output $c$
1 Choose random polynomials $(y_1, y_2)$, from the polynomial ring
   $\mathbb{Z}_p[x]/(x^n + 1)$ with integer coefficients from the set
   $\{-k, \ldots, -1, 0, 1, \ldots, k\}$ where $k$ is the predefined security parameter
2 $c = Hash(Transform((ay_1 + y_2)), m)$, 160-bit hash value of the higher
   order bits of $(ay_1 + y_2)$
3 $z_1 = s_1c + y_1$
4 $z_2 = s_2c + y_2$
5 If the coefficients of $z_1$ or $z_2$ are not in the range $[-(k-32), k-32]$ then
   go to Step 1.
6 $z_2' = Compress(ay_1 + y_2, p, k-32)$ compression of the polynomial $z_2$ into $z_2'$
7 if compression fails then
8     go to Step 1
9 end
```

Signature Verification

Algorithm 6 defines the signature verification process. In Step 1 range check is done for signature pair (z_1, z_2). In Step 2 hashed value is compared with c. For details please see [25].

Algorithm 6: Signature Verification Algorithm

```
Input  : Signature $(z_1, z_2')$ with integer coefficients in the range
         $[-(k-32), k-32]$, public key $t$, 160-bit hash , output c, message
         $m = \{0,1\}^*$
Output: “Verified” or “not verified”
if the coefficients of $z_1$ or $z_2$ are not in the range $[-(k-32), k-32]$ then
    return “not verified”
end
if $c = Hash(Transform(az_1 + z_2' - tc), m)$ then
    return ”verified”
else
    return “not verified
end
```

2.3 GPU Technologies

In this section we give a short introduction to GPU technologies and CUDA platform. 2000s manufacturers' main goal is to speed up the clock cycle of the processors. However limitations on manufacturing integration circuits made infeasible to get big performance gains on central processors. This causes the huge usage of parallel environments in multicore computers, notebooks and even mobile phones.

With these developments parallel environments have shifted from super computers to tools we use daily. In the meantime, graphic processing underwent a dramatic revolution. Revolutions started with 2D display accelerators used in personal computers and continued with 3Ds used in many applications like graphics and gaming. NVIDIA and ATI technologies evolved the graphic accelerators and made affordable to users on many applications [1].

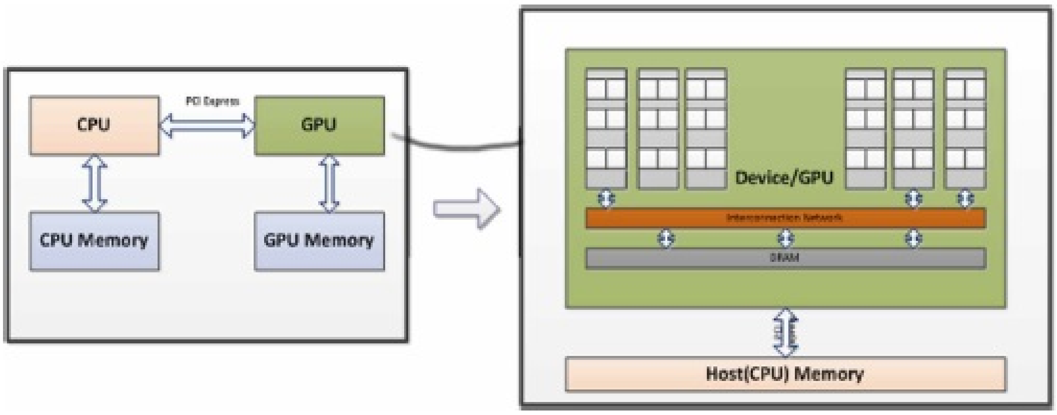


Figure 2.1: Simple CPU/GPU hardware configuration.

Early release of NVIDIA GPUs used Microsoft's DirectX standard. Then NVIDIA announced the first GPU cards GeForce 8800 GTX that is built with NVIDIA's CUDA. This new architecture has brought new components designed for strictly GPU computing and new alternatives for usage of graphic processors on general-purpose applications. In this study we prefer to use NVIDIA (Quadro 600) GPU, because it has more implementation alternatives compared to ATI technologies. Now we give a brief survey on hardware and software configuration of NVIDIA CUDA.

2.3.1 NVIDIA GPU Hardware Configurations

The main functionality of the "chipset" or "core logic" is connecting the CPU to outside world. Every input/output from network controllers, disk, keyboards, GPUs goes through this chipset [58]. GPUs were connected through these chipsets called PCI Express bus. Theoretically PCI Express designed to deliver about 500 MB/s of bandwidths which was not appropriate for GPUs. With new hardware configurations GPUs are designed up to 8G/s of bandwidths which is appropriate for parallel applications.Figure 2.1 shows the basic GPU architecture.
Multiple CPUs, CPU with integrated memory controller and integrated GPUs have different architectures that are widely used in GPU hardware for different needs. For interested readers, we recommend The CUDA Handbook [58] .

2.3.2 NVIDIA GPU Software Configurations

CUDA software designed in layer basis, at the deepest level starts with driver and continues with some tools such as driver APIs, runtime environment, libraries. In Figure 2.2 layered structure of CUDA software is given.
CUDA software, designed to operate on Windows, Linux and MacOS, is a parallel computing programming model maintained by NVIDIA. Since CUDA is an extension of the C programming language, applications are developed in C/C++ programming language by directly using CUDA driver API or special libraries for linear algebra operation, matrix operations etc.

2.3.3 CUDA Program Structure

A CUDA program consists of two parts which are run on either the host (CPU) or a device such as a GPU. The parts that have rich amount of data parallelism are implemented in the device code. The program has a single source code for both host and device code. The NVIDIA C Compiler (NVCC) compiles the host code on standard C compiler and runs it as an ordinary process. The device code is written using CUDA API including parallel functions, compiled by NVCC and then executed on a GPU device. Figure 2.4 shows the CUDA execution model. Each GPU kernel call launches CUDA grids and device code run on the GPU.A

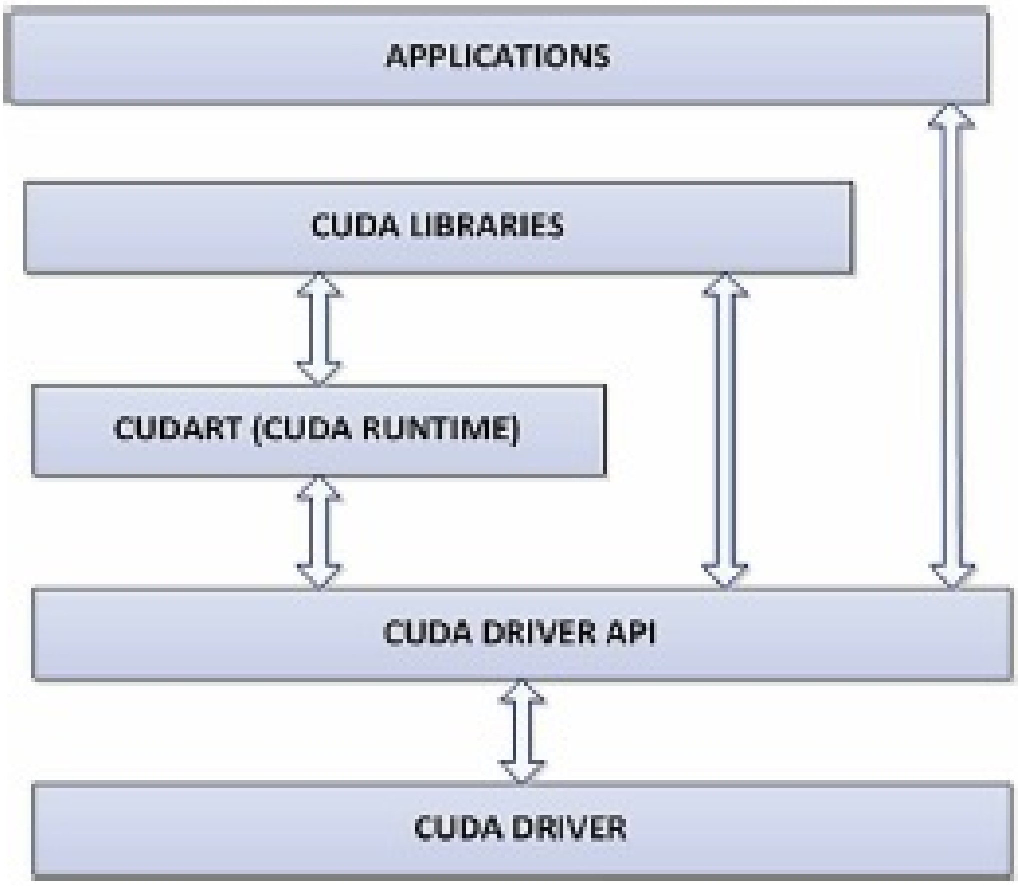
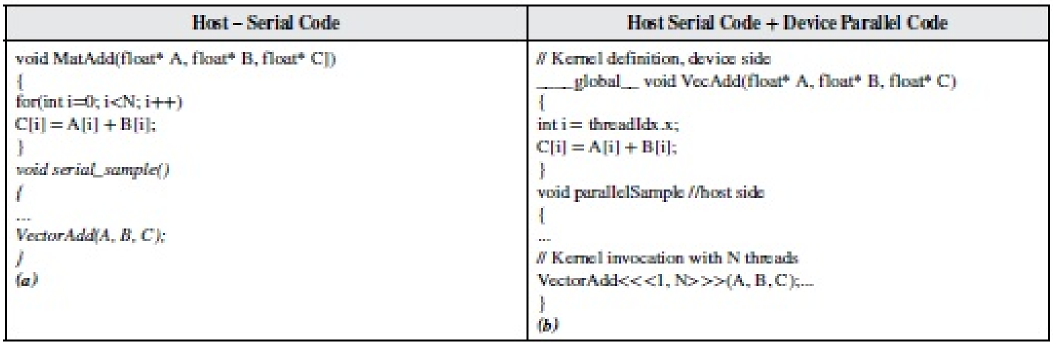


Figure 2.2: Layered structure of CUDA software.

sample code for parallel CUDA implementation is given in Figure (2.3).

CUDA parallel functions (kernel functions) use special thread structure that is composed of grids and blocks. At the basic level grids map to GPUs, blocks map to Multiprocessors (MP) and threads map to Stream Processors (SP). All threads have unique coordinates to handle which portion of the data to be processed. 2D hierarchy exists called blockId and threadId in CUDA runtime system for thread. Figure 2.4 shows the CUDA grid structure. CUDA grids have two dimensions which defines the size of the grid in x and y dimension. Each block consists of 2D or 3D thread structure in [x,y] or [x,y,z] dimensions. Both block and thread dimensions defines which thread will perform the required process.

The function to be parallelized called with kernel launch according to given block-thread structure. In the sample code VectorAdd function uses 1 block with N threads. Since only one block is called and 1D thread is used CUDA defined



Host – Serial Code	Host Serial Code + Device Parallel Code

void MatAdd(float* A, float* B, float* C])
{
for(int i=0; i<N; i++)
C[i] = A[i] + B[i];
}
void serial_sample()
{
...
VectorAdd(A, B, C);
}
(a)
	// Kernel definition, device side
	____global__ void VecAdd(float* A, float* B, float* C)
	{
	int i = threadIdx.x;
	C[i] = A[i] + B[i];
	}
	void parallelSample //host side
	{
	...
	// Kernel invocation with N threads
	VectorAdd<<<1, N>>>(A, B, C);...
	}
	(b)

Figure 2.3: Sample code.

threadIdx parameter is sufficient. For multiple blocks and 2D, 3D threads CUDA predefines parameters block dimension y and thread dimension z.

2.3.4 CUDA in Cryptographic Applications

Unlike previous parallel programming languages, CUDA gives an alternative way to easy development of parallel application on different fields such as cryptographic protocols. Several works are done for lots of cryptosystems on CUDA. The performance results are shared especially for block ciphers such as AES and DES. Also public key cryptographic protocols such as RSA, ECDSA and NTRU were implemented [28], [29], [48], [56]. In this thesis, we use CUDA API for polynomial multiplication operations. Note that CUDA has several libraries such as cuFFT and cuBLAS that are efficiently designed for specific operations. We also use the cuFFT library for polynomial multiplication operations which gives us a big performance gain on large bit sizes. cuFFT is the NVIDIA's Fast Fourier Transform product [1]. It consists of two separate libraries cuFFT and cuFFTW. The cuFFT library is designed to provide high performance on NVIDIA GPUs. The cuFFTW library is provided as a porting tool to enable users of FFTW to start using NVIDIA GPUs with a minimum amount of effort. We use the cuFFT library for comparison.

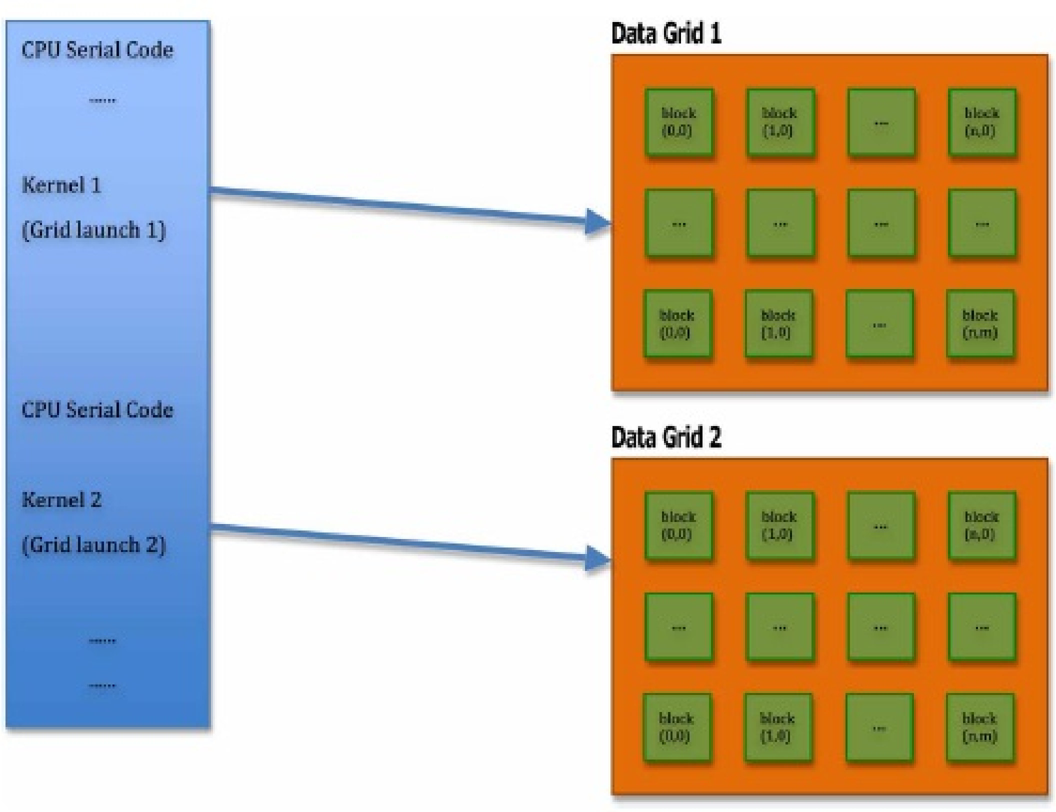


Figure 2.4: CUDA execution model.

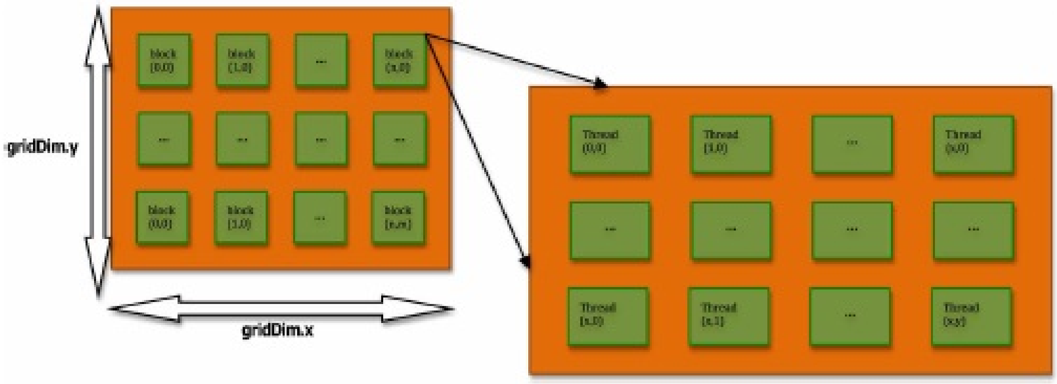


Figure 2.5: CUDA grid structure.

CHAPTER 3

MULTIPLICATION TECHNIQUES

In this chapter, polynomial multiplication techniques in lattice-based cryptographic schemes are discussed. Following section gives the algorithms specific to selected fields,namely $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ and $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$.

3.1 Multiplication over the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$

In this section we give the polynomial multiplication algorithms to be used for lattices defined in the field $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. Schoolbook, Parallelized Schoolbook, Fast Convolution and Fast Convolution with Sliding Window methods are discussed in the following sections.

3.1.1 Schoolbook Method

In Algorithm 7, the naive multiplication method called schoolbook is given. In this method, each coefficient of the first polynomial is multiplied with the other polynomial and then modular reduction operation according to p is performed. The multiplication is defined as follows:

Let $a(x) = \sum\limits_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum\limits_{i=0}^{n-1} b_i x^i$ be polynomials in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. Then, $c(x) = a(x)b(x)$ with $c_j = \sum\limits_{i=0}^{j} a_i b_{j-i} + \sum\limits_{i=j+1}^{n-1} a_i b_{n+j-i}$ by using $x^n \equiv 1 mod(x^n - 1)$.

The classical schoolbook algorithm has a quadratic complexity $\mathcal{O}(n^2)$ with n^2 multiplications and $(n-1)^2$ additions.

In Algorithms 8 and 9, parallelized version of schoolbook method is given. Algorithm 8 consists of two parts: CPU and GPU sides. In CPU side, there is a need to create GPU grids and blocks with respect to the degree of the polynomial. Then, from Step 2 to Step 4 allocating memory operation is performed for CUDA platform. In Algorithm 9, we give the GPU side of the schoolbook algorithm.

Algorithm 7: Schoolbook Algorithm

```
   Input  : $a(x) = \sum_{i=0}^{n-1} a_i x^i$, $b(x) = \sum_{i=0}^{n-1} b_i x^i \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$
   Output: $c(x) = a(x)b(x) = \sum_{i=0}^{n-1} c_i x^i$
 1 Set all coefficients of $c(x)$ to 0
 2 for $k$=0 to $n$ do
 3     for $i$=0 to $n$ do
 4         $c_k = c_k + b_i a_{(n+k-i)} \mod p$
 5     end
 6 end
 7 return $c(x)$
```

Parallel multiplication is performed in Step 5. In GPU side, each coefficients of $a(x)$ is multiplied by $b(x)$ and then added to $c(x)$ in threads. In Algorithm 8 Step 6, the results are turned back to the host (CPU Side).

Algorithm 8: Parallelized Schoolbook Method CPU and GPU side

```
   Input  : $a(x) = \sum_{i=0}^{n-1} a_i x^i$, $b(x) = \sum_{i=0}^{n-1} b_i x^i \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$
   Output: $c(x) = a(x)b(x) = \sum_{i=0}^{n-1} c_i x^i$
 1 grids, blocks = create gpu grids and blocks according to $n$
 2 allocate_cuda_memory(cuda_a)
 3 allocate_cuda_memory(cuda_b)
 4 allocate_cuda_memory(cuda_c)
 5 call gpu_schoolbook_procedure grids,blocks $(cuda\_a, cuda\_b, cuda\_c)$
 6 cuda_copy(cuda_c,c)
 7 return c
```

Algorithm 9: Parallelized Schoolbook Method GPU side

```
Input   : $cuda_a, cuda_b, cuda_c$
Output: $cuda_c$
$threadIdx = block\_id.x \cdot block\_dimension.x + threadIdx.x$
if $threadIdx > n$ then
    return
end
for $i$=0 to $n$ do
    $c\_threadIdx = c\_threadIdx + b\_threadIdx - ia\_i \mod p$
end
return c(x)
```

3.1.2 Fast Convolution Method

The most time consuming part of NTRU encryption and decryption are polynomial multiplications of products $r \cdot h$ and $e \cdot f$. According the design of the algorithm public key h and encrypted message e are almost randomly distributed modulo q. So we can choose r and f in a way that reduce the computation density. r and f are generally selected that have binary coefficients which results to get product without multiplication. The required computation is the hamming weight of the binary polynomial * polynomial degree addition and modulo q operation. By choosing low hamming weights one can gain significant performance gain. Fast convolution algorithm [46], [47] uses the following modular reduction fact:

$$x^n \equiv 1 \mod (x^n - 1), x^{n+1} \equiv x \mod (x^n - 1), ..., x^{2n-2} \equiv x^{n-2} \mod (x^n - 1)$$

With the help of these equations the modular multiplication of two elements a and b can be written in a matrix-vector product form as follows:

$$c(x) = \begin{bmatrix} a_0 & a_{n-1} & ... & a_2 & a_1 \\ a_1 & a_0 & ... & a_3 & a_2 \\ . & . & ... & . & . \\ . & . & ... & . & . \\ . & . & ... & . & . \\ a_{n-2} & a_{n-3} & ... & a_0 & a_{n-1} \\ a_{n-1} & a_{n-2} & ... & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ . \\ . \\ . \\ b_{n-2} \\ b_{n-1} \end{bmatrix}$$

Notice that $c_j = \sum_{i+k=jmodn} a_i b_k$ each row of this a matrix is the one cyclic shift of the previous row. In Algorithm 10 we give the fast convolution algorithm step by step. The complexity of fast convolution algorithm depends on the Hamming weight of $a(x)$. Assume that Hamming weight of $a(x)$ is e, then the required number of operations is $(e + 1)n$ (Step 4 and Step 8) and n (Step 8), additions and modular reductions, respectively.

In Algorithm 10 Step 4 is executed $d \cdot n$ times. In Step 4 there are two additions on e for index computation, one for addition of c and b index elements. Step 8 is executed n times. In Step 8 there are 2 additions one for index computation and one for c is addition. Also there are n reduciton. Total number of operation is $2n(d + 1)$ addition and n reduction.

Algorithm 10: Fast Convolution Algorithm

```
Input  : $a(x) = \sum_{i=0}^{n-1} a_i x^i,\ b(x) = \sum_{i=0}^{n-1} b_i x^i \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. $d$ is an
         array showing the index of 1's in a(x) and e is the number of 1's.
Output: $c(x) = a(x)b(x) = \sum_{i=0}^{n-1} c_i x^i$
Set all coefficients of c(x) to 0
for i=0 to e-1 do
    for j=0 to n-1 do
        $c_{j+d[i]} = c_{j+d[i]} + b_j$
    end
end
for i=0 to n-1 do
    $c_i = (c_i + c_{i+n}) \mod p$
end
return c(x)
```

3.1.3 Fast Convolution with Sliding Window Method

One can speed up the fast convolution method by using some patterns in the polynomials, which is called sliding window method for NTRU [36] , [37]. The basic idea lies on the structure of a convolution operation. Let $c = a \cdot b$ where $a, b, c \in \mathbb{R}$. Note that multiplication of $a \in \mathbb{R}$ and $ab \in \mathbb{R}$

This method needs some memory to store the results for the corresponding pattern. In order to apply multiplication with sliding window method , one needs to convert the coefficients to binary form. By searching for simple patterns like 1,11,101,1001, etc. the same coefficients are calculated only once and stored in the look-up table. The main idea is to find patterns that have many 1's and repeat many times in the coefficients for an efficient implementation. Note that the selected patterns do not share the same 1's.

Lemma 5. Let u be the number of 1's in the pattern and v be the number of occurrences for the selected pattern. Then, the required number of additions is $n(v + u - 1)$ over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$.

Lemma 6. [36, 37] Consider a task that chooses a bit according to a distribution where the probability that is selected is p. We repeat this task independently to choose coefficients of a binary polynomial. Let $\mathbb{Z}$ be the distance between two neighboring occurrences of 1's. Then $Pr[Z > d] = (1 - p)^d$.

By Lemma 5 patterns in different forms help to reduce the required number of additions. We have the following steps to use multiplication with sliding window method:

- **Finding Patterns:** We partition the binary string into short blocks of a fixed length, w, except for the parts containing consecutive zeroes. This is equivalent to a sliding window method with window size w in the context of modular exponentiation in RSA. Pattern search technique is given in Algorithm 11. In this study we focus on finding the patterns having only two 1's with the first and last position of the string for example 11, 101, 1001 and so on.

- **Recoding**: After using Algorithm 11, we make a list to show where the pattern ends in the binary representation. For example, let a=(1001011101) and w=3. Then, $d_0 = \{0, 6\}$ shows the positions of 1's not in the any pattern. $d_2 = \{5, 9\}$ gives the positions where the pattern (101) finishes (numbering is left to right). Note that since we use the 1's only once, there is no (11) pattern; so d_1 is an empty set.

- **Precomputation**: By using the selected patterns, a look-up table is computed.

- **Multiplication:** The coefficients of the $c(x)$ are computed as in fast convolution manner.

Algorithm 11 finds the patterns according to window size w and stores their indices. Scanning of bits for pattern search is done through right to left because of the Algorithm 12.

```
Algorithm 11: Pattern Search Algorithm
Input  : y is a binary string having n elements and w is the window size
Output: $d_0$, is an integer array representing the positions of ‘1’s not
        included in any pattern. $d_1$, $d_2$ and $d_{w-1}$ is also an array giving
        the positions of 1’s in y for 11, 101, . .., 100...1 of length w
        respectively
i = n − 1
while $i \geq 0$ do
    while  $i \geq 0$ and $y[i]! = 1$ do
        i=i-1
    end
    for j=1 to w-1 do
        if y[i-j] == 1 then
            append i to $d_j$
            i =i-j-1
        else
            append i to $d_0$
            i=i-w
        end
    end
end
return all $d_i$’s
```

Example 7. Example
Let w= 6 N= 34
0**3****8 10***15**19***24**28****34
10010000101100010001000010001011001
Pattern 0 {1} :[]
Pattern 1 {11} : []
Pattern 2 {101} : [30,10]
Pattern 3 {1001} : [34, 3]
Pattern 4 {10001} : [15]
Pattern 5 {100001} : [24]

In Algorithm 12, fast convolution with sliding window method is given. In this method the operations are done for the related pattern obtained by using Algorithm 11. Note that the number of stored integers is $n(w-1)$. Precomputation is done from Step 2 to Step 10. There are $w-1$ addition in Line 3 and $2(w-1)n$ additions in line 6. Multiplication is performed between Step 11 and Step 21.

Large values of w is not considered since amount of computation almost the same as the smaller ones while the amount of memory is proportional to w.

Algorithm 12: Fast Convolution with Sliding Window Method

```
Input  : For $0 \leq i \leq (w–1)d_i$ is an array having the positions $e_i$ of the
         pattern $p_i$ in the binary polynomial $a(x) \cdot b(x)$ is a polynomial of
         degree (n-1) and w is the window size.
Output: $c(x) = a(x)b(x) = \sum_{i=0}^{n-1} c_i x^i$
Set all coefficients of c(x) to 0
for $j$=0 to $w - 1$ do
    $b_{j+n} = b_j$
end
for $i$=1 to $w - 1$ do
    for $0 \leq j \leq n$ do
        $c_i[j] = b_j + b_{j+1}$
    end
end
Let $c_0 = b$ i.e. $c_0[k] = b \ for \ 0 \leq k \leq n$
for $i$ = 0 to $w$-1 do
    for $j$ =0 to $e_i$ do
        for $k$ = 0 to $n$ do
            $c_{k+d_i[]} = c_{k+d_i[]} + c_i[k]$
        end
    end
end
for $j$ = 0 to $n$ do
    $c_j = (c_j + c_{j+n}) \mod p$
end
return $c(x)$
```

3.1.4 Interleaved Montgomery Modular Multiplication

Montgomery multiplication is one of the widely used multiplication algorithms in cryptography. It's efficiency is one of the main reason of its usage. In Montgomery modular multiplication method one needs to transform the elements to the required form. For the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n-1)$, for given $a(x)$ and $b(x) \in \mathbb{R}_q$, first compute $a(x) \cdot b(x)$ in the form enabling very efficient computation and then transform the result to final computation [4].

In NTRU, polynomial arithmetic is performed in $\mathbb{R}_q$ i.e. $x^n = 1$. In Montgomery modular multiplication algorithm one needs $M'(x) \equiv -M(x)^{-1} \mod x^w$, where w is the word length of the target platform. In Lemma 8 we give the computation of $M'(x)$ for the NTRU case. Note that this also helps us to eliminate the precomputation phase.

Lemma 8. Let $M(x) = x^n - 1$ and $M'(x) \equiv -M(x)^{-1} \mod x^w$, where $w \leq n$. Then $M'(x) = 1$.

In Algorithm 13 we give modified interleaved Montgomery modular multiplication algorithm for NTRU. After using the observation in Lemma 8, we decrease the required number of multiplications by one with omitting the multiplication $M'(x)$ (see Step 4). Then, we replace the multiplication with $M'(x) = x^n - 1$ by shifting n times and one subtraction (see Step 5). Recall that shifting operation is almost free. We convert the multiplication with $M(x)$ to shifting and subtraction operations which improves the complexity of the algorithm. Since we are working on the Montgomery form, we need to convert the elements to the desired form. Conversion of the result is done by shifting operation (see Step 8 and 9). In Algorithm 13, the required number of multiplications is reduced to 1 (see Step 3) and the required number of additions is 3 (see Step 3 and 5).

Algorithm 13: Interleaved Montgomery Modular Multiplication Algorithm for NTRU

```
  Input  : $A(x) = \sum_{i=0}^{n-1} a_i x^{iw}, B(x) = \sum_{i=0}^{n-1} b_i x^{iw}, M = x^n - 1$ with $a_i, b_i \in \mathbb{Z}_q$
           where q is a prime power, $deg(A(x)) < deg(M(x)), deg(B(x)) <$
           $deg(M(x)), gcd(r(x), M(x)) = 1, r(x) = x^w$ and $n^w = \lceil \frac{n}{w} \rceil$.
  Output: $C(x) = A(x) \cdot B(x) mod M(x)$
1 Set all coefficients of C(x) to 0
2 for $i = 0$ to $n_w - 1$ do
3     $C(x) = C(x) + A(x) \cdot b_i(x) mod(M(x))$
4     $q(x) = C(x) mod(r(x))$
5     $C(x) = (C(x) + q(x) \cdot x^n - \frac{q(x)}{r(x)})$
6 end
7 $T(x) = (r(x))^{nw}$
8 $C(x) = C(x) \cdot T(x) mod(M(x))$
9 return $C(x)$
```

3.2 Multiplication over the Quotient Ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$

In this section, we give some of the selected algorithms: sparse polynomial multiplication, interleaved Montgomery modular multiplication, number theoretic transform in serial and parallel type and CUDA-based FFT (cuFFT). We focus on the arithmetic over the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ having important applications in lattice-based cryptographic schemes due to the applicability of FFT-based polynomial multiplication enabling efficient modular multiplication. Note that we also implement the schoolbook method over the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$. Since the idea is very similar to Algorithm 7 and Algorithm 8, we omit it.

3.2.1 Number Theoretic Transform

Number Theoretic Transform (NTT) algorithm was proposed in [51] to avoid rounding errors in Fast Fourier Transform (FFT). NTT, a Discrete Fourier Transform defined over a ring or a finite field, is used to multiply two integers and does not require arithmetic operations in complex numbers. The multiplication complexity is quasi-linear $\mathcal{O}(nlogn)$. The main idea is to transform polynomials to NTT form. Algorithm 14 describes the iterative NTT which is the modified version of [9, 5, 6] .

There are some restrictions on applying NTT algorithm:

- The degree of the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ n should divide $(p-1)$
- $w^n \equiv 1 \mod p$ and for each $i < n$, $w^i \not\equiv 1 \mod p$

Let w be the primitive n-th root of unity. For $a(x) = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Z}_p[x]$ by using w, $NTT(NTT_w(a))$ is defined as follows:

$$A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \mod p \ , \ i = 0, 1, ..., n-1$$

where $A = \{A_0, A_1, \ldots A_{n-1}\}$ is the NTT form.
The inverse transform $NTT_w^{-1}(A)$ is given as:

$$a_i = n^{-1} \sum_{j=0}^{n-1} A_j \omega^{-ij} \mod p \ , \ i = 0, 1, ..., n-1$$

By using Convolution Theorem arbitrary polynomials can be multiplied and then reduce according to chosen reduction polynomial. However appending n 0's to the inputs doubles the transform size [59]. To use NTT to multiply two elements in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$. the condition is $p \equiv 1 \mod 2n$ due to wrapped convolution approach [5, 6, 39]. This idea is given in lattice-based hash function SWIFFT which needs modular reduction operations in (x^n+1).

After computing NTT of two polynomials, the multiplication operation can be performed. Algorithm 14 gives polynomial multiplication with iterative NTT [5, 6]. In Algorithm 16 the parallel version of iterative NTT method is given. To make it efficient, we focus on "for" loops. Parallelization is achieved by determining the required number of threads. This algorithm needs data transfer between CPU and GPU. Thus, there is a delay and this causes inefficiency.

Algorithm 14: Iterative NTT algorithm

```
Input  : $a \in (Z/p\mathbb{Z})[x]/(x^n+1)$ and $\omega \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ is the
         primitive n-th root of unity
Output: $NTT_w(a)$
f = BitReverseCopy(a)
f = SumDiff(f)
n = 2ldn //length of the sequence
rn = element_Of_Order(n)
if reverse NTT then
    rn = rn(-1)
end
for ldm=1 to ldn do
    $m = 2^{ldm}$
    $mh = m/2$
    $dw = rn^{2^{ldnldm}}$
    $w = 1$
    for j=0 to mh-1 do
        for r=0 to n-m step m do
            $t1 = r + j$
            $t2 = t1 + mh$
            $v = f[t2] * w$
            $u = f[t1]$
            $f[t1] = u + v$
            $f[t2] = u - v$
        end
    end
end
return f
```

Algorithm 15: Polynomial multiplication with iterative NTT

```
Input  : $a(x), b(x) \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$
Output: $c(x) = a(x)b(x)$
ntta[]=iterative_NTT(a[], ldn, forward)
nttb[]=iterative_NTT(b[], ldn, forward)
for i=0 to n-1 do
    c[i]= ntta[i]nttb[i] mod p
end
c=iterative_NTT(c[], ldn, inverse)
for i=0 to n-1 do
    c[i]=c[i]/(ldn-1)
end
return c
```

Algorithm 16: Parallelized Iterative NTT Method (CPU and GPU Side)

```
Input  : $a(x), b(x) \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$
Output: $c(x) = a(x)b(x)$
f = BitReverseCopy(a)
f = SumDiff(f)
ldn base 2 logarithm on n
rn = elementOfOrder(n)
if reverse NTT then
    $rn = rn^{-1}$
end
for ldm=2 to ldn do
    $m = 2ldm$
    $mh = m/2$
    $dw = rn^{2^{ldnldm}}$
    create blocks and grids
    $callgpu\_ntt\_procedure \ll grids, blocks \gg (f, dw)$
end
return f
```

Algorithm 17: Parallelized Iterative NTT Method GPU side

```
Input  : $a(x), b(x) \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$
Output: $c(x) = a(x)b(x)$
thread_IdX = block_Id.x * block_Dimension.x+ threadIdx.x
thread_IdY = block_Id.y * block_Dimension.y+ threadIdx.y
if $thread\_IdX > 0$ or $thread\_IdY == 0$ then
    return
end
for $i = 0$ to $thread\_IdY$ do
    $w = w \cdot dw \mod p$
end
t1 = thread_IdX *mh + thread_IdY
t2= t1*mh
v= at2*w modp
u = at1
at1 = u +v modp
at1 = u -v modp
```

3.2.2 CUDA Fast Fourier Transform (cuFFT) Based Multiplication

The NVIDIA cuFFT library enables the users to have very fast FFT computations with an interface on the GPU by using CUDA platform [1]. cuFFT is optimized for a wide range application area from computational physics to signal processing. In this thesis we use cuFFT to obtain an efficient polynomial multiplication over the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$. In Algorithm 18 we give the parallel version of polynomial multiplication method using CUDA. In Step 5 the schedule is planned to have FFT value on GPU and then in Step 6 and Step 7 the computed values are stored. The component wise multiplication is performed in a parallel way in Step 8. Forward FFT is achieved in Step 9. The result is sent to host in Step 11. From Step 13 to Step 15 normalization of the computed values is performed by simply dividing the result to the polynomial degree n.

Algorithm 18: CUDA Fast Fourier Transform (cuFFT) Based Multiplication

```
   Input  : $a(x), b(x) \in (\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$
   Output: $c(x) = a(x)b(x)$
1  allocate_cuda_memory(cuda_a)
2  allocate_cuda_memory(cuda_b)
3  cuda_copy(a, cuda_a)
4  cuda_copy(b,cuda_b)
5  cufftPlan1d(planForward, n,CUFFT_D2Z, 1)
6  cufftExecD2Z(planForward, cuda_a, cuda_a)
7  cufftExecD2Z(planForward, cuda_b, cuda_b)
8  multiply_complex(cuda_a, cuda_b, cuda_a)
9  cufftPlan1d(planInverse, n,CUFFT_Z2D, 1)
10 cufftExecD2Z(planInverse, cuda_a, cuda_a)
11 copy the result from gpu to host
12 cuda_copy(cuda_a,c)
13 for $i=0$ to $n$ do
14     $c_i = \frac{c_i}{n}$
15 end
16 return c
```

3.2.3 Interleaved Montgomery Modular Multiplication Algorithm for $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$

Montgomery Modular Multiplication is widely used in elliptic curve systems due to its efficiency. In this study, we implement the Interleaved Montgomery Modular Multiplication proposed by [49]. The parameters are $n = 2^k$ and $p \equiv 1 \mod 2n$, where k is positive integer and p is prime. In Lemma we give the computation of $M'(x)$ for $\mathbb{R}_p$ case . By using this, one multiplication in interleaved Montgomery method turns out to the subtraction operation.

Lemma 9. Let $M(x) = x^n + 1$ and $M'(x) \equiv M(x)^{-1} \mod x^w$, where $w \leq n$. Then $M'(x) = -1$.

Now we have $M'(x) = -1$. In Step 4 we replace the multiplication by $M'(x)$ with multiplication by -1. Now that this can be also considered as an addition modulo p. Since in R-LWE based schemes coefficients of the elements are chosen from the set{-1,0,1}, this multiplication by -1 does not effect the efficiency of the algorithm. In Step 5, instead of multiplication with $M(x) = x^n + 1$ we use shifting the corresponding polynomial n times and then add it. With this observation we decrease the number of multiplication in the algorithm. In Step 8 and 9 we convert the elements to the desired form. Note that these are not the real multiplications, they are just shifting operations. The multiplication and addition complexity of Algorithm 19 is only 1 (see Step 3) and 3 (see Step 3 and 5), respectively.

Algorithm 19: Interleaved Montgomery Modular Multiplication Algorithm for $x^n + 1$

```
Input  : $A(x) = \sum_{i=0}^{n-1} a_i x^{iw}, B(x) = \sum_{i=0}^{n-1} b_i x^{iw}, M = x^n - 1$ with $a_i, b_i \in \mathbb{Z}_q$
         where q is a prime power, $deg(A(x)) < deg(M(x)), deg(B(x)) <$
         $deg(M(x)), gcd(r(x), M(x)) = 1, r(x) = x^w$ and $n^w = \lceil \frac{n}{w} \rceil$.
Output: $C(x) = A(x) \cdot B(x) mod M(x)$
1 Set all coefficients of C(x) to 0
2 for $i = 0$ to $n_w - 1$ do
3     $C(x) = C(x) + A(x) \cdot b_i(x) mod(M(x))$
4     $q(x) - C(x) mod(r(x))$
5     $C(x) = (C(x) + q(x) \cdot x^n - \frac{q(x)}{r(x)})$
6 end
7 $T(x) = (r(x))^{nw}$
8 $C(x) = C(x) \cdot T(x) mod(M(x))$
9 return $C(x)$
```

3.2.4 Sparse Polynomial Multiplication

In [10], it is noticed that the multiplication of polynomials could be performed using only additions because of the cyclic structure of the quotient ring $(x^n + 1)$. We extend this idea to the multiplication of two elements in $\mathbb{R}_p$.

Remark 10. Our observation is that some of signature schemes (for example, [19] , [24]) use sparse polynomials whose coefficients are in the set $\{-1, 0, 1\}$ and the number of nonzero coefficients is relatively small. Then, two of these polynomials or one of them completely random are multiplied. This observation yields a natural modification of the algorithms in [10].

The proposed method uses the easy way of reduction. Now we recall this idea. Note that we are working in $\mathbb{R}_p$. Then, $x^n \equiv -1 \mod (x^n + 1)$, $x^{n+1} \equiv -x \mod (x^n + 1)$, $x^{n+2} \equiv -x^2 \mod (x^n + 1), ..., x^{2n-2} \equiv -x^{n-2} \mod (x^n + 1)$. With the help of these equations, the multiplication of two elements $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum_{i=0}^{n-1} b_i x^i$ in $\mathbb{R}_p$ can be written in a matrix–vector form as follows:

$$c(x) = \begin{bmatrix} a_0 & a_{n-1} & ... & a_2 & a_1 \\ -a_1 & a_0 & ... & a_3 & a_2 \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ -a_{n-2} & -a_{n-3} & ... & a_0 & a_{n-1} \\ -a_{n-1} & -a_{n-2} & ... & -a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ . \\ . \\ . \\ b_{n-2} \\ b_{n-1} \end{bmatrix}$$

where c is the coefficient vector of c(x). Note that each row of this A matrix is one cyclic shift of the previous row and upper triangular part is multiplied by (-1). One can also formulate this multiplication as follows:

$$c_{i+j} \mod n = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (-1)^{\lfloor \frac{i+j}{n} \rfloor} a_i b_j$$

In Algorithm 20, we explain the sparse polynomial multiplication method step by step. Note that since $a_i, b_i \in \{-1, 0, 1\}$, we cannot apply the algorithm given in [10], [36], [37] directly for the multiplication in $\mathbb{R}_p$. While constructing Algorithm 20, we use the property of the matrix–vector product having the minus of elements in the upper triangular part.We add a new loop (from Step 8 to 12) for the elements $a_i = 1$. In Step 5, for each index of 1s in array d, we shift the coefficients vector of b by the number of index value and add to c. Similarly, in Step 10 for each index of -1s in array d, we shift the coefficients vector of b by the number of index value and subtract from c. It is easy to see that the arithmetic complexity of Algorithm 20 depends on the nonzero elements in the coefficient vector a. The complexity of determining the number of 1s and (-1)s is just 2n comparisons and thus precomputation step is linear.

Algorithm 20: Sparse Polynomial Multiplication

```
Input  : $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $b(x) = \sum_{i=0}^{n-1} b_i x^i$, are the elements of $\mathbb{R}_p$ with
         $a_i, b_i \in \{-1, 0, 1\}$, i.e. $a = (a_0, a_1, ..., a_{n-1})$ and
         $b = (b_0, b_1, ..., b_{n-1})$.d and f are arrays of e and g elements storing
         the index of 1s and (-1)s in $a(x)$ respectively.
Output: $c(x) = a(x)b(x) \mod (x^n + 1) = \sum_{i=0}^{n-1} c_i x^i$
Precomputation : Set $d[]$ and $f[]$
for i=0 to 2n-1 do
    $c_i$ =0 // set all coefficients of c(x) to 0
end
for i=0 to e-1 do
    for j=0 to n-1 do
        $c_{j+d[i]} = c_{j+d[i]} + b_j$ //add $a_i.b_j$ where $a_i = 1$
    end
end
for i= 0 to g-1 do
    for j= 0 to n-1 do
        $c_{j+f[i]} = c_{j+f[i]} - b_j$ //add $a_i.b_j$ where $a_i = -1$
    end
end
for i=0 to n-1 do
    $c_i = (c_i - c_{i+n}) \mod p$ // reduction of polynomial modulo $x^n + 1$ and
     its coefficients
end
return c(x)
```

Lemma 11. The required number of additions and subtractions in Algorithm 20 is $(g+1) \cdot n$ and $e \cdot n$ (in total bounded by $n^2 + n$, where e is the number of 1s and g is the number of (-1)s in the coefficient vector of $a(x)$, respectively.

Since the required number of additions and subtractions in Algorithm 20 depends on the number of nonzero elements in the coefficient vector of $a(x)$ (the multiplicand), Algorithm 20 gives better performance for the multiplication of polynomials having so many zeros. In [26], the hash function outputs a 512- coefficient polynomial having only 32 nonzero (-1 or 1) coefficients (see [26]). Then, Algorithm 20 is very efficient for this case. We also note that if the number of nonzero elements (-1 or 1) of a 512-coefficient vector is less than 340, then the proposed method is also efficient than NTT (see Remark 14).

Algorithm 20 can be used in any ring, i.e., it is independent of the choice of reduction polynomial. One can obtain sparse polynomial multiplication algorithm by changing the reduction part (steps 13-15 in Algorithm 20). Algorithm 20 can also be modified to be used to multiply to polynomials whose coefficients are in $\{-k, ..., 0, ..., k\}$, where k is a small integer. The main modifications are to add new loops for each coefficients in $\{-k, ..., 0, ..., k\}$ and then the multiplication of each b_i (see Steps 5 and 10) with the corresponding coefficient.

3.2.5 Sparse Polynomial Multiplication with Sliding Window

Extended pattern search
Using a detailed precomputation phase, the performance of Algorithm 20 can be improved significantly by storing some computations. We use the repeated patterns to decrease the arithmetic complexity of the polynomial multiplication. Then we adapt sliding window technique to Algorithm 20. Let w be the window length. In this paper, by a pattern of length w, we mean that (10...01), $(-10...0-1)$,$(10...0-1)$ or $(-10...01)$ with $\#0s = (w-2)$. For example, our pattern set includes $11, -1-1, -11, 1-1, 101, -10-1, 10-1, -101, ...$ With the help of this structure, if there is any repeated pattern, then we precompute and store some values, i.e., $b + x^l, -b - x^l, b - x^l$ or $-b + x^l$ where b is the coefficient array of $b(x)$ and $0 \leq l \leq w$. Note that if $l = 1$, then this corresponds to Algorithm 20.

Remark 12. There might be some patterns up to determined length in the coefficient array of $a(x)$. For example, let $a(x) = -1 + x - x^3 + x^4$, then the coefficient array is $(-1, 1, 0, -1, 1)$ and the repeated pattern is $\{-1, 1\}$. Therefore, the computation of the repeated pattern is performed only once. This, of course, helps us to improve the complexity of multiplication operation if the number of repeated patterns is satisfactory. To achieve this, we need some memory to store the precomputed values. Note that sparse polynomial multiplication with sliding window method is very efficient if the number of the repeated patterns is high. The intersection of different patterns yields an empty set, i.e., the patterns do not share any 1 or (-1).

In Algorithm 21, a detailed pattern search which finds the patterns in the coefficient array is given. This algorithm is the core part of the sparse polynomial multiplication with sliding window method since we need to find the repeated patterns. Algorithm 21 searches for specific patterns for given window length on polynomial coefficient. These specific patterns start with 1 or -1 and end with 1 or -1s. The start index of each pattern is stored in the arrays. Maximum length of the pattern is equal to window length. In Algorithm 21, we are looking for the repeated patterns such as $\{11, -1-1, -11, 1-1, 101, -10-1, 10-1, -101, ...\}$.

Sparse polynomial multiplication with sliding window method performs the polynomial multiplication using the properties of the specific patterns instead of only processing individual the coefficients equal to 1s and -1s. Sparse polynomial multiplication with sliding window method is given in Algorithm 22. This algorithm has a precomputational phase, i.e., one needs to find patterns using Algorithm 21. After determining the patterns, from steps 15 to 36 the results are computed and stored for selected patterns. In Algorithm 22, we precompute the subsequent coefficients addition/subtraction, i.e., $T_i[j] = b_j + b_{j+i}$ and $Tminus_i[j] = b_j - b_{j+i}$ where $i = 0, ..., n$ and $j = 1, ..., w$ (window length). The computations in the algorithm are performed using the following manner: For every element of each pattern in $pattern_i d_j$ we shift the value of T or $Tminus$ array by the value of index and add with c or subtract from c, respectively. The array T is used for the patterns which start and end with same value. The array $Tminus$ stands for the patterns which start and end with different values. Note that there is no

Algorithm 21: Extended Pattern Search Algorithm

```
Require: $a = (a_0, a_1, \ldots, a_{n-1})$ is the coefficient array of $a(x)$ with
    $a_i \in \{-1, 0, 1\}$ of length $n$ and $w$ is the window size.
Ensure: $pattern_0 d_0$ is an array representing the positions of separated 1s not
    included in any pattern $d_1, d_2, \ldots, d_{w-1}$ stand for patterns
    $(11, 101, \ldots., 10\ldots01)$ of length $w$ arrays representing the positions,
    respectively. $pattern_1 d_0$ is an array representing the positions of separated
    (-1)s not included in any pattern. $d_1, d_2, ..d_{w-1}$ stand for patterns
    $(-1-1, -10-1, \ldots., -10\ldots0-1)$ of length $w$ arrays representing the
    positions respectively $pattern_3 d_1$ is an array representing the positions for
    $(-11)$. $d_2, ..d_{w-1}$ stand for patterns $(-101, 10-1, \ldots., -10\ldots01)$ of length
    $w$ arrays representing the positions respectively
i= n-1
while $i \geq 0$ do
    while a[i] = 0  do
        i=i-1
    end while
    for j=1 to w-1  do
        if a[i] = 1 and a[i-j] = 1  then
            append i to $pattern_0 d_j$ // determining the positions of patterns starting with 1 and ending with 1 ;
            i = i-j-1;
            break;
        else if if a[i] = 1 and a[i-j] = 1  then
            append i to $pattern_2 d_j$ // determining the positions of patterns starting with 1 and ending with -1;
            i= i-j-1;
            break;
        else if if a[i] = -1 and a[i-j] = 1  then
            append i to $pattern_3 d_j$ // determining the positions of patterns starting with 1 and ending with -1;
            i= i-j-1;
            break;
        else
            append i to $pattern_1 d_j$ // determining the positions of patterns starting with -1;
            i= i-j-1;
            break;
        end if
        if j= w-1  then
            if a[i] = 1  then
                append i to $pattern_0 d_0$ //determining the positions of separated 1s not included in any pattern;
                i=i-w;
            else
                append i to $pattern_1 d_0$ //determining the positions of separated -1s not included in any pattern;
            end if
        end if
    end for
end while
return $pattern_i d_j$
```

need for multiplications of coefficients. Arithmetic complexity of Algorithm 22 is discussed in Lemma 13.

Lemma 13. Let w be window length. Let u_1 and u_2 be the number of 1s and (-1)s in the patterns , and v_1 and v_2 be the number of occurrences for the selected pattern, respectively. Then the required number of additions and subtractions is $n(u_1 + v_1 + w + 1)$ and $n(u_2 + v_2 + w + 1)$ over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$, respectively

Algorithm 22: Sparse polynomial multiplication with sliding window method

```
Require: $b\,(x) = \sum_{i=0}^{n-1} b_i x^i$, is an element in $\mathbb{R}_p$ with $b_i \in \mathbb{Z}_p$. Send $a(x)$ to
    Algorithm 21 then receive $pattern_i d_j$ for $0 \leq i \leq 3$ and $1 \leq j \leq w-1$
Ensure: $c(x) = a(x)b(x) \mod (x^n + 1)$
for i=0 to 2n-1 do
    $c_i = 0$ // set all coefficients of c(x) to 0
end for
for $j = 0$ to $w - 1$ do
    $b_{j+n} = b_j$
end for
for i=1 to w  do
    for for j=0 to n  do
        $T_i[j] = b_j + b_{j+i}$ // addition of two subsequent coefficients
        $Tminus_i[j] = b_j - b_{j+i}$ //subtraction of two subsequent coefficients
    end for
end for
for $k = 0$ to $n$ do
    $T_0[k] = b_k$
    $Tminus_0[k] = -b_k$
end for
```

Algorithm 23: Sparse polynomial multiplication with sliding window method Cont.

```
for i = 0 to w  do
    for j= 0 to $pattern_0d_i$  do
        for k= 0 to n  do
            $c_k + pattern_0d_i[j] = c_k + pattern_0d_i[j] + T_i[k]$ // addition of the patterns starting and ending with 1 and precomputed values
        end for
    end for
    for j= 0 to $pattern_1d_i$  do
        for k= 0 to n  do
            $c_k + pattern_1d_i[j] = c_k + pattern_1d_i[j] - T_i[k]$ // addition of the patterns starting and ending with -1 and precomputed values
        end for
    end for
    for j= 0 to $pattern_02_i$  do
        for k= 0 to n  do
            $c_k + pattern_2d_i[j] = c_k + pattern_2d_i[j] + Tminus_i[k]$ // addition of the patterns starting with 1 and ending with -1 and precomputed values
        end for
    end for
    for j= 0 to $pattern_03_i$  do
        for k= 0 to n  do

            $c_k + pattern_23_i[j] = c_k + pattern_3d_i[j] - Tminus_i[k]$ // addition of the patterns starting with -1 and ending with 1 and precomputed values
        end for
    end for
end for
for $i = 0$ to $n - 1$ do
    $c_i = c_i - c_{i+n}$  mod $p$ // reduction of polynomial modulo $x^n + 1$ and its coefficients modulo p
end for
return c(x)
```

Table 3.1: Comparison of the proposed method with NTT multiplication.

Operation	**NTT Mult.**	**Negative Wrapped Conv.**	**Algorithm**
Mult	3n log 2n + 4n	3n/2 log n + 5n	-
Add/Sub	6n log 2n + n	3n log n	33n
Mode p	9n log 2n + 4n	9n/2 log n+ 5n	n

Remark 14. The arithmetic complexity of NTT multiplication is asymptotically $\mathcal{O}(nlogn)$. One can efficiently perform polynomial multiplication in $\mathbb{R}_p$ with $9n\log 2n+4n$ coefficient multiplications, $6n\log 2n+n$ additions/subtractions and $3nlog2n+4n$ multiplications (see Table 3.1). Thus the proposed methods more efficient than NTT for sparse polynomial multiplication having up to 340 nonzero coefficients for $(\mathbb{Z}/p\mathbb{Z})[x]/(x^{512}+1)$ (see Remark 12). The proposed methods work for any positive integers n and p. However, NTT is only applicable for $p \equiv 1 \mod 2n$ where p is a prime power. In lattice-based cryptography $n = 2^k$ is chosen for efficiency and security reasons.

CHAPTER 4

IMPLEMENTATION DETAILS & EXPERIMENTAL RESULTS

In this chapter, we give the implementation details and timing results of the multiplication algorithms for NTRU and Signature schemes. We receive experimental results for several platforms.

4.1 Implementation Details

We implement the lattice-based library by using C++. We use CUDA programming language which has similar syntax with C++ and which can be adding as a library to our C++ project. We design the library in a layered manner. At the bottom, we have the global definitions which are used by all classes in common. The the lattice polynomial classes are implemented to define the lattice polynomial. NTRU and GLP lattices implemented in different classes since their arithmetic differs because of the quotient rings they are built on. In one layer above, classes for lattice polynomial arithmetic implemented. Polynomial addition, subtraction and ring based multiplication algorithms are implemented on these classes. On the top of the library, NTRU and GLP signature schemes are implemented.

4.2 Experimental Results for Multiplication Algorithms over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$

4.2.1 Performance Results for Intel(R) Xeon E3-1230 3.30GHz Platform

Timing results of the multiplication algorithms on NTRU given according to the platform defined in Table 4.1.To obtain more consistent results the algorithms are run 1000 times for the uniformly random polynomials in the multiplication operation. To compare the multiplication operation on the GPU using CUDA platform we fix $p = 49201153$ satisfying $p \equiv 1 \mod 2n$. In Table 4.1, we list the properties of experiment platform.

Table 4.1: The configuration of experiment platform

CPU	Intel(R) Xeon E3-1230 3.30 GHz
Memory	8 GB
Operating System	Windows 7 (64 bit)
GPU Accelerator	NVIDIA Quadro 600
GPU Memory	1 GB

Table 4.2 shows the performance results of multiplication algorithms to multiply two elements in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ for $n = 1024$, 2048, 4096 and 8192. Recall that in previous sections, we give the algorithms of schoolbook method, parallelized school method, fast convolution method and fast convolution method with sliding window for multiplication $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$.

Table 4.2: Timing results of multiplication in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ (n/second)

n	**1024**	**2048**	**4096**	**8192**
Schoolbook Method	14,856	58,918	238,165	967,287
Fast Convolution Method	1,768	6,999	28,109	109,441
Fast Convolution With Sliding Window Method	0,994	3,892	15,307	59,486
Parallelized Schoolbook Method	2,964	5,298	19,745	80,319

For $n = 1024$ fast convolution method is better than parallelized schoolbook method. However, for $n > 1024$ parallelized schoolbook method is much more efficient than fast convolution method.

In Figure 4.1 timing comparison of the selected multiplication algorithms is demonstrated. Note that since schoolbook method has the worst timing results, we omit it in the figure. According to the timing results, for every parameter of n, fast convolution with sliding window method is the best method to multiply two elements over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. Recall that in fast convolution with sliding window method the required number of additions is drastically reduced when we compare this with the fast convolution method.

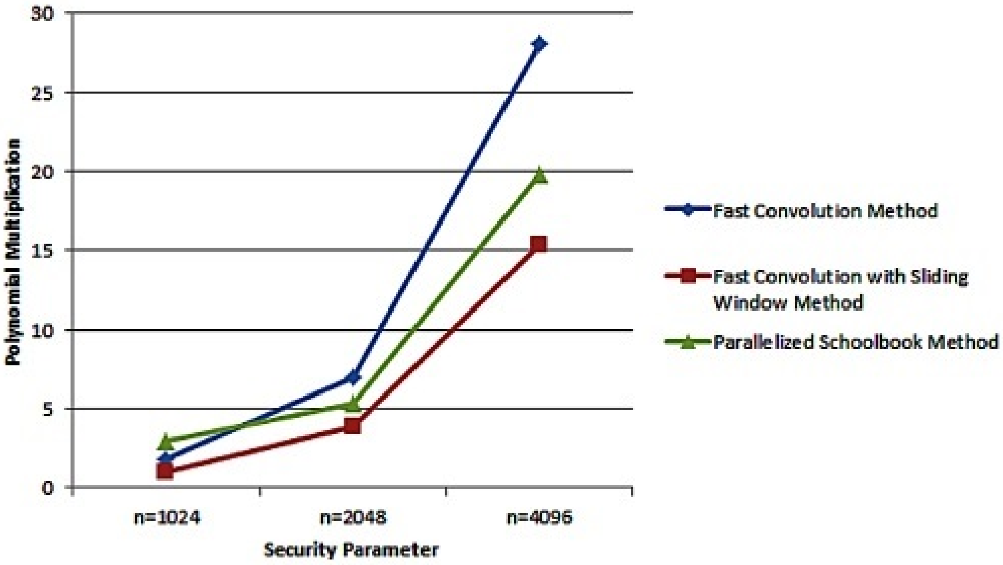


Figure 4.1: Timing comparison of multiplication methods over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ (second/n).

Performance Results for NTRUEncrypt

For NTRUEncrypt cryptosystem IEEEp1363.1 “Standard Specifications for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices” recommends a set of parameters for different security levels [32]. In Table 4.3,the parameters for NTRUEncypt with different security levels are summarized. NTRU-Encrypt cryptosystem is implemented according to each of these parameters.

Table 4.3: Recommended parameter sets for NTRUEncrypt in IEEEp1363.1

	N	p	q	d_f	d_g	d_m	d_r	Security Level
ees401ep1	401	3	2048	113	133	113	113	112
ees541ep1	541	3	2048	49	180	49	49	112
ees659ep1	659	3	2048	38	219	38	38	112
ees449ep1	449	3	2048	134	149	134	134	128
ees613ep1	613	3	2048	55	204	55	55	128
ees761ep1	761	3	2048	42	253	42	42	128
ees653ep1	653	3	2048	194	217	194	134	192
ees887ep1	887	3	2048	81	295	81	81	192
ees1087ep1	1087	3	2048	63	362	63	63	192
ees853ep1	853	3	2048	268	289	268	268	256
ees1171ep1	1171	3	2048	106	390	106	106	256
ees1499ep1	1499	3	2048	79	499	79	79	256

Table 4.4 reports the performance results of NTRUEncrypt cryptosystem with various multiplication algorithms for the different security levels shown in Table 4.3. Encryption operation is performed 1000 times to obtain consistent results. For each encryption process the required elements are randomly chosen.
The experimental results show that the performance of NTRUEncrypt does not directly depend on the polynomial degree. In other words, for the same security level (for example 128-bit security level one can choose one of ees449ep1, ees613ep1, ees761ep1) the choice of domain parameter set does not affect the performance. The timing results are very close to each other for quotient ring with different degrees. Other domain parameters have an important role on the timing. According to the timing results, using fast convolution with sliding window method in NTRUEncrypt improves the performance of the scheme. Recall that in Table 3 we use modulo $p = 49201153$. In NTRUEncrypt the reduction is done modulo $q = 2048$. This is almost free since this reduction equals to taking the least significant 11-bit of the result. This explains why the results in Table 4.2 and Table 4.4 are very close.

Table 4.4: Timing results of NTRUEncrypt for the IEEEp1363.1 parameter set (parameter set/second)

Parameter Set	**Schoolbook Method**	**Fast Conv. with Sliding Window**	**Fast Conv. Method**	**Security Level**
ees401ep1	2,863	0,177	0,274	112
ees541ep1	5,140	0,162	0,217	112
ees659ep1	7,614	0,181	0,196	112
ees449ep1	3,587	0,215	0,373	128
ees613ep1	6,612	0,197	0,253	128
ees761ep1	10,158	0,221	0,246	128
ees653ep1	7,467	0,316	0,584	192
ees887ep1	13,845	0,353	0,506	192
ees1087ep1	20,776	0,418	0,475	192
ees853ep1	12,679	0,648	1,199	256
ees1171ep1	24,149	0,573	0,787	256
ees1499ep1	39,739	0,645	0,794	256

In Figure 4.2 a comparison for NTRUEncrypt encryption process with different multiplication techniques is demonstrated. Recall that in Figure 4.1 fast convolution with sliding window gives better performance. Since the main operation in NTRUEncrypt is the polynomial multiplication, the implementation with this method NTRUEncrypt gives better performance.

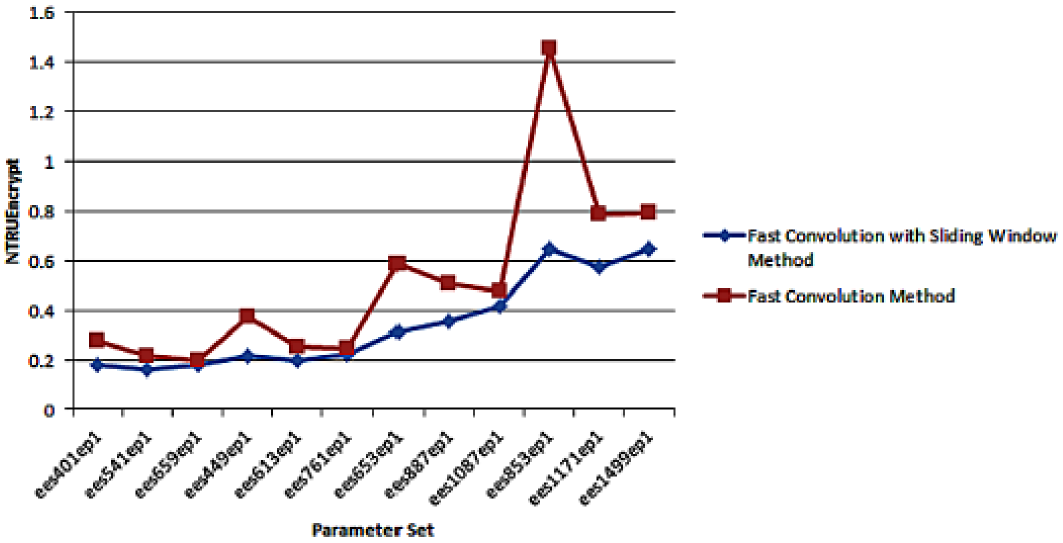


Figure 4.2: Timing results of NTRUEncrypt for the IEEEp1363.1 parameter set (second/parameter set).

4.2.2 Performance Results for NVIDIA Quadro 600 GPU Platform

In this section we give the performance results for the GPU platform using CUDA platform. We use NVIDIA Quadro 600 GPU having 96 cores. In Figure 4.3, polynomial multiplication algorithms are compared in view of the number of parallel multiplications per second on $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ by using the parameter sets given in Table 4.3. To perform the polynomial multiplication the required random data is generated on the GPU with CUDA platform. Since transferring data between CPU and GPU needs more time, we prefer this choice. According to the implementation results, polynomial multiplication in $\mathbb{R}_q$ is accelerated almost 29% by using Algorithm 13. Our design for ees401ep1 parameter set achieves the throughput of 12156 polynomial multiplications per second while it's 9391 in the original one. Note that degree of the polynomial has an important affect on the performance of parallel multiplication.

Table 4.5 summarizes our experimental findings for NTRUEncrypt from n = 401 up to n = 853. The timings for encryption operation are given for 1000 trials and the input of NTRUEncrypt is randomly generated. The data is generated on the CPU and then it's transferred to the GPU.
While implementing NTRUEncrypt, we use a set of parameters for different security level recommended in Table 4.3. We implement fast convolution and its sliding window version as described in previous sections. We also compare the proposed method with the original Montgomery modular multiplication method. According to the experimental results by using modified interleaved Montgomery multiplication method NTRUEncrypt is accelerated almost 35%. However, the proposed method is not the best choice for NTRUEncrypt. Fast convolution with sliding window method gives better performance since multiplication is performed by only additions and the required number of additions is drastically reduced when we compare this with the fast convolution method. Moreover, fast convolution method and its sliding window version have a nice structure for parallelization.

Table 4.5: Experimental results for NTRUEncrypt on the GPU using CUDA platform (second/parameter set)

Parameter Set	**ees401ep1**	**ees449ep1**	**ees653ep1**	**ees853ep1**
Fast Convolution with Sliding Window	0.179	0.233	0.350	0.749
Fast Convolution	0.265	0.384	0.621	1.678
Interleaved Montgomery	0.829	1.401	1.986	3.037
Original Montgomery	1.142	1.768	2.502	3.755

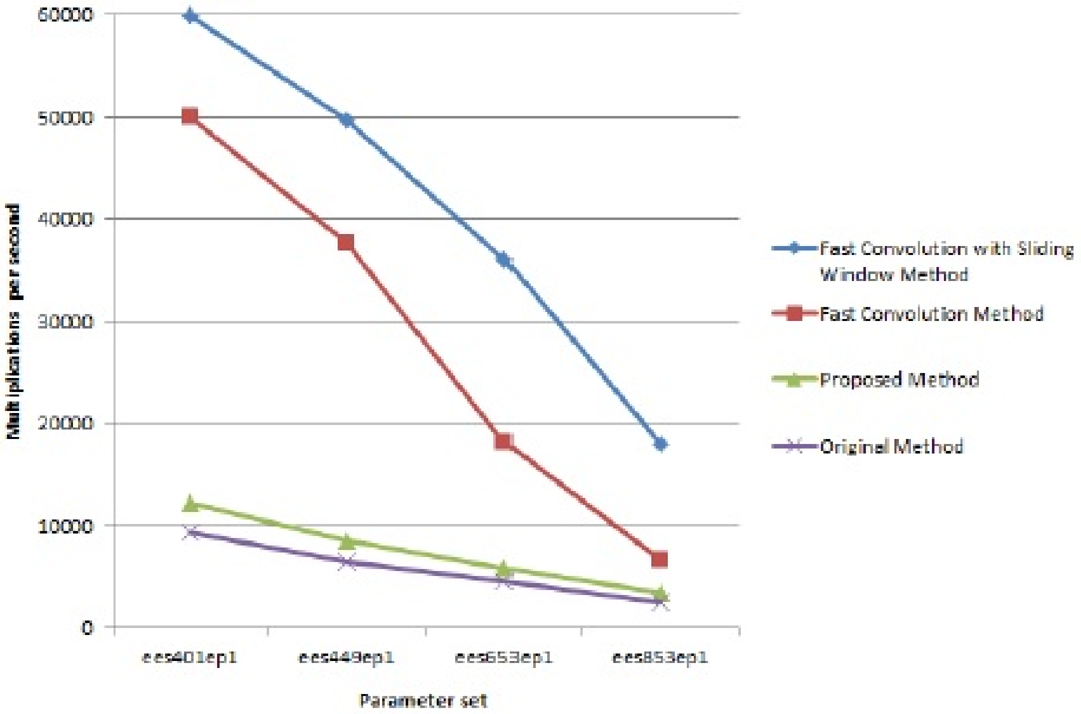


Figure 4.3: Comparison of multiplication over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$ (multiplication per second/parameter set).

4.3 Experimental Results for Multiplication Algorithms over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$

4.3.1 Performance Results for Intel(R) Xeon E3-1230 3.30GHz Platform

Timing results of the multiplication algorithms on GLP given according to the platform defined in Table 4.1.To obtain more consistent results the algorithms are run 1000 times for the uniformly random polynomials in the multiplication operation. In Table 4.1, we list the properties of experiment platform. The performance results of multiplication algorithms to multiply two elements in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ for $n = 1024$, 2048, 4096 and 8192 is given in Table 4.6. The polynomials are generated randomly whose coefficients are $\{-1, 0, 1\}$. Recall that we give the algorithms of schoolbook method, parallelized schoolbook method, iterative NTT, parallelized iterative NTT, interleaved Montgomery and cuFFT-based multiplication method over the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$.

Table 4.6: Timing results of multiplication of two elements in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ (n/second)

	1024	**2048**	**4096**	**8192**
Schoolbook Method	15,148	60,419	242,856	973,655
Iterative NTT	0,860	1,884	3,882	8,383
Parallelized Schoolbook Method	3,254	5,821	21,129	82,627
Parallelized Iterative NTT	6,851	14,343	3,983	118,326
cuFFT-Based Multiplication	1,318	2,240	2,204	3,512

In Figure 4.4 timing comparison of the selected multiplication algorithms to multiply two elements over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ is demonstrated. Note that since schoolbook method has the worst timing results, we omit it in the figure. According to the timing results, for the polynomials of degree $n = 1024$ and 2048, iterative NTT method is the most efficient one. For $n > 2048$ cuFFT-based multiplication has the best timing results since the library is designed to work with large data sets. The parallelization of cuFFT library is very effective when working with large data sets. Note that other parallelized methods such as iterative NTT do not give the expected improvement since the data transfer between CPU and GPU multiple times increases the execution time and affects the efficiency. cuFFT-based multiplication has a better performance since data transfer is performed once. This decreases the latency according to the other methods. Recall that cuFFT is an optimized library for the GPU.

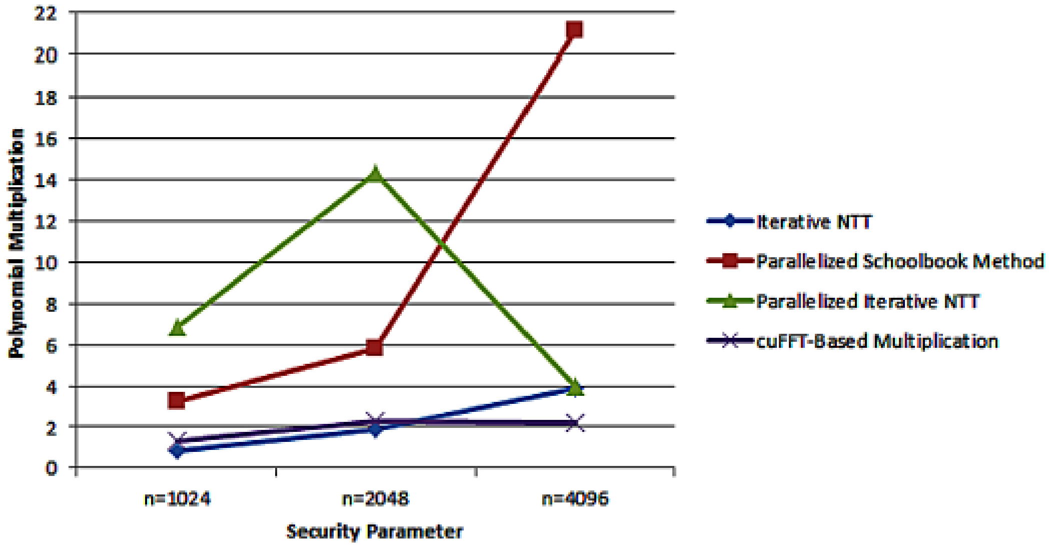


Figure 4.4: Timing comparison of multiplication methods over $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ (second/n).

Experimental Results for Signature Scheme

In this section we give the implementation results of the signature scheme defined over the polynomial $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$. Recall that we choose $p = 49201153$ satisfying $p \equiv 1 \mod 2n$. To obtain more accurate results, we run the signature generation and verification processes 1000 times.
Table 4.7 shows the timing results of signature generation process with various multiplication algorithms. According to the timing results, iterative NTT method has more efficient results than others for $n < 4096$. For $n \geq 4096$ signature generation with cuFFT-based multiplication is the fastest one. In signature generation process the most time consuming part is the polynomial multiplication. Serial implementation of the algorithms gives better results up to polynomial degree 8192.

Table 4.7: Timing results of signature generation (n/second)

n	**1024**	**2048**	**4096**	**8192**
Schoolbook Method	65,82	273,466	1191,049	5949,218
Iterative NTT	4,285	9,263	20,057	55,036
Parallelized Schoolbook Method	12,571	24,564	104,152	482,399
Parallelized Iterative NTT	29,016	65,104	166,621	485,218
cuFFT-Based Multiplication	5,846	9,919	18,258	25,685

In Figure 4.5, a timing comparison of signature generation phase with various multiplication algorithms defined in Multiplication Techniques section. The signature c is composed of 32 1's and -1's. Since the number of the coefficients different than 0 is very small, modified fast convolution method has better results. Recall that the efficiency of modified fast convolution method depends on Hamming weight of the input.

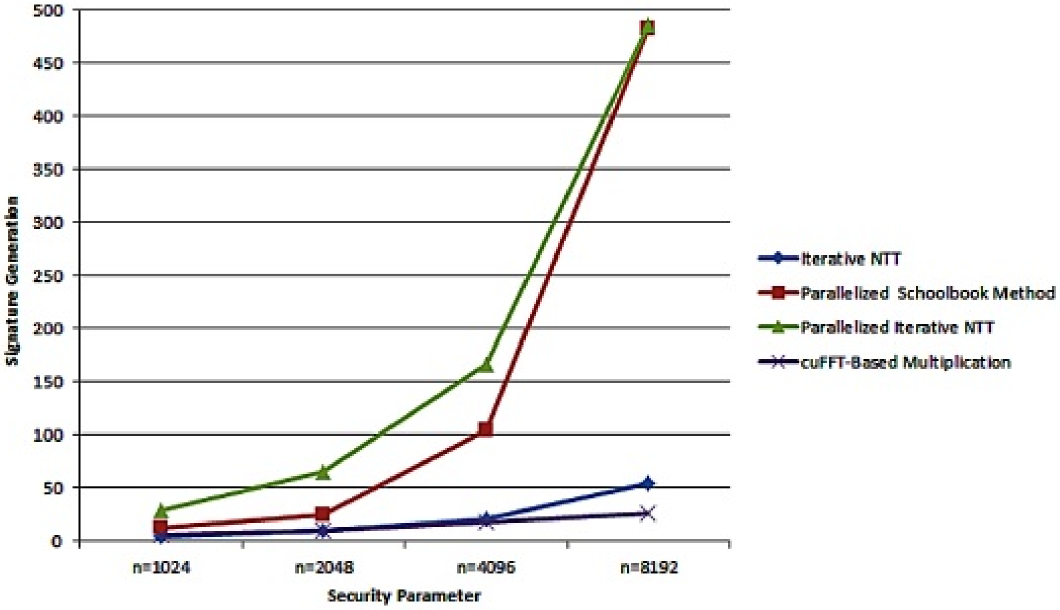


Figure 4.5: Timing comparison of signature generation phase with multiplication algorithms (second/n)

Table 4.8 shows the timing results of signature verification process with various multiplication algorithms. According to the timing results, as in signature generation phase iterative NTT method is more efficient than others for $n < 4096$. For $n \geq 4096$ signature verification with cuFFT-based multiplication is the most efficient one. These results show that parallelized algorithms have better performance for the higher security levels.

Table 4.8: Timing results of signature verification (n/second)

n	**1024**	**2048**	**4096**	**8192**
Schoolbook Method	31,048	126,125	504,425	2000,192
Iterative NTT	1,978	4,245	8,688	18,265
Parallelized Schoolbook Method	5,870	14,413	39,295	168,217
Parallelized Iterative NTT	14,831	28,021	13,381	38,976
cuFFT-Based Multiplication	2,914	4,290	6,539	8,892

In Figure 4.6 a timing comparison of signature generation phase with various multiplication algorithms defined in Multiplication Techniques section is demonstrated.

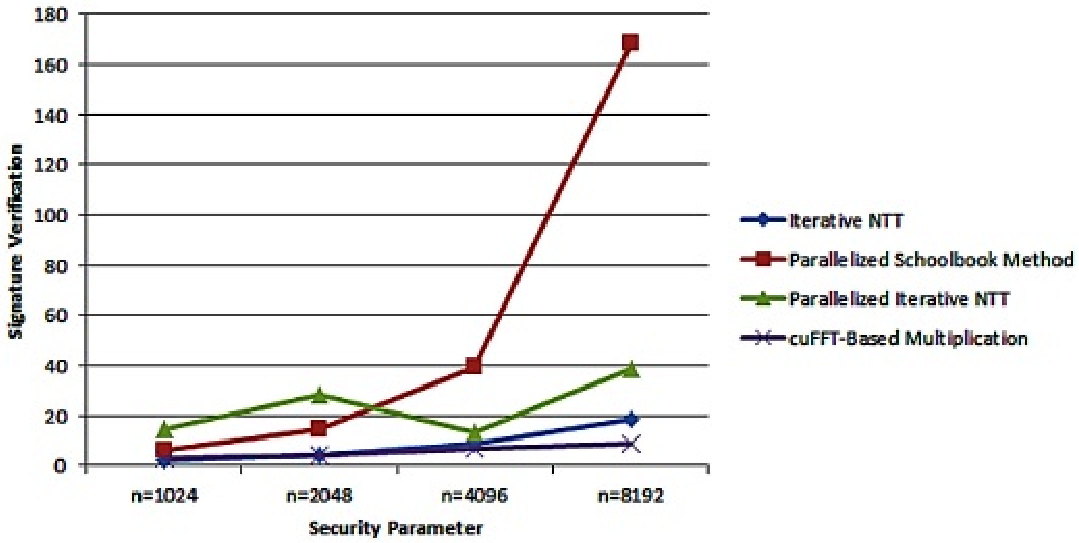


Figure 4.6: Timing comparison of signature verification phase with multiplication algorithms (second/n)

Performance Results for NVIDIA Quadro 600 GPU Platform

In this section we give the performance results for the GPU platform using CUDA platform. We use NVIDIA Quadro 600 GPU having 96 cores. In Table 4.9 the number of multiplications over the $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ for selected methods is given. We choose $p = 49201153$ satisfying $p \equiv 1 \mod 2n$. We generate the random data on the GPU.
According to the experimental results, Since cuFFT is optimized version of FFT on the GPU for the parallel processing, cuFFT-based multiplication gives the best throughput. Algorithm 19 gives better performance than the original one. We also note that the number of parallel multiplication decreases when one compares with Table 4.9. The reason is that taking modulo with $p = 49201153$ results a delay. The comparison results show that by using Algorithm 19 multiplying two elements in $\mathbb{R}_p$ is accelerated at least 19% compared to interleaved Montgomery modular multiplication.

Table 4.9: Experimental results for selected polynomial multiplication methods over the polynomial ring $\mathbb{R}_p$ on the GPU using CUDA platform (second/n)

Algorithms	**n=1024**	**n=2048**	**n=4096**
Parallelized Schoolbook Method	9478	5897	2515
cuFFT-based Multiplication	27650	15308	7691
Original Montgomery	9659	6034	2671
Interleaved Montgomery Method	8074	4965	2153

4.3.2 Performance Results for NVIDIA GeForce GTX 775M Platform

In this section, we give the performance results for the platform Intel Core i5 3.4GHz CPU and NVIDIA GeForce GTX 775M with 768 CUDA cores. We choose cuFFT library to implement NTT since cuFFT is an optimized library for the GPU This helps us to obtain more consistent results. The parallelized parts of the source code are implemented by using CUDA C++ as defined below. In this part, we fix $p = 1061093377$. Note that we use standard modulo reduction algorithm not effecting the overall performance of polynomial multiplication method.

Table 4.10 summarizes the performance results of the selected polynomial multiplication algorithms in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ on the GPU using CUDA platform. The polynomials are randomly generated in such a way that the coefficients of the multiplicand polynomial has 64 nonzero coefficients. Algorithm 20 and Algorithm 22 are implemented on CPU as serial. The proposed multiplication methods can also be parallelized but the operations on these algorithms have low computation density. The data transfer between CPU and GPU gets more time than the actual computation. As a result parallelized versions does not get better performance than serial implementations. According to the experimental results, modified fast convolution method is the most efficient algorithm for random polynomial multiplications (not sparse and not having special patterns).

Table 4.10: Timing results of polynomial multiplication methods

Algorithm	**n=1024**	**n=2048**	**n=4096**
Parallelized Schoolbook Method	4,508	4,786	5,079
cuFFT-based NTT method	0,062	0,131	0,291
Sparse Polynomial Multiplication	0,035	0,072	0,156
Sparse Polynomial Multiplication with Sliding Window	0,037	0,081	0,193

Remark 15. We would like to note that although in the implementation of the proposed methods the data transfer between CPU and GPU multiple times increases the execution time which affects the efficiency and in cuFFT-based NTT polynomial multiplication method data transfer is performed only once, the proposed methods have better performance for the defined data set.

Table 4.11 shows the timing results of signature generation process with various multiplication algorithms for $n = 1024$, 2048 and 4096 in seconds. We would like to note that since the polynomial multiplication in Algorithm 5 Step 2 ($a{\cdot}y1$) does not fit to our proposed methods, we use cuFFT-based NTT in that case. In Step 3, we use the proposed methods. According to the timing results, the performance of sparse polynomial multiplication (and also its sliding window version) is much more better than others. For $n = 4096$ signature generation with Algorithm 22 is the fastest one with a slight difference. Recall that in signature generation process, the signature c is composed of 32 1s and -1s. Since the number of the coefficients different than 0 is very small, sparse polynomial multiplication and its sliding window version have better results. Recall that the efficiency of sparse polynomial multiplication depends on Hamming weight of the input. However, low Hamming weight without repeated patterns causes the inefficiency of the sliding window version. Sparse polynomial multiplication with sliding window has better performance for uniformly random polynomial multiplication whose coefficients are in $\{-1, 0, 1\}$.

Table 4.11: Timing results of signature generation

Algorithm	**n = 1024**	**n = 2048**	**n = 4096**
Parallelized Schoolbook Method	12,571	24,564	104,152
cuFFT-based NTT Method	5,846	9,919	18,258
Sparse Polynomial Multiplication	2,656	5,534	13,297
Sparse Polynomial Multiplication with Sliding Window	2,782	5,845	13,273

Table 4.12 shows the timing results of signature verification process with various multiplication algorithms in seconds. As in signature generation case, since the polynomial multiplication in Algorithm 6 in Step 1 does not fit to our proposed methods, we use cuFFT-based NTT in that case. According to the timing results, as in signature generation phase sparse polynomial multiplication is more efficient than others.

Table 4.12: Timing results of signature verification

Algorithm	**n = 1024**	**n = 2048**	**n = 4096**
Parallelized Schoolbook Method	5,870	14,413	39,295
cuFFT-based NTT Method	2,914	4,290	6,539
Sparse Polynomial Multiplication	1,321	2,665	5,593
Sparse Polynomial Multiplication with Sliding Window	1,369	2,842	5,741

4.3.3 Performance Results for NVIDIA Geforce GT 555M Platform

In this section, we give cycle counts of the multiplication algorithms and signature generation and verification processes. To obtain more consistent results the algorithms are run 1000 times for the uniformly random polynomials in the multiplication operation. To compare the multiplication operation on the GPU using CUDA platform we fix $p = 8383489$ satisfying $p \equiv 1 \mod 2n$. We implement the algorithms using the NVIDIA Geforce GT 555M GPU having 144 CUDA cores on a notebook with the Intel Core i7-2670QM processor and 4GB memory.

In Table 4.13 we list the performance results of different multiplication algorithms which multiply two elements in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ for $n = 2^k$ with $k \in \{8, ..., 13\}$.The polynomials are generated randomly whose coefficients are in the set $\{-1, 0, 1\}$.

Table 4.13: Timing results of multiplication of two elements in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$ (n/second)

Degree n	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Iterative NTT	11475	24524	54407	125332	274217	719816
Parallelized Iterative NTT	16589	32569	72713	231960	689075	1860731
cuFFT-based	13970	26885	61089	134909	245964	450917
Sparse Polynomial Multiplication	14684	29163	75942	167857	331857	799078

According to the timing results, we observe that for the polynomials of degree $n \in \{256, 512, 1024, 2048\}$, the iterative NTT method is the most efficient one. However, for $n > 2048$, there is enough workload such that cuFFT-based multiplication outperforms the other methods since the library is designed to work with large data sets. The parallelization of cuFFT library is very effective when working with large data sets. Note that other parallelized methods, such as iterative NTT, do not give the expected improvement since the data transfer between CPU and GPU happens multiple times and increases the execution time affecting the efficiency negatively. cuFFT-based multiplication has a better performance since data transfer is performed merely once. This decreases the latency versus the other methods. Recall that cuFFT is an optimized library for GPU. The sparse polynomial multiplication method improves in performance with larger n since we prefer to use cuFFT-based multiplication for decomposed polynomials instead of schoolbook multiplication.

Here, we give the timing results of our implementation of the signature scheme proposed in [27]. It is defined over the polynomial ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$. Again, we run the signature generation and verification algorithm 1000 times and list in Table 4.14 our respective timing results under consideration of various multiplication algorithms. In the signature generation process the most time-consuming part is the polynomial multiplication. Hence, we make similar observations as in

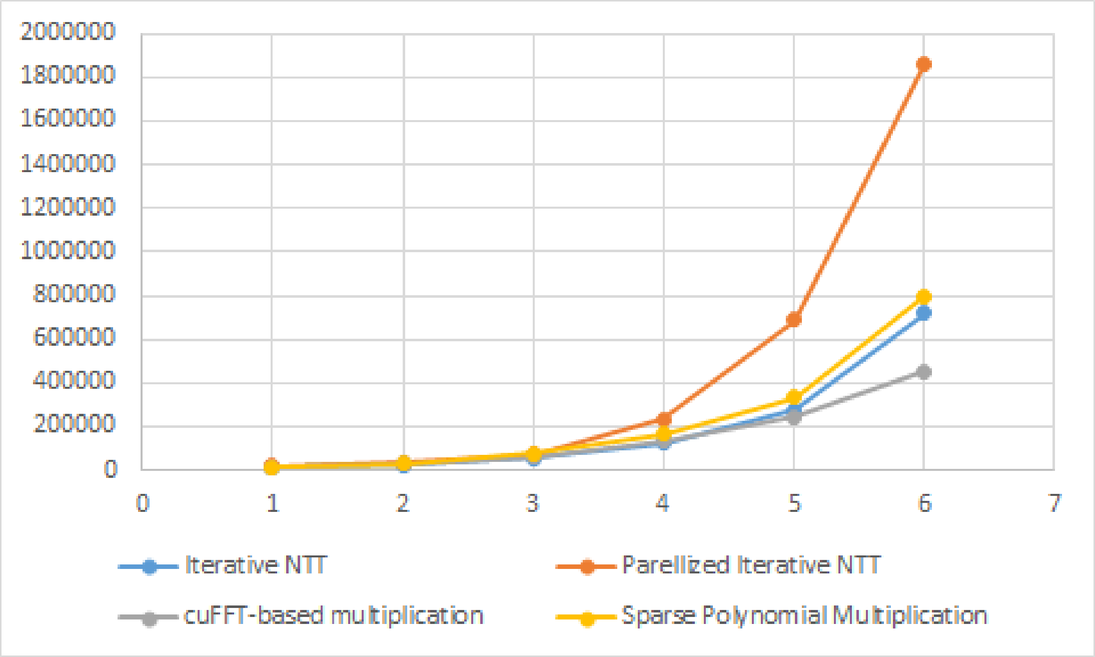


Figure 4.7: Comparison of polynomial multiplication results (cycle counts/parameter set).

the previous section. That is, according to the timing results, the signature generation process with iterative NTT for polynomial multiplication performs best for $n < 4096$ as in Table 4.14. For $n \in \{4096, 8192\}$ the signature generation with cuFFT-based multiplication is most efficient. The sparse polynomial multiplication method is of the same magnitude as the cuFFT-based multiplication and is a preferable choice for higher security levels where the degree n is chosen large. Also Figure 4.8 shows the graphic for signature generation. The timing results of the signature verification process with various multiplication algorithms is given in Table 4.15.

Table 4.14: Cycle counts for the signature generation algorithm with different multiplication methods

Degree n	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Iterative NTT	73866	158703	325530	702773	1534303	3808258
Parallelized Iterative NTT	102804	234901	472159	1850947	4975280	19548904
cuFFT-based Multiplication	92816	206719	359664	726804	1371674	2561705
Sparse Polynomial Multiplication	97085	219485	405977	857904	1964071	5081130

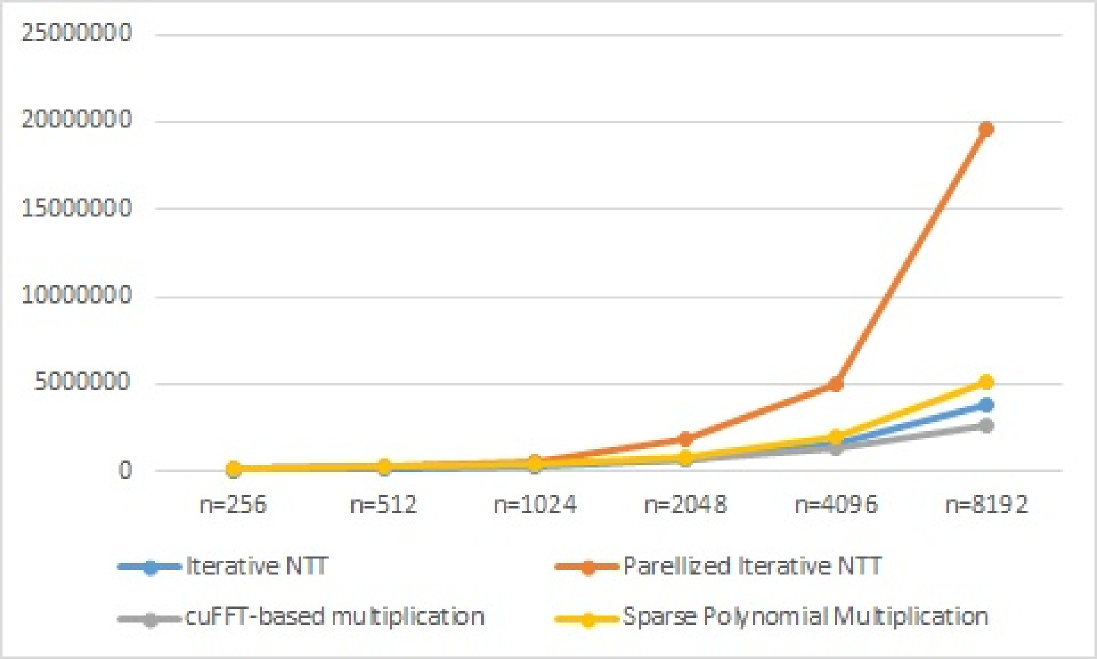


Figure 4.8: Comparison of signature generation results (cycle counts/parameter set).

Table 4.15: Cycle counts for the signature verification algorithm with different multiplication methods

Degree n	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Iterative NTT	40746	82331	171857	366431	815838	1971789
Parallelized Iterative NTT	58015	114089	295402	856910	2142230	1097502
cuFFT-based Multiplication	53618	90526	201976	424451	707590	1239700
Sparse Polynomial Multiplication	56419	99217	218006	480731	824003	2698207

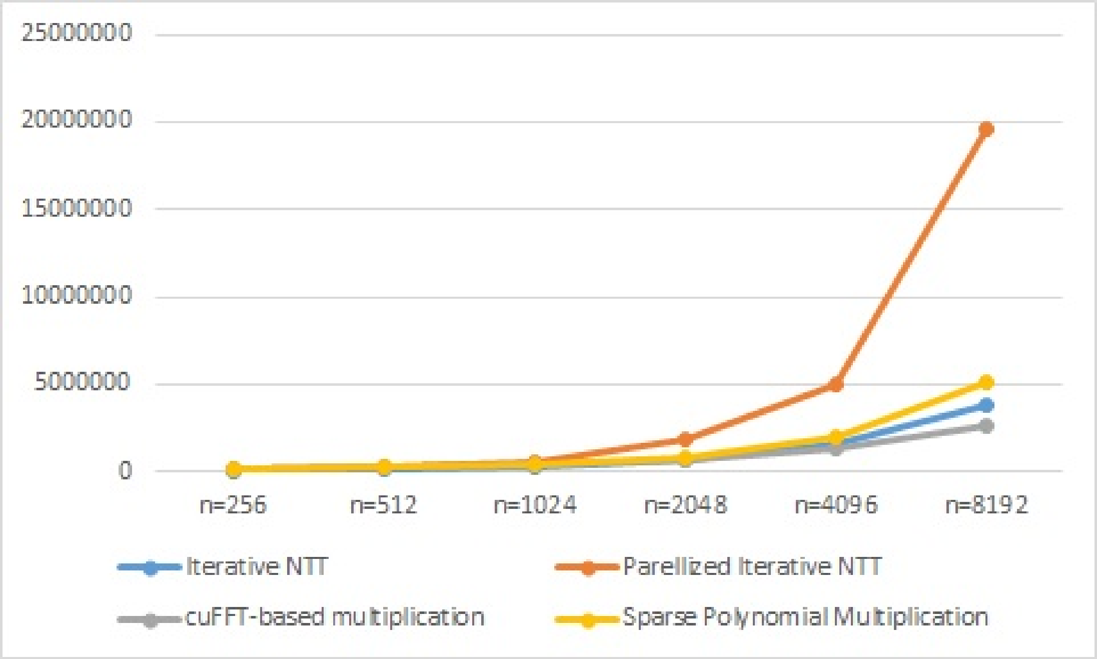


Figure 4.9: Comparison of signature verification results (cycle counts/parameter set).

CHAPTER 5

CONCLUSION

In this thesis, we discuss the computational aspects of lattice-based cryptographic schemes focused on NTRUEncrypt and GLP signature schemes. We give the details of the selected modular multiplication algorithms. We explain the improvements of the multiplication algorithms for CPU and GPU-based implementations. We also modify the polynomial multiplication methods considering the needs of the selected cryptosystems. We show that in some cases they give better performance results than iterative NTT method for the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n+1)$. Some of the multiplication methods running efficiently in CPU platform do not give the expected performance on the GPU.

First, we implement the fast convolution and fast convolution method with sliding window method for the quotient ring $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. The proposed methods work with the data input {0,1}. Since NTRU uses the inputs from the data set {-1,0,1}, we generalize these methods to handle NTRU data sets. According to the experimental results, fast convolution with sliding window method the number of required additions is drastically reduced when we compare this with the convolution method. GPU implementation of these methods are also done, but experiments gives inefficient results due to the low density characteristics of inputs.

Second, we give the required updates for Interleaved Montgomery Modular multiplication method to be used in lattice-based cryptographic schemes for the quotient rings $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ and $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n - 1)$. The major improvement is to reduce the required number of multiplications. Algorithm 13 and Algorithm 19 give better performance than the original one. We give an acceleration of interleaved Montgomery modular multiplication at least 19% on the GPU for lattice-based cryptography.

Finally, we present novel sparse polynomial multiplication methods working efficiently in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ for any positive integers n and p. We enhance the performance of some lattice-based signature schemes significantly by performing polynomial multiplication in $(\mathbb{Z}/p\mathbb{Z})[x]/(x^n + 1)$ with only addition and subtraction operations. Moreover, the proposed methods are easily modified to be used in other cryptographic applications having a sparse polynomial multiplication operation with a slight differences in reduction part since they are independent of the choice of reduction polynomial.According to the experimental results, the proposed methods are approximately 80% faster than NTT. Then,, we investigate how these algorithms perform when run on GPU. We show that some of the multiplication methods which efficiently run in the CPU platform do not give the expected performance on GPU. Data latency between CPU and GPU causes inefficient implementations, in particular, for small data sets since the CUDA platform is designed to work with large data sets. Every kernel call causes latency since kernel initialization is done for each call specific to CUDA platform. We obtain the highest efficiency of cuFFT-based multiplication method for the polynomial degree larger than 4096. We conclude that for an efficient GPU implementation of cryptographic schemes using the CUDA platform, more effort is required to optimize modular arithmetic operations.

REFERENCES

[1] Nvidia cuda toolkit documentation, Retrieved fromhttp://docs.nvidia.com/cuda/eula/index.html.

[2] Security innovation the application security company, Retrieved from https://www.securityinnovation.com/.

[3] S. Akleylek, E. Alkım, and Z. Y. Tok, Sparse polynomial multiplication for lattice based cryptography with small complexity, in *The Journal of Supercomputing*, Springer, 2016.

[4] S. Akleylek, M. Cenk, and F. Özbudak, On the generalisation of special moduli for faster interleaved montgomery modular multiplication, in *IET Information Security*, pp. 165–171, IET, 2013.

[5] S. Akleylek and Z. Y. Tok, Efficient arithmetic for lattice-based cryptography on gpu using the cuda platform, in *Signal Processing and Communications Applications Conference (SIU)*, pp. 854–857, IEEE, 2014.

[6] S. Akleylek and Z. Y. Tok, Efficient interleaved montgomery modular multiplication for lattice-based cryptography, in *Improving Information Security Practices through Computational Intelligence*, pp. 11–22, IEICE Electronics Express, 2014.

[7] S. Akleylek and Z. Y. Tok, Computational aspects of lattice-based cryptography on graphical processing unit, in *Improving Information Security Practices through Computational Intelligence*, pp. 255–284, IGI Global, 2016.

[8] S. Akleylek, Özgür Dağdelen, and Z. Y. Tok, On the efficiency of polynomial multiplication for lattice based cryptography on gpu using cuda, in *Second International Conference BalkanCrypSec*, pp. 155–168, Springer, 2015.

[9] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*, Springer-Verlag, 2011.

[10] D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury, Ntru in constrained devices, in *Cryptographic Hardware and Embedded Systems — CHES 2001*, pp. 262–272, Springer Berlin Heidelberg, 2001.

[11] D. J. Bernstein, J. Buchmann, and E. Dahmen, *Post-Quantum Cryptography*, Springer-Verlag Berlin Heidelberg, 2009.

[12] J. Buchmann, E. Dahmen, and A. Hülsing, Xmss - a practical forward secure signature scheme based on minimal security assumptions, in *In Proceedings of the 4th International Conference On Post-Quantum Cryptography, PQCRYPTO'11*, pp. 117–129, Springer Berlin Heidelberg, 2011.

[13] M. Campagna, Quantum safe cryptography and security, an introduction, benefits, enablers and challenges, in *ETSI White Paper*, ETSI, 2015.

[14] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, Report on post-quantum cryptography, Technical report, IEEE P1363.1 Group, 2016.

[15] M.-S. Chen, B.-Y. Yang, and D. Smith-Tone, Pflash - secure asymmetric signatures on smart cards, in *Lightweight Cryptography Workshop 2015*, pp. 337–350, NIST, 2015.

[16] N. Courtois, M. Finiasz, and N. Sendrier, How to achieve a mceliece-based digital signature scheme, in *Advances in Cryptology — ASIACRYPT 2001*, pp. 157–174, Springer Berlin Heidelberg, 2001.

[17] N. T. Courtois and M. D. Felke, On the security of hfe, hfev- and quartzs, in *PKC 2003*, pp. 337–350, Springer Berlin Heidelberg, 2003.

[18] A. D. K. Debra Cook, Cryptographics: Exploiting graphics cards for security, in *Advances in Information Security*, Springer Verlag, 2006.

[19] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, Lattice signatures and bimodal gaussians, in *Advances in Cryptology – CRYPTO 2013*, pp. 40–56, IACR-CRYPTO-2013, 2013.

[20] Fiat-Shamir and V. Lyubashevsky, Applications to lattice and factoring-based signatures, in *Advances in Cryptology – ASIACRYPT 2009*, pp. 598–616, Springer, 2009.

[21] K. Geissler and N. P. Smart, Lightweight asymmetric cryptography and alternatives to rsa: The ntru public-key cryptosystem, in *ECRYPT European Network of Excellence in Cryptology*, pp. 1219–1234, ACM New York, 2005.

[22] P. Giorgi, T. Izard, and A. Tisserand, Comparison of modular arithmetic algorithms on gpus, in *Proceedings of ParCo'09, Advances in Parallel Computing*, pp. 315–322, Springer Verlag, 2009.

[23] L. K. Grover, A fast quantum mechanical algorithm for database search, in *STOC '96 Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219, ACM, 1996.

[24] T. Guneysu, V. Lyubashevsky, and T. Poppelmann, Enhanced lattice-based signatures on reconfigurable hardware, in *IACR-CHES-2014*, pp. 262–272, IACR, 2015.

[25] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, Practical lattice-based cryptography: A signature scheme for embedded systems, in *CHES 2012 - 14th International Workshop, Leuven, Belgium*, pp. 530–547, Springer, 2012.

[26] T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe, Software speed records for lattice-based signatures, in *5th International Workshop, PQCrypto 2013*, pp. 67–82, Springer-Verlag Berlin Heidelberg, 2013.

[27] T. Güneysu, T. Pöppelmann, and V. Lyubashevsky, Lattice-based signatures: Optimization and implementation on reconfigurable hardware, IEEE Transactions on Computers, 2013.

[28] O. Harrison and J. Waldron, Efficient acceleration of asymmetric cryptography on graphics hardware, in *ECRYPT European Network of Excellence in Cryptology*, pp. 350–367, Springer Berlin Heidelberg, 2009.

[29] P. Hermans, F. Vercauteren, and B. Preneel, Speed records for ntru, in *Topics in Cryptology - CT-RSA 2010*, pp. 73–88, Springer Verlag, 2010.

[30] J. Hoffstein, J. Pipher, and J. H. Silverman, Ntru: A ring-based public key cryptosystem, in *Algorithmic Number Theory*, pp. 267–288, Springer Berlin Heidelberg, 1998.

[31] J. Hoffstein, J. Pipher, and J. H. Silverman, Ntru a public key cryptosystem, IEEE P1363.1 Group, 2007.

[32] J. Hoffstein, J. Pipher, and J. H. Silverman, P1363.1: Standard specifications for public-key cryptographic techniques based on hard problems over lattices, in *CHES 2012 - 14th International Workshop, Leuven, Belgium*, IEEE, 2008.

[33] J. Hoffstein, N. H.-G. J. Pipher, J. H. Silverman, and W. Whyte, Ntrusign digital signatures using the ntru lattice, Algorithmic Number Theory, pp. 122–140, 2003.

[34] J. Ioannidis, A. Keromytis, and M. Yung, Rainbow, a new multivariable polynomial signature scheme, in *EUROCRYPT 1999*, pp. 164–175, Springer Berlin Heidelberg, 2005.

[35] A. Kipnis, J. Patarin, and L. Goubin, Unbalanced oil and vinegar signature schemes, in *EUROCRYPT 1999*, pp. 206–222, Springer Berlin Heidelberg, 1999.

[36] M.-K. Lee, J. W. Kim, J. E. Song, and K. Park, Sliding window method for ntru, in *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007*, pp. 432–442, Springer-Verlag, 2007.

[37] M.-K. Lee, J. W. Kim, J. E. Song, and K. Park, Efficient implementation of ntru cryptosystem using sliding window methods, in *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, pp. 206–214, IEICE, 2013.

[38] V. Lyubashevsky, Lattice signatures without trapdoors, pp. 738–755, 2012.

[39] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, Swifft: A modest proposal for fft hashing, in *Fast Software Encryption, 15th International Workshop, FSE 2008*, pp. 54–72, Springer Berlin Heidelberg, 2008.

[40] A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan, On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption, in *STOC 2012*, pp. 1219–1234, ACM New York, 2012.

[41] T. Matsumoto and H. Imai, Public quadratic polynomial-tuples for efficient signature-verification and message-encryption, in *Advances in Cryptology — EUROCRYPT '88*, pp. 419–453, Springer-Verlag Berlin, 1988.

[42] R. J. McEliece, A public-key cryptosystem based on algebraic coding theory, Technical report, DSN Progress Report, 1978.

[43] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[44] R. C. Merkle, A digital signature based on a conventional encryption function, in *Advances in Cryptology — CRYPTO '87*, p. 369, Springer-Verlag Berlin, 1988.

[45] R. Misoczki and P. S. Barreto, Compact mceliece keys from goppa codes, in *Selected Areas in Cryptography*, pp. 376–392, Springer-Verlag Berlin, 2009.

[46] L. Mun-Kyu, K. J. Woo, S. J. Eu, and P. Kunsoo, Sliding window method for ntru, pp. 432–442, Springer-Verlag, 2007.

[47] L. Mun-Kyu, K. J. Woo, S. J. Eu, and P. Kunsoo, Efficient implementation of ntru cryptosystem using sliding window methods, pp. 206–213, IEICE Trans. Fundamentals, 2013.

[48] S. Neves and F. Araujo, On the performance of gpu public-key cryptography, in *2011 IEEE International Conference*, pp. 350–367, IEEE, 2011.

[49] C. O'Rourke and B. Sunar, Achieving ntru with montgomery multiplication, in *In CHES 2008 LNCS*, IEEE Transections On Computers, 2003.

[50] J. Patarin, Hidden fields equations (hfe) and isomorphisms of polynomials (ip): two new families of asymmetric algorithms, in *Eurocrypt'96*, pp. 33–48, Springer Berlin Heidelberg, 1996.

[51] J. M. Pollard, The fast fourier transform in a finite field, pp. 365–374, Mathematics of Computation, 1971.

[52] T. Pöppelmann and T. Güneysu, Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware, in *Progress in Cryptology – LATINCRYPT 2012*, pp. 139–158, Springer Verlag, 2012.

[53] T. G. Robert Szerwinski, Cryptographics: Exploiting graphics cards for security, in *Advances in Information Security*, pp. 79–99, Springer Verlag, 2008.

[54] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, pp. 1484–1509, SIAM Journal on Computing, 1997.

[55] R. Steinfeld, Ntru cryptosystem: Recent developments and emerging mathematical problems in finite polynomial rings., in *Algebraic Curves and Finite Fields: Cryptography and Other Applications*, pp. 1–33, De Gruyter, 2014.

[56] R. Szerwinski and T. Guneysu, Exploiting the power of gpus for asymmetric cryptography, in *In CHES 2008 LNCS*, pp. 79–99, Springer-Verlag, 2008.

[57] C. Tao, A. Diene, S. Tang, and J. Ding, Simple matrix scheme for encryption, in *PQCrypto 2013*, pp. 231–242, Springer Berlin Heidelberg, 2013.

[58] N. Wilt, *CUDA Handbook: A Comprehensive Guide to GPU Programmingy*, Addison-Wesley, 2011.

[59] F. Winkler, *Polynomial Algorithms in Computer Algebra*, Springer-Verlag Wien, 1996.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Yüce Tok, Zaliha
Nationality: Turkish (TC)
Date and Place of Birth: 19.01.1982, Uşak
Marital Status: Married
Phone: 0 543 9606516

EDUCATION

Degree	**Institution**	**Year of Graduation**
M.S.	METU IAM Cryptography Department	2007
B.S.	METU Computer Engineering Department	2004
High School	Uşak Anatolian High School	1999

PROFESSIONAL EXPERIENCE

Year	**Place**	**Enrollment**
03.2013 - ...	Mikes-Aselsan	Specialist Software Engineer
10.2008-02.2011	STM A.S.	Specialist Software Engineer
09.2006-10.2008	METU IAM	Specialist Software Engineer
01.2006-09.2006	Milsoft	Software Engineer
06.2004-01.2006	Aydın Yazılım	Software Engineer

PUBLICATIONS

Journals

[1] Akleylek, S., Alkım, E., Tok , Z.Y.,(2016) Sparse Polynomial Multiplication for Lattice Based Cryptography with Small Complexity, The Journal of Supercomputing DOI:10.1007/s11227-015-1570-1

[2]Akleylek, S. and Tok, Z.Y., (2014). Efficient Interleaved Montgomery Modular

Multiplication for Lattice-Based Cryptography. IEICE Electronics Express, 11-22, 1-6. DOI: .10.1587/elex.11.20140960

Chapters In a Book

[3] Akleylek, S., Tok, Z.Y., (2016)Computational Aspects of Lattice-Based Cryptography on Graphical Processing Unit. In: El Sayed El-Alfy et. al (Eds.), Improving Information Security Practices through Computational Intelligence, pp.255-284, IGI Global

International Conference Publications

[4] Akleylek, S., Dağdelen, Ö., Tok, Z.Y.,(2015) On the Efficiency of Polynomial Multiplication for Lattice Based Cryptography on GPU Using CUDA In the Preceedings of Second International Conference BalkanCrypSec (pp.155-168) DOI: 10.1007978-3-319-29172-7_10

[5] Akleylek S., Kırlar B.B., Sever Ö., Tok Z.Y., "A New Short Signature Scheme with Random Oracle from Bilinear Pairings", Journal of Telecommunications and Information Technology, 2011

[6] Akleylek S., Kırlar B.B., Sever Ö., Tok Z.Y., "Short Signature Scheme from Bilinear Pairings", NATO Information Assurance and Cyber Defense (IST-091), Antalya, Turkey, 2010.

[7] Akleylek S., Kırlar B.B., Sever Ö., Tok Z.Y.,, "Short Signature Scheme from Bilinear Pairings", Proceedings of Western European Workshop on Research in Cryptology (WEWoRC 2009), pp. Graz, Austria.

[8] Akleylek S., Kırlar B.B., Sever Ö., Tok Z.Y., "Arithmetic on Pairing-Friendly Fields", Proceedings of 3th International Information Security and Cryptography Conference (ISCTURKEY 2008), pp. 115-120, Ankara, Turkey

[9] Akleylek S., Kırlar B.B., Sever Ö., Tok Z.Y., Pairing-Based Cryptography: A Survey", Proceedings of 3th International Information Security and Cryptography Conference, (ISCTURKEY 2008), pp. 121-125, Ankara, Turkey.

National Conference Publications

[10] Akleylek, S., Tok, Z. Y. (2014). Efficient Arithmetic for Lattice-Based Cryptography on GPU Using the CUDA Platform. In Proceedings of the IEEE 22nd Signal Processing and Communications Applications Conference (pp.854-857). New York, NY: IEEE. doi:10.1109/SIU.2014.6830364

[11]Sedat Akleylek Hamdi Murat Yıldırım, Zaliha Yüce Tok Kriptoloji ve Uygu-

lama Alanları: Açık Anahtar Altyapısı ve Kayıtlı Elektronik Posta Akademik Bilişim'11 - XIII. Akademik Bilişim Konferansı Bildirileri , Malatya

Awards, Fellowships, and Grants

Best Research Paper Award at New York University Abu Dhabi Cyber Security Awereness Week (CSAW) 2016 in MENA Region (Published Paper in 2015-2016)