HIERARCHICAL NTT ARCHITECTURES ON FPGA

by
EMRE KOÇER

Submitted to the Graduate School of Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
December 2024

ABSTRACT

HIEARARCHICAL NTT ARCHITECTURES ON FPGA

EMRE KOÇER

COMPUTER SCIENCE & ENGINEERING, M.S. THESIS, DECEMBER 2024

Thesis Supervisor: Prof. Erkay Savaş

Keywords: Homomorphic Encryption, Hardware Acceleration, FPGA, Polynomial Multiplication, Hierarchical NTT, Modular Multiplication

Fully Homomorphic Encryption (FHE) represents a paradigm shift in cryptography, enabling computations directly on encrypted data while preserving privacy. However, the computational complexity inherent in FHE operations, particularly polynomial multiplications, presents a significant barrier to practical adoption. This thesis focuses on addressing these challenges by developing high-performance, hardware-accelerated architectures for FHE, with a particular emphasis on the optimization of negacyclic Number Theoretic Transform (NTT) operations.

The key contribution of this work is the introduction of a high speed, throughput-optimized, hierarchical 7-step NTT algorithm tailored for FPGA implementation. This approach builds on the negacyclic NTT method to enhance computational efficiency by leveraging hierarchical partitioning of polynomial operations. Compared to conventional 4-step NTT designs, the proposed architecture achieves superior scalability and resource efficiency, supporting polynomial sizes ranging from 2^{10} to 2^{16}. By utilizing a modular and pipelined hardware architecture, the design overcomes memory access bottlenecks, achieving significant improvement in NTT throughput compared to state-of-the-art software implementations.

The hierarchical design employs advanced techniques, including column independence and unrolled NTT stages, to maximize throughput while minimizing latency. Additionally, the modular reduction operations utilize Word-Level Montgomery (WLM) methods, enabling compatibility with varying security levels and FHE parameters. The results highlight the practicality of negacyclic NTT-based hi-

erarchical designs in high-throughput cryptographic applications, facilitating secure data processing in domains such as privacy-preserving machine learning, encrypted database queries, and secure cloud computing.

Our proposed design achieves significant performance improvements over state-of-the-art implementations for NTT operations, with up to 8.14× speed-up in average latency and 4.01× better Area-Time-Product (ATP) for high-performance FPGAs. Compared to leading solutions, our seven-step architecture demonstrates superior throughput and scalability across various ring dimensions ($\boldsymbol{n} = \boldsymbol{2}^{10}$ to $\boldsymbol{n} = \boldsymbol{2}^{16}$), offering up to 7.96× lower latency while maintaining resource efficiency. Furthermore, our design supports runtime-configurable parameters, making it a practical solution.

ÖZET

FPGA ÜZERINDE HIYERARŞIK STD YAPILARI

EMRE KOÇER

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, Aralık 2024

Tez Danışmanı: Prof. Dr. Erkay Savaş

Anahtar Kelimeler: Homomorfik Şifreleme, Hızlandırıcı Donanım, Polinom Çarpması, Hiyerarşik STD, Modüler Çarpma

Tam Homomorfik Şifreleme (Fully Homomorphic Encryption - FHE), kriptografi alanında bir paradigma değişimini temsil ederek, verilerin gizliliğini koruyarak şifreli veriler üzerinde doğrudan işlem yapılmasını mümkün kılmaktadır. Ancak, özellikle polinom çarpımları olmak üzere FHE işlemlerinde yer alan yüksek hesaplama karmaşıklığı, pratik kullanım için önemli bir engel teşkil etmektedir. Bu tez, FHE'nin bu zorluklarını ele almayı amaçlayarak, yüksek performanslı ve donanım hızlandırmalı mimariler geliştirmeye odaklanmaktadır. Özellikle, negatif döngüsel Sayı Teorisi Dönüşümü (negacyclic Number Theoretic Transform - NTT) işlemlerinin optimize edilmesi üzerine yoğunlaşılmıştır.

Bu çalışmanın önemli katkılarından biri, programlanabilir donanım aracı olan FPGA üzerinde uygulanmak üzere özelleştirilmiş, verimlilik odaklı, hiyerarşik bir 7-adımlı NTT algoritmasının tanıtılmasıdır. Bu yaklaşım, polinom işlemlerinin hiyerarşik olarak bölünmesinden yararlanarak, negatif döngüsel NTT prensiplerini temel almış ve hesaplama verimliliğini artırmıştır. 4-adımlı NTT tasarımlarıyla karşılaştırıldığında, önerilen mimari daha üstün ölçeklenebilirlik ve kaynak verimliliği sunmakta ve 2^{10} ile 2^{16} arasında değişen polinom boyutlarını desteklemektedir. Modüler ve ardışık işleme dayalı donanım mimarisi sayesinde, bellek erişim darboğazları aşılmış ve bilimsel literatürdeki en iyi uygulamalara kıyasla NTT verimliliğinde önemli bir iyileşme sağlanmıştır.

Hiyerarşik tasarım, sütun bağımsızlığı ve açılmış (unrolled) NTT döngüleri gibi gelişmiş teknikler kullanarak verimliliği en üst düzeye çıkarırken gecikmeyi en aza in-

dirmektedir. Ayrıca, modüler indirgeme işlemleri, farklı güvenlik seviyeleri ve FHE parametreleriyle uyumluluğu mümkün kılan Kelime Temelli Montgomery (Word-Level Montgomery - WLM) yöntemlerini kullanmaktadır. Elde edilen sonuçlar, negatif döngüsel NTT tabanlı hiyerarşik tasarımların yüksek verimli şifreleme uygulamalarının pratikliğini artırdığını ve bu tasarımların gizliliği koruyan makine öğrenimi, şifreli veri tabanı sorguları ve güvenli bulut bilişim gibi alanlarda güvenli veri işleme uygulamalarının yapılabilirliğini kolaylaştırdığını göstermektedir.

Önerilen tasarımımız, NTT işlemleri için bilimsel literatürdeki en iyi uygulamalara kıyasla önemli performans iyileştirmeleri sağlamaktadır; yüksek performanslı FPGA'larda ortalama gecikmede 8.14× hızlanma ve Alan-Zaman Ürünü (ATP) metriklerinde 4.01× iyileşme sunmaktadır. Bilimsel literatürdeki önde gelen çözümlerle karşılaştırıldığında, yedi adımlı mimarimiz, çeşitli halka boyutlarında ($\boldsymbol{n} = \mathbf{2}^{10}$ ile $\boldsymbol{n} = \mathbf{2}^{16}$ arasında) üstün işlem hacmi ve ölçeklenebilirlik göstererek, kaynak verimliliğini korurken 7.96× daha düşük gecikme sağlamaktadır. Ayrıca, tasarımımız çalışma zamanında yapılandırılabilir parametreleri destekleyerek pratik bir çözüm sunmaktadır.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Prof. Dr. Erkay Savaş, for his unwavering support and mentorship throughout my master's thesis journey. His expertise, insightful feedback, and thoughtful guidance have been instrumental in shaping the direction of my research. His patience and encouragement, especially during challenging moments, have motivated me to push beyond my limits and achieve my academic goals. I am truly fortunate to have had the opportunity to work under his supervision.

I would also like to sincerely thank my thesis jury members, Prof. Dr. Sıddıka Berna Örs Yalçın and Prof. Dr. Özcan Öztürk, for their time, valuable suggestions, and constructive criticism. Their expertise and detailed feedback have contributed significantly to the improvement of my work, and I truly appreciate their thorough evaluation.

A special thanks to Dr. Erdinc Ozturk for his guidance and assistance throughout my research journey. His advice and mentorship have been truly invaluable.

I want to extend my thanks to the CISEC group, especially Can Ayduman, Selim Kırbıyık, and Tolun Tosun, for their collaboration and support throughout my thesis. Their efforts and teamwork have enriched my research experience.

I want to extend my heartfelt gratitude to my friends, whose joy and wisdom have guided me throughout my journey. Their support and positive energy have illuminated my path and inspired me to embrace my future confidently. Their presence has enriched every moment of my master's journey, and I am deeply thankful for the joy they brought into my life.

Lastly, I would like to express my heartfelt thanks to my family, my father and my mother, for their love, encouragement, and thoughtfulness. Their constant support has been a source of strength during this journey.

This thesis is partially supported by the European Union's Horizon Europe research and innovation program under grant agreement No: 101079319 and by TUBITAK under Grant Number 122E222.

To my beloved mother...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

cc clock cycle 28, 30, 31, 32, 33

4NU 4-step NTT Unit 29

AU Automorphism Unit 29, 30, 33

BFV Brakerski / Fan-Vercauteren 4

BGV Brakerski-Gentry-Vaikuntanathan 4

BRAM block RAM 29, 32, 33, 34

BU Butterfly Unit 28, 32, 33

CKKS Cheon-Kim-Kim-Song 4

FHE Fully Homomorphic Encryption 4

HBM High-Bandwidth Memory 28

INU Merged NTT Unit 29, 30, 33, 34

LWE Learning-with Errors 2, 8

MMU Modular Multiplication Unit 32

NTT Number Theoretic Transform 16, 17, 29

PPFL Privacy-preserving Federated Learning 4

PPML Privacy-preserving Machine Learning 4

PQC Post-Quantum Cryptography 2

RLWE Ring Learning-with Errors 8, 9

TFHE Torus Fully Homomorphic Encryption 4

TGU Twiddle Generation Unit 29

TMU Twiddle Multiplication Unit 29, 30, 32, 33

1. INTRODUCTION

1.1 Classical Cryptosystems

Secure communication represents a critical challenge in today's interconnected world. The majority of internet communication occurs over unsecured channels, which pose significant risks of exposing sensitive user information. Consequently, the establishment of secure communication channels accessible to all participants in a network has become imperative. Such channels are predominantly constructed using classical encryption systems, such as the Rivest-Shamir-Adleman (RSA) algorithm (Rivest, Shamir & Adleman (1978)) and Elliptic Curve Cryptography (ECC) (Miller (1985)). These cryptographic schemes are grounded in mathematically intractable problems: RSA relies on the difficulty of integer factorization, while ECC is based on the Discrete Logarithm Problem (DLP). However, the emergence of quantum computing poses a significant threat to classical cryptosystems, as quantum algorithms, such as Shor's algorithm (Shor (1997)), can efficiently solve the underlying hard problems of many widely used cryptographic systems (e.g., integer factorization in RSA). In response, new cryptographic paradigms, known as Post-Quantum Cryptography (PQC), have been developed to resist quantum attacks. These paradigms ensure the secure exchange of cryptographic keys over untrusted communication channels by utilizing quantum-resistant public key cryptosystems.

1.2 Lattice-Based Cryptography

Lattice-based cryptography is a post-quantum cryptographic paradigm that relies on the mathematical structure of lattices which are discrete, periodic point arrangements in multi-dimensional space. Unlike traditional cryptographic schemes based on number-theoretic problems (e.g., RSA or ECC), lattice-based cryptosystems derive their security from hard computational problems such as the Shortest Vector Problem (SVP) (Micciancio (2005)) and the Learning-with Errors (LWE) (Regev (2024)) problem, which remain difficult to solve even with the advent of quantum computers. These problems are deeply rooted in high-dimensional geometry, where finding specific vectors or solving approximate equations becomes infeasible due to exponential complexity as dimensions increase. Lattice-based cryptography is not only resistant to quantum attacks but also highly versatile, supporting advanced cryptographic primitives such as fully homomorphic encryption (FHE), which allows computations on encrypted data without decryption and digital signatures. Furthermore, lattice-based schemes are efficient, with simple arithmetic operations and favorable performance for key generation, encryption, and decryption. This combination of quantum resistance, flexibility, and efficiency makes lattice-based cryptography a promising candidate for securing future communication systems and data against both classical and quantum adversaries.

1.2.1 Post-Quantum Cryptography

In the seminal work by Shor (Shor (1997)), he proposed an algorithm known as Shor's Algorithm, which demonstrated that cryptographic schemes such as RSA and discrete logarithm-based systems could be efficiently broken using quantum algorithms on classical computers. Consequently, developing quantum-secure encryption schemes resistant to such attacks became essential to create secure communication channels for future systems. These quantum-secure encryption schemes are so-called post-quantum secure systems that are resistant to quantum attacks.

Unlike traditional encryption schemes, Post-Quantum Cryptography (PQC) relies on the hardness of the Learning with Errors (LWE) (Regev (2024)) problem. This problem plays a critical role in both post-quantum cryptographic systems and homomorphic encryption, as it introduces an additional layer of error ensures that

quantum algorithms cannot easily invert the resulting ciphertext. The details of this RLWE problem are explained in Section 2.2.

1.3 Homomorphic Encryption

Homomorphic Encryption is a powerful cryptographic technique that enables computations to be performed directly on encrypted data without the need to decrypt it first. This property ensures that the privacy of sensitive data is maintained throughout the computation process, which is particularly useful in scenarios where data privacy is crucial, such as in cloud computing or secure data analysis. By allowing computations to occur on encrypted data, homomorphic encryption eliminates the need to expose sensitive information to potentially untrusted parties during the computation process.

The operations allowed in homomorphic encryption are typically classified into two categories: linear and nonlinear operations.

Linear operations include basic arithmetic operations such as addition and subtraction. These operations can be directly applied to encrypted data, maintaining their behavior as if the data were in plaintext form. For example, given two encrypted values, $\text{Enc}(a)$ and $\text{Enc}(b)$, the homomorphic encryption scheme enables the computation of $\text{Enc}(a+b)$, $\text{Enc}(a-b)$, and $\text{Enc}(a \cdot b)$ without the need to decrypt the values a and b. This allows for secure arithmetic operations on encrypted data, which is a fundamental property of many homomorphic encryption schemes.

Nonlinear operations, in contrast, are more complex and involve operations such as multiplication, shifting, and certain types of logical operations. For instance, shifting an encrypted value by a specified number of bits or performing mathematical operations like exponentiation. These operations typically require more advanced cryptographic techniques to support them efficiently. Moreover, they may introduce additional challenges, such as noise growth, which could eventually compromise the security of the ciphertext. To manage this noise, techniques like bootstrapping (Gentry, Halevi & Smart (2011)) are employed to reduce the noise and maintain the correctness of the encrypted data over multiple operations.

Homomorphic encryption schemes can be classified into two main types: Partially

Homomorphic Encryption (PHE) and Fully Homomorphic Encryption (FHE). PHE schemes support only specific types of operations, such as addition or multiplication, on encrypted data. For instance, Paillier's scheme (Paillier (1999)) is homomorphic with respect to the addition operation, meaning it preserves information during addition. Similarly, the ElGamal (Elgamal (1985)) cryptosystem provides a homomorphic scheme that supports multiplication, making it another example of a partially homomorphic encryption scheme, but this time under multiplication.

FHE, on the other hand, allows for both linear and nonlinear operations to be performed on encrypted data. This capability enables arbitrary computations to be carried out on encrypted data, making FHE a critical tool for privacy-preserving applications. By supporting both types of operations, FHE provides a powerful advantage: it allows computations to be performed on encrypted data without the need to decrypt it at any point in the process. This ensures that sensitive information remains protected throughout all stages of the operation, even during data transmission. The ability to perform arbitrary computations on encrypted data also opens up the possibility of secure outsourced computation on untrusted platforms. With FHE, computations can be offloaded to cloud servers or other external platforms without revealing sensitive data to the service provider, thereby ensuring the confidentiality of the data throughout the computation process.

Homomorphic encryption is a valuable tool for performing machine learning (ML) algorithms on encrypted data (Lee, Kang, Lee, Choi, Eom, Deryabin, Lee, Lee, Yoo, Kim & No (2021)), ensuring the privacy of both the data and the computation process. For instance, Privacy-Preserving Federated Learning (PPFL) and Privacy-Preserving Machine Learning (PPML) algorithms can be implemented using FHE, enabling collaborative learning without exposing sensitive data. Given its ability to securely process encrypted data, FHE is especially beneficial for applications involving highly sensitive information, such as healthcare and financial systems, where protecting user privacy is extremely important.

There are different homomorphic encryption schemes available in the literature, e.g. BFV (Brakerski (2012); Fan & Vercauteren (2012)), BGV (Brakerski, Gentry & Vaikuntanathan (2011)), CKKS (Cheon, Kim, Kim & Song (2016)) and TFHE (Chillotti, Gama, Georgieva & Izabachène (2018)). Most of these schemes are based on RLWE (Lyubashevsky, Peikert & Regev (2010)) problem. The core operation of the RLWE problem is polynomial multiplication because it is the most time-consuming operation. Time complexity for normal schoolbook multiplication is $\mathcal{O}(n^2)$ whereas for Number Theoretic based polynomial multiplication is $\mathcal{O}(n\log(n))$. Therefore, using NTT-based multiplication rather than schoolbook multiplication

for FHE schemes is more suitable.

It is generally infeasible to develop a solution that works for different parameter sets in FHE schemes. Therefore, it is usually preferred to work on specific parameters instead of generalized solutions. Most of the papers are based on ring size parametrization with specific prime moduli widths (Chen, Lu, Su & Chen (2024); Guo & Li (2023); Li, Ren, Du, Tu, Wang, Yin & Ouyang (2022); Nguyen, Kieu-Do-Nguyen, Pham & Hoang (2024); Ye, Yang, Kuppannagari, Kannan & Prasanna (2021)). These papers consider primes in special forms for reducing resource utilization. However, we support any prime that enables NTT computation without being restricted to special types of primes.

In Fully Homomorphic Encryption (FHE) applications, two critical performance metrics are *latency* and *throughput*. While both metrics play a significant role in evaluating the efficiency of FHE systems, throughput emerges as the more pivotal consideration due to the inherently data-intensive nature of FHE algorithms. FHE involves performing a huge number of computationally expensive homomorphic operations on encrypted data.

Latency, defined as the time required to complete a single operation or computation, becomes secondary in contexts where large volumes of data are processed. Conversely, throughput measures the total number of operations completed per unit of time and directly influences the practical usability of FHE in real-world scenarios, such as secure cloud computing, privacy-preserving machine learning, and encrypted database queries. The prioritization of throughput-based designs over latency-based ones stems from their suitability for parallelizable and batched workloads, which are typical in FHE applications. Hardware architectures and algorithmic optimizations targeting throughput can better leverage resources like multicore processors, FPGAs, or GPUs to execute a higher number of operations concurrently. For example, pipelined and parallel processing approaches maximize resource utilization and ensure a continuous flow of computations, which is crucial for maintaining high throughput.

Consequently, adopting throughput-centric strategies enhances the feasibility and scalability of FHE implementations, especially in enterprise-level or cloud-based deployments where high volumes of encrypted data need to be processed efficiently. This observation aligns with recent studies, which highlight that throughput-optimized hardware designs significantly outperform those focusing solely on minimizing latency (Roy, Turan, Jarvinen, Vercauteren & Verbauwhede (2019); Tan, Chiu, Wang, Lao & Parhi (2023)).

Various hardware architectures have been examined to accelerate Fully Homomorphic Encryption (FHE) applications. Most of these methods are based on GPUs and FPGAs. We focused on an FPGA-based approach, which is better suited for throughput. By employing numerous processing elements in parallel on the FPGA and developing dedicated hardware for specific blocks of the 7-step NTT algorithms, we presented highly efficient hardware implementations for negacyclic NTT algorithms.

In the literature, the majority of works are based on iterative NTT techniques. Bailey (Bailey (1989)) developed a different NTT architecture that is based on hierarchical structures. These algorithms have garnered attention due to their capacity to lead to highly parallelizable NTT architectures. Hierarchical NTT algorithms treat the input as a matrix and compute smaller NTTs both on the rows and columns of that matrix. Despite this approach leading to an increased total computational effort required for a single NTT operation, it is well-suited for throughput-focused FPGA implementations.

Recently, there has been a surge of interest among researchers in hierarchical NTT algorithms. Some papers in the literature call this hierarchical method a "4-Step" NTT algorithm, and there are several implementations for this 4-Step approach (Chen et al. (2024); Guan, Zhu, Huang, Lei, Wang, Jia, Chen, Zhang, Dong & Bian (2024); Liu, Tang, Song, Zhou, Yan & Yang (2024); Mert, Öztürk & Savaş (2020)). These architectures are mostly based on low resource utilization techniques by computing several smaller NTTs in parallel. However, these techniques are not suitable for high throughput approaches because these designs are not fully-pipelined implementations (except (Liu et al. (2024))), and they provide specific designs to fixed design parameters. The reason is that creating a fully-pipelined design with a 4-step approach is very hard to accomplish because resource utilization is very high for computing NTTs with a large size. The details of this discussion can be found in Section 3.1.

1.4 Our Contributions

We can outline our contributions in this thesis as follows:

- We devise a 4-step algorithm to function directly on the negacyclic ring, avoiding the pre-processing and post-processing stages required by existing tech-

niques. We prove that our method can be universally applied to hierarchical NTT strategies with any type of dimensional breakdown of the input polynomial.

- This thesis presents, to the best of our knowledge, the very first mathematically defined negacyclic 4-step and 7-step NTT algorithms. Negacyclic NTTs are essential for FHE applications as they reduce the input size for computation by half. Previous attempts at developing these algorithms were not algorithmically correct. However, we describe a mathematically accurate and implementable negacyclic NTT algorithm. Bailey's work (Bailey (1989)) paved the way for various approaches to the NTT algorithm, whereas previous papers primarily focused on using 4-step and iterative-NTT methods. The introduction of the 7-step NTT algorithm proved instrumental in enabling the deployment of hierarchical memory.

- We designed and implemented configurable negacyclic NTT hardware accelerators based on a hierarchical algorithm optimized for high-speed FPGA devices. Our implementation focuses on Alveo U280 and Virtex VC709 FPGAs, which have three distinct memory blocks: Block RAMs (BRAMs), DDR RAM, and High Bandwidth Memory (HBM) (only for Alveo U280). BRAMs are on-chip memories where we store twiddle factors and intermediate results, and HBMs and DDRs are off-chip memories where we store input and output results. The primary bottleneck in FPGA lies in HBM-FPGA communication, and this work aimed to overcome this challenge. We present a throughput-based approach where we select our coefficient throughput as the maximum IO bandwidth between HBM and FPGA.

- The proposed design is a parametric 7-step NTT algorithm for FPGA implementation that works with various parameters settings with five parameters, namely n_{11}, n_{12}, n_{21}, n_{22} and throughput (TP). Note that this TP represents how many coefficients we are consuming in each clock cycle. Every design changes with respect to those five parameters, and our parametric hardware generator generates the necessary modules to support those new settings.

2. BACKGROUND

2.1 Notation

- Lowercase italic letters, like a, represent integers. The function $\Gamma_l(\cdot)$ is the bit-reversing function for l-bit integers. The logarithm function log is base-2.
- Bold uppercase letters, such as $\mathbf{A}$, are used to represent matrices. Bold lowercase letters, like $\mathbf{a}$, denote vectors. Elements of matrices or vectors are accessed using square brackets, for example, $\mathbf{A}[i][j]$ for matrices and $\mathbf{a}[i]$ for vectors. The symbol $\odot$ represents element-wise multiplication for vectors or matrices, such as $\mathbf{a} \odot \mathbf{b}$.
- The symbol $\mathcal{R}_{q,n}$ refers to the cyclotomic polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$. Polynomials are represented using lowercase italic letters, for instance, $\boldsymbol{a}(x)$. The coefficients of a polynomial are represented with subscripts, like $\boldsymbol{a}_i$.

2.2 RLWE Problem

We focus on a specific variant of the LWE problem, namely the Ring Learning with Errors (RLWE) (Lyubashevsky, Peikert & Regev (2012)) problem. The RLWE problem is based on the computational hardness of distinguishing between two types of polynomial samples with embedded noise. The encryption and decryption procedures for this scheme are described in 2.2. Specifically:

- $a(x)$ is a uniformly random polynomial sampled from the ring $R_q = \mathbb{Z}[x]/(f(x))$

mod q, where $f(x)$ is a monic irreducible polynomial (typically $f(x) = x^n + 1$ for n a power of 2).

- $s(x)$ represents the secret polynomial chosen from a discrete Gaussian or uniform distribution over R_q.
- $e(x)$ is a small random error polynomial, typically sampled from a discrete Gaussian distribution, ensuring the noise is bounded to maintain decryption correctness.

This formulation leverages the algebraic structure of polynomial rings to provide both efficiency and security, making RLWE a cornerstone of modern post-quantum cryptographic schemes.

$$c(x) = \Big(a(x), b(x)\Big), \tag{2.1}$$

$$b(x) = a(x) \cdot s(x) + e(x), \tag{2.2}$$

In 2.2, retrieving the secret polynomial $s(x)$ from the ciphertext component $b(x)$ is computationally infeasible due to the presence of the error polynomial $e(x)$. The error polynomial introduces a level of noise that obfuscates the relationship between $a(x)$, $s(x)$, and $b(x)$, making it extremely difficult to recover the secret even if an attacker has access to the ciphertext. This difficulty comes from the fact that the error term $e(x)$ is carefully chosen to be small, but it distorts the linear relationship between $a(x)$ and $b(x)$, making it resistant to both classical and quantum attacks.

The introduction of the error polynomial also contributes to the security of the scheme in the context of quantum computing. Unlike traditional cryptographic schemes that rely on the hardness of problems such as factoring or discrete logarithms, which can be solved efficiently by quantum algorithms like Shor's algorithm, the RLWE problem is based on the hardness of lattice problems. Specifically, the problem is closely related to the Shortest Vector Problem (SVP) in lattices, which remains hard even for quantum computers. This makes RLWE-based encryption schemes quantum-secure, meaning that they resist attacks by quantum computers, thus providing a high level of security in the post-quantum era.

2.3 Convolutions

Positive-wrapped Convolution (PWC) and Negative-wrapped Convolution (NWC) operate on distinct polynomial rings, namely $\mathbb{Z}_q[x]/(x^n-1)$ and $\mathbb{Z}_q[x]/(x^n+1)$, respectively. Among these, the ring $\mathbb{Z}_q[x]/(x^n+1)$ is generally preferred due to its inherent advantages in terms of symmetry, computational efficiency, reduced vulnerabilities, and robustness. However, performing NWC using cyclic Number-Theoretic Transform (NTT) routines requires additional computational steps, (Pöppelmann & Güneysu (2012)).

The process involves a pre-processing step where the input vectors are scaled as follows:

$$\bar{\boldsymbol{a}}_i = \boldsymbol{a}_i \cdot \psi^i, \quad \bar{\boldsymbol{b}}_i = \boldsymbol{b}_i \cdot \psi^i, \quad \text{for } 0 \leq i < n.$$

Subsequently, the transformed vectors $\bar{\boldsymbol{a}}$ and $\bar{\boldsymbol{b}}$ are multiplied using cyclic NTT routines with a root of unity $\omega \equiv \psi^2 \mod q$, and the result is subjected to the inverse NTT to obtain $\bar{\boldsymbol{c}}$. Finally, post-processing is performed to retrieve the original coefficients:

$$\boldsymbol{c}_i = \bar{\boldsymbol{c}}_i \cdot \psi^{-i}, \quad \text{for } 0 \leq i < n.$$

This procedure enables efficient computation of NWC while maintaining the properties of the underlying polynomial ring.

2.4 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is widely regarded as a state-of-the-art method for performing polynomial multiplication. Compared to traditional schoolbook multiplication, which has a time complexity of $O(n^2)$, the NTT algorithm achieves a significantly better time complexity of $O(n \log n)$. Given two polynomials $\boldsymbol{a}(x), \boldsymbol{b}(x) \in \mathcal{R}_{q,n}$, their multiplication using the NTT algorithm is performed as follows:

$$\boldsymbol{a}(x) \cdot \boldsymbol{b}(x) = \mathrm{INTT}_n\Big(\mathrm{NTT}_n\big(\boldsymbol{a}(x)\big) \odot \mathrm{NTT}_n\big(\boldsymbol{b}(x)\big)\Big), \tag{2.3}$$

where $\odot$ denotes element-wise multiplication in the NTT domain, and NTT_n and INTT_n represent the forward and inverse NTT operations, respectively.

Polynomial multiplication in the ring $\mathcal{R}_{q,n}$ is also referred to as the *negative-wrapped convolution*, while the corresponding transform in this ring is known as the negacyclic NTT. For the negacyclic NTT to function correctly, the modulus q must satisfy the condition $q \equiv 1 \pmod{2n}$. Under this condition, a primitive $2n$-th root of unity, denoted by ψ, exists in $\mathbb{Z}_q$, where $\psi^n \equiv -1 \pmod{q}$.

Accordingly, the forward NTT computes the transformation $\hat{\mathbf{a}} = \mathrm{NTT}_n(\boldsymbol{a}(x))$, where each entry is evaluated as $\hat{\mathbf{a}}[i] = \boldsymbol{a}(\psi^{2i+1})$. The resulting multiplication in the NTT domain is performed element-wise, as shown in 2.3. The inverse NTT, INTT_n, reverses this process to retrieve the polynomial coefficients. Both the forward and inverse NTT are efficiently implemented using butterfly circuits.

Butterfly circuits are categorized into two primary types. The *Cooley-Tukey* (CT) butterflies, introduced in (Cooley & Tukey (1965)), are widely used for forward NTT_n computations. On the other hand, *Gentleman-Sande* (GS) butterflies, proposed in (Gentleman & Sande (1966)), are often employed for inverse INTT_n computations. These butterflies are defined as follows:

For two coefficients $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$, the CT butterfly is defined as:

$$\mathbf{a}'_i = \boldsymbol{a}_i + \boldsymbol{a}_j \cdot \zeta, \quad \mathbf{a}'_j = \boldsymbol{a}_i - \boldsymbol{a}_j \cdot \zeta, \tag{2.4}$$

while the GS butterfly is defined as:

$$\mathbf{a}'_i = \boldsymbol{a}_i + \boldsymbol{a}_j, \quad \mathbf{a}'_j = (\boldsymbol{a}_i - \boldsymbol{a}_j) \cdot \zeta, \tag{2.5}$$

where ζ is referred to as the *twiddle factor*, which is a power of the primitive root ψ.

Algorithm 1 Negacyclic Iterative NTT, It-NTT$_n(\cdot)$

Require: $\boldsymbol{a}(x) \in \mathcal{R}_{q,n}$ in natural order
Require: a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$
Ensure: $\mathbf{b} \in \mathbb{Z}_q^n$ in bit-reversed order

$\mathbf{b} \leftarrow \boldsymbol{a}(x)$
for t from 0 to $\log_2 n - 1$ **do**
  $m \leftarrow 2^t$
  for i from 0 to $m-1$ **do**
    $z_1 \leftarrow 2 \cdot i \cdot \frac{n}{2m}$
    $z_2 \leftarrow z_1 + \frac{n}{2m}$
    for z from z_1 to $z_2 - 1$ **do** ▷ CT butterflies
      $U \leftarrow \mathbf{b}[z]$, $V \leftarrow \mathbf{b}[z+t]$
      $\mathbf{b}[z] \leftarrow (U + V \cdot \psi^{\Gamma_l(m+i)}) \mod q$
      $\mathbf{b}[z+t] \leftarrow (U - V \cdot \psi^{\Gamma_l(m+i)}) \mod q$
    end for
  end for
end for
return b

Algorithm 2 Negacyclic Iterative Inverse NTT, It-INTT$_n(\cdot)$

Require: $\mathbf{a} \in \mathbb{Z}_q^n$
Require: a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$, $n_1 \cdot n_2 = n$
Ensure: $\boldsymbol{b}(x) \in \mathcal{R}_{q,n}$ where $b(x) = \text{INTT}_n(\mathbf{a})$

$\mathbf{b} \leftarrow \boldsymbol{a}(x)$
$m \leftarrow N, t \leftarrow 1$
while $m > 1$ **do**
  $j_1 \leftarrow 0$
  $h = m/2$
  for i from 0 to $h-1$ **do**
    $j_2 = j_1 + t - 1$
    for j from j_1 to j_2 **do** ▷ GS butterflies
      $U \leftarrow \mathbf{b}[j]$, $V \leftarrow \mathbf{b}[j+t]$
      $\mathbf{b}[j] \leftarrow (U + V) \mod q$
      $\mathbf{b}[j+t] \leftarrow (U - V) \times \psi^{-\Gamma_l(h+i)} \mod q$
    end for
  end for
  $t \leftarrow 2 \times t$, $m \leftarrow m/2$
end while
return $\mathbf{b} \leftarrow \mathbf{b} \times n^{-1} \mod q$ ▷ all $\mathbf{b}[i]$ are multiplied with n^{-1}

The iterative form of the NTT algorithm employs either CT or GS butterflies (Mert,

Öztürk & Savaş (2019)). During each stage of the algorithm, $n/2$ butterflies are computed, and there are $\log n$ stages in total. As a result, the NTT achieves a time complexity of $O(n \log n)$, in contrast to the $O(n^2)$ complexity of traditional schoolbook multiplication. This efficiency has established the NTT as a foundational algorithm for polynomial arithmetic in modern cryptographic and computational applications. Algorithms for CT-based forward NTT algorithm can be seen in 1 and GS-based inverse NTT algorithm can be seen in 2.

2.5 Residue Number System (RNS)

The Residue Number System (RNS) is a modular arithmetic framework that enables efficient representation and computation of large integers. A large integer X can be represented in RNS by decomposing it into a set of residues with respect to pairwise coprime moduli $q_0, q_1, \ldots, q_{\lambda-1}$. Formally, the representation is defined as $X \xrightarrow{\text{RNS}} \{X_0, X_1, \ldots, X_{\lambda-1}\}$, where each residue X_i is computed as $X \mod q_i$ for $i = 0, 1, \ldots, \lambda - 1$. This modular decomposition reduces the computational complexity of arithmetic operations by breaking them into smaller independent modular computations.

The process of decomposition in RNS maps computations over a large modulus Q, where $Q = \prod_{i=0}^{\lambda-1} q_i$, into smaller modular computations over the individual moduli q_i. Mathematically, this decomposition is expressed as:

$$\mathcal{R}_{Q,n} \simeq \prod_{i=0}^{\lambda-1} \mathcal{R}_{q_i,n}, \quad \text{where } \mathcal{R}_{q_i,n} = \mathbb{Z}_{q_i}[x]/(x^n + 1).$$

This property allows polynomial computations over Q to be distributed into parallel computations over smaller moduli rings.

The reconstruction of the original number X from its residues $\{X_0, X_1, \ldots, X_{\lambda-1}\}$ is achieved through the Chinese Remainder Theorem (CRT). The CRT provides a method to recombine the smaller residues into a single result modulo Q. The reconstruction formula is given as:

$$X = \sum_{i=0}^{\lambda-1} (X_i \cdot m_i \cdot M_i \mod q_i) \mod Q,$$

where $m_i = \frac{Q}{q_i}$ represents the partial modulus, and $M_i \equiv m_i^{-1} \mod q_i$ is the modular

multiplicative inverse of $m_i \mod q_i$. This process ensures that X is uniquely defined modulo Q.

In the context of Fully Homomorphic Encryption (FHE), the RNS representation is critical because it can optimize computational efficiency. By enabling parallel computations on residues, RNS significantly reduces the computational overhead of arithmetic operations. Moreover, it minimizes carry propagation, a common bottleneck in modular arithmetic, thereby improving overall performance. This efficiency is particularly advantageous when handling large polynomial degrees and moduli, as required in FHE schemes.

Additionally, the decomposition of large computations into smaller modular operations allows RNS to support scalability in FHE implementations. This scalability facilitates the use of higher security parameters and more complex encrypted computations without incurring prohibitive computational costs. Furthermore, the modular decomposition and reconstruction mechanisms naturally align with batching techniques, which allow multiple ciphertexts to be packed and processed simultaneously.

In summary, RNS enables efficient modular arithmetic by transforming large computations into smaller, independent modular operations and subsequently recombining the results using CRT. This property makes RNS a cornerstone of efficient FHE implementations, supporting high performance, scalability, and practicality in cryptographic computations.

3. HIERARCHICAL NTT ALGORITHMS

3.1 Hierarchical NTT Algorithms

In the 1980s, Bailey introduced an alternative approach to the iterative NTT, based on hierarchical memory (Bailey (1989)). As previously discussed, Field-Programmable Gate Arrays (FPGAs) offer various types of memory, and it is crucial to design the system effectively in order to achieve maximum throughput. When dealing with larger ring sizes, ranging from 2^{12} to 2^{16}, the memory access pattern becomes more complex compared to smaller ring sizes due to the stride size of the input to the Butterfly Unit (BU)s. To illustrate this, consider the access pattern for the coefficients in the first stage, as shown in 1: $\boldsymbol{a}_0, \boldsymbol{a}_4, \boldsymbol{a}_1, \boldsymbol{a}_5, \boldsymbol{a}_2, \boldsymbol{a}_6, \boldsymbol{a}_3, \boldsymbol{a}_7$. In contrast, for $n = 4$, the access pattern in the first stage is $\boldsymbol{a}_0, \boldsymbol{a}_2, \boldsymbol{a}_1, \boldsymbol{a}_3$, which represents an easier case. Generally, an n-point NTT requires strides of length $n/2$. If the stride size could be reduced across the stages during the NTT transformation, throughput would increase. Bailey's algorithm addresses this issue by operating on a matrix rather than a polynomial vector, which introduces independent column operations. By working with matrices instead of vectors, the algorithm facilitates operations on independent data, thereby optimizing the memory access pattern and improving overall performance.

3.1.1 Four-step NTT

Although many studies in the literature have utilized Bailey's method (Chen et al. (2024); Guan et al. (2024); Liu et al. (2024); Mert et al. (2020)), most of them concentrate on the 4-Step NTT algorithm. This algorithm is named after the four

steps it consists of, which are described below:

- n_2 NTT_{n_1}
- Twiddle Multiplication
- Matrix Transpose
- n_1 NTT_{n_2}

where the ring size of the input polynomial $n = n_1 \cdot n_2$. The existing literature on 4-Step algorithms focused on the cyclic NTT case, where the reduction polynomial is $x^n - 1$.

Algorithm 3 Negacyclic 4-Step NTT, $\text{4S-NTT}_n^{n_1}(\cdot)$

Require: $\boldsymbol{a}(x) \in \mathcal{R}_{q,n}$
Require: a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$, $n_1 \cdot n_2 = n$
Ensure: $\mathbf{b} \in \mathbb{Z}_q^n$ where $\mathbf{b} = \text{NTT}_n(\boldsymbol{a}(x))$

1: $\mathbf{A} \in \mathbb{Z}_q^{n_1 \times n_2} \leftarrow a$ ▷ represent $\boldsymbol{a}$ as a matrix s.t. $\mathbf{A}[i][j] = \boldsymbol{a}_{i \cdot n_2 + j}$
2: **for** $i = 0 \rightarrow n_2 - 1$ **do**
3: $\quad \mathbf{A}^T[i] = \text{It-NTT}_{n_1}(\mathbf{A}^T[i])$ ▷ NTT_{n_1} on columns of A using Algorithm 1 and ψ^{n_2}
4: **end for**
5: **for** $i = 0 \rightarrow n_1 - 1$ **do**
6: $\quad$ **for** $j = 0 \rightarrow n_2 - 1$ **do**
7: $\quad\quad t \leftarrow 2n - \Big(n_1 - 1 - 2 \cdot \Gamma_{n_1}(i)\Big) \cdot j \mod 2n$
8: $\quad\quad \mathbf{A}[i][j] = \mathbf{A}[i][j] \cdot \psi^t$
9: $\quad$ **end for**
10: **end for**
11: **for** $i = 1 \rightarrow n_1 - 1$ **do**
12: $\quad \mathbf{A}[i] = \text{It-NTT}_{n_2}(\mathbf{A}[i])$ ▷ NTT_{n_2} on rows of $\mathbf{A}$ using Algorithm 1 and ψ^{n_1}
13: **end for**
14: $\mathbf{b} \leftarrow \mathbf{A}$ ▷ flatten $\mathbf{A}$ s.t. $\mathbf{b}[i \cdot n_2 + j] = \mathbf{A}[i][j]$
15: **return b**

Naturally, allocating more resources regarding time/area is needed to apply this solution. To overcome this overhead, we present a variant of 4-Step algorithm in 3, that is dedicated to negacyclic NTT. The trick in our solution is the formulation of the powers of the twiddles (Line 7) for the multiplication between two sets of NTTs. We should note that one sets $t \leftarrow i \cdot j$ in the cyclic version of 4-Step NTT.

In this method, the first step involves performing n_2 instances of n_1-point NTTs on the columns of the matrix $\mathbf{a}(x)$. This operation differs from a standard negacyclic n_1-point NTT, which typically computes $\log(n_1)$ stages of the overall NTT process. Notably, these operations can be executed in parallel, as all n_1-point NTTs are independent of each other. Following the n_1-point NTTs, an intermediate mul-

tiplication (line 8) is performed to combine the intermediate results, ensuring the correctness of the negacyclic NTT outputs. To achieve this, a specific formulation (line 7) is developed and applied to compute the correct powers of twiddle factors required for the intermediate multiplication. Finally, n_1 n_2-point NTTs are carried out to compute the final result. Retrieving the correct order from the matrix $\mathbf{A}$ is a straightforward operation and can be implemented with minimal complexity.

The main advantage of this algorithm is that it is well suited for small ring sizes. There exist 4-Step NTT solutions in the literature that allows up to $n = 2^{16}$ (Chen et al. (2024); Guan et al. (2024); Liu et al. (2024)). On the other hand, for extreme ring sizes such as ($2^{18} \leq n \leq 2^{24}$), re-using some modules from the 4-Step hardware for multiple steps of the algorithm is necessary, which would avoid developing a pipelined design.Since we can't provide fully pipelined architecture, it will be useful if we can use same hardware again. Therefore, fully pipelined architecture will be useless for larger ring sizes.

As previously mentioned, we focus on the ring sizes in the range $2^{10} \leq n \leq 2^{16}$ but with a different perspective. We implemented various different hardware solutions using a 7-Step algorithm instead of a 4-Step approach. 7-Step creates a better solution for higher ring sizes because it provides smaller NTT partitions than other hierarchical designs. Working on smaller partitions allows us to have an unrolled NTT approach for this implementation, so we provide high throughput solutions for larger ring sizes.

3.1.2 7-Step NTT

The 7-Step algorithm is another hierarchical NTT algorithm. Compared to the 4-Step, it employs a deeper divide-and-conquer strategy. The polynomial is represented in a higher-dimensional space rather than a 2-dimensional matrix. The algorithm's name is derived from the following seven steps, which are detailed below:

- n_{12} $\text{NTT}_{n_{11}}$
- Twiddle Multiplication + Matrix Transpose
- n_{11} $\text{NTT}_{n_{12}}$
- Twiddle Multiplication + Matrix Transpose

- n_{22} $\text{NTT}_{n_{21}}$
- Twiddle Multiplication + Matrix Transpose
- n_{21} $\text{NTT}_{n_{22}}$

where $n = n_{11} \cdot n_{12} \cdot n_{21} \cdot n_{22}$. Indeed, the 7-Step NTT algorithm can be viewed as a recursive application of the 4-Step NTT. 4 presents the details of the 7-Step NTT algorithm in a recursive manner dedicated to the negacyclic case. Observe that, the iterative NTT calls from 3 are replaced with 4-Step NTT calls in 4.

Algorithm 4 Negacyclic 7-Step NTT

Require: $\boldsymbol{a}(x) \in \mathcal{R}_{q,n}$
Require: a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$, $n_{11} \cdot n_{12} \cdot n_{21} \cdot n_{22} = n$
Ensure: $\mathbf{b} \in \mathbb{Z}_q^n$ where $\mathbf{b} = \text{NTT}_n(\boldsymbol{a}(x))$
1: $n_1 \leftarrow n_{11} \cdot n_{12}$ $\quad$ $n_2 \leftarrow n_{21} \cdot n_{22}$
2: $\mathbf{A} \in \mathbb{Z}_q^{n_1 \times n_2} \leftarrow \boldsymbol{a}$ ▷ represent a as a matrix s.t. $\mathbf{A}[i][j] = \boldsymbol{a}_{i \cdot n_2 + j}$
3: **for** $i = 0 \rightarrow n_2 - 1$ **do**
4: $\quad \mathbf{A}^T[i] = \text{4S-NTT}_{n_1}^{n_{11}}(\mathbf{A}^T[i])$ ▷ NTT_{n_1} on columns of A using Algorithm 3 and ψ^{n_2}
5: **end for**
6: **for** $i = 0 \rightarrow n_1 - 1$ **do**
7: $\quad$ **for** $j = 0 \rightarrow n_2 - 1$ **do**
8: $\quad\quad t \leftarrow 2n - \Big(n_1 - 1 - 2 \cdot \Gamma_{n_1}(i)\Big) \cdot j \mod 2n$
9: $\quad\quad \mathbf{A}[i][j] = \mathbf{A}[i][j] \cdot \psi^t$
10: $\quad$ **end for**
11: **end for**
12: **for** $i = 1 \rightarrow n_1 - 1$ **do**
13: $\quad \mathbf{A}[i] = \text{4S-NTT}_{n_2}^{n_{21}}(\mathbf{A}[i])$ ▷ NTT_{n_2} on rows of $\mathbf{A}$ using Algorithm 3 and ψ^{n_1}
14: **end for**
15: $\mathbf{b} \leftarrow \mathbf{A}$ ▷ flatten $\mathbf{A}$ s.t. $\mathbf{b}[i \cdot n_2 + j] = \mathbf{A}[i][j]$
16: **return b**

In a nutshell, Algorithm 4 provides a more hieararchical approach than 4-Step NTT algorithm considering better partitioning. There are two 4-Step NTT operations (line 4 and line 13) and one additional intermediate multiplication (line 8). This approach provides better pipelining performance than 4-Step NTT because it provides even higher independence in between coefficients.

The 7-Step NTT has different characteristics compared to the 4-Step NTT considering practicality. It allows better optimization for leveraging the throughput. Using 4-Step NTT architectures for throughput-optimized systems is impractical even with modern FPGAs because of lacking resources. Our design purpose is to produce **TP** coefficients at each clock cycle (cc), the number of BUs must be carefully configured to achieve this goal. The 7-Step approach addresses the **TP** constraint because it

further partitions the computations compared to the 4-Step. As an intuition, consider the following example for $n = 2^{16}$. For the 4-Step NTT, a natural parameter set is $\mathbf{TP} = n_1 = 2^8 = n_2 = 2^8$. For implementing a fully pipelined system in this setting, we would need $2^7 \cdot 8 \cdot 2$ BUs for implementing $N = 2^8$ NTT blocks, and we would need 2^8 multiplier modules for twiddle multiplication. In total, we would need $2^7 \cdot 8 \cdot 2 + 2^8 = 2304$ multipliers. On the other hand, for the 7-Step NTT, the following partitioning can be performed, $n_{11} = 2^4$, $n_{12} = 2^4$, $n_{21} = 2^4$ and $n_{22} = 2^4$. Also, three twiddle multiplication units are needed, each of them containing 2^4 multipliers. For the four iterative NTT modules in a pipelined manner, $2^3 \cdot 4 \cdot 4$ BUs are needed. In total, we need $2^3 \cdot 4 \cdot 4 + 2^4 \cdot 3 = 176$ multipliers. As the example highlights, the further partitioning of the input, as achieved with the 7-Step NTT, decreases the resource allocation significantly. However, there is a penalty in terms of the **TP** . Note that in the example, the $\mathbf{TP} = 256$ for 4-Step, while it is 16 for 7-Step NTT. As explained earlier in this work, the communication of the FPGA with the host system has certain limitations. For example, modern High-Bandwidth Memory (HBM)s provide 1024-bit throughput. Then, the NTT engine can consume at most 2^7 coefficients, assuming each of them is 32-bit. Therefore, having the ability to configure the through with respect to n and q and the communication bandwidth is desired.

4. HARDWARE ARCHITECTURE

This section presents a parametric Butterfly design and an IO-optimized and pipelined hardware architecture that implements the 7-Step NTT algorithm detailed in 4. To the best of our knowledge, this is the first-in-literature implementation of the negacyclic 7-Step NTT algorithm on hardware.

4.1 Butterfly Architecture

Butterfly is the fundamental operation in NTT. It enables us to compute NTT results recursively so that we don't need to change the mode of operation throughout the computation time.

There are two main butterfly operation techniques in the literature: Cooley-Tukey (Cooley & Tukey (1965)) and Gentleman-Sande (Gentleman & Sande (1966)). Both of these mode of operations can be used for both forward NTT and inverse NTT interchangeably. In this work, we preferred to use Cooley-Tukey for calculating NTT operations and Gentleman-Sande for calculating inverse NTT operations.

Cooley-Tukey (CT) operation takes 4 inputs, A and B, two uniformly random integers in $\mathbf{Z}_q$, W is the so-called twiddle factor and q is the prime number that we operate on. There are two outputs of CT operation, namely, Even and Odd. Even (E) output of CT is $A + b \times W \mod q$ and Odd (E) output of CT is $A - B \times W$. The top-level diagram of CT operation can be found in Figure 4.1. It can be understood from the figure that there are 3 main building blocks of this circuit: Modular Adder, Modular Subtractor and Modular Multiplier.

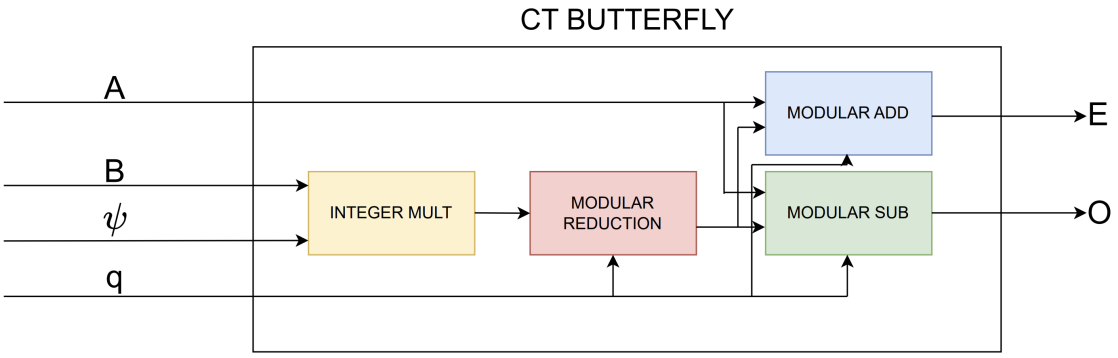

Figure 4.1 Cooley-Tukey Butterfly Architecture

4.1.1 Modular Adder and Modular Subtractor

Modular addition and modular subtraction are important operations in our architecture. However, these operations can be implemented trivially because it doesn't need many resources and doesn't introduce a critical path in our implementation compared to modular multiplication units.

Modular adder and subtractor operations take 3 inputs: A and B, in $\mathbf{Z}_q$, and a prime number q. The operation is $(A+B) \mod q$ and $(A-B) \mod q$ for adder and subtractor, respectively.

4.1.2 Modular Multiplication

Modular multiplication is the most fundamental building block of the butterfly circuit used in our design. This is because the computations are performed over a ring, requiring numbers to be reduced after multiplication to ensure they fit within the constraints of the FPGA during operation. However, implementing modular multiplication on an FPGA is a non-trivial task due to the computationally expensive operations it involves, which are not inherently hardware-friendly.

For example, to multiply two w-bit integers and reduce the result modulo a prime q, two main operations are required: integer multiplication and modular reduction. Integer multiplication involves performing a $w \times w$-bit multiplication, which is inherently resource-intensive and not well-suited for hardware implementation. Therefore, it is essential to find methods that make this operation more hardware-friendly. Details on integer multiplication will be provided in subsequent sections.

A similar challenge arises with modular reduction. On CPUs, modular reduction is efficiently implemented due to the availability of division and rounding operations using general purpose processors. However, on FPGAs, division operations are inefficient and require significant computational resources, making modular reduction a bottleneck. To address this challenge, researchers have developed several modular reduction techniques tailored for FPGA implementations. Some of the most notable methods include Plantard (Plantard (2021)), Montgomery (Montgomery (1985)), Barrett (Barrett (1987)), and K2-RED (Bisheh-Niasar, Azarderakhsh & Mozaffari-Kermani (2021)), which are specifically designed to overcome the inefficiencies of hardware division and improve the performance of modular arithmetic operations.

Barrett reduction (Barrett (1987)) is one of the earliest modular reduction methods introduced and is known for its efficiency in CPU-based implementations. The method begins by calculating the reciprocal μ with respect to the given modulus m. Subsequently, for a given input x, an approximate quotient is computed. Finally, a comparison is made to ensure the correct remainder is obtained. Barrett reduction is particularly well-suited for FPGA implementations due to its inherent parallelism and efficient use of hardware resources. However, it requires larger operand sizes compared to the WLM reduction technique. As a result, the WLM reduction method is often a more suitable approach than Barrett reduction in scenarios where smaller operand sizes are desired.

Plantard reduction (Plantard (2021)) is a more recent method in the literature. It involves calculating an auxiliary modulus by stretching the input numbers to operate within this new modulus. The reduction is then performed through a series of modular operations to arrive at the final result. While this method is conceptually innovative, the intermediate calculations required for these reductions are complex and computationally expensive. As a result, this method was not selected for our implementation.

K2-RED reduction (Bisheh-Niasar et al. (2021)) is a relatively new approach designed to accelerate post-quantum cryptography (PQC) operations on FPGAs. This method relies on the specific structure of prime moduli q to achieve efficient reduction. However, the applicability of K2-RED is limited by its requirement for primes of a particular form, restricting the range of primes that can be used. Homomorphic encryption schemes, which require a wide range of primes to support various security levels, are incompatible with this limitation. Consequently, K2-RED was not chosen for our design, as it would fail to accommodate the flexibility needed for such schemes.

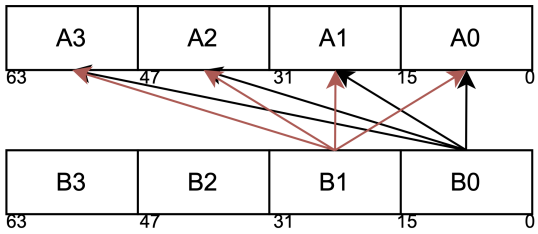

Figure 4.2 64x64-bit Symmetric Partition Integer Multiplication with 16 DSPs

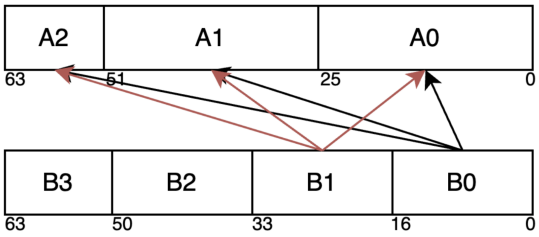

Figure 4.3 64x64-bit Aymmetric Partition Integer Multiplication with 12 DSPs

Based on these considerations, our modular multiplication design prioritizes flexibility and efficiency, avoiding methods that are either computationally expensive or limited in scope for FPGA implementations. We chose to use a reduction in Word-Level-Montgomery (WLM) in our implementation.

4.1.3 Integer Multiplication

In the literature, various multiplication techniques have been proposed for performing $w \times w$-bit multiplications. Many of these methods are specifically designed to leverage the architecture of Digital Signal Processing (DSP) blocks on Field Programmable Gate Arrays (FPGA) devices. These DSP blocks typically include highly efficient Application Specific Integrated Circuits (ASIC) implementations for 18×27 signed multiplications and accumulations, with an additional input provided. In this study, we also aim to optimize the utilization of DSP blocks to enhance the overall efficiency of the design.

The work presented (Mert et al. (2020)) employs a symmetric partitioning strategy to compute 32×32-bit multiplications. This method requires $2 \times 2 = 4$ smaller 16×16-bit multiplications to produce the final result. Given the 32-bit operand size, this approach is likely the most efficient, as the intermediate results remain manageable and can be readily accumulated into subsequent computations. However, this strategy presents challenges when extended to 64-bit multiplications.

For 64-bit multiplication, the symmetric partitioning approach would necessitate $4 \times 4 = 16$ smaller multiplications before the results could be accumulated. Such an implementation would forfeit the advantage provided by the asymmetric input sizes of the DSP blocks. To address this, we propose an alternative input partitioning method that better exploits the available DSP block sizes. Our approach follows a methodology similar to that described (Hirner, Mert & Roy (2023)), wherein standard tiling is employed.

4.1.4 Modular Reduction

Modular reduction presents a significant challenge when implemented on Field-Programmable Gate Array (FPGA) platforms. The primary difficulty arises from the necessity of the division operation in conventional modular reduction approaches, as typically utilized in CPU-based systems. Unlike addition, subtraction, and multiplication, the division operation is inherently complex and not efficiently supported on FPGA architectures.

In contrast, addition, subtraction, and multiplication operations can be efficiently implemented and accelerated on FPGA platforms due to their inherent parallelism and resource-efficient characteristics. Consequently, alternative methods that circumvent the need for explicit division must be employed to realize modular reduction on FPGA.

As discussed in the preceding sections, the literature offers several techniques to address this issue. Among these, the Word-Level Montgomery method (Mert, Karabulut, Öztürk, Savaş, Becchi & Aysu (2020)) emerges as the most suitable approach for FPGA-based architectures, particularly those designed to accelerate Fully Homomorphic Encryption (FHE) applications.

The WLM (Word-Level Montgomery) algorithm is outlined in 5. This algorithm calculates the modular reduction results iteratively, breaking down computations into smaller components that are more manageable for hardware implementations. The original Montgomery reduction algorithm (Montgomery (1985)) involves one $K \times 2K$ and one $K \times K$-bit multiplication to perform the modular reduction. While effective, this approach requires handling large operand sizes, which can be challenging for hardware implementations.

In contrast, the WLM algorithm transforms these multiplications into multiple L smaller multiplications of size $w \times (K - w)$ bits. Although the number of operations

Algorithm 5 Word-Level Montgomery Reduction Algorithm for NTT-friendly primes Mert et al. (2020)

Input: $C = A \cdot B$ (a $2K$-bit positive integer)
Input: q (a K-bit modulus), $q = q_H \cdot 2^w + 1$
Input: $w = \log_2(2n)$ (word size)
Output: $Res = C \cdot R^{-1} \pmod{q}$ where $R = 2^{w \times L} \pmod{q}$
$L \leftarrow \lceil \frac{K}{w} \rceil$
$T1 \leftarrow C$
for $i = 0 \rightarrow L2$ **do**
    $T1_H \leftarrow T1 >> w$
    $T1_L \leftarrow T1 \pmod{2^w}$
    $T2 \leftarrow$ two's complement of $T1_L$
    $cin \leftarrow T2[w-1] \vee T1_L[w-1]$
    $T1 \leftarrow T1_H + (q_H \cdot T2[w-1:0]) + cin$
end for
$T4 \leftarrow T1 - q$
if $T4 < 0$ **then**
    $Res \leftarrow T1$
else
    $Res \leftarrow T4$
end if

may increase as the parameter w decreases, the operand sizes are significantly smaller than in the original Montgomery approach. This tradeoff allows for an efficient hardware realization of the algorithm. By reducing operand sizes, the WLM algorithm is particularly well-suited for FPGA implementations, where resource constraints often necessitate optimization of operand size over operation count.

The WLM algorithm takes a single input, which is a $2K$-bit positive integer $C \in \mathbf{Z}_q$. The modulus q is represented in the form $q = q_H \cdot 2^w + 1$. This specific representation is essential to enable compatibility with the NTT (Number Theoretic Transform) algorithms, which often require such forms of q to achieve efficient modular arithmetic.

During each iteration of the WLM process, the lower half of the result from the previous iteration is multiplied by q_H, producing the input for the next iteration. This iterative process operates "word by word," where the size of a word corresponds to the parameter w. This word-level processing is the distinguishing feature of the WLM algorithm and is the reason it is referred to as "Word-Level Montgomery" in the literature. By iterating word by word, the algorithm minimizes operand sizes at each step, making it highly effective for FPGA implementations, which benefit from smaller, localized computations.

Furthermore, in the context of this thesis, the modular multiplication results ob-

tained using the WLM algorithm align with those reported in the literature. However, the 7-Step NTT architecture proposed in this work outperforms existing designs regarding computational efficiency and resource utilization, making it superior to other architectures found in previous studies. This demonstrates the effectiveness of combining the WLM algorithm with an optimized NTT architecture for modular arithmetic computations in specialized hardware environments.

4.2 IO-Optimized 7-Step NTT Architecture

4.2.1 Design Principles

4.2.1.1 Column Independence

4-Step and 7-Step algorithms utilize column independence by working on independent columns. In other words, there is no dependency between the processing of different columns of the matrix representation. On the other hand, the iterative approach requires operating on the whole polynomial at each step. Therefore, it is feasible to parallelize multiple stages of the iterative approach. In this work, the advantage of this characteristic is widely observed. It is crucial for the main design goal, which is throughput maximization.

Column independence between NTT columns creates a way to operate on different columns of the NTT at the same time. Our main aim is to maximize the throughput, so we need to increase the parallel operations as much as possible. If we have a dependency on the coefficient that we calculate, we would have stall cycles to make coefficients available before the next computation.

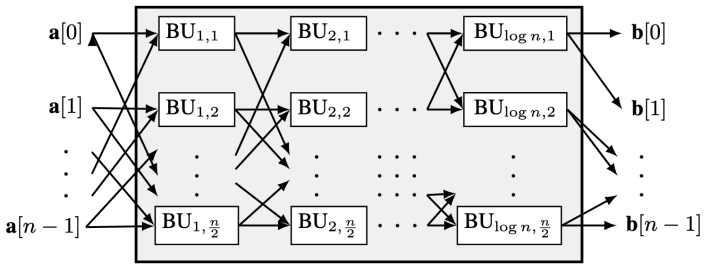

Figure 4.4 Loop-unrollling illustration: NTT_n requires $\log n \cdot (n/2)$ Butterfly, main block of INU

4.2.1.2 NTT-unrolling

NTT-unrolling refers to the fact that there are dedicated BUs for every stage of the iterative NTT, so each stage is processed at each *cc*. This strategy is demonstrated in 4.4. In the ideal scenario, every *cc* will have n coefficient output (since there is no stall.). However, this would require an enormous number of resources for the BUs in an iterative NTT of large size (*e.g.*, for $n = 2^{12}$, $2^{11} \cdot 12$ BUs that is infeasible in today's technology). However, 4-Step and 7-Step NTTs decrease the resources needed for fully pipelined stages as these approaches instantiate iterative NTTs of smaller sizes, as explained in previous sections.

Loop-unrolling is a fundamental technique in GPU and FPGA implementations that increase the parallelization and number of operations per unit of time. However, the tradeo-ff of this approach is that it increases resource allocation, as stated before. Therefore, it is fundamental to consider the resources before increasing the unrolling level of the implementation. For the next calculations, assume $TP = n$ for the iterative model because all coefficients depend on each other. For 4-Step approach, assume $TP = n_1 = n_2$ and for 7-Step approach assume $TP = n_{11} = n_{12} = n_{21} = n_{22}$. For iterative approaches, we would need $\frac{n}{2} \times \log n$. In 4-Step implementation (assume symmetric partition, $n_1 = n_2$), we would need $\frac{n_1}{2} \times \log n_1 \times 2 \quad + \quad n_1$ BUs. The reason is that we have two same pipelined iterative NTT blocks for $n = n_1$. In 7-Step implementation, we would need $\frac{n_{11}}{2} \times \log n_{11} \times 4 \quad + \quad 3 \times n_{11}$ because we would have 4 iterative. As an example, if we have 2^{16} ring size we would have $(2^8, 2^8)$ and $(2^4, 2^4, 2^4, 2^4)$ partitions for 4-Step and 7-Step, respectively. Consequently, we would need $2^{15} \times 16$ BUs for iterative approach, $2^7 \times 8 \times 2 \quad + \quad 2^8$ for 4-Step and $2^3 \times 4 \times 4 \quad + \quad 3 \times 2^4$. Compared to the other two methods, 7-Step is the only feasible option for FPGA because the other two approaches require huge amounts of resources.

4.2.1.3 Coefficient Throughput

Our solution incorporates a unique feature. We provide a parametric hardware generator that operates based on design parameters **TP** , $n11, n_{12}, n_{21}, n_{22}$ for any given $\mathcal{R}_{q,n}$. As previously discussed, the modern FPGAs with HBMs suffer from limited bandwidth, which constrains the throughput of the NTT engine. Our solution addresses this problem by considering the IO bandwidth during the generation of throughput-oriented hardware. The goal of this concentration is to fully utilize

the communication bandwidth.

4.2.1.4 Address Generation

Address generation is a fundamental challenge in NTT architectures on FPGA. Existing methods in the literature (Ayduman, Koçer, Kırbıyık, Can Mert & Savaş (2023); Kurniawan, Duong-Ngoc & Lee (2023); Liu, Kuo, Mo & Su (2023); Mert et al. (2020); Su, Yang, Yang, Yang & Liu (2022); Yang, Kuppannagari, Kannan & Prasanna (2022); Ye et al. (2021)) entail complex address generation units for computing NTTs. On the other hand, in SDF/MDC (Hirner et al. (2023); Tan, Wang, Lao, Zhang & Parhi (2021); Ye, Cheung & Huang (2022)) architectures, scalable implementations are provided for different ring sizes. In this work, we introduce a simple address-generation method for smaller NTTs. We provide twiddle factors for each NTT stage and create a fully pipelined system without using extra address generation logic. This system can produce NTT outputs in each clock cycle after populating pipeline stages. This architecture paves the way for a fully pipelined implementation with a simpler address generation and can be generalized to different ring sizes.

4.2.2 Overall Architecture

In the proposed design, there are 10 main modules, which include INUs 1-4, AUs 1-3, TMUs 1-3 and TGU. Details of this architecture are shown in 4.5. In general, iterative NTT units handle pipelined and relatively small NTT operations. Twiddle multiplication units perform the task of multiplying intermediate coefficients by powers of ψ during iterative NTT steps. To achieve maximum throughput, the entire implementation is fully pipelined. The AUs are responsible for rotating coefficient outputs to generate proper BRAM read/write addresses for subsequent iterative NTT computations.

It is worth noting that INUs 1-2, combined with TMU 1 and AU 1, form a 4-Step NTT structure. This implementation corresponds to the algorithm in 3 and is used in Line 4 of 4, where it is referred to as 4NU 1. Similarly, INUs 3-4, along with TMU 3 and AU 3, form another 4-Step NTT, which is called in Line 13 of 4 and referred to as 4NU 2.

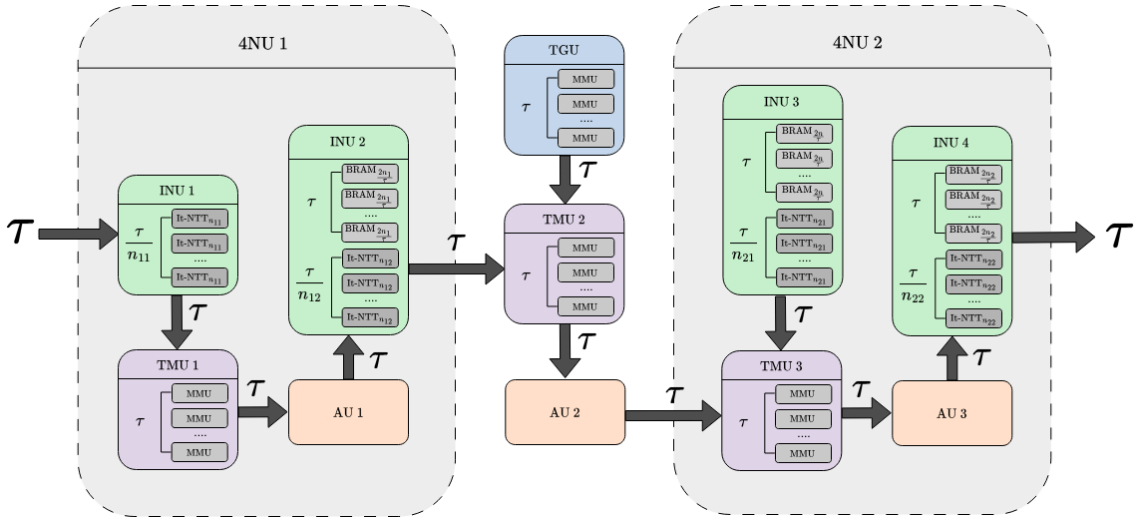

Figure 4.5 Pipelined 7-Step NTT architecture.

The proposed 7-Step architecture focuses on high throughput, determined by the parameter **TP** . During each *cc*, this 7-Step NTT design processes **TP** new coefficients as input and produces **TP** coefficients as the result of the NTT operation. The throughput of all the units described earlier is also **TP** . As expected, the output stream becomes valid after some *cc*, depending on the latency of the architecture. 4.1 illustrates the *cc* latencies for each module. The total latency of the proposed architecture depends on the size of the input ring, the input modulus size ($\log_2 q$), and the parameter **TP** .

An example of the general execution for $n = 2^4$ using the r-Step NTT architecture is shown in 4.6. Here, the partitions are of size r, with $\mathbf{TP} = r$. Initially, coefficients are fed as $(0-8, 4-12, 1-9, 5-13, ...)$, as the 4-Step NTT algorithm follows the same computation pattern as traditional iterative NTTs. After INU 1, the first twiddle multiplication is performed, and inputs are rotated for INU 2. Subsequently, the second twiddle multiplication (TMU 2) is computed, and inputs are rotated again (AU 2) at the end of the first column $(0, 4, 8, 12)$. This ensures that elements from the same row (e.g., $(0, 1, 2, 3)$) are stored in different BRAMs. For subsequent blocks, the computations are carried out in a manner similar to the earlier blocks.

4.2.2.1 Merged NTT Unit (INU) 1

INU 1 is the first NTT block responsible for directly processing the raw input coefficients of the 7-Step NTT. It executes multiple parallel n_{11}-point iterative NTTs based on the parameter **TP** . When the throughput $\mathbf{TP} = n_{11}$, this module inputs

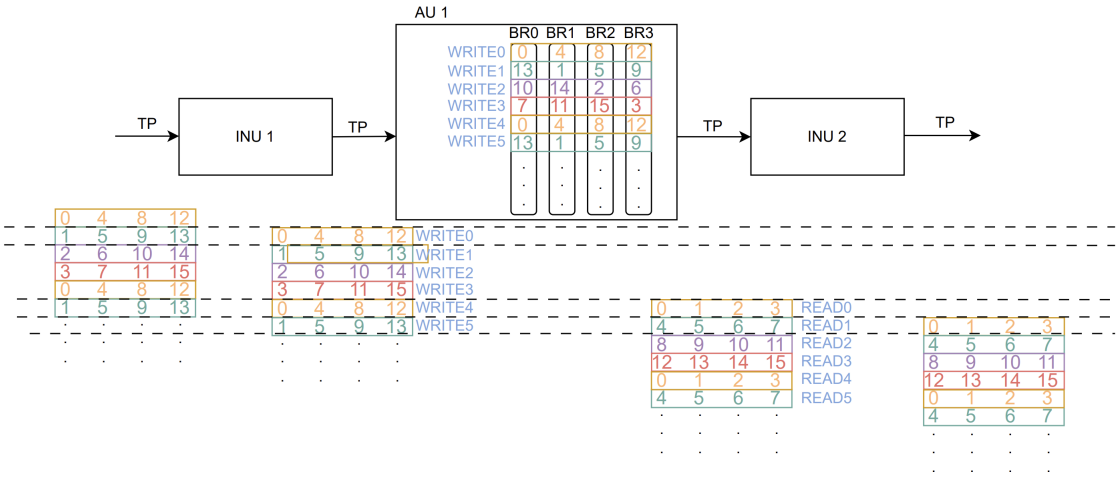

Figure 4.6 A sample execution for the pipelined 4-Step NTT architecture for $n = 16$, $n_1 = n_2 = 4, TP = 4$.

Table 4.1 Latencies of different units in the proposed architecture.

Unit	**Latency**
MMU	$(\lceil \frac{\log_2 q}{13} \rceil \cdot 2 + 2)$
BU	MMU+2
BRAM R/W	1
INU 1	$\log n_{11} \cdot BU + 3$
INU 2	$\log n_{12} \cdot BU + 3$
INU 3	$\log n_{21} \cdot BU + 3$
INU 4	$\log n_{12} \cdot BU + 3$
TMU 1	MMU + 3
TMU 2	MMU + 3
TMU 3	MMU + 3
AU 1	3
AU 2	3
AU 3	3
4NU 1	INU1 + TMU1 + AU1 + INU2
4NU 2	INU3 + TMU3 + AU3 + INU4
7-step NTT	4NU1 + TMU2 + 4NU2

and outputs n_{11} coefficients per cc due to its pipelined structure. More generally, $\mathbf{TP}/n_{11}$ parallel n_{11}-point iterative NTTs are used to generate $\mathbf{TP}$ coefficients per cc, assuming $\mathbf{TP}$ is divisible by n_{11}. This module corresponds to the set of column NTTs described in Line 3 of 3.

Table 4.2 Authomorphism (Rotator) Unit (AU) operation pattern with respect to counters.

Unit	Counter Bit-length
AU 1	$\log_2 \frac{n_{11} \cdot n_{12}}{TP}$
AU 2	$\log_2(n_{11} \cdot n_{12} \cdot n_{21} \cdot n_{22})$
AU 3	$\log_2(n_{21} \cdot n_{22})$

Each NTT stage is fully unrolled, resulting in $\log n_{11}$ stages. Dedicated blocks of BUs are allocated for the distinct stages, leading to a total of $(n_{11}/2) \cdot \log(n_{11}) \cdot \mathbf{TP}/n_{11}$ BUs to meet the **TP** requirement. Intermediate results between the stages are stored in registers, avoiding the use of BRAMs for this purpose. Similarly, twiddle factors are stored in registers. Recall from 2.4 that $n_{11} - 1$ twiddle factors are required for the n_{11}-point iterative negacyclic NTTs.

4.2.2.2 Twiddle Multiplication Unit (TMU)s 1-3

As discussed earlier, the overall architecture includes three TMU units. These units handle the output coefficients from the preceding iterative NTT stages. Specifically, TMU 1 and 3 are responsible for performing the multiplications defined in Line 8 of 3, while TMU 2 carries out the twiddle multiplication described in Line 9 of 4. Each of these units is equipped with a set of BRAMs to store pre-computed twiddle factors. For TMU 1, 2, and 3, the twiddle factors n_1, n, and n_2, respectively, are stored in the BRAMs.

To meet the **TP** requirement, each TMU contains **TP** independently operating, pipelined MMUs. At every *cc*, these MMUs are fed by **TP** independent BRAMs. The size of each BRAM is determined by the number of twiddle factors required for each TMU, divided by **TP** , as depicted in 4.5. Apart from storing pre-computed twiddle factors, no additional BRAMs are utilized in the TMUs. Intermediate results produced by the MMUs are stored in registers.

4.2.2.3 Authomorphism (Rotator) Unit (AU)s 1-3

Similar TMUs, there are three AUs in the pipeline that handle the output from the TMUs. Each AU has its own internal counter and shifts its input by a fixed offset based on the current counter value. The counter bit-lengths vary for each AU and depend on n_{11}, n, n_{21}, and **TP** , as shown in 4.2. The rotation of the coefficients ensures that they are stored in the correct positions in BRAMs for the following INUs. For AU1 and AU3, the rotation happens at each *cc* to correctly position the coefficients for the next It-NTT block. However, AU2 performs rotation after the first column computation is complete. Since each column element must be stored separately for the second 4S-NTT, the rotation begins after a few *cc*s, unlike for AU1 and 3. As explained in Line 3 and Line 12 of 3, 4S-NTT processes one block of It-NTTs for the rows and another block for the columns of the matrix **A**. Thanks to the rotation of output coefficients from INU 1 and INU 3 (It-NTT on columns) by AU 1 and AU 3, the input coefficients for INU 2 and INU 4 (It-NTT on rows) can be accessed in a single *cc* from the appropriate BRAMs. The same principle applies for AU 2 and the 7-Step NTT (4). The operation of AUs is illustrated in 4.6. It's important to note that if the input coefficients cannot be retrieved immediately from BRAMs in INUs 2-4, stall cycles will occur, reducing throughput. Since AUs generate their **TP** -coefficient output with 1 *cc* latency, there is no intermediate data, so a BRAM for storing intermediates is unnecessary.

4.2.3 Merged NTT Unit (INU)s 2-4

Similar to INU 1, INUs 2-4 implement a fully pipelined and negacyclic It-NTT as mentioned in Line 12, Line 3, and Line 12 of 3. These units operate on n_{12}, n_{21}, and n_{22} points, respectively. Additionally, note that the It-NTTs carried out by INUs 3-4 correspond to the 4S-NTT call in Line 13 of 4, while the one implemented by INU 2 relates to the 4S-NTT call in Line 4 of 4.

In terms of BUs and the number of parallel It-NTTs, the structure of INUs 2-4 is identical to INU 1. However, INUs 2-4 differ by storing the outputs of the preceding AUs in BRAMs. This ensures efficient and correct operation. As explained earlier, BRAMs are used because they enable It-NTTs to process the transpose of the matrix compared to the prior INU. For example, INU 4 processes rows while INU 3 processes columns.

To start operations, the preceding INU must finish generating all the input coefficients. Thus, the BRAM size must match the number of points in the It-NTT being processed (e.g., n_{12} for INU 2). However, for efficient computation, newly generated

coefficients must also be stored in BRAMs. Specifically, the BRAM size is twice the number of points in the implemented It-NTT (e.g., $2n_{12}$ for INU 2). This is due to the read/write mode switching in BRAMs, where half of the BRAM blocks are in read mode, and the other half are in write mode at any given time.

4.2.4 Twiddle Generation Unit (TGU)

The 4-Step algorithm includes a step called twiddle multiplication, where each matrix element is multiplied by a power of ψ, as outlined in Line 7 of 3. This process involves n twiddle factors, which are later utilized. For large n, pre-computing these factors can be challenging due to the significant memory required. In FHE applications, computations often involve multiple values of q, requiring a set of twiddle factors for each q. Several studies in the literature explore methods for generating twiddle factors on-the-fly.

Let $\mathbf{p}_j[i]$ represent $\psi^{(2\cdot i-n_1+1)\cdot j}$. Then, $\mathbf{p}_{j+1}[i] = \mathbf{p}_j[i] \cdot \mathbf{p}_1[i] = \psi^{(2\cdot i-n_1+1)\cdot(j+1)}$. By pre-computing only $\mathbf{p}_1$, one can compute $\mathbf{p}_j$ for all $j > 2$. This method reduces the storage requirement to n_1 twiddle factors.

In pipelined designs like ours, the actual number of precomputed twiddle factors may be slightly higher. To generate **TP** twiddle factors in each *cc* using this approach, $\mathbf{p}_1$ to $\mathbf{p}_k$ are precomputed, where $k = \lceil (Tp \cdot \mathcal{L})/n_1 \rceil$, and $\mathcal{L}$ is the latency of the MMU. The TGU then outputs $\mathbf{p}_{j+k}[i']$ during each *cc*, computed from $\mathbf{p}_j[i']$ within $\mathcal{L}$ clock cycles, for $j > k$, $Tp \cdot t \leq i' < \min(Tp \cdot (t+1), n_1)$, and $t \leq \lfloor n_1/Tp \rfloor$.

5. EVALUATION

In this section, a comprehensive evaluation of our proposed solution is presented.

5.1 Implementation

The proposed 7-Step architecture is realized using Verilog HDL. To enable reconfigurability, Python scripts are developed to automate the generation of RTL designs based on the given parameters, namely, $n, n_{11}, n_{12}, n_{21}, q$, and the desired throughput **TP** . The implemented ring sizes range from $n = 2^{10}$ to $n = 2^{16}$, while the coefficient moduli vary from $\log q = 32$ to $\log q = 64$. As stated earlier, the target FPGAs used for implementations are the AMD-Xilinx Alveo U280[1] (XCU280) and Virtex VC709[2], with synthesis and implementation performed using Vivado 2023.2[3]. The XCU280 FPGA features 1,303,680 Look-up Table (LUT)s, 2,607,360 Flip-flop (FF)s, 9024 DSPs[4], and 2016 BRAM36E1s. The VC709 FPGA features 433,200 LUTs, 866,400 FFs, 3600 DSPs and 1470 BRAM36E1s.

For decomposing n, we use a symmetric partitioning strategy where $n_{11} = n_{12} = n_{21} = n_{22}$ is selected if this configuration is feasible. If not, we assign a larger size to the first dimension. For instance, when $n = 2^{13}$, the values are set as $n_{11} = 2^4$ and $n_{12} = n_{21} = n_{22} = 2^3$. Similarly, for $n = 2^{15}$, we choose $n_{11} = n_{12} = n_{21} = 2^4$ and $n_{22} = 2^3$. This strategy is based on the observation that the initial blocks handle a lighter computational load compared to the later blocks.

---

[1] `https://www.xilinx.com/products/boards-and-kits/alveo/u280.html`

[2] `https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html#resources`

[3] `https://www.xilinx.com/products/design-tools/vivado.html`

[4] `https://docs.amd.com/r/en-US/ug958-vivado-sysgen-ref/DSP48E2`

5.2 Results and Comparison

Tables 5.1 and 5.2 present a comparative analysis of our implementations against state-of-the-art designs. The comparison is divided into two tables to account for the two primary categories of FPGAs: high-performance and moderate-performance devices. High-performance FPGAs, such as Alveo or Ultrascale, generally achieve superior performance compared to their moderate-performance counterparts, such as Virtex devices. Given the distinct characteristics of these FPGA families, the results for moderate-performance FPGAs are compared with those of the Virtex-7, while the results for high-performance FPGAs are compared with those of the Alveo U280.

The average latency is computed over 100 consecutive NTT operations. The Area-Time-Product (ATP) metric, widely used in the literature (Ye et al. (2022)), is calculated as Latency $(\mu s) \times (\mathrm{LUT} + \mathrm{FF}/2 + 100 \times \mathrm{DSP} + 300 \times \mathrm{BRAM})$. The average latency is used in the ATP computation because this study primarily focuses on maximizing throughput. Most of the compared implementations are based on iterative NTT architectures (Hirner et al. (2023); Kurniawan et al. (2023); Liu et al. (2023); Yang, Wang, Yang, Zhang & Yang (2023)), while some recent works, including ours, employ hierarchical approaches (Chen et al. (2024); Guan et al. (2024); Liu et al. (2024); Mert et al. (2020); Wang & Gao (2023)). In all cases, our proposed design achieves superior timing and ATP results. For each category, we report the design corresponding to the maximum value of **TP** that is implementable on the target FPGA.

Table 5.1 Comparison with Literature (Alveo & Ultrascale FPGAs)

Work	**Arch.**	**Platform**	$\log q$	n	Tp	**LUT/FF/DSP/BRAM**	**F (MHz)**	**Lat. (cc)**	**Avg. (μs)**	**ATP ($\cdot 10^{-3}$)**
Yang et al. (2022)	Iter.	XCU200	28	2^{10}	32	95 / 104 / 640 / 80	210	236	1.12 (**4.25x**)	0.264 (**4.60x**)
Ours	7-Step	XCU280	32	2^{10}	16	76.1 / 94.6 / 864 / 24	250	66	0.26 (**1.00x**)	0.057 (**1.00x**)
Ours	7-Step	XCU280	32	2^{12}	32	137 / 160 / 1632 / 40	250	130	0.52 (**1.00x**)	0.204 (**1.00x**)
Ours	7-Step	XCU280	64	2^{11}	16	185 / 204 / 2640 / 40	250	130	0.52 (**1.00x**)	0.293 (**1.00x**)
Kurniawan et al. (2023) ‡	Iter.	V.Ultrascale+	60	2^{12}	-	74.5 / 61.4 / 288 / 155	250	951	3.804 (**7.26x**)	0.687 (**1.22x**)
Guan et al. (2024)	4-Step	XCU280	64	2^{12}	-	523 / 1478 / 6518 / 34.5	300	351	1.17 (**2.23x**)	2.251 (**4.01x**)
Ours	7-Step	XCU280	64	2^{12}	32	356.2 / 375.5 / 5040 / 72	250	131	0.52 (**1.00x**)	0.560 (**1.00x**)
Ayduman et al. (2023)	Iter.	XCU280	32	2^{13}	-	29.1 / 21.5 / 224 / 64	181.8	1690	9.29 (**4.47x**)	0.84 (**1.60x**)
Ours	7-Step	XCU280	32	2^{13}	16	89.8 / 107.9 / 1008 / 32	250	518	2.07 (**1.00x**)	0.53 (**1.00x**)
Ayduman et al. (2023)	Iter.	XCU280	32	2^{14}	-	29.1 / 21.5 / 224 / 96	181.8	3612	19.87 (**4.79x**)	1.81 (**1.61x**)
Ours	7-Step	XCU280	32	2^{14}	16	94.2 / 112.2 / 1056 / 48	250	1036	4.14 (**1.00x**)	1.12 (**1.00x**)
Kurniawan et al. (2023) ‡	Iter.	V.Ultrascale+	60	2^{14}	-	74.5 / 61.4 / 288 / 155	250	4340	17.36 (**4.18x**)	3.13 (**1.05x**)
Ours	7-Step	XCU280	64	2^{14}	16	234.1 / 255.7 / 3280 / 96	250	1036	4.14 (**1.00x**)	2.97 (**1.00x**)
Kurniawan et al. (2023) ‡	Iter.	V.Ultrascale+	60	2^{15}	-	74.5 / 61.4 / 288 / 155	250	8435	33.74 (**8.14x**)	6.09 (**1.09x**)
Ours	7-Step	XCU280	64	2^{15}	32	444.6 / 459.6 / 6160 / 176	250	1036	4.14 (**1.00x**)	5.56 (**1.00x**)
Ours	7-Step	XCU280	32	2^{16}	32	169.5 / 191.5 / 2016 / 152	250	2070	8.28 (**1.00x**)	4.24 (**1.00x**)
Kurniawan et al. (2023) ‡	Iter.	V.Ultrascale+	60	2^{16}	-	74.5 / 61.4 / 288 / 155	250	16627	66.5 (**7.96x**)	12.00 (**1.01x**)
Wang & Gao (2023)	3-D†	XCU250	64	2^{16}	-	267.1 / 328.4 / 2736 / 2126	165	62700	380 (**45.5x**)	510.2 (**43.3x**)
Ours	7-Step	XCU280	64	2^{16}	32	460 / 470 / 6320 / 280	248	2070	8.34 (**1.00x**)	11.78 (**1.00x**)

†: employs 3-D decomposition of n as $2^6 \cdot 2^6 \cdot 2^4$. ‡: restricted to special primes.

Table 5.2 Comparison with Literature (Virtex-7 FPGAs)

Work	**Arch.**	**Platform**	$\log q$	n	Tp	**LUT/FF/DSP/BRAM**	**F (MHz)**	**Lat. (cc)**	**Avg. (μs)**	**ATP ($\cdot 10^{-3}$)**
Hirner et al. (2023)	Iter.	Virtex-7	28	2^{10}	-	6.4 / 3.7 / 18 / 2	150	1035	6.9 (**18.82x**)	0.073 (**1.78x**)
Ours	7-Step	Virtex-7	32	2^{10}	16	27.5 / 41.2 / 576 / 17	180	66	0.37 (**1.00x**)	0.041 (**1.00x**)
Ours	7-Step	Virtex-7	32	2^{11}	16	30.5 / 44.4 / 624 / 17	180	130	0.722 (**1.00x**)	0.087 (**1.00x**)
Liu et al. (2023)	Iter.	Virtex-7	32	2^{12}	-	24.6 / 23.9 / 352 / 80	207	777	3.75 (**2.60x**)	0.359 (**1.94x**)
Mert et al. (2020)	4-Step	Virtex-7	32	2^{12}	-	70 / 70 / 599 / 129	200	460	2.30 (**1.59x**)	0.468 (**2.53x**)
Ours	7-Step	Virtex-7	32	2^{12}	16	32.4 / 47.2 / 666 / 18	180	260	1.44 (**1.00x**)	0.185 (**1.00x**)
Hirner et al. (2023)	Iter.	Virtex-7	60	2^{12}	-	21.8 / 19 / 220 / 16	150	2070	13.80 (**8.81x**)	0.802 (**1.22x**)
Yang et al. (2023)	Iter.	Virtex-7	60	2^{12}	-	19.1 / 17.86 / 216 / 88	270	6144	22.76 (**14.52x**)	1.73 (**2.63x**)
Ye et al. (2022)	Iter.	Virtex-7	60	2^{12}	-	17 / 11 / 286 / 24.5	150	4125	27.50 (**17.56x**)	1.61 (**2.45x**)
Liu et al. (2024)	4-Step	Virtex-7	64	2^{12}	-	18.9 / 26.7 / 266 / 24	211	2490	11.80 (**7.53x**)	0.779 (**1.19x**)
Liu et al. (2024)	4-Step	Virtex-7	64	2^{12}	-	9.2 / 12.6 / 133 / 24	241	4596	19.07 (**12.18x**)	0.687 (**1.05x**)
Ours	7-Step	Virtex-7	64	2^{12}	16	112 / 140 /2220 / 52	166	260	1.57 (**1.00x**)	0.657 (**1.00x**)
Ours	7-Step	Virtex-7	32	2^{13}	16	35.8 / 51.7 / 720 / 28	181	520	2.87 (**1.00x**)	0.41 (**1.00x**)
Chen et al. (2024)	4-Step	Virtex-7	32	2^{14}	-	26.9/26.9/144/32.5	200	4320	21.6 (**3.74x**)	1.39 (**1.54x**)
Ours	7-Step	Virtex-7	32	2^{14}	16	39 / 55 / 768 / 44	180	1040	5.78 (**1.00x**)	0.904 (**1.00x**)
Ours	7-Step	Virtex-7	64	2^{14}	16	134 / 161 / 2560 / 104	166	1040	6.27 (**1.00x**)	3.14 (**1.00x**)
Ours	7-Step	Virtex-7	64	2^{15}	16	166 / 195 / 3120 / 170	164	2060	12.56 (**1.00x**)	7.87 (**1.00x**)
Roy et. al. (2018)	Iter.	Virtex-6	30	2^{16}	-	72.6 / 63.1 / 250 / 84	100	47795	477.95 (**20.4x**)	73.77 (**13.71x**)
Chen et al. (2024)	4-Step	Virtex-7	30	2^{16}	-	30.8 / 36.2 / 160 / 128	196	16758	85.5 (**3.65x**)	8.83 (**1.64x**)
Ours	7-Step	Virtex-7	32	2^{16}	16	53 / 70 / 984 / 144	175	4100	23.43 (**1.00x**)	5.38 (**1.00x**)
Hirner et al. (2023)	Iter.	Virtex-7	64	2^{16}	-	31.3/30/300/255	135	59400	440 (**16.42x**)	67.23 (**3.58x**)
Ours	7-Step	Virtex-7	64	2^{16}	16	181 / 200 / 3264 / 310	153	4100	26.80 (**1.00x**)	18.77 (**1.00x**)

In a typical FHE setting with $\boldsymbol{n = 2^{12}}$ and $\log \mathbf{q} = \mathbf{32}$, our design achieves $\mathbf{1.94}\times$ and $\mathbf{2.53}\times$ lower ATP compared to the existing solutions (Liu et al. (2023)) and (Mert et al. (2020)), respectively, for Virtex-7 devices (see Table 5.2). This performance advantage arises from the adoption of the 7-Step NTT architecture, which enables higher throughput. Consequently, the proposed design completes an NTT operation in 1.44 μs on average, achieving up to $\mathbf{2.60}\times$ speed-up.

The advantages of our design persist in the 64-bit scenario with $n = 2^{12}$ for high-performance FPGAs in Table 5.1, with an average latency improvement of $\mathbf{2.23}\times$ over the best design in the literature (Guan et al. (2024)), which offers the minimum average latency among the existing solutions. Notably, apart from (Guan et al. (2024)), none of the other existing implementations provide a fully pipelined architecture. When compared to iterative NTT architectures with near 64-bit word-size and $n = 2^{12}$, our design exhibits $\mathbf{7.26}\times$ and $\mathbf{8.81}\times$ lower average latency than (Kurniawan et al. (2023)) for high-performance FPGAs and (Hirner et al. (2023)) for moderate-performance FPGAs, respectively. Additionally, while accelerating the NTT operation at this scale, our design achieves up to $\mathbf{4.01}\times$ better ATP than state-of-the-art implementations (Table 5.1), demonstrating the resource efficiency of our speed-optimized design.

Consistent with our design goals, the effectiveness of our solution scales with ring sizes from $\boldsymbol{n = 2^{10}}$ to $\boldsymbol{n = 2^{16}}$. For $\log q = 32$, our design outperforms the leading existing solutions (Chen et al. (2024)) for $\boldsymbol{n = 2^{14}}$ (Table 5.2) and (Ayduman et al. (2023)) for $\boldsymbol{n = 2^{13}}$ (Table 5.1), achieving $\mathbf{3.74}\times$ and $\mathbf{4.47}\times$ better average latency, respectively. Similarly, for $\log q = 64$, our solution offers up to $\mathbf{4.18}\times$ speed-up for $n = 2^{13}$ (Table 5.1). For $\boldsymbol{n = 2^{15}}$ and 64-bit, an $\mathbf{8.14}\times$ speed-up is achieved for high-performance FPGAs in Table 5.1. For the relatively smaller ring size of $\boldsymbol{n = 2^{10}}$ and 32-bit word size, our design outperforms the literature by $\mathbf{4.25}\times$ and $\mathbf{18.82}\times$ in terms of average latency for high-performance and moderate-performance FGPAs, respectively. The ATP of the proposed design also exhibits slight advantages over these implementations, further emphasizing the resource efficiency of our solution.

For the largest ring size $\boldsymbol{n = 2^{16}}$ in FHE applications, (Kurniawan et al. (2023)) is the closest competitor to our solution for high-performance FPGA devices (see Table 5.1). However, our design achieves a $\mathbf{7.96}\times$ improvement in average latency while maintaining a comparable ATP. It is worth noting that (Kurniawan et al. (2023)) only supports special primes, which provide resource utilization advantages over general NTT primes. Moreover, the prime modulus in that design is fixed, limiting its applicability for operations requiring multiple prime moduli, a critical requirement in FHE applications. In contrast, our proposed design allows for run-

time configuration of the prime modulus and twiddle factors, making it more practical for FHE scenarios. For more generic designs, our solution achieves a $\mathbf{3.65}\times$ and $\mathbf{45.5}\times$ reduction in average latency for $\log_2 q = 32$ and $\log_2 q = 64$, respectively, when compared to (Chen et al. (2024)) and (Wang & Gao (2023)). Additionally, our design outperforms these alternatives regarding ATP, which is consistent with the broader analysis.

6. Conclusion

In this work, we presented a modified four-step Number Theoretic Transform (NTT) algorithm that directly operates in the negacyclic ring $\mathcal{R}_{q,n}$, enhancing its compatibility with Fully Homomorphic Encryption (FHE) schemes. Additionally, we introduced an FPGA-based hardware accelerator implementing the 7-Step NTT algorithm, designed to support a wide range of parameters q and n commonly employed in FHE applications. The proposed solution prioritizes high throughput, low BRAM utilization, and scalability to address the diverse requirements of modern FHE systems.

Our architecture was implemented on the Alveo U280 FPGA and demonstrated significant performance improvements, achieving up to two orders of magnitude speed-up over existing designs in the literature. By leveraging an efficient design for modular arithmetic and optimizing data flow, our implementation achieves a competitive balance between flexibility and performance, catering to a variety of cryptographic use cases.

The experimental results underscore the versatility and efficiency of our approach. Our design achieves substantial improvements in both computational speed and hardware resource utilization, positioning it as a viable solution for FHE acceleration. The high clock frequency (approximately 250-300 MHz) further reinforces the practicality of the proposed architecture in real-world applications, ensuring its relevance in time-sensitive cryptographic workloads.

6.1 Future Work

While this work provides a robust foundation for accelerating the NTT in FHE applications, several avenues for future research remain. One promising direction

involves exploring alternative dimensional decompositions of the ring dimension n, which could further optimize resource utilization and enhance scalability. Additionally, integrating modular reduction techniques for special primes could significantly improve the efficiency of modular arithmetic operations.

Further optimization of the modular multiplication unit through advanced hardware techniques is another area of potential improvement. By investigating novel algorithms and architectural designs, future work can enhance both the computational efficiency and adaptability of the proposed solution. These enhancements would enable even greater performance gains while maintaining the flexibility required for emerging cryptographic applications.

In summary, the contributions of this work lay the groundwork for efficient, scalable, and adaptable FHE hardware accelerators. By addressing the outlined future challenges, this research can pave the way for further advancements in secure and efficient cryptographic computations.

BIBLIOGRAPHY

Ayduman, C., Koçer, E., Kırbıyık, S., Can Mert, A., & Savaş, E. (2023). Efficient design-time flexible hardware architecture for accelerating homomorphic encryption. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, (pp. 1–7).

Bailey, D. H. (1989). Ffts in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, (pp. 234–242). Association for Computing Machinery.

Barrett, P. (1987). Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Odlyzko, A. M. (Ed.), *Advances in Cryptology — CRYPTO' 86*, (pp. 311–323)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Bisheh-Niasar, M., Azarderakhsh, R., & Mozaffari-Kermani, M. (2021). High-speed ntt-based polynomial multiplication accelerator for post-quantum cryptography. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, (pp. 94–101).

Brakerski, Z. (2012). Fully homomorphic encryption without modulus switching from classical GapSVP. Cryptology ePrint Archive, Paper 2012/078.

Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2011). Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277.

Chen, X., Lu, W., Su, T., & Chen, D. (2024). Shp-fsntt: A scalable and high-performance ntt accelerator based on the four-step algorithm. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, (pp. 1–5).

Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2016). Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421.

Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2018). TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Paper 2018/421.

Cooley, J. W. & Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, *19*(90), 297–301.

Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, *31*(4), 469–472.

Fan, J. & Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144.

Gentleman, W. M. & Sande, G. (1966). Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, (pp. 563–578).

Gentry, C., Halevi, S., & Smart, N. P. (2011). Better bootstrapping in fully homomorphic encryption. Cryptology ePrint Archive, Paper 2011/680.

Guan, Z., Zhu, Y., Huang, Y., Lei, L., Wang, X., Jia, H., Chen, Y., Zhang, B., Dong, J., & Bian, S. (2024). Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (pp. 1–6).

Guo, W. & Li, S. (2023). Split-radix based compact hardware architecture for crystals-kyber.

Hirner, F., Mert, A. C., & Roy, S. S. (2023). Proteus: A pipelined NTT architecture generator.

Kurniawan, S., Duong-Ngoc, P., & Lee, H. (2023). Configurable memory-based ntt architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *70*(10), 3942–3946.

Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., & No, J.-S. (2021). Privacy-preserving machine learning with fully homomorphic encryption for deep neural network.

Li, Z., Ren, J., Du, G., Tu, Z., Wang, X., Yin, Y., & Ouyang, Y. (2022). An area-efficient large integer ntt-multiplier using discrete twiddle factor approach. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *70*(2), 751–755.

Liu, C., Tang, D., Song, J., Zhou, H., Yan, S., & Yang, F. (2024). Hmntt: A highly efficient mdc-ntt architecture for privacy-preserving applications. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, GLSVLSI '24, (pp. 7–12)., New York, NY, USA. Association for Computing Machinery.

Liu, S.-H., Kuo, C.-Y., Mo, Y.-N., & Su, T. (2023). An area-efficient, conflict-free, and configurable architecture for accelerating ntt/intt.

Lyubashevsky, V., Peikert, C., & Regev, O. (2010). On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, (pp. 1–23). Springer.

Lyubashevsky, V., Peikert, C., & Regev, O. (2012). On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Paper 2012/230.

Mert, A. C., Karabulut, E., Öztürk, E., Savaş, E., Becchi, M., & Aysu, A. (2020). A flexible and scalable ntt hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (pp. 346–351).

Mert, A. C., Öztürk, E., & Savaş, E. (2019). Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, (pp. 253–260).

Mert, A. C., Öztürk, E., & Savaş, E. (2020). Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *28*(2), 353–362.

Micciancio, D. (2005). *Shortest Vector Problem*, (pp. 569–570). Boston, MA: Springer US.

Miller, V. S. (1985). Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, (pp. 417–426). Springer.

Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of Computation*, *44*, 519–521.

Nguyen, T.-H., Kieu-Do-Nguyen, B., Pham, C.-K., & Hoang, T.-T. (2024). High-speed ntt accelerator for crystal-kyber and crystal-dilithium.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. volume 5, (pp. 223–238).

Plantard, T. (2021). Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, *9*(3), 1506–1518.

Pöppelmann, T. & Güneysu, T. (2012). Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *Progress in Cryptology–LATINCRYPT 2012:*, (pp. 139–158). Springer.

Regev, O. (2024). On lattices, learning with errors, random linear codes, and cryptography.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, *21*(2), 120–126.

Roy, S. S., Turan, F., Jarvinen, K., Vercauteren, F., & Verbauwhede, I. (2019). Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*, (pp. 387–398). IEEE.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, *26*(5), 1484–1509.

Su, Y., Yang, B.-L., Yang, C., Yang, Z.-P., & Liu, Y.-W. (2022). A highly unified reconfigurable multicore architecture to speed up ntt/intt for homomorphic polynomial multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *30*(8), 993–1006.

Tan, W., Chiu, S.-W., Wang, A., Lao, Y., & Parhi, K. K. (2023). Parentt: Low-latency parallel residue number system and ntt-based long polynomial modular multiplication for homomorphic encryption.

Tan, W., Wang, A., Lao, Y., Zhang, X., & Parhi, K. K. (2021). Pipelined high-throughput ntt architecture for lattice-based cryptography. In *2021 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, (pp. 1–4).

Wang, C. & Gao, M. (2023). Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, (pp. 1–9).

Yang, S., Wang, J., Yang, B., Zhang, F., & Yang, C. (2023). Reconfigurable multicore array architecture and mapping method for rns-based homomorphic encryption. In *AEU - International Journal of Electronics and Communications*, volume 161.

Yang, Y., Kuppannagari, S. R., Kannan, R., & Prasanna, V. K. (2022). Nttgen: a framework for generating low latency ntt implementations on fpga. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, (pp. 30–39). Association for Computing Machinery.

Ye, T., Yang, Y., Kuppannagari, S. R., Kannan, R., & Prasanna, V. K. (2021). Fpga acceleration of number theoretic transform. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*, (pp. 98–117)., Berlin, Heidelberg. Springer-Verlag.

Ye, Z., Cheung, R. C. C., & Huang, K. (2022). Pipentt: A pipelined number theoretic transform architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *69*, 4068–4072.