TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Accelerating Fully Homomorphic Encryption (FHE) schemes with FPGAs

Author:
Georgios AGORITSIS

Thesis Committee:
Prof. Sotirios IOANNIDIS
Prof. Apostolos DOLLAS
Prof. Georgios KARYSTINOS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

September 13, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Accelerating Fully Homomorphic Encryption (FHE) schemes with FPGAs

by Georgios AGORITSIS

This thesis delves into the exploration of FPGA acceleration possibilities for Fully Homomorphic Encryption (FHE), focusing on the OpenFHE library. Following a software profiling procedure, the thesis centers its attention on the Number Theoretic Transform (NTT) as a key acceleration point and investigates state-of-the-art optimization techniques from the existing literature. The chosen platform is the Xilinx Alveo U50 FPGA card, with the OpenFHE library as the foundation. The study demonstrates how a hardware-based project designed for the Alveo U50 card can be integrated into OpenFHE. Although this research does not introduce a novel acceleration method, it primarily focuses on showcasing the integration of an application developed in Xilinx Vitis IDE with OpenFHE. An essential contribution is the provision of a functional NTT implementation that supports various NTT sizes, complementing OpenFHE. The implementation encompasses an in-place forward NTT with the Cooley-Tukey butterfly and Harvey's modular multiplication optimization, utilizing High-Level Synthesis (HLS). This work stands as the first hardware accelerator for OpenFHE, deployed on a Xilinx FPGA, supporting up to 2^{18}-point Number Theoretic Transforms without FPGA reconfiguration. The accelerator is seamlessly integrated with OpenFHE as a hardware component. While the project does not yield an immediate speedup for OpenFHE, it lays a solid foundation for future acceleration endeavors. Future work directions and optimization suggestions are identified and conclude the thesis.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Accelerating Fully Homomorphic Encryption (FHE) schemes with FPGAs

by Georgios AGORITSIS

Αυτή η εργασία διερευνά τη δυνατότητα χρήσης Αναδιατασσόμενης Λογικής (FPGA) για την επιτάχυνση της Πλήρους Ομομορφικής Κρυπτογράφησης (FHE), εστιάζοντας στη βιβλιοθήκη OpenFHE. Μετά από μια software profiling διαδικασία, η εργασία επικεντρώνεται στο Number Theoretic Transform (NTT) ως κύριο στόχο επιτάχυνσης και διερευνά την υφιστάμενη βιβλιογραφία για υπάρχουσες τεχνικές βελτιστοποίησης. Ως πλατφόρμα υλοποίησης επιλέγεται η FPGA κάρτα Xilinx Alveo U50. Η μελέτη δείχνει τον τρόπο με τον οποίο η υλοποίηση ενός αλγορίθμου στην κάρτα Alveo U50 μπορεί να ενσωματωθεί στο περιβάλλον της OpenFHE. Παρόλο που δεν παρουσιάζεται μια νέα μέθοδος επιτάχυνσης, η εργασία επικεντρώνεται κυρίως στην ενσωμάτωση μιας εφαρμογής που αναπτύχθηκε στο περιβάλλον του Xilinx Vitis IDE με τη βιβλιοθήκη OpenFHE. Στα πλαίσια της εργασίας παρέχεται μια πλήρως λειτουργική υλοποίηση του NTT που είναι ισοδύναμη με την υπάρχουσα υλοποίηση του NTT από την OpenFHE. Η υλοποίηση γίνεται σε High-Level Synthesis (HLS) και περιλαμβάνει έναν in-place ευθύ NTT μετασχηματισμό, ο οποίος χρησιμοποιεί την πεταλούδα των Cooley και Tukey, καθώς και την βελτιστοποίηση του modular multiplication από τον D. Harvey. Η παρούσα εργασία περιλαμβάνει τον πρώτο επιταχυντή σε αναδιατασσόμενη λογική για την OpenFHE, που εγκαθίσταται σε μια Xilinx FPGA, υποστηρίζει NTT μεγέθους έως και 2^{18}, χωρίς επαναπρογραμματισμό της FPGA και ενσωματώνεται απρόσκοπτα στο περιβάλλον της βιβλιοθήκης. Αν και δεν καταφέρνει να επιταχύνει τη βιβλιοθήκη, θέτει τη βάση για μελλοντικές προσπάθειες επιτάχυνσης. Τέλος, η εργασία ολοκληρώνεται με την πρόταση ιδεών βελτιστοποίησης και κατευθύνσεων για μελλοντικές εργασίες.

Acknowledgements

First and foremost, I would like to express my gratitude to Prof. Sotirios Ioannidis, my supervisor, for enabling me to work on this thesis and for the unwavering trust he placed in me. I would also like to thank Andreas Brokalakis for his guidance and all his support and availability during the past months. His understanding and contribution were substantial. Special thanks to Prof. Apostolos Dollas and Prof. Georgios Karystinos for being members of the thesis committee and for assessing my work. Last but not least, I would like to thank my family and my friends for the support and encouragement throughout my studies. Their presence, feedback and contribution have shaped me to this day.

Georgios Agoritsis
Chania 2023

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CC	**C**lock **C**ycle
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
CRT	**C**hinese **R**emainder **T**heorem
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
DTT	**D**ata **T**ransfer **T**ime
FF	**F**lip **F**lops
FHE	**F**ully **H**omomorphic **E**ncryption
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
KL	**K**ernel **L**atency
KCT	**K**ernel **C**omputation **T**ime
LUT	**L**ook **U**p **T**able
NTT	**N**umber **T**heoretic **T**ransform
TFET	**T**otal **F**unction **E**xecution **T**ime
PL	**P**rogrammable **L**ogic
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SRL	**S**hift **R**egister **L**ogic
URAM	**U**ltra **R**andom **A**ccess **M**emory

Dedicated to my family and friends…

Chapter 1

Introduction

1.1 Motivation

Cryptography plays a vital role in safeguarding the security and confidentiality of sensitive data across diverse applications. One of the most groundbreaking advancements in cryptography is Fully Homomorphic Encryption (FHE) [12], an encryption scheme that allows users to perform computations on encrypted data without decrypting it. This remarkable property opens up a wide range of possibilities for secure computation in scenarios where privacy is of utmost importance.

The history of FHE dates back to the early 1970s when researchers began exploring the possibility of performing computations on encrypted data. However, it wasn't until Craig Gentry's groundbreaking work in 2009 [24] that FHE became a practical reality. Gentry introduced the concept of bootstrapping, which enabled the evaluation of arbitrary circuits on encrypted data, thus overcoming the limitations of earlier attempts. His work opened up new avenues for secure data processing and sparked widespread interest in FHE research and development.

FHE finds application in numerous domains where data privacy is paramount. Industries such as healthcare, finance, cloud computing, and machine learning can benefit from secure data processing without compromising confidentiality. FHE enables secure outsourcing of computations, confidential data sharing, and privacy-preserving machine learning algorithms.

Several FHE schemes have been developed over the years, including the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [8], the Brakerski/ Fan- Vercauteren (FV) scheme [21], [7], and the Cheon-Kim-Kim-Song (CKKS) scheme [13]. Each scheme has its own strengths and is suited to different types of

applications and performance requirements. The underlying mathematical problem that powers most of FHE schemes is the Ring Learning With Errors (RLWE) problem [9]. RLWE provides a foundation for constructing secure and efficient FHE schemes, allowing computations to be performed on encrypted data while maintaining the security guarantees.

To make FHE practical and viable for real-world applications, software libraries implementing FHE have been developed. These libraries provide high-level abstractions and tools for working with FHE schemes, making it easier for developers to integrate FHE into their applications. Examples of existing FHE libraries include OpenFHE [2], Palisade [50] (the predecessor of OpenFHE), and Microsoft SEAL [56].

OpenFHE is a state-of-the-art open-source software library specifically designed to facilitate the use and development of FHE applications. It provides a comprehensive set of tools and functionalities for working with various FHE schemes, including key generation, encryption, and decryption operations, while it also complies with the Homomorphic Encryption post-quantum security standards [33]. OpenFHE aims to make FHE accessible and usable for developers, researchers, and practitioners, promoting collaboration and innovation in the field of secure data processing.

While software libraries like OpenFHE make the use of FHE schemes easily accessible to developers, performance issues remain as an adoption obstacle. FHE schemes encrypt plaintexts into high degree polynomials and perform computations over these polynomials, thus increasing the complexity of computations. Number Theoretic Transform (NTT) [38] has been proposed as a way to reduce the complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$ and has been widely adopted throughout the FHE schemes as a fundamental computational primitive. Indeed, by using performance analysis tools [34], we identified that NTT computations account for more than 30% of the overall computation time in typical scenarios.

In an effort to address the performance issues that plague the adoption of FHE schemes, this work conducts a feasibility study on using reconfigurable hardware accelerators to offload computationally-significant functions of the OpenFHE library. According to our performance measurements, a natural starting point is the offloading of the NTT computations. Our goal is to create hardware accelerated functions that are fully functional and equivalent to those included in the OpenFHE library (i.e. without limitations on specific data sizes or operations) and integrate them in the library in a manner that

developers who employ the OpenFHE library need not be aware of the complications or other issues related to using hardware accelerators. As such, we hope that our work can be used as a reference point for future works targeting OpenFHE library acceleration using Xilinx FPGAs.

In conclusion, Fully Homomorphic Encryption (FHE) has the potential to transform data privacy and security by enabling computations on encrypted data. With the development of schemes such as BGV, FV, and CKKS, FHE is finding applications in various domains. Existing software libraries, such as OpenFHE, provide valuable resources for FHE implementation and development. The integration of FPGA-based accelerators with the OpenFHE library can potentially open up new possibilities for accelerating FHE computations. This work is the first attempt of accelerating the NTT used in OpenFHE with the Xilinx FPGAs (more specifically Xilinx Alveo U50) without any functional limitations.

1.2 Scientific Contributions

We use Intel HEXL-FPGA [35] and the work of Mert et al. [40] as the starting point of our work. Our target platform is Xilinx Alveo U50 Data Center Accelerator Card. The goal of this thesis is to design a functional NTT accelerator on Alveo U50 and integrate it with OpenFHE [2]. Our work is the first attempt to accelerate OpenFHE using a Xilinx Alveo FPGA to the best of our knowledge and can be used for future work as a template to further optimize OpenFHE acceleration. In more details:

- We confirm that NTT can be a bottleneck operation in OpenFHE, due to multiple NTT transformations used in internal high-level functions. Note that the NTT is a critical component among all FHE high-level functions and is the starting point of any existing FHE accelerator.
- We provide an alternative way of integrating our design with OpenFHE library, instead of using the library's Hardware Abstraction Layer (HAL), to simplify hardware acceleration research on OpenFHE.
- We provide a functional and OpenFHE-equivalent NTT implementation supporting NTT sizes up to 2^{18} in Xilinx Vitis HLS, using and combining existing work for NTT in literature, in order to create a baseline for future research.

- We identify further optimization approaches and future work directions at the last chapters of this document.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** Chapter 2 provides the necessary theoretical background concepts related to the goal of this thesis.
- **Chapter 3 - Related Work:** Chapter 3 provides a brief overview of existing works in literature and the thesis approach.
- **Chapter 4 - Defining the Architecture:** Chapter 4 includes OpenFHE library software profiling results and presents the proposed accelerator's architecture.
- **Chapter 5 - FPGA Implementation:** Chapter 5 report on the FPGA-based design and implementation of the proposed accelerator.
- **Chapter 6 - Results:** Chapter 6 reports the performance and resource utilization results of the proposed FPGA design.
- **Chapter 7 - Conclusions:** Chapter 7 concludes the thesis and provides the final conclusions, alongside proposals for future work directions.

Chapter 2

Theoretical Background

2.1 Learning With Errors (LWE) and Lattice-based Cryptography

Learning With Errors (LWE) [9] is a lattice-based cryptographic problem that was introduced in 2005 and forms the basis of many homomorphic encryption schemes, including FHE. In the LWE problem, one must find a small integer secret vector hidden within a set of noisy linear equations. These equations introduce errors to make it computationally difficult to recover the secret vector from the noisy data.

The problem formulation is as follows:

$$c[i] = (a[i], a[i] * s + e[i]) \mod q \quad \text{for } i = 0, 1, 2, ...$$

where:

$c[i]$ is a ciphertext,

$a[i]$ is a random vector chosen uniformly from a discrete set,

s is the secret vector (small integer vector) that the adversary aims to find,

$e[i]$ is a small noise term,

q is a large modulus.

The noise term $e[i]$ is generated from a probability distribution, and it is what adds the "noise" to the equations. The LWE problem involves finding the secret vector s from a collection of such noisy equations.

LWE has a search version and a decision version:

- Search LWE problem: Given a set of computed pairs of points $(a[i], a[i] * s + e[i])$, find s.
- Decision LWE problem: Given a set of computed pairs of points $(a[i], a[i] * s + e[i])$ and some random points, determine the points that originate from s. The random points are generated using a uniform distribution.

The main challenge in the LWE problem is that the noise terms make it computationally difficult to directly recover the secret vector s. The security of LWE relies on the assumption that solving the system of equations and extracting the secret vector s is hard even for powerful adversaries.

LWE has been widely used to demonstrate the security of various cryptosystems. However, their efficiency is limited by the fact that their keys are matrices randomly generated over a small integer q in the ring Z_q, causing their dimension to increase linearly as security parameters grow. To address this efficiency concern, ring variants have been introduced.

In 2009, a variant of LWE called **Ring Learning with Errors (Ring-LWE)** was proposed as a solution [9]. RLWE extends the LWE problem to polynomial rings. Instead of using vectors, RLWE operates on polynomials within a ring, such as the polynomial ring modulo a power of two. The RLWE problem involves finding the secret polynomial amidst errors introduced through the polynomial ring.

One significant advantage of Ring-LWE over its LWE counterpart is the reduction in key size by a factor of n. For a message that would require several thousand bits to be secured using LWE, Ring-LWE accomplishes the same with only hundreds of bits. This makes Ring-LWE a more feasible and efficient system, especially for implementations with constrained computational resources. OpenFHE uses Ring-LWE as its main security guarantee.

2.2 Fully Homomorphic Encryption (FHE) and FHE Schemes

Fully Homomorphic Encryption (FHE), is an encryption scheme that allows users to perform computations on encrypted data without decrypting it. Whilst FHE was initially introduced in the 1970s, it was in 2009 that Craig Gentry

[24] proved that FHE can be used in practise. Starting from a Somewhat Homomorphic Encryption (SWHE) scheme, Gentry proved how it can be transformed in a FHE scheme using the bootstrapping technique [8]. In SWHE schemes one can evaluate a ciphertext in the form of a low-degree polynomial until the noise increases up to a point that any further operation on the ciphertext will make it undecryptable. In a FHE scheme, the ciphertext can be a polynomial of any size and boostrapping is used as a noise maintenance technique, to allow more computations on the encypted data.

Bootstrapping can be defined as a ciphertext refresh operation that reduces the generated noise into levels that allow more operations to be performed [66]. In schemes like CKKS, bootstrapping is performed in a periodic manner [10] and is a memory and compute intensive operation. In schemes like CGGI, bootstrapping is performed after each bit-wise operation. As bootstrapping can be a bottleneck operation for FHE, a great part of the community has focused in accelerating this high level function, and the main focus usually is the intensive memory access required by the algorithm. Following their existing work [10], the same research team recently proposed an FPGA accelerator that supports bootstrapping [1] by optimizing the required memory access pattern.

Following Gentry's work, multiple FHE schemes have been proposed in literature, the Brakerski-Gentry-Vaikuntanathan (BGV) [8], the Fan-Vercauteren (FV) [7], the Cheon-Kim-Kim-Song (CKKS) [13] and the Chillotti - Gama - Georgieva - Izabachene (CGGI) [14] schemes to name the most important. Choosing the right FHE scheme for an application depends on both the application (machine learning computations, encrypted queries in databases, etc.) and the data type used in the application (integer numbers, floating point numbers, etc.).

Grouping FHE schemes based on the supported data type, three classes of schemes are formed [2]. The first include the BGV and BFV schemes and support modular arithmetic over finite fields in the form of vectors of integers modulo a number. The second group include among others the CGGI scheme and support boolean circuits. The third and most recent group is the one that comprises of the CKKS scheme. In this scheme, vectors of real and complex numbers are supported, enabling approximate computations. All these groups, are based on the hardness of the RLWE problem [9], due to the noise added during homomorphic operations, such as encryption and key generation.

Plaintexts in FHE depend on the scheme used and can have the form of vectors of integers, vectors of real/complex numbers or boolean values. A parameter p, called the plaintext modulo, is also defined based on the worst multiplication scenario of the dataset. This parameter is later used to define the ciphertext modulo q, which is used internally in FHE operations. The bigger the ciphertext modulo, the more operations can be performed on the ciphertext without using bootstrapping. Ciphertexts are arrays of hundreds or thousands of integers in all schemes that map to polynomial coefficients, which means that plaintexts are encrypted into polynomials and the size of the polynomials is a power of two, typically in the range $[2^{10}, 2^{17}]$. In FHE, the basic mathematical structure used is a polynomial ring (denoted by R_q) of dimension $n = \phi(m)$, where m is the index of the $m - th$ cyclotomic polynomial, ϕ is Euler's totient function, and q is the coefficient modulus of the ring. n,m, and q are all positive integers. One can view this ring as the set of polynomials of degree less than n with integer coefficients in $\{-q/2, \ldots, q/2\}$. The nature of the ciphertexts pose a challenge for FHE on its own, due to the fact that polynomial arithmetic can be expensive as the polynomial size increases.

To tackle this challenge, multiple techniques are used. Regarding modular arithmetic, efficient modular reduction algorithms such as Barrett reduction [26] or Montgomery reduction [41] have been proposed. To accelerate the main bottleneck, the polynomial multiplication, the Number Theoretic Transform (NTT) [38] has been used. There is a great variety of NTT algorithm optimizations in the literature [39], [32], [53], [51]. Using the NTT, the complexity of polynomial multiplication is reduced from $O(n^2)$ to $O(n \log n)$. To further accelerate the NTT and its efficiency, a Residue Number System (RNS) representation is used on the initial polynomial [25]. This means, that multiple polynomials with smaller coefficients are calculated and NTTs can be performed in parallel. The set of RNS calculated polynomials represent the original polynomial. The main advantage of this approach is that it can bound arithmetic to be performed for example in 64 bits variables without overflow, regardless of the initial coefficient bit requirements (that can be up to multiple hundreds bits). All these approaches have been integrated in all state-of-the-art software libraries available.

2.3 Modular Arithmetic

A key component of lattice-based cryptography and hence FHE, is the modular arithmetic operations. Among those modular arithmetic operations, the modular multiplication is of specific interest. That is due to the fact that multiplication greatly increases the size of the result which often exceeds the target modulo, thus modular reduction will be applied more frequently during multiplication compared to modular addition. In addition, the Number Theoretic Transform (NTT), one of the main bottlenecks for FHE, utilizes modular multiplication to form its result. Consequently, modular reduction techniques are of specific interest for FHE, the most important of which include the Montgomery modular multiplication [43], the Barrett modular multiplication [26] and a recently proposed algorithm from Plantard [48]. Plantard's publication also includes an overview of a variety of modular reduction algorithms for unsigned arithmetic. Modular reduction algorithms for signed arithmetic have also been proposed [4]. Below, we present Montgomery (algorithm 1), Barrett (algorithm 2) and Plantard (algorithm 3) modular multiplication algorithms.

A common approach between these methods is using the fact that multiplications and divisions with powers of two are implemented using bit-shift operations in hardware. In addition, the $\mod$ operation can be viewed as a bit masking operation that maintains only the target bits of a word. Regarding Montgomery's method, one of the most used algorithms for modular multiplication [48], its result is scaled by the factor 2^{-n}. On the contrary, Barrett's method returns directly the correct result, but is slightly more expensive than Montgomery multiplication in terms of computational resources.

Algorithm 1 Montgomery Modular Multiplication

Require: X, Y $(0 < X, Y < q)$ as input values. q is an odd modulus $(q < 2^n)$.
R such that $R = (-P^{-1}) \pmod{2^n}$.
Ensure: $Z = XY2^{-n} \pmod{q}, (0 \leq Z \leq q)$.
function MONTGOMERYMODMUL(X, Y, R, q, n)
$Z \leftarrow (XY + q \cdot (XYR \pmod{2^n}))/2^n$
if $Z \geq q$ **then**
$Z \leftarrow Z - q$
return Z

Plantard's method is in fact a special case of the Montgomery modular multiplication [4], which operates on $2n$-bit words and computes the multiplication result in a n-bit word, taking advantage of the property that $2n$-bit word

Algorithm 2 Barrett Modular Multiplication

Require: X, Y $(0 < X, Y < q)$ as input values. q is a modulus. R such that $R = \lfloor 2^{2n}/q \rfloor$.
Ensure: $Z = XY \pmod{q}, (0 \leq Z \leq q)$.
  function BARRETTMODMUL(X, Y, R, q, n)
    $Z \leftarrow (XY - q \cdot (((XY)/2^{n-1})R)/2^{n+1}))$
    if $Z \geq 2q$ **then**
      $Z \leftarrow Z - 2q$
    else if $Z \geq q$ **then**
      $Z \leftarrow Z - q$
  return Z

multiplication can be viewed as a modular multiplication modulo 2^{2n}. In Plantard modular multiplication, one less multiplication is performed compared to Montgomery's method.

Algorithm 3 Plantard Modular Multiplication

Require: X, Y $(0 < X, Y < q)$ as input values. q is a modulus. R such that $R = q^{-1} \pmod{2^{2n}}$.
Ensure: $Z = XY(-2^{-2n}) \pmod{q}, (0 \leq Z \leq q)$.
  function PLANTARDMODMUL(X, Y, R, q, n)
    $Z \leftarrow \lfloor (q \cdot (\lfloor XYR \pmod{2^{2n}}/2^n \rfloor + 1)/2^n \rfloor$
    if $Z = q$ **then**
      $Z \leftarrow Z - q$
  return Z

2.4 Residue Number System (RNS)

When polynomial multiplication is studied, the Residue Number System (RNS) representation is the starting point of any acceleration attempt. An introduction to the properties of the RNS can be found at [22]. In the RNS domain, a polynomial with coefficients modulo Q is decomposed into multiple polynomials with coefficients modulo q_i for $i = 0, 1, 2, ...$ In other words, the moduli q_i are chosen so as $Q = \prod(q_i)$ for $i = 0, 1, 2, ...$ The foundation of the RNS system is the Chinese Remainder Theorem [45]. Operations such as addition, subtraction and multiplication can be performed for every q_i in parallel, as:

$$[c_1, c_2, c_3] = [(a_1 \oplus b_1) mod q_1, (a_2 \oplus b_2) mod q_2, (a_3 \oplus b_3) mod q_3]$$

where $\oplus$ is one of the following operations $[+, -, *]$.

Choosing the smaller moduli q_i values is also important and depends on the effective range of numbers that the initial modulo Q covers. A great effort has been put in studying what are the best candidates for a moduli set selection. [45] includes a summary of different moduli sets found in the literature up to 2011. While more recent works have been published, we do not focus more on it, as choosing a moduli set was out of the scope of this thesis.

2.5 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a special case of the Discrete Fourier Transform (DFT) over finite fields [49] and it is also used to efficiently compute the polynomial multiplication between high degree polynomials [38]. That is due to the fact that using the NTT the complexity of polynomial multiplication drops from $O(n^2)$ to $O(n \log n)$. A comprehensive review of NTT and state-of-the-art optimizations can be found at [55]. Prior to explaining the NTT, we describe the polynomial multiplication over the rings $\mathbb{Z}_q[X], \mathbb{Z}_q[X]/(X^n-1), \mathbb{Z}_q[X]/(X^n+1)$ and then we will also define the primitive $n-th$ root of unity.

Let A(x) and B(x) be two polynomials of degree $n-1$ in the ring $R_q = \mathbb{Z}_q[X]$:

$$A(x) = a_0 + a_1 x + \ldots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1}$$
$$B(x) = b_0 + b_1 x + \ldots + b_{n-2} x^{n-2} + b_{n-1} x^{n-1}$$

The polynomial multiplication is defined as:

$$C(X) = A(X) \cdot B(X) = \sum_{k=0}^{2n-2} c_k x^k$$

where $c_k = \sum_{i=0}^{k} a_i b_{k-i} \pmod q \in \mathbb{Z}_q[X]$. Thus, polynomial multiplication can also be viewed as the linear convolution between the coefficients of A(x) and B(x). In addition to the linear convolution, the cyclic convolution and the negacyclic convolution should also be defined.

For the cyclic convolution, consider that A(x) and B(x) are again two polynomials of the same degree, but in the $R_q = \mathbb{Z}_q[X]/(X^n-1)$ ring. Then, the

polynomial multiplication is defined as:

$$C(X) = A(X) \cdot B(X) = \sum_{k=0}^{n-1} c_k x^k$$

where $c_k = \sum_{i=0}^{k} a_i b_{k-i} + \sum_{i=k+1}^{n-1} a_i b_{k+n-i} \pmod{q} \in \mathbb{Z}_q[X]/(X^n - 1)$.

The negacyclic convolution for A(x) and B(x) is defined in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ instead.

$$C(X) = A(X) \cdot B(X) = \sum_{k=0}^{n-1} c_k x^k$$

where $c_k = \sum_{i=0}^{k} a_i b_{k-i} - \sum_{i=k+1}^{n-1} a_i b_{k+n-i} \pmod{q} \in \mathbb{Z}_q[X]/(X^n + 1)$.

To form the cyclic or the negacyclic convolution of C(x), where C(x) is the result of the linear convolution, one has to perform a modular reduction using $(x^n - 1)$ or $(x^n + 1)$ respectively:

$$C(x)_{cyclic} = C(x) \mod (x^n - 1)$$
$$C(x)_{negacyclic} = C(x) \mod (x^n + 1)$$

Regarding the primitive $n - th$ root of unity, we follow the definition provided by [55]. Note that a ring can have multiple primitive $n - th$ roots of unity.

Definition 2.1: Let $\mathbb{Z}_q$ be an integer ring modulo q, and $n - 1$ the polynomial degree of the aforementioned polynomials A(x) and B(x). Define ω as primitive $n - th$ root of unity in $\mathbb{Z}_q$ if and only if:

$$\omega^n \equiv 1 \pmod{q}$$

and

$$\omega^k \not\equiv 1 \pmod{q}$$

for $k < n$.

For the forward cyclic NTT, let $a(x) = a_0 + a_1 x + \ldots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1} \in \mathbb{Z}_q[X]/(X^n - 1)$ be our polynomial of degree $n - 1$ and ω a primitive $n - th$ root of unity in $\mathbb{Z}_q[X]$.

$$\hat{a} = NTT(a) \text{ where } \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \mod q \quad \text{for } i = 0, 1, \ldots, n-1$$

The inverse cyclic NTT can be defined as:

$$a = INTT(\hat{a}) \text{ where } a_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij} \mod q \quad \text{for } i = 0, 1, \ldots, n-1$$

The polynomial multiplication of A(x) and B(x) using the NTT is defined as:

$$C(x) = INTT(NTT(A(x)) \cdot NTT(B(x))) \in \mathbb{Z}_q[X]/(X^n - 1)$$

where $\cdot$ represents the coefficient-wise multiplication between each coefficient of the initial polynomials. Note that the result of the cyclic transforms is in $\mathbb{Z}_q[X]/(X^n - 1)$, while FHE operates in $\mathbb{Z}_q[X]/(X^n + 1)$, mentioned as R_q in section 2.2. In order for the polynomial multiplication to produce a result in the $\mathbb{Z}_q[X]/(X^n + 1)$ ring, a negacyclic NTT transform has to be used.

In the negacyclic NTT the primitive $2n - th$ root of unity in $\mathbb{Z}_q$ must be defined as:

$$\psi^{2n} \equiv 1 \pmod{q}$$

The following property must also be true:

$$\psi^2 \equiv \omega \pmod{q}$$

To perform the negacyclic polynomial multiplication, one has to multiply each coefficient of the initial polynomials A(x) and B(x) with a power of ψ. Powers of ψ are also called twiddle factors. The modified polynomials are:

$$A'(x) = a_0 + \psi a_1 x + \ldots + \psi^{n-2} a_{n-2} x^{n-2} + \psi^{n-1} a_{n-1} x^{n-1}$$
$$B'(x) = b_0 + \psi b_1 x + \ldots + \psi^{n-2} b_{n-2} x^{n-2} + \psi^{n-1} b_{n-1} x^{n-1}$$

and the negacyclic polynomial multiplication is now computed as:

$$C(x) = (1, \psi^{-1}, \ldots, \psi^{-(n-1)}) \cdot INTT(NTT(A'(x)) \cdot NTT(B'(x)))$$

where $C(x) \in \mathbb{Z}_q[X]/(X^n+1)$ and $\cdot$ represents the coefficient-wise multiplication between each coefficient of the polynomials and between the multiplied coefficients and the inverse powers of ψ. In other words, the negacyclic forward NTT can be defined as:

$$\hat{A} = NTT^{\psi}(A) \text{ where } \hat{a_i} = \sum_{j=0}^{n-1} a_j \psi^i \omega^{ij} \mod q \quad \text{for } i = 0, 1, \ldots, n-1$$

and the inverse negacyclic NTT as:

$$A = INTT^{\psi}(\hat{A}) \text{ where } a_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a_j} \psi^{-i} \omega^{-ij} \mod q \quad \text{for } i = 0, 1, \ldots, n-1$$

Following the naive approach, multiple optimizations have been proposed in literature [39]. To avoid multiplying each input coefficient with the appropriate power of ψ and then performing the NTT, Roy et al. [53] proposed that powers of ψ can be merged with the powers of ω as $\psi^i \omega^{ij} = \psi^{2ij+i}$, using the property $\psi^2 \equiv \omega \pmod q$. The negacyclic NTT can now be defined as:

$$\hat{A} = NTT^{\psi}(A) \text{ where } \hat{a_i} = \sum_{j=0}^{n-1} a_j \psi^{2ij+i} \mod q \quad \text{for } i = 0, 1, \ldots, n-1$$

A similar optimization is available due to Pöppelmann et al. [51] for the inverse negacyclic NTT, which can now be defined as:

$$A = INTT^{\psi}(\hat{A}) \text{ where } a_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a_j} \psi^{-(2ij+i)} \mod q \quad \text{for } i = 0, 1, \ldots, n-1$$

To further accelerate the NTT, a divide and conquer technique was proposed by Cooley and Tukey [16] in 1965 and is called Cooley-Tukey (CT) butterfly (figure 2.1) in literature. The turning point of their work was using the same power of ψ in two calculations, thus reducing the number of calculations in the entire NTT. Terms like $(A - \psi^k B)$ and $(A + \psi^k B)$ are calculated once and used by other butterfly units as the transform progresses. Using the CT butterfly is possible as long as n is a power of two for a n-point NTT.

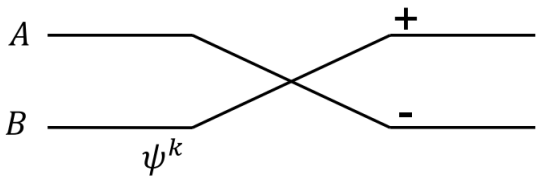

FIGURE 2.1: Cooley-Tukey Butterfly.

A similar technique, mainly applied to the inverse NTT, was published by Gentleman and Sande [23] in 1966 and is called Gentleman-Sande (GS) butterfly (figure 2.2) in literature.

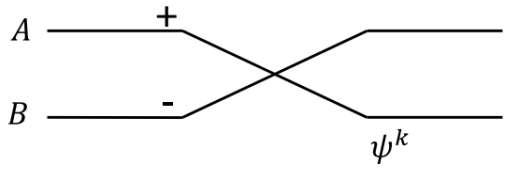

FIGURE 2.2: Gentleman-Sande Butterfly.

Using many butterflies, one can form high-degree NTT and INTT transforms. We refer the reader to figures 2.3 and 2.4, where a 8-point forward CT-based NTT and a 8-point GS-based INTT are displayed. Note that both transforms use the powers of ψ in the way proposed by Roy et al. [53] and Pöppelmann et al. [51].

An important remark on the below figures is the order of the input and output coefficients. One can notice that in the case of CT-based forward NTT the input is in normal order, while the output's order is mixed. On the contrary, the GS-based inverse NTT has its input in mixed order and the output in normal order. In literature, researchers refer to this mixed order as bit-reversed order [6] [55] [38]. The bit-reversed order applies to the index i used to access each coefficient in the coefficient vector $[a_0, a_1, \ldots, a_{n-1}]$. For an index i with binary representation $0bi_0i_1 \ldots i_{log_2(n)}$ this process can be defined as:

$$BitReverse(0bi_0i_1 \ldots i_{log_2(n)}) = 0bi_{log_2(n)}i_{log_2(n)-1} \ldots i_1i_0$$

For example, for the 8-point NTT transform ($n = 8$), index 0 in binary is $0b000$ and in bit-reversed representation is $0b000 = 0d0$, while index 1 in binary representation is $0b001$ and in bit-reversed representation is $0b100 = 0d4$, and so on.

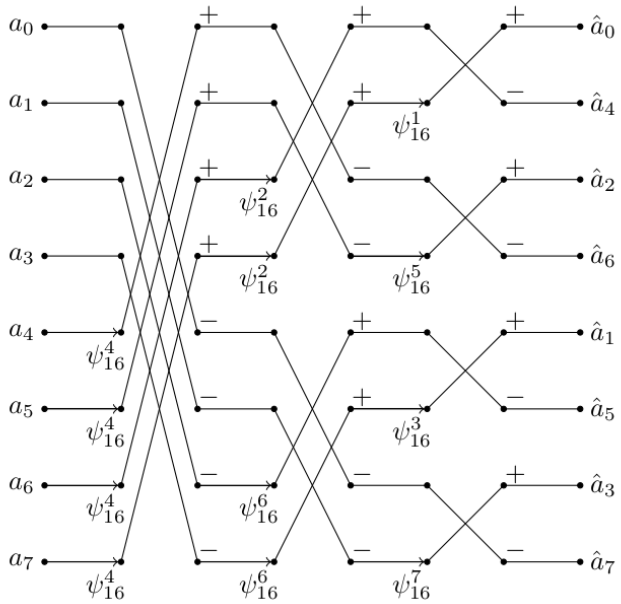

FIGURE 2.3: Forward 8-point NTT with Cooley-Tukey (CT) butterfly

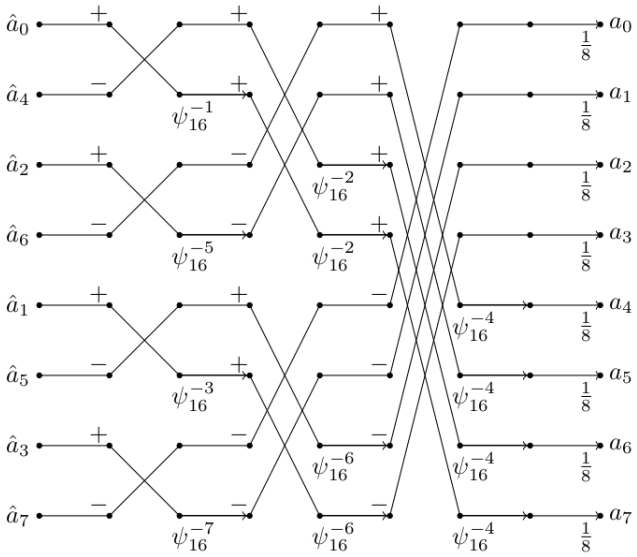

FIGURE 2.4: Inverse 8-point NTT with Gentleman-Sande (GS) butterfly

Combining the CT-based forward NTT and the GS-based inverse NTT, we manage to avoid reordering the output coefficients, as the GS INTT restores the bit-reversed ordered coefficients that the CT NTT produced [39]. We take the Cooley-Tukey Radix-2 NTT transform and the Gentleman-Sande Radix-2

inverse transform from [39] and present it in algorithms 4 and 5. We modify the presentation of the algorithms slightly, so as to follow the conventions used in OpenFHE library. Both of CT and GS-based NTT/INTTs can be found in their reverse forms (from bit-reversed ordered input to normal ordered output and normal ordered input to bit-reversed ordered output respectively). However, we omit presenting them here, as both of these versions are out of the scope of this work.

Algorithm 4 In-Place Cooley-Tukey Forward NTT

Require: $a = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$ in normal-ordering. n is a power of two. q is a prime such that $q \equiv 1 \pmod{2n}$. $\psi_{rev} \in \mathbb{Z}_q^n$ are the powers of ψ in bit-reversed order.
Ensure: $a \leftarrow NTT^{\psi}(a)$ in bit-reversed order.

```
function CooleyTukeyRadix2NTT(a, ψ_rev, n, q)
    t ← n ≫ 1
    logt1 ← log2(t)
    for m = 1; m<n; m=2*m do
        for i = 0; i<m; i++ do
            j1 ← i ≪ logt1
            j2 ← j1 + t
            indexOmega ← m + i
            omega ← ψ_rev[indexOmega]
            for indexLo = j1; indexLo<j2; ++indexLo do
                indexHi ← indexLo + t
                X0 ← a[indexLo]
                X1 ← a[indexHi]
                                              ▷ Cooley-Tukey Butterfly
                a[indexLo] ← X0 + omega · X1 (mod q)
                a[indexHi] ← X0 − omega · X1 (mod q)
        t ← t ≫ 1
        logt1 ← logt1 − 1
    return
```

Further optimizing the butterflies has also been studied. Harvey in his paper [32] utilizes Shoup's butterfly method used in NTL library [58] to optimize the CT and GS butterflies. This technique uses precomputed values based on the powers of ψ, so as to accelerate the modular multiplication used in each butterfly and reduce the modular correction steps. Algorithm 6 below presents the optimized CT butterfly. Note that in Harvey's paper algorithm 6 is presented as the inverse butterfly, while in our case we use this version for the forward transform. To efficiently use this approach, the powers of ψ and the preconditioned powers of ψ should have been precomputed and provided as input to the algorithm.

Algorithm 5 In-Place Gentleman-Sande Inverse NTT

Require: $a = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$ in bit-reversed ordering. n is a power of two. q is a prime such that $q \equiv 1 \pmod{2n}$. $\psi_{rev}^{-1} \in \mathbb{Z}_q^n$ are the powers of ψ^{-1} in bit-reversed order.
Ensure: $a \leftarrow INTT^{\psi}(a)$ in normal ordering.

```
function GentlemanSandeRadix2INTT(a, ψ⁻¹_rev, n, q)
    t ← 1
    logt1 ← 1
    for m = n; m>1; m=m/2 do
        for i = 0; i<m; i++ do
            j1 ← i ≪ logt1
            j2 ← j1 + t
            indexOmega ← m + i
            omega ← ψ⁻¹_rev[indexOmega]
            for indexLo = j1; indexLo<j2; ++indexLo do
                indexHi ← indexLo + t
                X0 ← a[indexLo]
                X1 ← a[indexHi]
                                          ▷ Gentleman-Sande Butterfly
                a[indexLo] ← X0 + X1 (mod q)
                a[indexHi] ← (X0 − X1) · omega (mod q)
        t ← t ≪ 1
        logt1 ← logt1 + 1
    for i = 0; i<n; i++ do
        a[i] ← a_i · n^{-1} (mod q)
    return
```

Algorithm 6 Harvey/Shoup Cooley-Tukey Butterfly

Require: X, Y $(0 < X, Y < 4 * q)$ as input values and q as modulus.
A root of unity W such that $0 < W < q$ and $W' = \lfloor W\beta/q \rfloor, (0 < W' < \beta)$, where β is the processor word bit-size (e.g. $\beta = 64$ for 64-bit word-size).
Ensure: $X' = X + WY \pmod{q}$, $Y' = X - WY \pmod{q}, (0 \leq X', Y' \leq 4q)$.

```
function HarveyCTButterfly(X, Y, W, W', q)
    if X ≥ 2q then
        X ← X − 2q
    Q ← ⌊W'Y/β⌋
    T ← WY − Qq (mod β)
    X' ← X + T
    Y' ← X − T + 2q
return X',Y'
```

2.6 OpenFHE Open-source Library

Moving from the theoretical investigation of FHE to the practical one, there is a great variety of software libraries implementing FHE schemes. These libraries provide high-level abstractions and APIs for working with FHE schemes, making it easier for developers to integrate FHE into their applications. Among all available libraries, we choose to work with OpenFHE [2], a C++ open-source library implementing all major FHE schemes (BGV, BFV, CKKS, CGGI and more). OpenFHE follows the design of the PALISADE library [50], merged with selected capabilities of HElib [31], HEAAN [11] and FHEW [18]. The latest stable version release is 1.0.4.

OpenFHE also complies with the Homomorphic Encryption post-quantum security standards available in [33] and currently supports only RNS variants of each scheme. OpenFHE is also the first FHE library that supports OpenMP multithreading. The main advantage of these libraries is the fact that by providing a non-scheme-specific API, the user does not need to understand the implementation of the internal low level functions, neither worry for their performance.

When using OpenFHE, the user should choose the plaintext modulo P for the input data of the FHE application. The encrypted data operate on a different modulus Q which is larger than the modulus P. The plaintext modulus P and the ciphertext modulus Q serve different purposes in the homomorphic encryption scheme. In more details:

- Plaintext Modulus (P): The plaintext modulus P is a parameter that determines the range of values that can be encrypted and operated on in the plaintext space. It defines the maximum value that can be represented in the plaintext domain. Any plaintext value to be encrypted must be smaller than the plaintext modulus P. The choice of P depends on the application requirements and the desired precision or granularity of computations.
- Ciphertext Modulus (Q): The ciphertext modulus Q is a parameter that defines the size of the ciphertext space and affects the security level and noise growth during homomorphic operations. It determines the maximum value of the encrypted ciphertexts. The choice of Q is typically larger than the plaintext modulus P to accommodate the intermediate results and noise introduced during homomorphic operations.

When performing homomorphic computations, the ciphertexts are manipulated using homomorphic operations (e.g., addition, multiplication) without decrypting them. The operations are performed on the encrypted data, preserving the confidentiality of the underlying plaintext. During the homomorphic operations, the ciphertext modulus Q is critical as it affects the noise growth. As the homomorphic operations are executed, the noise in the ciphertext accumulates, potentially leading to decryption errors or reduced security. Hence, choosing an appropriate ciphertext modulus Q is important to balance security and efficiency. In summary, the plaintext modulus P determines the maximum value that can be encrypted in the plaintext space, while the ciphertext modulus Q affects the security level and noise growth during homomorphic operations in the encrypted ciphertext space.

A high-level view of the steps one needs to follow so as to use OpenFHE's API in the target application is:

1. Select FHE parameters (plaintext modulo, etc.)
2. Generate public-private key pair
3. Encode plaintext data
4. Encrypt
5. Perform computations
6. Decrypt (optional)
7. Decode (optional)

Another advantage of OpenFHE is the very active community of researchers working on it. A forum is available and users can submit questions and discuss on FHE related topics, facilitating further someone's introduction to the library. OpenFHE also includes all performance optimization features mentioned in previous sections, such as the modular reduction techniques, the RNS system and the NTT. Regarding the modular reduction, the library internally implements the Barrett and Montgomery reduction, which are mainly used in modular multiplication operations.

In section 2.2, the basic mathematical structure of FHE was presented, the polynomial quotient ring R_q of dimension $n = \phi(m)$, where m is the index of the $m - th$ cyclotomic polynomial, ϕ is Euler's totient function, and Q or q is the coefficient modulus of the ring. OpenFHE uses a specific quotient ring that is defined as $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. This ring can be also

viewed as the set of polynomials of degree less than n with integer coefficients in $\{-q/2, \ldots, q/2\}$. OpenFHE defines three data representation types: Poly, NativePoly and DCRTPoly. Based on the definitions provided in the OpenFHE documentation:

- A Poly is a single-CRT representation using big integer types (e.g. 128 bit) as coefficients, and supporting a large modulus q.
- A NativePoly is a single-CRT representation using native integer types, which limits the size of the coefficients and the modulus q to 64 bits.
- A DCRTPoly is a double-CRT representation.

This means that while Poly uses a single large modulus q, DCRTPoly uses a set of smaller moduli. Hence, computations with DCRTPoly data type is much faster than with Poly data types, because DCRTPoly operations can fit into the native bitwidths of commodity processors.

OpenFHE internally represents polynomial ring elements as being either in COEFFICIENT or EVALUATION format. It is generally computationally less expensive to carry on all operations in the evaluation form. Multiplication is currently implemented only in the EVALUATION format. The coefficient representation of a is a vector containing the coefficients in normal order:

$$a = [a_0, a_1, ..., a_{\phi(m)-1}] \quad \in \mathbb{Z}_q$$

By selecting m and q as two integers in a way that the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ contains a primitive $m - th$ root of unity (denoted by ω), we construct vector b for the evaluation representation as:

$$b = [b_0, b_1, ...] \quad \in \mathbb{Z}_q$$

where

$$b_i = a(\omega^i) \pmod{q} \quad \text{for} \quad i \quad \in \mathbb{Z}_q^*$$

As a note for all i we have the equality $(a \mod (X - \omega^i)) = a(\omega^i) = b_i$, meaning that the evaluation representation of a is just a Chinese Remainder Theorem-based polynomial representation [25].

In RLWE-based FHE schemes (BGV, BFV and CKKS), the coefficient modulus q is a very large integer and can be a few hundred bits in size. That means we would need to use multi-precision arithmetic to perform polynomial arithmetic in R_q. Multi-precision arithmetic is known to be inefficient due to its

serial nature. DCRT helps us convert these multi-precision arithmetic operations into native arithmetic that uses processor-register-size operations. DCRTPoly can be in both coefficient and evaluation format. Having a polynomial in R_q, the procedure to create a DCRTPoly starts by decomposing the modulo q as a product of $(t+1)$ small primes $(p_0, p_1, ..., p_t)$. The size of each small prime is typically 32, 40, 50 and up to 60 bits. This allows us to factor our polynomial into smaller polynomials using the Chinese Remainder Theorem. The result is a matrix of dimensions $(t+1) \times n$ where n is the ring size or the polynomial degree. The coefficients of this matrix are native integers and in coefficient format.

Having two polynomials, coefficient-wise modular addition or subtraction can be performed directly in their coefficient representation as:

$$C_{ij} = A_{ij} \oplus B_{ij} \pmod{p_i}$$

where $\oplus$ can be $[+, -]$ and $i = 0, 1, ..., t$.

However, polynomial multiplication between two polynomials in R_q is currently supported only in evaluation format. Switching formats requires using the Number Theoretic Transform (NTT). NTT and Inverse NTT operations take $O(n \log n)$ time using current best known algorithms, where n is the ring dimension. Applying the NTT transform into our $(t+1) \times n$ matrix, yields another matrix of dimension $(t+1) \times n$. The key-point here is that the schoolbook polynomial multiplication can now be substituted by a coefficient-wise modular multiplication:

$$C_{ij} = A_{ij} * B_{ij} \pmod{p_i}$$

where $i = 0, 1, ..., t$.

We remind that addition and subtraction can also be computed in the evaluation format in the same way as in the coefficient format. The conversion back from evaluation format to coefficient format may not always be needed, and we can do further operations in the evaluation representation.

In OpenFHE, in-place and out-of-place Number Theoretic Transform variants are available. The library implements the negacyclic NTT transform in order to perform transformations within the $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ ring. For the forward NTT transform, the Cooley-Tukey radix-2 butterfly with normal-ordered input and bit-reversed output is used. To avoid, reordering the

output, the Gentleman-Sande radix-2 butterfly with bit-reversed input and normal-ordered output is used for the inverse NTT. The library also precomputes the necessary roots-of-unity for each modulo q_i and stores it, so as to avoid re-computing the roots if the same modulo is used in the same application run. This approach is also followed for the preconditioned roots of unity factors that are used for Harvey's efficient modular multiplication algorithm.

OpenFHE internally sets the necessary parameters. For the negacyclic Number Theoretic Transform (NTT), the library sets a modulo Q that is decomposed into a number of smaller moduli q_i and a ring of a constant size. Selecting Q and q_i as prime numbers, the library ensures that the modular arithmetic system satisfies certain desirable properties. Prime numbers have the property that every non-zero element is invertible modulo Q, meaning that there exists a unique multiplicative inverse for each non-zero element in the ring. This property is crucial for performing division and inversion operations in the polynomial quotient ring. The size of the ring, which is usually a power of two and determined by the parameter Q, should be large enough to accommodate the input data and to preserve the desired precision in the transformed domain. This choice ensures that the polynomial remains within the ring during the NTT computation.

Overall, OpenFHE is in the center of attention in the Fully Homomorphic Encryption community, as it implements all available FHE schemes and integrates most of available optimizations both in the algorithmic level (e.g. NTT optimizations) and in the software implementation level (e.g. deploying acceleration techniques such as multithreading). OpenFHE also includes benchmarks for all important low-level and high-level functions, under the benchmark folder of its source files. Researchers aiming to compare OpenFHE performance with other existing libraries, using some reference benchmarks, can you use the work of Gouert et al. [28] by modifying slightly the PALISADE's benchmarks provided. Instructions of how to migrate a project from PALISADE to OpenFHE are provided in OpenFHE's documentation.

Chapter 3

Related Work

3.1 Accelerating FHE Schemes

While the usefulness of FHE schemes is undeniable and consolidated efforts like the OpenFHE library are a step towards their wider adoption, there is a significant barrier that hinders their practical use: their real-world performance. When measuring the performance of FHE compute, a comparison is typically made against equivalent computations on the plain text version of the data. The encryption methods to enable FHE can increase the size of the data by 100-1000x, and then compute on that data is 10000x to 1 million times slower than conventional compute. To put this into perspective, one second of compute on the raw data can take from 3 hours to 12 days.

As such, FHE scheme acceleration is a significantly active field of research within the FHE community. Two main components are targeted: the computational intensive polynomial operations and the highly time consuming ciphertext maintenance operations. There is a plethora of accelerators designed for different hardware platforms, such as CPUs, GPUs, FPGAs and ASICs [54], [59], [61], [52], [1], [6], [35], [37], [17], [57]. A systematic comparison of existing FHE accelerators is provided in [66], while an overview of researched acceleration methods can be found in [27].

No matter the targeted platforms, accelerators are usually designed either to accelerate high-level functions of a specific FHE scheme, such as key-switching, modulus switching and bootstrapping [59], [61], [1], or optimize low-level functions, common among different FHE schemes, such as the Number Theoretic Transform (NTT) and the polynomial multiplication [52], [6], [35], [5], [42].

3.2 The FPGA Perspective

A Field Programmable Gate Array (FPGA) is an integrated circuit (IC) designed to be configured after manufacturing. Similarly to an ASIC device, an FPGA implements custom circuits that perform a specified logic functionality. This is achieved by programming its logic elements and signal routing instead of etching a circuit on a silicon substrate. As such, FPGAs cannot provide neither the performance nor the energy efficiency that can be achieved with ASIC devices. But at the same time, due to its reconfiguration capabilities, an FPGA can be vastly more flexible, adaptable and easier to deploy and use post fabrication. Modern solutions that employ High Level Synthesis design tools and standardized deployment environments (in SoCs or PCIe-based accelerator boards) further lower the design and deployment difficulties for FPGA-based solutions.

Compared to software programmable devices such as CPUs or GPUs, a compute function implemented in an FPGA is realised at a far lower level (logic gate level) with significantly less overheads attributed to software elements. This apparently means that performance and energy efficiency can be significantly higher but programmability, ease of development and flexibility are to a certain degree sacrificed. Therefore, an FPGA can be seen as a middle ground between general-purpose processors and application-specific circuits in terms of performance, energy efficiency and programmability. This makes them ideal for accelerating compute functions that cannot be effectively handled by CPUs or GPUs and at the same time they are not standardised enough or fixed to the extend that an ASIC design cost may be justified.

This is the reason that the FHE research community has heavily relied on using FPGA devices to explore acceleration opportunities in order to make the FHE schemes significantly faster. For example, the Intel proposed HEXL library [6], tries to take advantage of the AVX-512 Instruction Set to accelerate the NTT and the polynomial multiplication on general purpose processors (CPU). However, while it achieves high single core acceleration, when multithreading is enabled, the overall performance decreases mainly due to the caused heat dissipation that forces the CPU to reduce its working frequency.

To overcome these limitations, Intel has also published a High Level Synthesis (HLS) FPGA-based accelerator (Intel HEXL-FPGA [35]) that supports the same functionality as Intel HEXL and is open-source. Their work supports operations on polynomials of degree in range $[2^{10}, 2^{15}]$. The FPGA approach

improves another limitation of HEXL, the slow memory access, but the HLS design does not manage to optimally utilize the FPGA resources. Microsoft proposed HEAX [52] in 2020, a multi core FPGA-based architecture targeting the NTT, the modular multiplication and the key-switching function through different hardware modules. HEAX needs to be reconfigured to support different cryptographic parameters, such as different polynomial degree, however it outperforms Intel HEXL-FPGA. HEAT [59], another FPGA-based accelerator, was designed to accelerate primitive operations of the BFV scheme (e.g. addition, subtraction, multiplication, modulus switching, NTT). HEAT operates on polynomials of degree up to 4096 (2^{12}), limiting its practical use.

The main disadvantage of many FPGA accelerators is the fact that the supported parameters are fixed and limited, leading to designs of limited generality. Dedicated architecture (ASIC) accelerators supporting different FHE schemes have also been proposed (F1 [54] supporting polynomials of maximum degree 2^{14}, ARK [37] supporting polynomials of maximum degree 2^{16}). A common issue regarding ASIC approaches is the designs' throughput. FHE applications launched on ASICs will not always succeed in fully utilizing its available resources, leading to highly underutilized designs. While it is obvious that accelerating low-level functions propagates to high-level functions as well, accelerating directly the ciphertext maintenance operations is crucial so as to make FHE practical. FAB [1] is a recently proposed FPGA accelerator supporting bootstrapping and polynomials of degree up to 2^{16}. FAB outperforms F1 ASIC-based design for a logistic regression application but does not manage to reach the performance of ARK. ARK also targets FHE high-level functions and deploys on-the-fly data generation to reduce the size of the plaintexts used in bootstrapping.

Accelerating only the NTT is also common in the literature. Various accelerators have been proposed for different platforms, targeting different NTT algorithm versions [40], [19], [5], [42], [44]. These works are the result of many NTT proposed optimizations [39], [32], [53], [51] and they usually support a limited range of NTT parameters (e.g. polynomial degree, ciphertext modulo bits, etc.). For example, the work of Mert et al. [40] includes designing a NTT accelerator on a Xilinx Virtex-7 FPGA, using Verilog, High-Level Synthesis (HLS) and a RISC-V ISA and compare the resuls of each design approach. The maximum supported polynomial degree is 2^{12}. Note that most works implement a different NTT algorithm for the forward transform, indicating the variety of NTT-related works in the literature. HLS designs for the

NTT have also been proposed, each focusing on optimizations based on the supported NTT parameters and the available FPGA platform [40], [47], [36] [46].

Recent works on Number Theoretic Transform (NTT) accelerators have also focused on area efficient [19] and configurable designs [20]. As accelerating the NTT is only one part of FHE acceleration attempts, area efficient and configurable designs are essential in the case where someone attempts to accelerate more functions beyond the NTT. For example, supporting fast and efficient NTTs of size 2^{17} can utilize a significant amount of resources for some designs in an FPGA. If more kernels are to be included, such as an inverse NTT kernel or a key-switching kernel, area efficiency of the kernels design is paramount. Overall, optimizing hardware designs for the NTT depends on the use-case, the platform used and the experience of the designer with the available technology and implementation methods.

3.3 Thesis Approach

According to the aforementioned, we believe that offloading computationally intensive parts of the FHE computations on hardware accelerators implemented in FPGA is a viable path towards increasing the performance of FHE schemes. However, in order to ensure adoption and increase the impact of these efforts, we need to ensure that compatibility with the de-facto standard FHE library (OpenFHE) is maintained, while ease of use and deployment remains high without introducing any significant restrictions (e.g. limited functionality, support only for specific data sizes, etc).

Therefore, in this work we aim to provide a hardware-based accelerator integrated into OpenFHE. We employ a Xilinx Alveo U50 available in our data center and mainly focus on showing how a Xilinx Vitis IDE-developed application can be integrated with OpenFHE. We are going to provide a functional NTT implementation supporting various NTT sizes that can be tested alongside OpenFHE with the goal that applications employing the vanilla OpenFHE library can seamlessly employ its hardware-accelerated version as well.

Chapter 4

Defining the Architecture

4.1 OpenFHE Library Profiling

Software profiling is the process of analyzing the runtime behavior of a software application to gather information about its execution characteristics, such as function calls, memory usage and execution time. Profiling helps developers identify performance bottlenecks, memory leaks, and areas of the code that can be optimized for better efficiency. Profiling tools, like *gperftools* (Google Performance Tools) [34], provide insights into how the program is actually running, helping developers make informed decisions about code optimization and resource allocation. We use *gperftools* as our main profiling tool, in order to determine the computational bottlenecks of OpenFHE library. *gperftools* is an open-source collection of profiling and performance optimization tools developed by Google. It includes tools for CPU and heap profiling, memory debugging, and code coverage analysis.

To configure OpenFHE for *gperftools*, one needs to add the below code lines in the CMakeLists.txt included in the OpenFHE source directory.

```
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -lprofiler")
set( CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -lprofiler")
set( CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -lprofiler")
```

Following this step, the user needs to open a new terminal session in the gperftools directory (∼/gperftools) and execute the below code lines:

```
export OMP_NUM_THREADS=1
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=test.prof \
        /home/user/openfhe_folder/build_folder/example
google-pprof --web \
        /home/user/openfhe_folder/build_folder/example  test.prof
```

Using the code above, the results will be displayed in a new web browser session. Alternatives for different result format can be found at [30]. To ensure the best results, OpenFHE's multithreading must be disabled by setting the OMP_NUM_THREADS terminal environment variable to 1.

We use the above settings to get profiling results for the different schemes of OpenFHE using the following files: *example.cpp*[1] (C++ code developed during the thesis, which includes the same dataset computations for BGV, BFV, CKKS schemes), *logistic_regression_ a16_int.cpp/ logistic_regression_a16_fp.cpp* (modified C++ code files compatible with OpenFHE out of T2 benchmarks [28]), *boolean.cpp/ boolean-ap.cpp* (C++ code example included in OpenFHE), *simple- ckks- bootstrapping.cpp/ iterative- ckks- bootstrapping.cpp/ advanced-ckks-bootstrapping.cpp* (C++ code example included in OpenFHE).

We keep track of the most called functions and we identify the key operations of each scheme. Our results can be found in the figures 4.1 and 4.2. Our profiling results are in line with those presented in Section 3.2 confirming the focus on accelerating functions such as the NTT. We focus on the BFV, BGV and CKKS scheme results, to avoid memory bound problems of TFHE/FHEW schemes, due to large key sizes and large intermediate data of these schemes.

Based on figures 4.1 and 4.2, we notice that the majority of our code examples consume a significant amount of time at the ***SwitchFormat*** function, because of multiple function calls. Bootstrapping (***EvalBootstrap*** function) and multiplication (***EvalMult*** function) are other bottleneck operations. However, ***EvalBootstrap*** is solely used for the CKKS scheme, while the ***EvalMult*** high-level function consists of some ***SwitchFormat*** function calls. Having no prior experience with Fully Homomorphic Encryption, we choose to work with accelerating the ***SwitchFormat*** function, which breaks down to the forward and the inverse Number Theoretic Transform (NTT). The NTT has been identified as an FHE bottleneck operation multiple times in literature. For the rest of this thesis, we focus on the forward NTT, however our work can be modified to apply for the inverse NTT as well.

[1]example.cpp is provided in the private GitHub repository for this thesis (https://github.com/parasecurity/FHE_FPGA/tree/main/OpenFHE_FPGA)

OpenFHE Function name	File (.cpp) example (BFV)	example (BGV)	example (CKKS)	logistic_regression _a16_int (BFV)	logistic_regression _a16_fp (CKKS)
lbcrypto CryptoContextImp **Encrypt**	24.7%	29.3%	13.7%	20.3%	22.5%
lbcrypto CryptoContextImp **Decrypt**	12.3%%	20.9%	**57.5%**	-	-
lbcrypto CryptoContextImp **EvalMult**	**60%**	**45.7%**	24.6%	43.1%	25%
lbcrypto CryptoContextImp **EvalRotate**	-	-	-	-	-
lbcrypto CryptoContextImp **EvalBootstrap**	-	-	-	-	-
lbcrypto LeveledSHEBFVRNS **RelinearizeCore**	12.7%	-	-	12.5%	-
lbcrypto BinFHEScheme **BootstrapGateCore**	-	-	-	-	-
lbcrypto LeveledSHECKKSRNS **ModReduceInternalInPlace**	-	-	9.3%	-	-
lbcrypto SchemeBase **KeySwitchCore**	6.4%	26.3%	13.7%	6.2%	20.8%
lbcrypto DCRTPolyImpl **ApproxModDown**	-	14.7%	7%	-	6.7%
lbcrypto DCRTPolyImpl **ModReduce**	-	19%		-	-
lbcrypto DCRTPolyImpl **SwitchFormat**	38.1%	25.9%	12.7%	23.4%	14.2%
lbcrypto PolyImpl **SwitchFormat**	**49.9%**	**39%**	19%	**43.8%**	**33.3%**
lbcrypto DiscreteUniformGeneratorImpl **GenerateInteger**	-	-	-	10.9%	19.2%
lbcrypto PseudoRandomNumberGenerator **GetPRNG**	1.9%	2.5%	-	4.7%	7.5%
intnat NativeVectorT **SwitchModulus**	-	4.3%	2.3%	-	-
intnat NativeVectorT **ModMulEq**	6.8%	10.4%	3.9%	4.7%	10.8%
intnat NumberTheoreticTransformNat **ForwardTransformToBitReverseInPlace**	27.2%	22.7%	10.8%	**35.9%**	29.2%
intnat NumberTheoreticTransformNat **InverseTransformFromBitReverseInPlace**	23.7%	17.2%	8.2%	7.8%	4.2%

<20% 20-30% 30-40% 40-50% >50%

FIGURE 4.1: Function execution time % out of C++ code total execution time (1/2)

OpenFHE Function name	File (.cpp) boolean (CGGI)	boolean-ap (DM)	simple-ckks-bootstrapping (CKKS)	iterative-ckks-bootstrapping (CKKS)	advanced-ckks-bootstrapping (CKKS)
lbcrypto CryptoContextImp **Encrypt**	4.3%	1.1%	-	-	-
lbcrypto CryptoContextImp **Decrypt**	-	-	-	-	-
lbcrypto CryptoContextImp **EvalMult**	-	-	14.9%	20.1%	15%
lbcrypto CryptoContextImp **EvalRotate**	-	-	-	13.7%	10.6%
lbcrypto CryptoContextImp **EvalBootstrap**	-	-	**47.8%**	**79.4%**	**57.9%**
lbcrypto LeveledSHEBFVRNS **RelinearizeCore**	-	-	-	-	-
lbcrypto BinFHEScheme **BootstrapGateCore**	9.9%	2.7%	-	-	-
lbcrypto LeveledSHECKKSRNS **ModReduceInternalInPlace**	-	-	10.6%	14.2%	8.8%
lbcrypto SchemeBase **KeySwitchCore**	-	-	17.8%	25.2%	19.8%
lbcrypto DCRTPolyImpl **ApproxModDown**	-	-	8.8%	-	13.2%
lbcrypto DCRTPolyImpl **ModReduce**	-	-	-	-	-
lbcrypto DCRTPolyImpl **SwitchFormat**	-	-	27.9%	28.5%	22%
lbcrypto PolyImpl **SwitchFormat**	6.7%	19.4%	**31.6%**	**33.8%**	**30%**
lbcrypto DiscreteUniformGeneratorImpl **GenerateInteger**	**52.7%**	28.5%	21.5%	6.8%	19.4%
lbcrypto PseudoRandomNumberGenerator **GetPRNG**	14%	14.7%	7%	-	7.3%
intnat NativeVectorT **SwitchModulus**	-	-	-	-	-
intnat NativeVectorT **ModMulEq**	4.7%	2.5%	11.1%	10.2%	6.6%
intnat NumberTheoreticTransformNat **ForwardTransformToBitReverseInPlace**	6.3%	19.1%	26.9%	28.2%	23.8%
intnat NumberTheoreticTransformNat **InverseTransformFromBitReverseInPlace**	-	-	4.5%	5.6%	5.9%

<20% 20-30% 30-40% 40-50% >50%

FIGURE 4.2: Function execution time % out of C++ code total execution time (2/2)

To ensure that results are in the quotient ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$, OpenFHE

SwitchFormat function calls an implementation of a negacyclic NTT (negative wrapped convolution (NWC) NTT) with a Cooley-Tukey butterfly for the forward transform (coefficient to evaluation format) and with a Gentleman-Sande butterfly for the inverse transform (evaluation to coefficient format). Combining these two different NTT configurations, a bit-reordering procedure is avoided, as the CT-NTT uses inputs in normal order and yields a bit-reversed output, while GS-NTT uses inputs in bit-reversed order and yields a normal-ordered output.

4.2 Acceleration Target

Following our profiling results, we define the acceleration target of this thesis as the NTT function:

```
void NumberTheoreticTransformNat<VecType>::
    ForwardTransformToBitReverseInPlace(const VecType& rootOfUnityTable,
                                        const VecType& preconRootOfUnityTable,
                                        VecType* element)
```

that can be found at the **src/core/include/math/hal/transformnat-impl.h** file of the library source files [2]. OpenFHE computes once the necessary roots-of-unity ψ and ψ^{-1} for each modulo q_i used and stores it in the rootOfUnityTable and rootOfUnityInverseTable variables respectively, to avoid recomputing it in case it is needed during the same application run. The library also uses Harvey's optimization and pre-computes some factors based on each ψ_j of the rootOfUnityTable that corresponds to the same modulo q_i and stores it in the preconRootOfUnityTable. This is also done for the rootOfUnityInverseTable and the inverse NTT operation. The rootOfUnityTable and preconRootOfUnityTable vectors include the precomputed roots of unity and preconditioned roots of unity in bit-reversed order. The element vector is the vector of coefficient to perform the forward NTT on. An overview is provided in algorithm 7. HarveyModMul function can be found at **src/core/include/math/hal/ubintnat.h** file and for an overview the reader is pointed to algorithm 8.

To determine our target architecture, we had to compare existing NTT hardware accelerators and the design choices that come with each one of them. Table 4.1 demonstrates the supported parameters and the characteristics of each studied accelerator. We focus on accelerators with publicly accessible source codes in our study.

Algorithm 7 OpenFHE's In-Place Forward CT-NTT

Require: $a = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$ in normal-ordering. n is a power of two usually in the range $[2^{10}, 2^{17}]$. q is a prime such that $q \equiv 1 \pmod{2n}$. $\psi_{rev} \in \mathbb{Z}_q^n$ are the powers of ψ in bit-reversed order and $\psi_{rev}^{prec} \in \mathbb{Z}_q^n$ are the preconditioned powers of ψ in bit-reversed order.

Ensure: $a \leftarrow NTT(a)$ in bit-reversed order.

function FORWARDTRANSFORMTOBITREVERSEINPLACE($a, \psi_{rev}, \psi_{rev}^{prec}$)

$t \leftarrow n \gg 1$

logt1 $\leftarrow$ log2(t)

for `m = 1; m<n; m=2*m` **do**

for `i = 0; i<m; i++` **do**

$j1 \leftarrow i \ll$ logt1

$j2 \leftarrow j1 + t$

indexOmega $\leftarrow m + i$

omega $\leftarrow \psi_{rev}$[indexOmega]

precOmega $\leftarrow \psi_{rev}^{prec}$[indexOmega]

for `indexLo = j1; indexLo<j2; ++indexLo` **do**

indexHi $\leftarrow$ indexLo $+ t$

loVal $\leftarrow$ a[indexLo] ▷ X0

hiVal $\leftarrow$ a[indexHi] ▷ X1

omegaFactor = HARVEYMODMUL(hiVal, omega, precOmega, q)
▷ Harvey's Modular Multiplication W*X1 mod q

hiVal $\leftarrow$ loVal $+$ omegaFacTor ▷ X0 + (W*X1 mod q)

▷ Correction steps

if hiVal $\geq$ q **then**

hiVal $\leftarrow$ hiVal $-$ q ▷ X0 + (W*X1 mod q) - q

if loVal $<$ omegaFactor **then**

loVal $\leftarrow$ loVal $+$ q ▷ X0 = X0 + q

loVal $\leftarrow$ loVal $-$ omegaFacTor ▷ X0 - W*X1 mod q

▷ Cooley-Tukey Butterfly

a[indexLo] $\leftarrow$ hiVal ▷ X0 + W*X1 mod q

a[indexHi] $\leftarrow$ loVal ▷ X0 - W*X1 mod q

$t \leftarrow t \gg 1$

logt1 $\leftarrow$ logt1 $- 1$

return

Algorithm 8 OpenFHE's Harvey Modular Multiplication

Require: $value1$ and $value2$ as the operands to perform the modular multiplication on. mod is the modulus and a prime such that $mod \equiv 1 \pmod{2)n}$. $\psi_{rev}^{prec} \in \mathbb{Z}_m od^n$ is a preconditioned power of ψ.
Ensure: $value1 * value2 \pmod{q}$.
  function HARVEYMODMUL($value1, value2, \psi_{rev}^{prec}, mod$)
    $q \leftarrow$ HighWordOf$(value1 * \psi_{rev}^{prec})$
                ▷ e.g. keeping 64 high bits of a 128-bit value
    yprime $\leftarrow value1 * value2 - q * mod$
    if yprime - mod ≥ 0 **then**
      yprime $\leftarrow$ yprime - mod
    return yprime

TABLE 4.1: A comparison of existing NTT accelerators with OpenFHE supported parameters

Name	Implementation	Max Mod. Q (bits)	NTT supported ring size (N)	Optimizations	Notes
OpenFHE [2]	C++	Native word size modulo q (64 bits) & small moduli q_i up to 60 bits	All powers of two	NWC (Inv)NTT, includes CT, GS, Harvey butterflies, Barrett, Montgomery reduction, etc.	-
Intel HEXL [6]	AVX512 & C++ API (CPU)	62	$2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}$	NWC (Inv)NTT, CT, GS, Harvey butterflies, Barrett reduction	Requires precomputation of twiddle factors but avoids bit reversing operations. Includes dyadic multiplication functionality.
Intel HEXL-FPGA [35]	HLS & C++ API (Stratix 10 FPGA)	62	$2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}$	NWC (Inv)NTT, CT, GS, Harvey butterflies, Barrett reduction	Requires precomputation of twiddle factors but avoids bit reversing operations. Includes dyadic multiplication functionality.
Griffinfly [29]	HLS (Varium C1100 FPGA)	64 (const. modulo $2^{64} - 2^{32} + 1$)	$2^{12}, 2^{24}$	Goldilocks NTT	-
Supranational [44]	System Verilog & C++API (Varium C1100 FPGA)	64 (const. modulo $2^{64} - 2^{32} + 1$)	$2^{12}, 2^{24}$	Goldilocks NTT, constant geometry RAM access pattern	Includes dynamic twiddle factors generation and bit reverse operations
Parametric NTT [40]	Verilog/ HLS/ RISC-V (Virtex 7 FPGA)	8-64	Up to 2^{15}	NWC (Inv)NTT, GS butterfly, Montgomery Modular Multiplication	Includes design-time constant twiddle factors precomputed based on input parameter

Following the aforementioned comparison, we investigated whether OpenFHE uses the same parameters throughout an application run. The ciphertext modulus $Q = q_0 \cdot q_1 \ldots \cdot q_k$, which is a product of multiple smaller

primes, might change depending on the scheme used, the specific choice of some FHE-specific algorithms, and the FHE operations that the application performs. Furthermore, there is an auxiliary ciphertext modulus $P = p_0 \cdot p_1 \ldots \cdot p_k$ that is used in certain algorithms, such as key switching. Once all the moduli q_i's and p_i's are selected for a certain application, they will remain the same throughout the execution of the application. The ring dimension will also remain the same. If the application restarts, there is a slight chance that the moduli q_i's and p_i's might change due to some introduced randomness. However, the ring dimension is unlikely to change.

We remind that another NTT parameter requirement is that for each modulus q_i (or p_i), NTT requires a 2N-th primitive root of unity modulo q_i (or p_i) (also called ψ or twiddle factors), where N is the ring dimension. These parameters are fixed once generated, but they may change from one run of the application to another. For a given application, in word-wise FHE schemes such as BFV, BGV and CKKS, only fixed-size NTTs (e.g. 2^{16} points NTT) will be performed (based on the selected ring-size). In binary-wise FHE schemes such as DM (also known as FHEW) and CGGI (also known as TFHE), both implemented in OpenFHE, the application keeps switching between different ring sizes internally.

We conclude that twiddle factors generation is out of the scope of our work, as OpenFHE already includes functions generating these parameters. In addition, reordering the NTT output is also not required, as OpenFHE uses the CT butterfly for the forward transform and the GS butterfly for the inverse transform.

Table 4.2 contains the resource utilization of each work included in table 4.1, if available. For the Parametric NTT, resource utilization results are available from [40] and [20]. For Supranational, resource utilization results are available at their GitHub repository [44]. In the table, we list each work along with its defined ring size, the target platform and the resource utilization. We also provide percentages to denote the resource utilization of each work out of the overall resources available on each platform.

TABLE 4.2: Target platforms and resource utilization of existing NTT accelerators

Design	Ring size	Device	Resource utilization [% of overall device resources]				
			LUT	REG	DSP	BRAM	URAM
Suprana-tional [44]	2^{12}	Varium C1100	327,707 [43.68%]	547,426 [35.15%]	2,880 [48.42%]	136 [11.23%]	64 [10.00%]
Parametric NTT [40]	2^{10}	Virtex-7	17,188 [3.97%]	-	96 [2.66%]	48 [3.26%]	-
Parametric NTT [40]	2^{12}	Virtex-7	99,384 [22.94%]	-	992 [27.55%]	176 [11.97%]	-

4.3 Target Architecture

To define the target architecture of this thesis, we use the results provided in sections 4.1 and 4.2. We also take into consideration the functional requirements of OpenFHE. Our target architecture for our NTT accelerator can be categorized as the OpenFHE requirements, the target NTT version and the modular reduction algorithm, the input/output interface, the target platform and the integration with OpenFHE. We start with OpenFHE requirements.

OpenFHE Requirements:

Input Data (from OpenFHE):

- Vector of power-of-two size in quotient ring size range $[2^{10}, 2^{18}]$ containing polynomial coefficients of size up to 60bits (due to smaller moduli decomposition of initial ciphertext modulus) – **max data size =** 2^{18} * 60 bits = 15728640 bits = 1966080 bytes = **1,875 Mb**
- Ciphertext modulus up to 60bits - **max data size = 60 bits**
- Powers of ψ/twiddle factors (same size with input vector – unique for each smaller moduli q_i and precomputed once) - **max data size =** 2^{18} * 60 bits = 15728640 bits = 1966080 bytes = **1,875 Mb**
- (optional – if Harvey's optimization is used) Preconditioned powers of ψ (same size with input vector – unique for each smaller moduli q_i and precomputed once) - **max data size =** 2^{18} * 60 bits = 15728640 bits = 1966080 bytes = **1,875 Mb**
- **Total input size requirements:** about **5,625Mb**

Output Data (to OpenFHE):

- Vector of power-of-two size containing polynomial coefficients in NTT domain – **max data size =** 2^{18} * 60 bits = 15728640 bits = 1966080 bytes = **1,875Mb**
- **Total output size requirements:** about **1,875Mb**

NTT Algorithm:

- Negative Wrapped Convolution (NWC)/Negacyclic Cooley-Tukey (CT) Forward NTT

We choose to implement the Negative Wrapped Convolution (NWC) Number Theoretic Transform (NTT) to support the polynomial multiplication over the quotient ring $\mathbb{Z}_q[X]/(X^n+1)$ and to be compatible with the NTT implementation of OpenFHE. Note that only fixed-size NTTs will be performed for a given application (based on the selected ring-size). We can also use Harvey's optimization. In this case, we do not need an additional modular reduction module. We remind the reader that the target NTT algorithm can be found at algorithm 7.

Accelerator Input/Output Interface

Regarding the input and output interface between the accelerator and OpenFHE, the requirements are listed below:

Input data format:

- Vector of power-of-two size in range $[2^{10}, 2^{18}]$, containing polynomial coefficients of size up to 60bits (due to smaller moduli decomposition of the initial ciphertext modulus) - **maximum data width**: 2D Array of max size $[2^{18} – 1 : 0][59 : 0]$
- Ciphertext modulus up to 60bits - **data width: 60 bits**
- Powers of ψ/twiddle factors - data width: **maximum data width**: 2D Array of max size $[2^{18} – 1 : 0][59 : 0]$
- (optional - if Harvey's optimization is used) Preconditioned powers of ψ - **maximum data width**: 2D Array of max size $[2^{18} – 1 : 0][59 : 0]$

Using the **uint64_t** data type for the above vectors, each vector requires $2^{18} *$ 64 bits = 16777216 bits = 2097152 bytes = **2Mb**. So, a total of 6Mb will be needed in the worst case scenario, to store all necessary data on the FPGA.

Output data format:

- Vector of power-of-two size $[2^{10}, 2^{18}]$ containing polynomial coefficients in NTT domain – **maximum data width**: 2D Array of max size $[2^{18} - 1 : 0][59 : 0]$

Again, using the **uint64_t** data type for this vector, $2^{18} * 64$ bits = 16777216 bits = 2097152 bytes = **2Mb** need to be transferred back from the FPGA at the end of the transform.

Target Platform:

As already mentioned, our target FPGA platform is the Xilinx Alveo U50 FPGA. We refer the reader to table 5.1 for an overview of the available platform resources.

OpenFHE Integration:

OpenFHE has been designed to allow integration of different hardware accelerators, such as GPUs, FPGAs and ASICs [2]. It uses a Hardware Abstraction Layer (HAL) designed to allow acceleration of polynomial arithmetic operations. HAL introduces OpenFHE's ability to instantiate multiple different mathematical back-ends, allowing the user to choose the desired configuration through a cmake option of the CMakeLists.txt file. The Number Theoretic Transform (NTT) is included in a layer supported by HAL and hence one can replace or override an OpenFHE NTT function implementation, just by creating a new back-end.

Intel HEXL [6] is the only officially integrated hardware accelerated back-end with OpenFHE. OpenFHE provides a configurator to allow installation of Intel HEXL alongside the library. As already mentioned, Intel HEXL uses the Intel Advanced Vector Extensions 512 (AVX512) instruction set to implement the polynomial operations with word-sized primes on 64-bit Intel processors. When the polynomial layer is accelerated, this propagates to higher FHE operations within the library.

Still, we choose not to use HAL to integrate our design with OpenFHE. While this may not be the optimal case, we follow this approach to reduce the development time of our accelerator's integration with the library and provide an alternative for researchers focusing only at testing different acceleration methods for OpenFHE, instead of focusing on the integration of their design with the library.

More technical details about the integration of our work with the library are provided in section 5.5.

Chapter 5

FPGA Implementation

5.1 Tools Used

5.1.1 Vitis IDE

Xilinx Vitis IDE (Integrated Development Environment) is a unified platform for developing both software and hardware designs targeting Xilinx FPGAs and SoCs (System-on-Chips). Vitis provides a high-level approach, where developers can use high-level programming languages, such as C/C++, to design hardware modules. Users can use a set of optimization options and directives, which the tool manages accordingly in order to create an optimized hardware design. An overview of Vitis development flow can be found in figure 5.1.

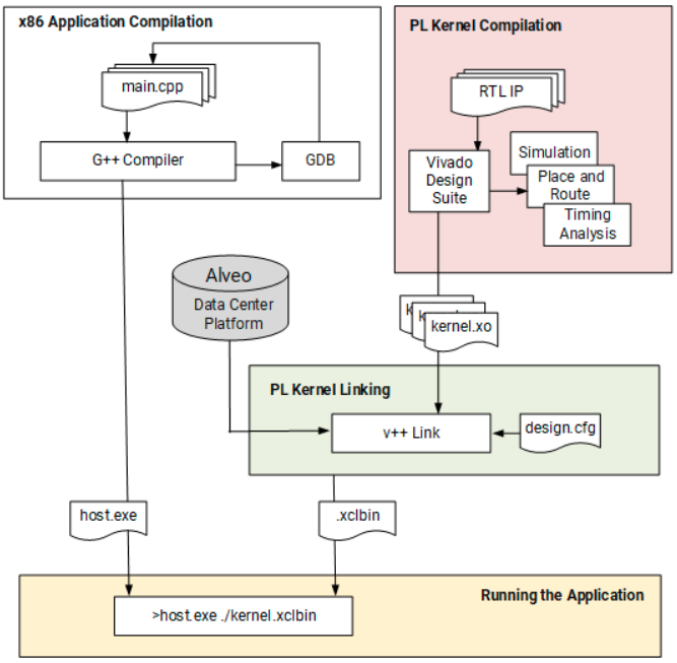

FIGURE 5.1: Xilinx Vitis Development Flow

For the purposes of this thesis, we used both **Xilinx Vitis IDE 2020.1** and **Xilinx Vitis IDE 2022.2** versions.

5.1.2 High Level Synthesis (HLS)

Historically, the development of FPGA designs was a challenging process. Engineers had to use low-level hardware design tools and a deep understanding of digital hardware design was required. To overcome this challenge, the engineering community introduced the High Level Synthesis (HLS) approach. HLS is an automated design process that creates hardware designs out of high-level descriptions using typical programming languages (such as C or C++). The main benefit of using HLS is the faster development time of hardware designs for both hardware and software engineers. The Vitis HLS Development Flow is displayed in figure 5.2.

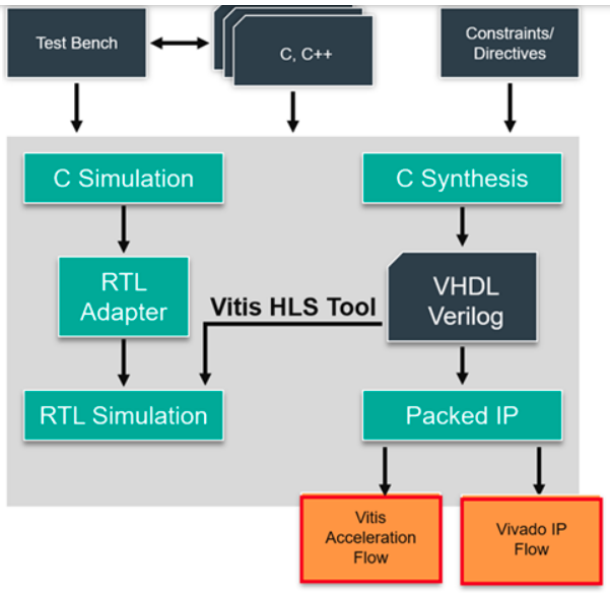

FIGURE 5.2: Xilinx Vitis HLS Development Flow

Vitis HLS is a high-level synthesis tool that translates C/C++ functions into device logic using programmable logic (PL) elements and RAM/DSP blocks. Vitis HLS synthesizes the C/C++ code into an RTL design and packages it as a compiled object (.xo) file that can be imported into the Vitis environment. There can be multiple functions targeted on the FPGA, each being a separate kernel. Vitis HLS will synthesize these kernels one by one and generate separate .xo files. Xilinx Vitis HLS documentation is available at [62], while example code and tutorials are available at a dedicated GitHub repository [63].

While HLS may not lead to the optimal hardware design, it minimizes development time of hardware functions compared to using Hardware Description Languages, such as System Verilog. We choose to use HLS, to reduce the development time of our accelerator, as the main objective of this thesis was first to establish our knowledge around FHE and alternative ways of integrating OpenFHE with our design and then design a forward NTT transform hardware kernel. In that way, we have created a baseline work for future reference for researchers aiming to further accelerate OpenFHE library using Xilinx tools and FPGAs.

5.1.3 Vitis pragmas and optimizations

Vitis HLS comes with a set of optimization directives, that the user needs to use in order to instruct the compiler of how to handle specific parts of the high-level code and produce an optimized hardware design. These optimization directives are also called **pragmas**. Directives and Pragmas are two sides of the same coin. They only differ in the way of usage. While directives are specified in a configuration file, pragmas are specified in specific areas of the high-level C/C++ source code. In this section, we introduce the directives used in the context of this thesis in the form of HLS pragmas.

Array Partition:

The ARRAY_PARTITION pragma partitions an array into smaller arrays or individual elements. This pragma is usually used to increase the amount of read and write ports of a memory by assigning multiple small memories to a variable instead of one large memory. It potentially improves the throughput of the design under study. However, it requires more memory instances or registers, which can also be a limitation depending on the target application.

Bind Storage:

The BIND_STORAGE pragma is used to assign a variable to a specific memory type in the RTL manually. If this pragma is not used, Vitis automatically decides of what memory type should be used for a specific variable of the source code. The user can decide about the memory ports for read or write purposes, as well as for the implementation of the memory (BRAM, URAM, SRL).

Interface:

The INTERFACE pragma specifies how RTL ports are created from the function description during interface synthesis. It can be only used on the top-level function of the HLS code. The ports in the RTL implementation are derived from the arguments of the top-level function and the corresponding data types. It also defines the execution control protocol of the HLS code (execute, idle, complete, ready, etc.)

Unroll:

The UNROLL pragma transforms the target loops by creating multiples copies of the loop body. A loop can be partially unrolled using the factor argument. This creates a number of loop body copies and reduces the loop iteration accordingly. By unrolling a loop, data access and throughput can be increased. Loops remain rolled if the UNROLL pragma is not used.

Pipeline:

The PIPELINE pragma allows the concurrent execution of operations in a function or a loop. A pipelined loop can drastically increase the hardware performance, however appropriate code structuring should take place before using the pipeline pragma, so as to avoid data and memory dependency issues.

More details about pragmas and HLS can be found at [62].

5.2 FPGA Platform

The target FPGA platform for the purposes of this thesis is the Xilinx Alveo U50 Data Center Accelerator Card (figure 5.3) installed on a Dell server (kronos.mhl.tuc.gr) of the Microprocessor and Hardware Laboratory (MHL) of the Technical University of Crete (TUC). This Alveo data center card is PCI-Express Gen3x16 compliant, designed to accelerate compute intensive applications such as machine learning, data analytics, and video processing. An overview of its features can be found at table 5.1.

The Vitis IDE tool supports all Alveo card versions and requires the Xilinx Runtime (XRT) library to be installed and enabled. XRT provides an API and drivers for the host program to connect with the target FPGA platform and handles transactions between the host program and hardware kernels. We use the **xilinx_u50_gen3x16_ xdma_201920_3** version of the Alveo U50 as our platform configuration. Detailed documentation about the card, as well as

the necessary files to setup the platform for the Vitis IDE 2022.2 environment are accessible at [3].

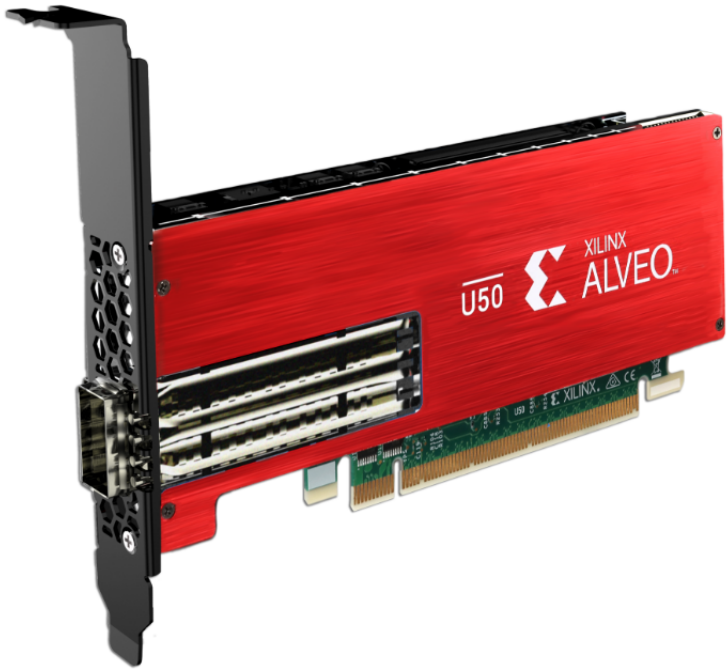

FIGURE 5.3: Xilinx Alveo U50 Data Acceleration Card

TABLE 5.1: Available resources of Xilinx Alveo U50 FPGA

Name	LUT	REG	DSP	BRAM	URAM	HBM
Xilinx Alveo U50	872K	1.743K	5,952	1,344 (x36Kbit)= 47.3Mbit= 5.9Mbyte	640 (x288Kbit)= 180Mbit= 22.5Mbyte	8Gb (316 GB/s)

5.3 Design Space Exploration

We implemented an **in-place forward NTT** with the **Cooley-Tukey butterfly** and **Harvey's modular multiplication optimization**, equivalent to algorithm 7. We chose the in-place variant, so as to reduce the overall memory footprint. As already mentioned, our accelerator is designed using **High-Level Synthesis (HLS)**. Our algorithm consists of $log_2(n)$ stages, where in each stage $n/2$ butterfly operations will be executed, n being a power of two. The inputs of the algorithm consist of a vector of n coefficients, a vector of n precomputed (by OpenFHE) roots of unity ψ, a vector of n precomputed (by OpenFHE)

preconditioned roots of unity ψ for Harvey's optimization and the current NTT size n.

As a starting point of our design, we created a new **Vitis application project** and we adopted the code architecture of algorithm 4 proposed in [40] (algorithm 9 below), an HLS-friendly iterative forward NTT algorithm that uses the Gentleman-Sande butterfly and the Montgomery modular reduction algorithm. In this version of the algorithm, the inner loop of the naive in-place Gentleman Sande NTT (see algorithms 4 or 5 for similar code structure) is decomposed into four loops to help the HLS compiler to synthesize an efficient design. The original inner loop is now split into an index calculation loop, a memory read loop, an operation loop and a memory write loop. Algorithm 9 deploys the array partition pragma for the vectors a and ω, the pipeline pragma for the butterfly loop and the unroll pragma for the four inner loops.

We modify the code to implement the Cooley-Tukey butterfly and we use the HLS implementation of Harvey's optimization out of Intel HEXL-FPGA source code [35]. The HLS ARRAY_PARTITION, the HLS PIPELINE and the HLS UNROLL pragmas were also used, as in [40]. The pipeline directive is used so as to pipeline the four small loops that substituted the initial inner loop. Each of these small loops is also fully unrolled and are configured to utilize 8 parallel butterfly units (NBU = 8). We confirm the findings of [36] that a write-after-read (WAR) dependency between the memory read loop and the memory write loop is not resolved by the Vitis tool and we modify our code according to El-Kady et. al [36] and Cohen [15] works.

The key point of this modification is the observation made by Cohen [15], that for each butterfly the two indices used, differ in their parity. The parity is defined as one if the number of ones in the binary representation of the index is odd and as zero if the number of ones is even. Hence by assigning the input coefficient into two memories according to the index parity and by utilizing 2 parallel butterfly units (NBU = 2), we implement the CT-based NTT provided in [36]. Note that that El-Kady et. al test their code for NTT of size $n = 256 = 2^8$, while Mert et. al [40] for NTT size up to $n = 4096 = 2^{12}$. Our approach aims to support NTT size up to 2^{18}. Also note that contrary to our work, both of [40] and [36], do not use the optimizations of Roy [53] for the powers of ψ, neither Harvey's optimization [32]. Using these optimizations means storing more data on the FPGA, which increases the memory requirements of our design.

Algorithm 9 HLS-friendly NTT algorithm with GS-butterfly [40]

Require: $a = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$. n is a power of two. q is a prime such that $q \equiv 1 \pmod{2n}$. $\omega \in \mathbb{Z}_q^n$ are the twiddle factors. NBU is the number of butterfly units for parallel operation processing.

Ensure: $a \leftarrow NTT(a)$.

function NTT(a, ω, NBU, n)

$t \leftarrow n \gg 1$

logt1 $\leftarrow$ log2(t)

STAGE_LOOP:

for `i = 0; i < logt1; i++` **do**

BUTTERFLY_LOOP:

for `s=0; s<t; s=s+NBU` **do**

IDX_CALC_LOOP:

for `bu=0; bu<NBU; bu++` **do**

j[bu] $\leftarrow (s + bu) \gg (logt1 - 1 - i)$

k[bu] $\leftarrow (s + bu)\&((t \gg 1) - 1)$

ie[bu] $\leftarrow$ j[bu] $\cdot (1 \ll (logt1 - i)) +$ k[bu]

io[bu] $\leftarrow$ ie[bu] $\cdot (1 \ll (logt1 - i - 1))$

iw[bu] $\leftarrow (1 \ll i) \cdot$ k[bu]

MEM_READ_LOOP:

for `bu=0; bu<NBU; bu++` **do**

U[bu] $\leftarrow$ a[ie[bu]]

V[bu] $\leftarrow$ a[io[bu]]

W[bu] $\leftarrow \omega$[iw[bu]]

OP_LOOP:

for `bu=0; bu<NBU; bu++` **do**

E[bu] $\leftarrow$ (U[bu] + V[bu]) $\pmod{q}$

O[bu] $\leftarrow$ (U[bu] $-$ V[bu]) $\cdot$ W[bu] $\pmod{q}$

MEM_WRITE_LOOP:

for `bu=0; bu<NBU; bu++` **do**

a[ie[bu]] $\leftarrow$ E[bu]

a[io[bu]] $\leftarrow$ O[bu]

return a

We also attempted to use the same pragma directives used in [36] with the NBU set to 2 in our design for NTT sizes in the range $[2^{10}, 2^{18}]$. However, when the design is synthesized, a memory dependency between the memory read loop and the memory write loop is encountered, despite the various array partition settings tested. We observed that we cannot use complete array partition (which would resolve the issue) for the coefficient array sizes we operate on, as the complete array partition can be applied to arrays up to 1024 elements [40]. The tool manages to resolve the dependency issue but the working frequency of the accelerator is decreased. Hence using the PIPELINE pragma reduces the achieved frequency of our accelerator. However, the overall performance is better when the design is pipelined.

Another design goal is to avoid off-chip memory accesses for the intermediate results. We use BRAM and URAM resources available on the Alveo U50 to avoid read and write operations on the global memory. The total memory requirements for all NTTs in range $[2^{10}, 2^{18}]$ can be met with the available on-chip memory resources. Our design allocates the necessary memory resources to handle a 2^{18}-point NTT and thus it can support any NTT size up to 2^{18} without FPGA reconfiguration. We use and test the BIND_STORAGE pragma with different options to assign the coefficients, the rootOfUnityTable and the preconRootOfUnityTable variables in available memory resources. We remind the reader that based on the defined architecture of section 4.3, for 2^{18}-point NTT, 6Mb of memory is required, which exceeds the 5.9Mb of BRAM resources available on the Alveo U50, hence using URAM resources is required. Using the BIND_ STORAGE pragma for all three aforementioned variables of our design resulted in better results compared to using the BIND_ STORAGE pragma for the coefficients only and letting the tool automatically assign the appropriate memory type for the other two variables.

Algorithm 10 is a high-level overview of our work. We explore various versions with different NBU parameter and pragmas selection in order to identify the optimal design. Table 5.2 includes the resource utilization results for each tested case, while table 5.3 includes latency results for the different versions under study.

In table 5.3, the #CC column refers to the necessary clock cycles of a circuit consisting of NBU butterflies to generate the result. The n-point NTT BL column refers to the NBU parallel butterfly operations latency for each n-point NTT measured in microseconds. To get these latency results, we use

Algorithm 10 Implemented HLS-friendly CT-NTT

Require: $coef = (c_0, c_1, ..., c_{n-1}) \in \mathbb{Z}_q^n$. n is a power of two in the range $[2^{10}, 2^{18}]$. q is a prime such that $q \equiv 1 \pmod{2n}$. $\psi_{rev} \in \mathbb{Z}_q^n$ are the powers of ψ in bit-reversed order and $\psi_{rev}^{prec} \in \mathbb{Z}_q^n$ are the preconditioned powers of ψ in bit-reversed order. NBU is the number of butterfly units for parallel operation processing. Stages is the number of stages required to execute the NTT.

Ensure: $a \leftarrow NTT(a)$.

function NTT($coef, \psi_{rev}, \psi_{rev}^{prec}, q, n, stages$)

$t \leftarrow n \gg 1$

STAGES_LOOP:

for `i = 0; i < stages; i++` **do**

BUTTERFLY_LOOP:

for `s=0; s<t; s=s+NBU` **do**

INDEX_CALC:

for `bu=0; bu<NBU; bu++` **do**

j[bu] $\leftarrow (s + bu) \gg (stages - 1 - i)$

k[bu] $\leftarrow (s + bu)\&((t \gg i) - 1)$

indexLo[bu] $\leftarrow$ j[bu] $\cdot\, (1 \ll (stages - i)) +$ k[bu]

indexHi[bu] $\leftarrow$ indexLo[bu] $+ (1 \ll (stages - i - 1))$

indexOmega[bu] $\leftarrow (1 \ll i) +$ j[bu]

parity_arr[bu] $\leftarrow$ parity(indexLo[bu])

not_parity_arr[bu] $\leftarrow$ not(parity_arr[bu])

MEM_READ:

for `bu=0; bu<NBU; bu++` **do**

omega[bu] $\leftarrow \psi_{rev}$[indexOmega]

precOmega[bu] $\leftarrow \psi_{rev}^{prec}$[indexOmega]

X0[bu] $\leftarrow$ coef[indexLo[bu] $\gg 1$][not_parity_arr[bu]] ▷ X0

X1[bu] $\leftarrow$ coef[indexHi[bu] $\gg 1$][parity_arr[bu]] ▷ X1

BUTTERFLY_OPERATION:

for `bu=0; bu<NBU; bu++` **do**

▷ equivalent to X1·W - Q·q (mod β)

Q $\leftarrow \lfloor$X1[bu] $\cdot$ precOmega[bu]$/\beta \rfloor$

yprime $\leftarrow$ X1[bu] $\cdot$ omega[bu] $-$ Q $\cdot\, q$

hiVal[bu] $\leftarrow$ X0[bu] $+$ yprime ▷ X0 + (W·X1 mod q)

loVal[bu] $\leftarrow$ X0[bu] $-$ yprime ▷ X0 - (W·X1 mod q)

MEM_WRITE:

for `bu=0; bu<NBU; bu++` **do**

coef[indexLo[bu] $\gg 1$][not_parity_arr[bu]] $\leftarrow$ hiVal[bu]

coef[indexHi[bu] $\gg 1$][parity_arr[bu]] $\leftarrow$ loVal[bu]

return

TABLE 5.2: Design Space Exploration Resource Utilization Results

NBU	Pragmas			Resource Utilization Results				
	Array Partition	Bind Storage	Pipeline	LUT	REG	DSP	BRAM	URAM
8	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	No	19002 [2.17%]	17007 [0.9%]	284 [4.7%]	1028 [88.1%]	64 [10%]
8	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	Yes	19002 [2.17%]	17007 [0.9%]	284 [4.7%]	1028 [88.1%]	64 [10%]
16	coefficients = 32 omega = 32 precOmega = 32	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	32611 [4.25%]	27600 [1.71%]	594 [9.98%]	4 [0.34%]	192 [30.1%]
16	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	Yes	29835 [3.89%]	28618 [1.78%]	594 [9.98%]	1028 [88.1%]	64 [10%]
16	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	32611 [4.25%]	27600 [1.71%]	594 [9.98%]	4 [0.34%]	192 [30.1%]
16	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	No	32254 [4.25%]	27371 [1.71%]	588 [9.88%]	4 [0.34%]	192 [30.1%]
32	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	Yes	49311 [6.43%]	48343 [3%]	1170 [19.7%]	1028 [88.1%]	64 [10%]
32	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = auto precOmega = auto	Yes	49311 [6.43%]	48343 [3%]	1170 [19.7%]	1028 [88.1%]	64 [10%]
32	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	52556 [6.81%]	47393 [2.9%]	1170 [19.7%]	4 [0.34%]	192 [30.1%]
64	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = auto precOmega = auto	Yes	90978 [10.4%]	87367 [5%]	2322 [39%]	1028 [88.1%]	64 [10%]

the following formula:

$$BL = \#CC \cdot times \cdot clock\_period = NBU \cdot times \cdot \frac{1}{F},$$

where *times* is the number of times the NBU butterfly units should be used to complete a n-point NTT. For ease of reading, we present BL results for 4096, 8192 and 16384-point NTTs only.

We choose the optimal design based on the minimum BL value of table 5.3 and we summarize the results in the tables 5.4 and 5.5.

TABLE 5.3: Design Space Exploration Latency Results

NBU	Pragmas			Latency Results		
	Array Partition	Bind Storage	Pipeline	F (MHz)	#CC	n-point NTT BL (μs)
8	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	No	172.9	33	n=4096: 586 n=8192: 1270 n=16384: 2736
8	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	Yes	164.1	33	n=4096: 617 n=8192: 1338 n=16384: 2882
16	coefficients = 32 omega = 32 precOmega = 32	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	161.5	36	n=4096: 342 n=8192: 741 n=16384: 1597
16	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	Yes	155.1	36	n=4096: 356 n=8192: 772 n=16384: 1663
16	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	173.7	36	n=4096: 318 n=8192: 689 n=16384: 1485
16	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	No	183.2	65	n=4096: 544 n=8192: 1180 n=16384: 2543
32	coefficients = 64 omega = 32 precOmega = 32	coefficients = URAM T2P omega = auto precOmega = auto	Yes	136.1	68	n=4096: 383 n=8192: 831 n=16384: 1790
32	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = auto precOmega = auto	Yes	136.1	68	n=4096: 383 n=8192: 831 n=16384: 1790
32	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	135.3	68	n=4096: 385 n=8192: 836 n=16384: 1801
64	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = auto precOmega = auto	Yes	121.9	132	n=4096: 415 n=8192: 900 n=16384: 1940

TABLE 5.4: Target Design Resource Utilization Results

NBU	Pragmas			Resource Utilization Results				
	Array Partition	Bind Storage	Pipeline	LUT	REG	DSP	BRAM	URAM
16	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	32611 [4.25%]	27600 [1.71%]	594 [9.98%]	4 [0.34%]	192 [30.1%]

TABLE 5.5: Target Design Latency Results

NBU	Pragmas			Latency Results		
	Array Partition	Bind Storage	Pipeline	F (MHz)	#CC	n-point NTT BL (μs)
16	coefficients = 64 omega = 64 precOmega = 64	coefficients = URAM T2P omega = URAM T2P precOmega = URAM T2P	Yes	173.7	36	n=4096: 318 n=8192: 689 n=16384: 1485

5.4 NTT Accelerator Design

We split our design in two parts, the host code, which sets up the FPGA and provides an API for the integration with OpenFHE, and the kernel code, which includes the Number Theoretic Transform implementation in HLS. All relevant project files can be found in the thesis private GitHub repository[1].

[1] https://github.com/parasecurity/FHE_FPGA/tree/main/OpenFHE_FPGA

5.4.1 Host Code Design

The host code in Vitis IDE is written in **C++ language** using the **OpenCL API** [64]. The Xilinx Runtime (XRT) API [65] can also be used. We split the host code into two files, the *ntt.hpp* file, that contains the function headers, etc. and the *ntt.cpp* file that contains the API implementation. The provided NTT API is:

```
/// @brief
/// @function NTT
/// Calls the Number Theorectic Transform.
/// @param[in/out] coeff_poly vector of polynomial coefficients
/// @param[in] root_of_unity_powers vector of twiddle factors
/// @param[in] precon_root_of_unity_power vector of twiddle
///             factors for the constant (Harvey's optimization)
/// @param[in] coeff_modulus stores the coefficient modulus
/// @param[in] n stores the polynomial size
///
void alveo_accelerator::NTT(uint64_t* coeff_poly,
        const uint64_t* root_of_unity_powers,
        const uint64_t* precon_root_of_unity_powers,
        uint64_t coeff_modulus,
        uint64_t n);
```

OpenFHE redefines the data types that are used in the library, so as to create a data type configurability and abstraction. Data compatibility between OpenFHE and C++ native data types can be ensured using the reinterpret_cast C++ function. The NTT function can optionally contain data correctness tests for the data movement between OpenFHE and the accelerator, to ensure that OpenFHE data are type casted correctly and are compatible with the accelerator's data types.

OpenCL specific variables like cl::Context, cl::CommandQueue, cl::Kernel, cl::Program and cl::Platform need to be initialized accordingly. The user is provided with the **program_device function** that is used to initialize the aforementioned variables and to program the Alveo U50 device with the binary file (xclbin file) containing the NTT design, generated by Vitis. This function should be called at the beginning of an OpenFHE application, prior to any setup and initialization step.

```
/// @brief
/// @function program_device
```

```
/// Programs the target FPGA using the internally defined binary (xclbin) file.
/// The function also contains the binary file path (xclbin file) used to
/// program the Alveo U50 device.
///
void alveo_accelerator::program_device();
```

The program_device function will traverse all available platforms to find the target Xilinx platform. Once the platform is found, the code will create a program using the binary file (xclbin file) and it will load it into the platform. The cl::Context and the cl::CommandQueue variables are used to control the hardware kernels. Next, inside the NTT function, we assign the input data from OpenFHE into cl::Buffer variables and we enqueue the buffers, so as the data can be migrated into the kernel space. Once the data have been migrated, the kernel is launched using the enqueueTask command. When the kernel execution has been completed, the output data need to be migrated back from the kernel space to the host code and OpenFHE space.

5.4.2 Kernel Code Design

The kernel code in Vitis IDE is written in **HLS** and implements an **in-place forward NTT** with the **Cooley-Tukey butterfly** and **Harvey's modular multiplication optimization**, equivalent to algorithm 7. The in-place variant was chosen to reduce the overall memory footprint. As already mentioned, algorithm 10 is the high-level overview of our work. We remind that our implementation consists of $log_2(n)$ stages, where in each stage, $n/2$ butterfly operations will be executed, where n is a power of two. The inputs of the algorithm consists of a vector of n coefficients, a vector of n precomputed (by OpenFHE) roots of unity ψ, a vector of n precomputed (by OpenFHE) preconditioned roots of unity ψ for Harvey's optimization and the current NTT size n.

Based on the design space exploration results presented in section 5.3, we set the number of paraller butterfly units (NBU) to 16. We choose to implement a pipelined NTT, as using the PIPELINE pragma improves the accelerator's performance. To avoid off-chip memory accesses for the intermidiate resutls, we use the BIND_STORAGE pragma, to assign the coefficients, the rootOfUnityTable and the preconRootOfUnityTable variables in URAM memories only.

To move the data between the FPGA and the host code, we use the INTERFACE pragma with m_axi and s_axilite modes in the top-level function of

the kernel, based on available Xilinx documentation and tutorials. We have ensured that our design can fit in the FPGA and is functional and OpenFHE-equivalent for all supported NTT sizes. The hardware generated for the BUTTERFLY_LOOP of our algorithm (algorithm 10) will be used as many times required to execute the n-point NTT, leading to a universal NTT design which supports the desired n-point NTTs defined in our architecture.

Table 5.6 includes all HLS pragmas used in our design. After multiple synthesis attempts, we find that using the specific HLS pragmas, when the number of parallel butterfly units (NBU) is set to 16, is the best option in order to support all NTT sizes in range $[2^{10}, 2^{18}]$. In section 5.3, we tested our design for NBU set to 32 and 64, however the working frequency of the device was decreased, while the number of clock cycles for the defined butterfly units increased, resulting in a slower design. Setting the NBU to 16 outperforms a design with NBU set to 8 as well, as the NBU=16 design achieves higher working frequency and higher operation parallelism.

TABLE 5.6: Deployed HLS Pragmas in algorithm 10

Code	**HLS Pragma**
j, k, indexLo, indexHi, indexOmega	array_partition complete
parity_arr, not_parity_arr	array_partition complete
omega, precOmega, X0, X1	array_partition complete
loVal, hiVal	array_partition complete
coef, omega, precOmega	array_partition block factor 64
coef, omega, precOmega	bind_storage ram_t2p uram
BUTTERFLY_LOOP	pipeline
INDEX_CALC	unroll
MEM_READ, MEM_WRITE	unroll
BUTTERFLY_OPERATION	unroll

5.5 Integration with OpenFHE

This section introduces a new integration approach of Xilinx FPGA-based accelerated back-ends with the OpenFHE library. Our proposed integration method has been tested with Xilinx FPGAs and hardware designs generated using the Xilinx Vitis IDE 2020.1 or 2022.2 platforms and consists of two steps. In the first step, we need to create a dynamic library (.so) using the Vitis environment. This library will contain our hardware implementation. Having generated and configured the library, we need to include this design in OpenFHE. These two steps are further explained below.

5.5.1 Generating an .so file using Xilinx Vitis IDE

In order to generate an .so file out of a Vitis application project, one needs to modify the Vitis compiler default settings. In particular, one needs to add the **-fPIC** flag to the CXXFLAGS and the **-shared, -Wl,-soname= libalveontt.so.1 -o libalveontt.so.1.0** flags in the LDFLAGS of the makefile for all build configurations (Emulation-SW, Emulation-HW, Hardware). Building now the project generates the **libalveontt.so.1.0** file. The CXXFLAGS and the LDFLAGS of the Vitis generated makefile should look like:

```
CXXFLAGS += -std=c++1y -DVITIS_PLATFORM=$(VITIS_PLATFORM) -D__USE_XOPEN2K8
            -I$(XILINX_XRT)/include/
            -I/tools/Xilinx/Vitis_HLS/2022.2/include/ -O0 -g -Wall -c
            -fmessage-length=0 -fPIC
LDFLAGS += -luuid -lxrt_coreutil -lxilinxopencl -lpthread -lrt -lstdc++
            -L$(XILINX_XRT)/lib/ -shared
            -Wl,-soname= libalveontt.so.1 -o libalveontt.so.1.0
            -Wl,-rpath-link,$(XILINX_XRT)/lib
```

Navigate into the active build configuration folder (Emulation-SW, Emulation-HW, Hardware) of the project, open a new terminal session and execute the following code lines:

```
sudo mv libalveontt.so.1.0 /usr/local/lib
sudo ln -sf /usr/local/lib/libalveontt.so.1.0 /usr/local/lib/libalveontt.so.1
sudo ln -sf /usr/local/lib/libalveontt.so.1.0 /usr/local/lib/libalveontt.so
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

The path */usr/local/lib* should be added to the /etc/ld.so.conf file. This can be done by creating or opening the file, if already created, and adding the */usr/local/lib* in a line. The file can be created or opened using the vim text editor with the below command:

```
sudo vim /etc/ld.so.conf
```

The final step is executing the *ldconfig* command in the terminal:

```
sudo ldconfig
```

When the above procedure is done once, most of the steps can be omitted when configuring a newer version of the .so file generated in Vitis. In this case, the user should only execute the below code lines in a terminal session at the active build configuration folder (Emulation-SW, Emulation-HW, Hardware) of the project, once the .so file is generated:

```
sudo mv libalveontt.so.1.0 /usr/local/lib
sudo ldconfig
```

A useful tutorial about dynamic libraries can be found at [60].

5.5.2 Configuring OpenFHE

To configure OpenFHE for using the libalveontt.so, we modify the **src/core/CMakeLists.txt** file of the library, by adding the following code lines:

```
if(BUILD_STATIC)
add_library(alveontt STATIC IMPORTED GLOBAL)
set_target_properties(alveontt PROPERTIES IMPORTED_LOCATION \
        "/usr/local/lib/libalveontt.so")
endif()

if(BUILD_SHARED)
add_library(alveontt SHARED IMPORTED GLOBAL)
set_target_properties(alveontt PROPERTIES IMPORTED_LOCATION \
        "/usr/local/lib/libalveontt.so")
endif()
```

and we modify the below code lines from:

```
if( BUILD_SHARED )
set (CORELIBS PUBLIC OPENFHEcore ${THIRDPARTYLIBS} ${OpenMP_CXX_FLAGS})
target_link_libraries (OPENFHEcore ${THIRDPARTYLIBS} ${OpenMP_CXX_FLAGS})
add_dependencies( allcore OPENFHEcore)
endif()
```

into:

```
if( BUILD_SHARED )
set (CORELIBS PUBLIC OPENFHEcore ${THIRDPARTYLIBS} ${OpenMP_CXX_FLAGS})
target_link_libraries (OPENFHEcore ${THIRDPARTYLIBS} ${OpenMP_CXX_FLAGS})
add_dependencies( allcore OPENFHEcore)
find_package(OpenCL REQUIRED)
target_link_libraries(OPENFHEcore OpenCL::OpenCL)
target_link_libraries(OPENFHEcore alveontt)
endif()
```

As already mentioned, the target NTT algorithm is located in the **src/core/include/math/hal/intnat/transformnat-impl.h** OpenFHE file. Appendix A includes the updated code in the file. Note that only the modified parts of

the code are included. The ntt.hpp file is included in the OpenFHE code by using the #include directive. For the original file, the reader is referred to the OpenFHE source files.

Now that the updated file is in place, open a new terminal session in the OpenFHE directory and execute the below code lines:

```
export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu
source /tools/Xilinx/Vitis/2022.2/settings64.sh
source /opt/xilinx/xrt/setup.sh
export PLATFORM_REPO_PATHS=/opt/xilinx/platforms
export OMP_NUM_THREADS=1
```

The last command disables the multithreading functionality of OpenFHE. The library includes some examples in the *bin/examples/* folder, once the library has been built. In the folder where the executables are located the below command must be executed in the terminal session.

```
emconfigutil --platform xilinx_u50_gen3x16_xdma_201920_3
```

To test that the configuration of the library is successful, suppose we want to execute the **bin/examples/pke/simple-ckks-bootstrapping** example. To do so, ensure that the emconfigutil command has been executed for the bin/examples/pke/ folder or execute it as:

```
cd bin/examples/pke
emconfigutil --platform xilinx_u50_gen3x16_xdma_201920_3
cd ../../../
```

If the example is to be executed in either the Software Emulation mode or the Hardware Emulation mode, one must execute:

```
export XCL_EMULATION_MODE=sw_emu
```

or

```
export XCL_EMULATION_MODE=hw_emu
```

If the example is to be executed directly on the FPGA and one of the above commands has been executed once, the XCL_EMULATION_ MODE variable should be unset with the below command:

```
unset XCL_EMULATION_MODE
```

The example can be executed in the terminal session by typing:

```
./bin/examples/pke/simple-ckks-bootstrapping
```

Chapter 6

Results

In this chapter, we describe the performance, the resource utilization and the power consumption of our proposed accelerator deployed on a Xilinx Alveo U50 FPGA. Our design was synthesized and implemented using the **Xilinx Vitis IDE 2020.1** tool with the **xilinx_u50_gen3x16_xdma_201920_3** platform configuration that was available in the Microprocessor and Hardware Laboratory (MHL) at the Technical University of Crete (TUC).

6.1 FPGA Resource Utilization and Performance

Tables 6.1 and 6.2 report on the resource utilization and timing measurements of our NTT accelerator on the Alveo U50 board. The results correspond to the NTT kernel with Number of Butterfly Units **(NBU) set to 16**. We observe that our design utilizes a small amount of the available platform resources (mainly URAM resources with 2 clock cycles latency) and achieves a **clock frequency of 173.7 MHz**.

TABLE 6.1: Alveo U50 Post-Route Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	32611	743808	4.25
REG	27600	1577349	1.71
DSP	594	5948	9.98
BRAM	4	1163	0.34
URAM	192	636	30.1

We measured the latency of the parallel butterfly units (NBU=16), which correspond to the four inner loops of algorithm 10, to be 36 clock cycles. We execute multiple NTTs of each size in range $[2^{10}, 2^{18}]$ and we note down the mean time of the total execution time on the FPGA. It should be noted that

the reported times in Table 6.2 are the times observed by the user application, i.e. they are measured from the software application that invokes the hardware accelerator.

Therefore, we define as start time the moment when the host application calls the function that invokes the hardware accelerator and as the end time, we define the moment that this function returns. In Table 6.2, Total Function Execution Time (TFET) is calculated as (end time - start time) and is measured in microseconds, while Kernel Computation Time (KCT) only counts the hardware kernel execution time for each NTT size. In other words, KCT measures the required time to execute an n-point NTT without taking into account the data transfers to and from the Alveo board. The Data Transfer Time (DTT) measures the time needed to copy the required data to the FPGA board and retrieve the results once the computation is completed. The required initialization time of these cl::Buffer variables as well as all steps required to prepare the data prior to transfering them to the accelerator, is included in the Total Function Execution time. We also calculate the kernel computation (KC) percentage (%) and the data transfer (DT) percentage (%) out of the Total Function Execution time entries listed in table 6.2. Results for our work were obtained by executing the **./bin/benchmark/ntt-benchmark** benchmark[1]. This benchmark is our modification of the lib-benchmark built-in benchmark that can be used to test all different NTT sizes.

The KC and the DT percentages define the main cost of using our accelerator. For small NTT sizes, the data transfer costs are significant compared to the kernel execution time and are mostly attributed to the processes required to setup and execute the transfers. This can be observed through the non-linear scaling of the data transfer times for small data sizes (doubling the amount of data required, results in a small increase in data transfer time for example when moving from 1024 to 2048 NTT size - for larger data sets, e.g. moving from 2^{17} NTT size to 2^{18}, the data transfer time follows an almost similar increase). For the larger NTT sizes (2^{14} and above), the kernel computation time is dominant. These observations indicate that this is a compute-bound computation and the main focus of future optimizations should be centered around the improvement of the kernel in order to yield any significant performance benefits.

[1]ntt-benchmark.cpp is provided in the private GitHub repository for this thesis https://github.com/parasecurity/FHE_FPGA/tree/main/OpenFHE_FPGA/1.%20Alveo_accelerator/benchmarking/ntt-benchmark.cpp

TABLE 6.2: NTT Hardware Accelerator Timing Results (KC(%) and DT(%) correspond to the percentage of Kernel Computation Time and Data Transfer Time out of the Total Function Execution Time respectively)

NTT size	**Kernel Computation Time (μs)**	**Data Transfer Time (μs)**	**Total Function Execution Time (μs)**	**KC (%)**	**DT (%)**
1024 (2^{10})	168.80	123.39	558.56	30.02	22.09
2048 (2^{11})	303.10	138.75	679.51	44.60	20.04
4096 (2^{12})	488.60	172.76	1102.81	44.30	15.66
8192 (2^{13})	983.70	221.05	1766.32	55.69	12.51
16384 (2^{14})	1958.90	273.22	3026.75	64.70	9.02
32768 (2^{15})	3915.60	313.32	5894.50	66.40	5.30
65536 (2^{16})	7174.50	475.97	10904.30	65.79	4.36
131072 (2^{17})	16493.80	686.70	23524.10	70.10	2.90
262144 (2^{18})	34411.10	1228.47	49858.50	69.00	2.40

Another conclusion that can be drawn from the results presented in Table 6.2 is that there is a significant time required in order to prepare the data to be transferred to the FPGA accelerator. This is reflected in the time difference between Total Function Execution Time and the sum of the Kernel Computation Time and Data Transfer Time. Depending on the NTT size, a 30% to 50% is spent on copying data between buffers prior to their transfer to the FPGA board. This is an obvious optimization point that if resolved, can yield significant performance uplift.

To program the FPGA device, the program_device function, that comes along our proposed API, should be called once. We measure the required time to program the FPGA with our binary file to be between 70-130 ms.

TABLE 6.3: Program Device Latency

Function	**Required Time (ms)**
program_device	70-130

Among the different related works presented in Section 3, our HLS results can be directly compared with those from Mert et. al [40] for a 4096-point NTT. In this work, results from both HLS and Verilog implementations are presented and can provide an interesting insight for future optimizations. Table 6.4 demonstrates the relevant comparison results (it should be noted that smaller NTT size results are also available in their paper, however none

of them has been tested with 60-bit coefficient size). We remind that our design supports any coefficient size up to 64 bits, with no further configuration needed. OpenFHE bounds the coefficients to be up to 60 bits. We observe, that our design, for NBU set to 16, outperforms Mert's HLS approach in terms of the NTT operation clock cycles, while the Verilog approach outperforms both Mert's HLS approach and our work. The design of Mert achieves reduced resource utilization for NBU set to 8 for both HLS and Verilog compared to our work. The main difference between this work and Mert's designs is the maximum NTT size supported, which in the case of Mert et al. is 4096 (2^{12}). This significantly reduces the memory requirements, as BRAM resources can host all NTT related data.

TABLE 6.4: Results and comparison for NTT implementations of size $n = 4096$

Version	NBU	LUT	REG	DSP	BRAM	URAM	# of CC	Time (μs)
This Work (Alg. 10)	16	32611	27600	594	4	192	55296	488.6
Alg. 4 [40] HLS	8	17768	-	360	128	-	73731	-
Alg. 4 [40] Verilog	8	23215	-	248	176	-	3276	26.2

Note that our work and the work of Mert focus on slightly different NTT versions and optimizations and target different platforms (Alveo U50 FPGA and Virtex-7 FPGA respectively). Our design uses both powers of ψ and the preconditioned powers of ψ, instead of the simple powers of ω used by Mert's algorithm. We also use Harvey's modular multiplication optimization instead of the Montrgomery modular multiplication. Comparing our results with the Verilog results provided in their paper, we conclude that using Verilog can significantly optimize the NTT design and we encourage future works on accelerating OpenFHE to focus on using Verilog or System Verilog for their hardware design.

Mert et al. report 26.2μs latency for their 4096 point NTT Verilog version (125 MHz on a Virtex-7 FPGA). This time corresponds to the kernel execution time only, without taking into consideration the data movement between the host and the FPGA. They do not provide latency results for their 4096 point NTT HLS version. Our design has a kernel execution latency of 488.6μs when NBU is set to 16 (173.7 MHz on an Alveo U50 FPGA), approximately 18 times slower.

6.2 Performance Evaluation

In this section, we will compare our performance results with both the performance of OpenFHE native NTT software function and the performance of Intel HEXL-FPGA for NTT sizes 4096 (2^{12}), 8192 (2^{13}), 16384 (2^{14}). Results for Intel HEXL-FPGA are provided in [66] and correspond only to kernel execution time. No data transfer time is measured. These results are not official Intel results (Intel does not publish any related measurements), but results from the researchers of [66] that replicated Intel's design. In their publication, they do not provide explicit numbers for the results, but only a graph, thus in figure 6.2 below we provide Intel HEXL-FPGA approximate performance results.

Results for OpenFHE library were obtained using the **./bin/benchmark/ntt-benchmark** executable. We launch OpenFHE on a Ubuntu 20.04 LTS machine running on an Intel Core i7-8750H CPU (12 x 4100MHz CPUs and CPU caches: L1 Data 32 KiB (x6), L1 Instruction 32 KiB (x6), L2 Unified 256 KiB (x6), L3 Unified 9216 KiB (x1)). The NTT performance reported for each benchmark corresponds to the mean execution time out of multiple single-thread NTT operations that contain the computation of the roots of unity and the preconditioned roots of unity needed by the CT NTT of OpenFHE. We also measure the time needed only for the CT NTT operation in OpenFHE. The software performance results can be found in table 6.5.

TABLE 6.5: Performance of Software NTT (OpenFHE)

NTT size	**Software NTT (μs)**	**Software Twiddle Factor Generation & NTT (μs)**
1024 (2^{10})	35.9	72.5
2048 (2^{11})	61.18	162
4096 (2^{12})	103.76	354
8192 (2^{13})	216.75	778
16384 (2^{14})	481.95	1772
32768 (2^{15})	965.23	3837
65536 (2^{16})	2099.7	7484
131072 (2^{17})	4147.86	16487
262144 (2^{18})	9268.65	36463

For performance results of our work, the reader is referred to table 6.2. A comprehensive comparison between the OpenFHE native NTT implementation and our hardware accelerated NTT is provided in Table 6.6. We remark

that the Software NTT column refers only to the software-based NTT calculation time required. For our work (hardware NTT), we provide data in three columns, namely Kernel Computation Time (KCT), Data Transfer Time (DTT) and Total Function Execution Time (TFET). We calculate the speedup by comparing the Software NTT time with the Total Function Execution Time. We define a negative speedup for when OpenFHE native NTT function outperforms our work. Our results showcase that if the kernel design and data transfer are optimized, NTTs of size above 2^{14} could reach and maybe outperform OpenFHE's native NTT software-based implementation. These results are visualised in Figure 6.1.

TABLE 6.6: Performance comparison of Software NTT time (NTT function included in OpenFHE library) and Hardware NTT time (kernel computation, data transfers, total hardware function execution time)

NTT size	**Software NTT (μs)**	**Hardware NTT (μs)**			**Speedup**
		Kernel Computation Time (μs)	**Data Transfer Time (μs)**	**Total Function Execution Time (μs)**	
1024 (2^{10})	35.9	168.8	123.39	558.56	-15.5x
2048 (2^{11})	61.18	303.1	138.75	679.51	-11.1x
4096 (2^{12})	103.76	488.6	172.76	1102.81	-10.6x
8192 (2^{13})	216.75	983.7	221.05	1766.32	-8x
16384 (2^{14})	481.95	1958.9	273.22	3026.75	-6.3x
32768 (2^{15})	965.23	3915.6	313.32	5894.5	-6.1x
65536 (2^{16})	2099.7	7174.5	475.97	10904.3	-5.2x
131072 (2^{17})	4147.86	16493.8	686.7	23524.1	-5.6x
262144 (2^{18})	9268.65	34411.1	1228.47	49858.5	-5.4x

Figure 6.2 provides the performance comparison of the NTT computations of OpenFHE, Intel HEXL-FPGA and our work. Note that in the case of OpenFHE, it includes only the NTT computation execution time results and not the overall software NTT time (that would include the twiddle factor generation as well). In the same figure, only kernel execution time results are available for both Intel HEXL and our work. For a 4096-point NTT, our design has a kernel execution time (KCT) of 488.6 μs, while the kernel execution time along the data initialization and data movement operations between the host and the FPGA (TFET) can reach up to 1102.81 μs.

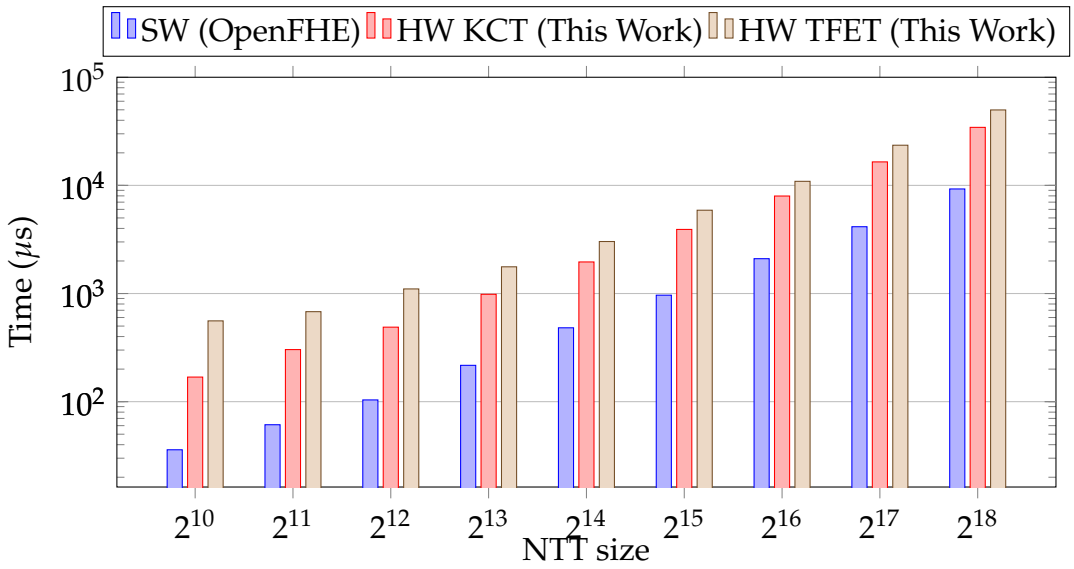

FIGURE 6.1: Performance comparison of software NTT function call (OpenFHE) and hardware NTT function call (This Work). For the hardware NTT function call the Kernel Computation Time (KCT) and Total Function Execution Time (TFET) are included.

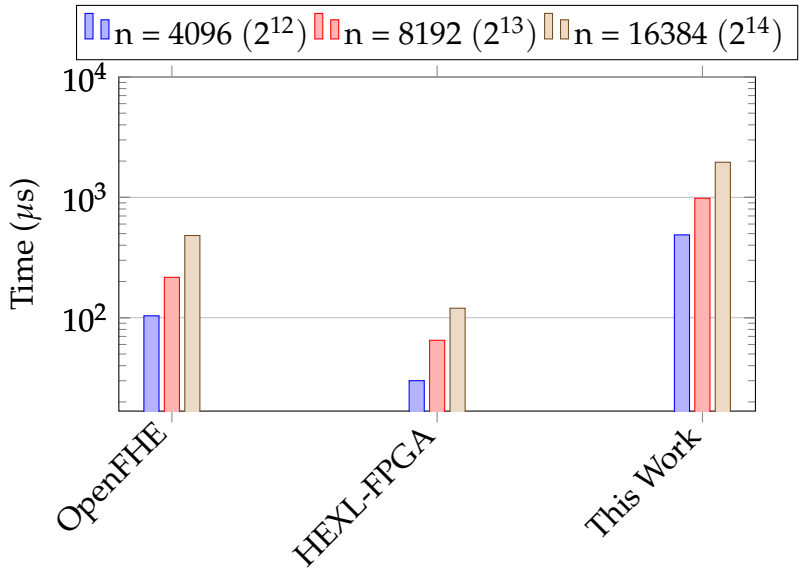

FIGURE 6.2: Comparison of NTT implementations (NTT computation time only)

We observe that our kernel design is approximately 15 times slower than the design of HEXL-FPGA. This can be explained by both the more complex design choices used in Intel HEXL-FPGA and the smaller set of supported parameters. Supporting NTTs of maximum size 2^{15} can significantly increase

the performance of a design, as less memory resources are needed and the HLS tools can achieve better place and route results.

Intel HEXL-FPGA [35] does not report the achieved clock frequency and does not use the PIPELINE HLS pragma, but it achieves a better performance mainly because of the data scheduling procedure in the available butterfly units. They also use a dynamic operation reordering, which is used to change the flow of the butterfly units input and output data, so as to avoid data and memory dependencies in their code. They also provide slightly different HLS compiled designs based on the configured NTT size, while in our case we use a generic HLS design that can support all target NTT sizes. Intel's design also uses a different set of tools and optimizations based on Intel platforms. Resource utilization data were not available.

Our design achieves a clock frequency of 173.7 MHz. While Xilinx Vitis 2022.2 tool achieves higher frequencies for our design due to better implementation optimizations, we use Xilinx Vitis 2020.1 tool which was the available tool on the MHL server. We observe that our design does not manage to accelerate the NTT operation of OpenFHE. This is mainly the result of the low working frequency achieved by our design, the small number of parallel butterflies (NBU=16) and the naive data transfer approach. We remind the reader that when NBU was set to 32 or 64 for our design, the working frequency was reduced and the performance of our design dropped. We believe that better memory management can significantly improve the performance of our design. For example, using BRAM resources as small caches, in order to provide data to the available butterfly units, could potentially decrease the design's latency, due to more efficient place and route results. Regarding the generated kernel, the data dependency between the different stages of the NTT is the main design bottleneck. To solve this data dependency, the butterfly operations could be reordered, to avoid requesting data that are not yet out of the pipeline. We highlight that this is an algorithmic problem and restructuring the algorithm is needed to further optimize the NTT algorithm.

Observing the OpenFHE results of table 6.5 yields another acceleration target candidate. We remind that the third column reports the sum of the required time to generate the root of unity powers and the preconditioned roots of unity factors (for Harvey's optimization) needed by the NTT and the software NTT time. The roots of unity, also known as twiddle factors, are unique for each modulo q_i used in OpenFHE. However, in an OpenFHE application the moduli q_i can be finite and reused and the NTT size is fixed. That means

that any NTT with a modulo q_i will use the same twiddle factors and the same preconditioned Harvey's factors. Following this observation, if consecutive NTTs with the same modulo q_i are scheduled on our FPGA accelerator, it is not necessary to transfer the aforementioned factors and the data transfer cost is avoided. While we do not provide this functionality in our design, it would reduce the data transfer cost and thus it would improve the accelerator's performance.

On another note, while we considered the twiddle factor generation out of the scope of this work, we believe that generating those factors on the FPGA when necessary, could significantly improve an accelerator's performance, as the data preparation and transfer costs would be completely avoided. As initializing the cl::Buffer variables and transferring their data on the FPGA can take up to around 30-40% of the total function call time, minimizing them would produce significant performance uplift. Thus, we suggest that future work should include twiddle factor generation functionality alongside the NTT kernel on the FPGA. Deploying both of the aforementioned suggestions, one would be able to substitute a higher-level OpenFHE function and thus provide better acceleration results compared to our work. That means that instead of substituting the target function:

```
void NumberTheoreticTransformNat<VecType>::
    ForwardTransformToBitReverseInPlace(const VecType& rootOfUnityTable,
                                        const VecType& preconRootOfUnityTable,
                                        VecType* element)
```

one could accelerate the below OpenFHE function that includes the necessary NTT factor precomputation and the NTT computation function call (also included in the **src/core/include/math/hal/transformnat-impl.h** file:

```
void ChineseRemainderTransformFTTNat<VecType>::
    ForwardTransformToBitReverseInPlace(const IntType& rootOfUnity,
                                        const usint CycleOrder,
                                        VecType* element)
```

Last but not least, we consider the energy impact of our solution compared to the software OpenFHE library. To get representative results, we executed an existing OpenFHE example (ckks-noise-flooding) which includes 2^{16}-point NTTs, iteratively, for 400 times. Each iteration yields 184 calls to the NTT functions. In that way, we simulate a Fully Homomorphic Encryption-based

application, where multiple encryption, evaluation, bootstrapping and decryption operations will be performed. In the case of the hardware accelerated OpenFHE, since the whole application is executed, the effects of operating, programming and maintaining active the FPGA board are also taken into consideration.

Both, pure software and hardware-accelerated, applications were executed on the same Dell R530 server installed in TUC's data center facilities. The server has two Intel Xeon E5-2630v4 processors (10-cores/20-threads per CPU, base frequency of 2.2GHz and Turbo frequency of 3.1GHz) and has a single Xilinx Alveo U50 card installed. Through the server's out-of-bound management module (Dell iDRAC) we were able to monitor the overall power consumption of the system during the execution of the two applications. No other user-level processes were allowed to run on the server during benchmarking times. We kept the server in idle state for at least 15 minutes between each test to get accurate measurements and measured an idle-state power consumption of 140 watts.

The example needs 8.3 minutes to complete when executed on a CPU using OpenFHE native implementations and 19.1 minutes to complete when OpenFHE's NTT has been substituted with our FPGA implementation. To calculate the overall energy consumption for the two cases, one can use the following formula:

$$Energy(Joule) = Power(Watt) \cdot Time(Seconds)$$

Table 6.7 includes energy results for both the software run of the application and the hardware (FPGA-based) run.

TABLE 6.7: Power Consumption and Energy Efficiency comparison of a FHE application when NTTs are launched only in Software (SW OpenFHE - SW NTT case) and when NTTs are launched on the FPGA (SW OpenFHE - HW NTT case) - OpenFHE in both cases is launched on Software

Case	**Power (W)**	**Time (min)**	**Energy (KJ)**
Idle	140	-	-
SW OpenFHE - SW NTT	157	8.3	78.186
SW OpenFHE - HW NTT	154	19.1	176.484

We observe that using our approach leads to higher overall energy consumption. However, this is the result of both non highly optimized NTT kernel and

data transfer operations that roughly double the required execution time. We remark that when OpenFHE uses our accelerator as a hardware component, the mean power consumption is slightly better than when OpenFHE library is launched solely on the CPU. A highly optimized FPGA-based NTT design could potentially decrease the energy consumption of the OpenFHE library if the required total function execution time was decreased.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis conducted an in-depth study of potential FPGA acceleration targets for Fully Homomorphic Encryption and more specifically for the OpenFHE library. It identifies the Number Theoretic Transform (NTT) as the starting point of any acceleration attempt and elaborates on the state-of-the-art optimizations available in the literature. The results of the proposed design, revealed that accelerating the OpenFHE library using a High-Level-Synthesis (HLS) approach can be both demanding and challenging. While it does not necessarily yields the optimal results, it remains easier compared to an HDL-based approach.

To the best of our knowledge, this work is the first hardware accelerator for OpenFHE that is implemented on a Xilinx Alveo U50 FPGA and supports up to 2^{18}-point Number Theoretic Transforms without FPGA reconfiguration. Our accelerator is fully integrated with OpenFHE as a hardware component. While our work does not manage to provide a speedup for OpenFHE, we identify several potential optimization opportunities for future work reference. A lower-level approach (in the form of an HDL-based design) can be employed for the implementation of the critical computations within the NTT butterflies to achieve higher working frequencies, while the HLS approach may be preserved for the control paths of the accelerator. Reordering butterfly operations to avoid data and memory dependencies in the design, enabling concurrent data transfers and kernel execution and generating NTT-related factors directly on the FPGA, should be further examined so as to improve the overall performance of the accelerator. We also observed that migrating to newer versions of the Vitis tools (2022.1 vs 2020.1) provided a measurable QoR improvement. Last but not least, optimizations on the host

code domain dealing with software buffers and data copies during the communication with the hardware accelerator are also possible and we measured a potential opportunity to reduce overall execution time by as much as 50%.

Overall, we aspire to make our work the foundation of any further acceleration attempt of Fully Homomorphic Encryption schemes, included in OpenFHE library, with FPGAs. Since FHE schemes are only recently becoming practical to use, we consider that it is important to consolidate related research efforts towards the OpenFHE library that seems to become the standard. Our work alleviates restrictions and other requirements that make a hardware accelerated version of the library less accessible to application developers and researchers outside the field of hardware design and allows the easy integration of any worthwhile research outcomes, be it on the software or hardware components of the library.

7.2 Future Work

Concluding our work, we provide some directions for future research regarding hardware acceleration of OpenFHE library. In particular, future researchers should consider:

- modifying the design to achieve higher clock frequencies.
- enabling concurrent data transfers (from the host to the FPGA) and kernel execution.
- enabling BRAM-based caching for efficient memory usage.
- implementing twiddle factor and other NTT essential factor generation on the FPGA.
- deploying multiple instances of the same kernel and scheduling multiple NTT operations accordingly. As OpenFHE supports multithreading, supporting multiple NTT computations scheduling on the FPGA can make the accelerator even more practical when using the library.
- using Verilog or System Verilog to achieve the best acceleration results and the minimum resource utilization on the target platform, thus allowing more hardware accelerated functions to fit on the FPGA.
- including inverse NTT and element-wise multiplication units to substitute the polynomial multiplication functions in OpenFHE.

- accelerating other OpenFHE high-level functions such as key-switching and bootstrapping.

Appendix A

OpenFHE Modified Source Code

Appendix A includes the modified OpenFHE source code of the **src/core/include/math/hal/intnat/transformnat-impl.h** file. We split the below code in two parts. Part 1 of the below code was not included in the original version of the file, while part 2 of the code is a modification of the existing **ChineseRemainderTransformFTTNat<VecType>::ForwardTransformToBitReverseInPlace** function. The main difference is that we substitute the native function call **NumberTheoreticTransformNat<VecType>().ForwardTransformToBitReverseInPlace()** with our implemented FPGA-based function **alveo_accelerator ::NTT()**. We maintain the native OpenFHE NTT function call for data correctness testing and timing purposes.

```
[...]

// Part 1
#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
#define CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY 1
#include <CL/cl2.hpp>
#include "/path/to/vitis/project/include/ntt.hpp"

[...]

// Part 2
template <typename VecType>
void ChineseRemainderTransformFTTNat<VecType>::
    ForwardTransformToBitReverseInPlace(const IntType& rootOfUnity,
                                        const usint CycloOrder,
                                        VecType* element) {
```

```
if (rootOfUnity == IntType(1) || rootOfUnity == IntType(0)) {
    return;
}
if (!lbcrypto::IsPowerOfTwo(CycloOrder)) {
    OPENFHE_THROW(lbcrypto::math_error,
                "CyclotomicOrder is not a power of two");
}
usint CycloOrderHf = (CycloOrder >> 1);
if (element->GetLength() != CycloOrderHf) {
    OPENFHE_THROW(lbcrypto::math_error,
                "element size must be equal to CyclotomicOrder / 2");
}
IntType modulus = element->GetModulus();

auto mapSearch = m_rootOfUnityReverseTableByModulus.find(modulus);
if (mapSearch == m_rootOfUnityReverseTableByModulus.end() ||
        mapSearch->second.GetLength() != CycloOrderHf) {
    PreCompute(rootOfUnity, CycloOrder, modulus);
}
std::cout<<"************************"<<std::endl;
std::cout<<"NTT progress: START"<<std::endl;
// Generate data correctness test files
// into /path/to/test/files/directory
std::string path = "/path/to/test/files/directory";
std::ofstream myfile;
myfile.open (path+"/ntt_parameters.txt");
myfile << "Parameter n - ring size = " << n;
myfile << "\nModulo q = " << modulus;
myfile << "\nLog2 Modulo q: "<<log(modulus.ConvertToDouble())/log(2);
myfile << "\nInput array length: "<< n;
myfile << "\nInput array length log2: "<< log(n.ConvertToDouble())/log(2);
myfile.close();
myfile.open (path+"/ntt_input.txt");
for (uint32_t i=0; i<n; i++){
    myfile << (*element)[i] << "\n";
}
myfile.close();
myfile.open (path+"/rootOfUnity_br_input.txt");
for (uint32_t i=0; i<n; i++){
    myfile << m_rootOfUnityReverseTableByModulus[modulus][i] << "\n";
}
```

```
myfile.close();
myfile.open (path+"/preconRootOfUnity_br_input.txt");
for (uint32_t i=0; i<n; i++){
    myfile << m_rootOfUnityPreconReverseTableByModulus[modulus][i] << "\n";
}
myfile.close();

// write expected result in file
VecType ptr(element->GetLength(),modulus);
//VecType* ptr = temp_vec[0];
for (uint32_t i=0; i<n; i++){
   ptr[i] = (*element)[i];
}
// Measure CPU time of native function call
std::chrono::duration<double> cpu_time(0);
auto cpu_start = std::chrono::high_resolution_clock::now();
// Call OpenFHE Native NTT function for timing purposes
NumberTheoreticTransformNat<VecType>().ForwardTransformToBitReverseInPlace(
    m_rootOfUnityReverseTableByModulus[modulus],
    m_rootOfUnityPreconReverseTableByModulus[modulus], &ptr);
auto cpu_end = std::chrono::high_resolution_clock::now();
cpu_time = std::chrono::duration<double>(cpu_end - cpu_start);
std::cout << "CPU time: ";
std::cout << cpu_time.count()*1000 << " ms" << std::endl;
myfile.open (path+"/ntt_expected_output.txt");
for (uint32_t i=0; i<n; i++){
    myfile << ptr.at(i) << "\n";
}
myfile.close();

// Cast input data
auto* data = reinterpret_cast<uint64_t*>(&element->at(0));
auto* rootsOfUnity =
    reinterpret_cast<uint64_t*>(&m_rootOfUnityReverseTableByModulus[modulus][0]);
auto* preconRootsOfUnity =
    reinterpret_cast<uint64_t*>(&m_rootOfUnityPreconReverseTableByModulus[modulus][0]);
uint64_t coeff_modulus = modulus.ConvertToInt();

// Call Alveo NTT implementation
alveo_accelerator::NTT(data, rootsOfUnity, preconRootsOfUnity,
                       coeff_modulus, n.ConvertToInt());
```

```
    std::cout<<"NTT progress: COMPLETED"<<std::endl;
    std::cout<<"************************"<<std::endl;
}

[...]
```

References

[1] R. Agrawal, L. de Castro, G. Yang, *et al.*, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption", in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 882–895. DOI: 10.1109/HPCA56546.2023.10070953.

[2] A. Al Badawi, J. Bates, F. Bergamaschi, *et al.*, "Openfhe: Open-source fully homomorphic encryption library", in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 53–63. [Online]. Available: https://github.com/openfheorg/openfhe-development.

[4] D. Aoki, K. Minematsu, T. Okamura, and T. Takagi, "Efficient word size modular multiplication over signed integers", in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022, pp. 94–101. DOI: 10.1109/ARITH54963.2022.00026.

[5] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "High-speed ntt-based polynomial multiplication accelerator for post-quantum cryptography", in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, 2021, pp. 94–101. DOI: 10.1109/ARITH51176.2021.00028.

[6] F. Boemer, S. Kim, G. Seifu, F. D.M. de Souza, and V. Gopal, "Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52", in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 57–62, ISBN: 9781450386562. DOI: 10.1145/3474366.3486926. [Online]. Available: https://doi.org/10.1145/3474366.3486926.

[7] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp", in *Annual Cryptology Conference*, Springer, 2012, pp. 868–886.

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping", *ACM Trans. Comput. Theory*, vol. 6, no. 3, Jul. 2014, ISSN: 1942-3454. DOI: 10.1145/2633600. [Online]. Available: https://doi.org/10.1145/2633600.

[9] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe", *SIAM Journal on computing*, vol. 43, no. 2, pp. 831–871, 2014.

[10] L. de Castro, R. Agrawal, R. Yazicigil, *et al.*, *Does fully homomorphic encryption need compute acceleration?*, Cryptology ePrint Archive, Paper 2021/1636, https://eprint.iacr.org/2021/1636, 2021. [Online]. Available: https://eprint.iacr.org/2021/1636.

[11] J. H. Cheon, A. Kim, M. Kim, and Y. Song, *HEAAN*, https://github.com/snucrypto/HEAAN, 2016.

[12] J. H. Cheon, A. Costache, R. C. Moreno, *et al.*, "Introduction to homomorphic encryption and schemes", *Protecting Privacy through Homomorphic Encryption*, pp. 3–28, 2021.

[13] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers", in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, Springer, 2017, pp. 409–437.

[14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds", in *Advances in Cryptology – ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–33, ISBN: 978-3-662-53887-6.

[15] D. Cohen, "Simplified control of fft hardware", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 6, pp. 577–579, 1976.

[16] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series", *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[17] D. B. Cousins, Y. Polyakov, A. A. Badawi, *et al.*, *Trebuchet: Fully homomorphic encryption accelerator for deep computation*, 2023. arXiv: 2304.05237 [cs.CR].

[18] L. Ducas and D. Micciancio, *FHEW*, https://github.com/lducas/FHEW, 2017.

[19] P. Duong-Ngoc, S. Kwon, D. Yoo, and H. Lee, "Area-efficient number theoretic transform architecture for homomorphic encryption", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 3, pp. 1270–1283, 2023. DOI: 10.1109/TCSI.2022.3225208.

[20] P. Duong-Ngoc and H. Lee, "Configurable mixed-radix number theoretic transform architecture for lattice-based cryptography", *IEEE Access*, vol. 10, pp. 12 732–12 741, 2022. DOI: 10.1109/ACCESS.2022.3145988.

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption", *Cryptology ePrint Archive*, 2012.

[22] H. L. Garner, "The residue number system", in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, ser. IRE-AIEE-ACM '59 (Western), San Francisco, California: Association for Computing Machinery, 1959, pp. 146–153, ISBN: 9781450378659. DOI: 10.1145/1457838.1457864. [Online]. Available: https://doi.org/10.1145/1457838.1457864.

[23] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit", in *Proceedings of the November 7-10, 1966, fall joint computer conference*, 1966, pp. 563–578.

[24] C. Gentry, "Fully homomorphic encryption using ideal lattices", in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09, Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178, ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. [Online]. Available: https://doi.org/10.1145/1536414.1536440.

[25] C. Gentry, S. Halevi, and N. P. Smart, *Homomorphic evaluation of the aes circuit*, Cryptology ePrint Archive, Paper 2012/099, https://eprint.iacr.org/2012/099, 2012. [Online]. Available: https://eprint.iacr.org/2012/099.

[26] R. Géraud, D. Maimuţ, and D. Naccache, "Double-speed barrett moduli", in *The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, Springer, 2016, pp. 148–158.

[27] Y. Gong, X. Chang, J. Mišić, V. B. Mišić, J. Wang, and H. Zhu, "Practical solutions in fully homomorphic encryption–a survey analyzing existing acceleration methods", *arXiv preprint arXiv:2303.10877*, 2023.

[28] C. Gouert, D. Mouris, and N. G. Tsoutsos, *Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks*, Cryptology ePrint Archive, Paper 2022/425, https://eprint.iacr.org/2022/425, 2022. [Online]. Available: https://eprint.iacr.org/2022/425.

[29] *Griffinfly - a submission to the zprize competition under the category, "accelerating ntt operations on an fpga"*, https://github.com/KULeuven-COSIC/Griffinfly-ZPRIZE-FPGA-NTT, 2023.

[31] S. Halevi and V. Shoup, *HElib*, https://github.com/homenc/HElib, 2014.

[32] D. Harvey, "Faster arithmetic for number-theoretic transforms", *Journal of Symbolic Computation*, vol. 60, pp. 113–119, 2014, ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747717113001181.

[33] *Homomorphic encryption standardization*, 2023. [Online]. Available: https://homomorphicencryption.org/introduction/.

[34] G. Inc., *Gperftools*, https://github.com/gperftools/gperftools, 2023.

[35] Y. Meng, S. Butt, Y. Wang, Y. Zhou, S. Simoni, *et al.*, *Intel Homomorphic Encryption Acceleration Library for FPGAs (version 2.0)*, https://github.com/intel/hexl-fpga, 2022.

[36] A. El-Kady, A. P. Fournaris, T. Tsakoulis, E. Haleplidis, and V. Paliouras, "High-level synthesis design approach for number-theoretic transform implementations", in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2021, pp. 1–6. DOI: 10.1109/VLSI-SoC53125.2021.9607003.

[37] J. Kim, G. Lee, S. Kim, *et al.*, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse", in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254. DOI: 10.1109/MICRO56248.2022.00086.

[38] Z. Liang and Y. Zhao, *Number theoretic transform and its applications in lattice-based cryptosystems: A survey*, 2022. arXiv: 2211.13546 [cs.CR].

[39] P. Longa and M. Naehrig, *Speeding up the number theoretic transform for faster ideal lattice-based cryptography*, Cryptology ePrint Archive, Paper 2016/504, https://eprint.iacr.org/2016/504, 2016. [Online]. Available: https://eprint.iacr.org/2016/504.

[40] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform", *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2829–2843, 2022. DOI: 10.1109/TC.2020.3017930.

[41] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture", in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 2019, pp. 253–260. DOI: 10.1109/DSD.2019.00045.

[42] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture", in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 2019, pp. 253–260. DOI: 10.1109/DSD.2019.00045.

[43] P. L. Montgomery, "Modular multiplication without trial division", *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[44] *Nantucket - a submission to the zprize competition under the category, "accelerating ntt operations on an fpga"*, https://github.com/supranational/zprize-fpga-ntt, 2023.

[45] K. Navi, A. S. Molahosseini, and M. Esmaeildoust, "How to teach residue number system to computer scientists and engineers", *IEEE Transactions on Education*, vol. 54, no. 1, pp. 156–163, 2011. DOI: 10.1109/TE.2010.2048329.

[46] D. T. Nguyen, V. B. Dang, and K. Gaj, "High-level synthesis in implementing and benchmarking number theoretic transform in lattice-based post-quantum cryptography using software/hardware codesign", in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, Eds., Cham: Springer International Publishing, 2020, pp. 247–257, ISBN: 978-3-030-44534-8.

[47] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems", *IEEE Embedded Systems Letters*, vol. 12, no. 4, pp. 133–136, 2020. DOI: 10.1109/LES.2019.2960457.

[48] T. Plantard, "Efficient word size modular arithmetic", *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1506–1518, 2021. DOI: 10.1109/TETC.2021.3073475.

[49] J. M. Pollard, "The fast fourier transform in a finite field", *Mathematics of computation*, vol. 25, no. 114, pp. 365–374, 1971.

[50] Y. Polyakov, R. Rohloff, G. W. Ryan, and D. Cousins, *PALISADE lattice cryptography library (release 1.11.5)*, https://palisade-crypto.org/, 2021.

[51] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers", in *Progress in Cryptology – LATINCRYPT 2015*, K. Lauter and F. Rodríguez-Henríquez, Eds., Cham: Springer International Publishing, 2015, pp. 346–365, ISBN: 978-3-319-22174-8.

[52] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data", in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[53] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor", in *Cryptographic Hardware and Embedded Systems – CHES 2014*, L. Batina and M. Robshaw, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391, ISBN: 978-3-662-44709-3.

[54] N. Samardzic, A. Feldmann, A. Krastev, *et al.*, "F1: A fast and programmable accelerator for fully homomorphic encryption", in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[55] A. Satriawan, I. Syafalni, R. Mareta, I. Anshori, W. Shalannanda, and A. Barra, "Conceptual review on number theoretic transform and comprehensive review on its implementations", *IEEE Access*, 2023.

[56] *Microsoft SEAL (release 4.1)*, https://github.com/Microsoft/SEAL, Microsoft Research, Redmond, WA., Jan. 2023.

[57] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "Cuda-accelerated rns multiplication in word-wise homomorphic encryption schemes", *Cryptology ePrint Archive*, 2022.

[58] V. Shoup *et al.*, "Ntl: A library for doing number theory", 2001.

[59] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data", in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398. DOI: 10.1109/HPCA.2019.00052.

[61] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga", *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020. DOI: 10.1109/TC.2020.2988765.

[66] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, "Sok: Fully homomorphic encryption accelerators", *arXiv preprint arXiv:2212.01713*, 2022.

External Links

[3] "Alveo u50 data center accelerator card", [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u50.html.

[30] "Guide to gperftools", [Online]. Available: https://developer.ridgerun.com/wiki/index.php/Profiling_with_GPerfTools.

[60] "Static, shared dynamic and loadable linux libraries", [Online]. Available: http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html.

[62] "Vitis high-level synthesis user guide", [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction.

[63] "Vitis hls introductory examples reporitory", [Online]. Available: https://github.com/Xilinx/Vitis-HLS-Introductory-Examples.

[64] "Vitis unified software platform documentation: Application acceleration development", [Online]. Available: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis.

[65] "Xilinx runtime (xrt) api", [Online]. Available: https://xilinx.github.io/XRT/master/html/xrt_native_apis.html.