Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

5-2019

Design of a Flexible Schoenhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis

Kevin Millar
kdm8162@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Millar, Kevin, "Design of a Flexible Schoenhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis" (2019). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Design of a Flexible Schönhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis

Kevin Millar

Design of a Flexible Schönhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis

Kevin Millar

May 2019

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

R·I·T | Kate Gleason
College of ENGINEERING

Department of Computer Engineering

Design of a Flexible Schönhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis

Kevin Millar

Committee Approval:

Dr. Marcin Łukowiak *Advisor* Date
RIT, Department of Computer Engineering

Dr. Stanisław Radziszowski Date
RIT, Department of Computer Science

Dr. Sonia López Alarcón Date
RIT, Department of Computer Engineering

Acknowledgments

I would like to thank my family for all of their support throughout the years, I would not be where I am today without them. Specifically, thank you Mom, Dad, Caela, Kristen, Matthew, Kelly, Mary, Natalie, Katie and Brian.

I would like to thank my girlfriend of seven years, Sarah Hannigan. Thank you for all of your support and putting up with me even if I had to work on my thesis while we talked.

I would like to thank all of my friends for helping me and pushing me to keep working even when things seemed impossible. I would especially like to thank Humza Syed, no one works harder than you do, and I have been incredibly fortunate to have you as my closest friend and roommate these past five years. I would also like to thank Stephanie Soldavini, Dakota Folger, Michael Shullick, Chris Fernandez, Max Proskauer, Andrew Ramsey, Emily Reynolds, Hunter Newman, Andrew Hamrick, Nick Higby, Nathan Reed, and the RIT Honors squad. You have all been incredible and supportive friends, and I wish I could write a paragraph about each and every one of you.

I would like to thank my friends who worked alongside me in the Applied Cryptography and Information Security (ACIS) lab including Cody Tinker, Prathibha Rama, Eric Scheler, Jason Blocklove, and Thomas Cenova. Your insight and company were always greatly appreciated.

Finally, I'd like to thank my professors, advisors, and committee members who have made all of this possible: Dr. Kurdziel for sharing his industry experience and teaching me the basics of cryptography through his entertaining lectures, Dr. Radziszowski for sharing his deep understanding and passion for the mathematics behind cryptography, Dr. López Alarcón for her help and support, and Dr. Łukowiak for guiding me through the research process and encouraging me to think more about the big picture.

Abstract

Homomorphic Encryption (HE) is a promising field because it allows for encrypted data to be sent to and operated on by untrusted parties without the risk of privacy compromise. The benefits and applications of HE are far reaching, especially in regard to cloud computing. However, current HE solutions require resource intensive arithmetic operations such as high precision, high degree polynomial multiplication resulting in a minimum computational complexity of $O(n \log n)$ on standard CPUs though application of the Fast Fourier Transform (FFT). These operations result in poor overall performance for HE schemes in software and would benefit greatly from hardware acceleration.

This work aims to accelerate the multi-precision arithmetic operations used in HE with specific focus on an implementation of the Schönhage-Strassen FFT based multiplication algorithm. It is to be incorporated into a larger HE library of arithmetic functions tuned for High-Level Synthesis (HLS) that enables flexible solutions for hardware/software systems on reconfigurable cloud resources. Although this project was inspired by HE, it could be incorporated within a generic mathematical library and support other domains. The developed FFT based polynomial multiplier exhibits flexibility in the selection of security parameters facilitating its use in a wide range of HE schemes and applications. The design also displayed substantial speedup over the polynomial multiplication functions implemented in the Number Theory Library (NTL) utilized by software based HE solutions.

Contents

Signature Sheet **i**

Acknowledgments **ii**

Abstract **iii**

Table of Contents **iv**

List of Figures **vi**

List of Tables **vii**

Acronyms **x**

1 Introduction **1**
1.1 Motivation 1
1.2 Objective 2

2 Background **4**
2.1 Related Work 4
2.2 High-Level Synthesis 5
2.3 Homomorphic Encryption 5
2.3.1 Learning with Errors 6
2.3.2 Ring Learning with Errors 6
2.3.3 Basic Scheme 8
2.4 Arithmetic Software Libraries 10
2.5 Multiplication Algorithms 11
2.6 Polynomial vs. Integer Multiplication 13
2.7 Basic Convolution 15
2.8 Discrete Fourier Transform 15
2.9 Fast Fourier Transform 16
2.10 FFT Polynomial Multiplication 18
2.11 FFT Modular Polynomial Multiplication 19
2.12 FFT Integer Multiplication 21
2.13 Modular Reduction Algorithms 22

2.13.1 Barret Reduction . . . 22
2.13.2 Montgomery Reduction . . . 23

3 FFT Based Integer Multiplier 25
3.1 Initial Implementation . . . 25
3.2 Arbitrary Precision Data Types . . . 27
3.3 Arithmetic Modulo $2^N + 1$ . . . 28
3.3.1 Addition . . . 29
3.3.2 Multiplication by 2^k . . . 29
3.3.3 Subtraction . . . 30
3.3.4 Division by 2^k . . . 30
3.3.5 Modular Reduction . . . 31
3.3.6 Results . . . 31
3.4 Parameter Selection . . . 33
3.5 Results . . . 34

4 FFT Based Polynomial Multiplier 37
4.1 Initial Implementation . . . 37
4.2 Loop Structure of the FFT . . . 42
4.3 Pipelining . . . 46
4.4 Loop Unrolling . . . 52
4.5 Design Parameters . . . 56
4.6 Initial Results . . . 57
4.7 Resource Allocation . . . 63
4.8 Synthesis Results . . . 64
4.9 Implementation Results . . . 72

5 Conclusion 78

Bibliography 79

List of Figures

1.1 Operation on private data in the cloud through a conventional encryption scheme . . . 1
1.2 Operation on private data in the cloud through a homomorphic encryption scheme . . . 2
1.3 Anticipated design flow of an HE application utilizing the accelerated HLS library developed in this work . . . 3

2.1 Design flow and deployment of an HLS application in the cloud . . . 5

4.1 Basic block diagram of the FFT based polynomial multiplier . . . 38
4.2 Structure of an 8-point decimation-in-frequency FFT . . . 43
4.3 Butterfly computation of a decimation-in-frequency FFT . . . 43
4.4 Structure of an 8-point decimation-in-time FFT . . . 44
4.5 Butterfly computation of a decimation-in-time FFT . . . 45
4.6 Basic block diagram of the FFT with "ping-pong" memory buffer . . . 48
4.7 Index mapping of a 16-point FFT . . . 53
4.8 Speedup of the pipelined and loop unrolled polynomial multiplier with resource sharing versus NTL . . . 67
4.9 Speedup of the pipelined polynomial multiplier with resource sharing versus NTL . . . 71
4.10 Implementation speedup of the pipelined and loop unrolled polynomial multiplier with resource sharing versus NTL for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 . . . 74
4.11 Implementation speedup of the pipelined polynomial multiplier with resource sharing versus NTL for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 . . . 76

List of Tables

2.1 Recommended Security Parameters for RLWE-based HE Schemes [19] 8
2.2 Computational complexities of common multiplication algorithms . . 12
2.3 Operand size thresholds for multiplication using Multiple Precision Integers and Rationals (MPIR) [25] 13

3.1 Available resources for part xczu9eg-ffvb1156-2-i-es2 26
3.2 Synthesis results of the base integer multiplier 27
3.3 Synthesis results of the integer multiplier with arbitrary precision data types 28
3.4 Synthesis results of the integer multiplier with efficient arithmetic modulo $2^N + 1$ 32
3.5 Synthesis results of the integer multiplier with efficient modular reduction by $2^N + 1$ 32
3.6 Final synthesis results of the integer multiplier 33
3.7 Efficiency of the parameter k for various operand sizes 34
3.8 Synthesis results of the final integer multiplier, values with * indicate that they do not fit on the target device 35
3.9 Timing results of the final integer multiplier 36

4.1 Synthesis results of the initial polynomial multiplier 40
4.2 Timing results of the initial polynomial multiplier 40
4.3 Timing results of the polynomial multiplier with main loops pipelined 41
4.4 Timing results of the polynomial multiplier with improved FFT loop structure 46
4.5 Timing results of the improved FFT 46
4.6 Timing results of the FFT with pipelined loops 51
4.7 Timing results of the polynomial multiplier with pipelined FFT . . . 51
4.8 Timing results of the FFT with partially unrolled *fft_group* loop . . . 56
4.9 Timing results of the polynomial multiplier with partially unrolled FFT loops 56
4.10 Synthesis results of the base polynomial multiplier for each security configuration 57
4.11 Resource utilization of the base polynomial multiplier for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2 58

4.12 Timing results of the base polynomial multiplier for each security configuration . . . 59
4.13 Synthesis results of the pipelined and loop unrolled polynomial multiplier for each security configuration . . . 60
4.14 Resource utilization of the pipelined and loop unrolled polynomial multiplier for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2 . . . 61
4.15 Timing results of the pipelined and loop unrolled polynomial multiplier 62
4.16 Synthesis results of the pipelined and loop unrolled polynomial multiplier with resource sharing for each security configuration . . . 64
4.17 Resource utilization of the pipelined and loop unrolled polynomial multiplier with resource sharing for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2 . . . 65
4.18 Timing results of the pipelined and loop unrolled polynomial multiplier with resource sharing for each security configuration . . . 66
4.19 Synthesis results of the pipelined polynomial multiplier with resource sharing for each security configuration . . . 68
4.20 Resource utilization of the pipelined polynomial multiplier with resource sharing for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2 . . . 69
4.21 Timing results of the pipelined polynomial multiplier with resource sharing for each security configuration . . . 70
4.22 Available resources for various Xilinx FPGAs . . . 70
4.23 Implementation results of the pipelined and loop unrolled polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 . . . 72
4.24 Implementation resource utilization of the pipelined and loop unrolled polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 . . . 73
4.25 Implementation timing results of the pipelined and loop unrolled polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 . . . 73
4.26 Implementation results of the pipelined polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 . . . 74

4.27 Implementation resource utilization of the pipelined polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 75
4.28 Implementation timing results of the pipelined polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2 75

Acronyms

ASIC Application Specific Integrated Circuit

AWS Amazon Web Services

BGV Brakerski-Gentry-Vaikuntanathan

BRAM Block RAM

DFT Discrete Fourier Transform

DIF Decimation In Frequency

DIT Decimation In Time

DSP Digital Signal Processing

FFT Fast Fourier Transform

FHE Fully Homomorphic Encryption

FLINT Fast Library for Number Theory

FPGA Field-Programmable Gate Array

FV Fan-Vercauteren

GMP GNU Multiple Precision

HDL Hardware Description Language

HE Homomorphic Encryption

HLS High-Level Synthesis

IFFT Inverse Fast Fourier Transform

II Initiation Interval

LUT Lookup Table

LWE Learning with Errors

MPFR Multiple Precision Floating-Point Reliable

MPIR Multiple Precision Integers and Rationals

NTL Number Theory Library

NTT Number Theoretic Transform

RLWE Ring Learning with Errors

RTL Register Transfer Level

SHE Somewhat Homomorphic Encryption

WNS Worst Negative Slack

Chapter 1

Introduction

1.1 Motivation

As cloud computing grows in popularity, solutions like Amazon Web Services (AWS) [1] are becoming more desirable as an affordable means by which to utilize computing resources. Though cloud resources offer many benefits, the off-loading of private data to third party systems for computation introduces new privacy risks. Conventional cryptographic solutions do not solve this problem as protected data requires decryption to allow for operation on these shared computing resources as illustrated in Figure 1.1. This image shows the transfer of multiple encrypted plaintext data to

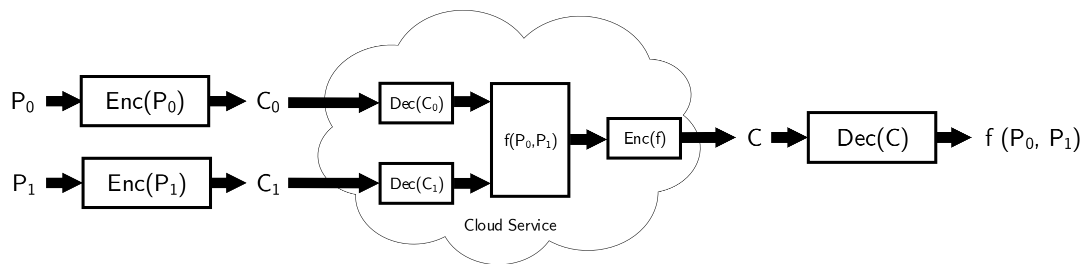

Figure 1.1: Operation on private data in the cloud through a conventional encryption scheme

a cloud service for operation. The receiving platform must decrypt the data to perform operations between the underlying plaintexts, and the result is then encrypted and sent back to the user. A potential solution to this problem is Homomorphic

Encryption (HE) which allows for arbitrary operations to be performed on encrypted data without exposing the underlying plaintext to untrusted parties. This allows for data secured with an HE scheme to be operated on homomorphically and the result returned to the user without security compromise as depicted in Figure 1.2. This

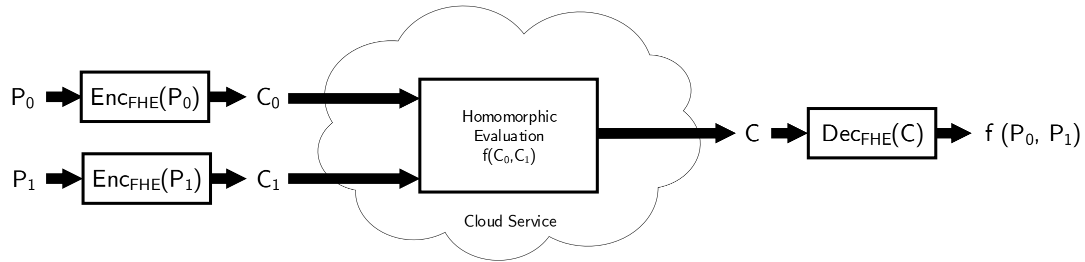

Figure 1.2: Operation on private data in the cloud through a homomorphic encryption scheme

illustration shows the transfer of encrypted data to a cloud service with operations performed homomorphically on the underlying plaintext. The still encrypted result is then returned to the user for decryption. This removes the necessity of decrypting the data before operation and retains the privacy of the data even when evaluated on cloud resources. Though HE schemes do exist, current solutions require complex arithmetic operations that are computationally resource intensive.

1.2 Objective

The goal of this work was to accelerate the resource intensive arithmetic operations heavily relied upon in many HE schemes to enable secure cloud computing. This was achieved through the continued development of a library containing arithmetic functions accelerated through High-Level Synthesis (HLS) to allow for the flexible design of hardware/software systems on reconfigurable cloud resources. The development flow of an application utilizing this library is shown in Figure 1.3. This diagram shows the design flow of an HE application in which the computationally intensive

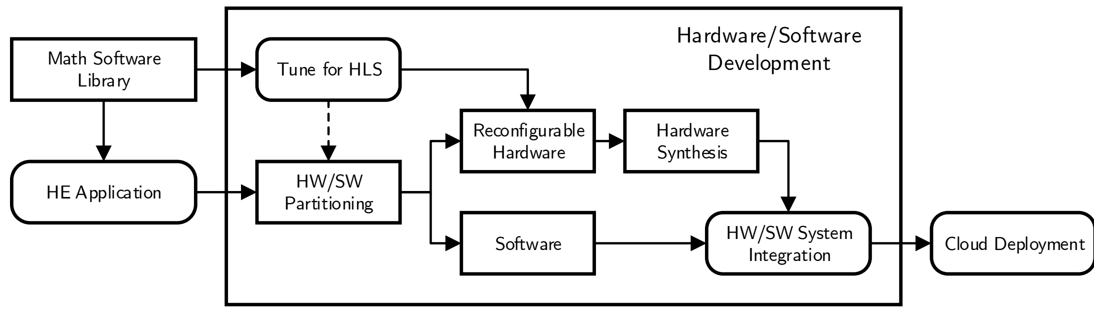

Figure 1.3: Anticipated design flow of an HE application utilizing the accelerated HLS library developed in this work

operations are partitioned for hardware acceleration through the HLS accelerated math library. These operations are then designated for execution on a hardware co-processor that is synthesized for the available target reconfigurable resources. Integration is performed by establishing communication between the main program running on a conventional CPU and the reconfigurable resources, and the full system is deployed within a cloud environment. Although this project directly targeted HE, the functions developed could be incorporated within a generic mathematical library and support theoretical computations. This focused explicitly on the design and development of a flexible Fast Fourier Transform (FFT) based multiplier through HLS to offer improved performance over software solutions.

Chapter 2

Background

2.1 Related Work

The hardware design of large-scale FFT multipliers for HE has been explored in various works such as [2, 3, 4]. Each of these works designed custom hardware targeting either Application Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) platforms using conventional Hardware Description Language (HDL) development and implementation techniques. Michael Foster began work on a potential case study for the creation of an arithmetic library for hardware accelerated HE in [5]. This work designed and developed a flexible Karatsuba multiplier with Vivado HLS achieving a maximum theoretical speedup up to 136 times the Fast Library for Number Theory (FLINT) arithmetic software library. A similar accelerator for HE was created using the Karatsuba multiplication algorithm through the application of hardware/software co-design techniques in [6] specifically targeting the Fan-Vercauteren (FV) HE scheme introduced in [7]. A hardware accelerated FFT algorithm for HE acceleration was designed in [8] with Vivado HLS achieving a maximum speedup of 6.9 times the same algorithm run on an Intel Core i7-5600U CPU at 2.6GHz. Mkhinini et. al designed a flexible RNS-based large polynomial multiplier through HLS for HE and achieved significant speedup over software implementations [9, 10, 11].

2.2 High-Level Synthesis

HLS is a method by which software written in a programming language such as C/C++ can be converted into HDL, synthesized, and implemented in hardware. This allows for a simpler method of hardware/software co-design and deployment of flexible hardware accelerated designs. The inclusion of FPGAs in the cloud allows for the deployment of hardware accelerated designs through HLS as shown in Figure 2.1.

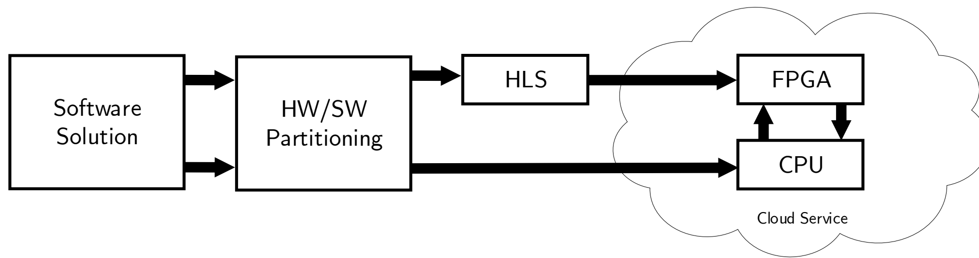

Figure 2.1: Design flow and deployment of an HLS application in the cloud

The HLS tool utilized in this work is Xilinx Vivado HLS [12] which transforms a C/C++ program into a synthesizable HDL model for Xilinx FPGAs. This tool provides various directives that can be applied to the software design to guide synthesis toward the desired hardware structure. These directives include the UNROLL directive which specifies the number of times a loop should be unrolled to implement parallel hardware processors to decrease latency and the PIPELINE directive which specifies the placement of registers between hardware operations to allow for increased throughput.

2.3 Homomorphic Encryption

HE cryptographic schemes allow for an arbitrary number of operations to be performed on encrypted data. This has far reaching benefits as they allow for potentially

sensitive information to be encrypted and operated on by untrusted parties without compromising the underlying data. Though Somewhat Homomorphic Encryption (SHE) schemes allowing for a limited number of operations on encrypted data have existed for a long time, the first Fully Homomorphic Encryption (FHE) scheme enabling boundless operations on encrypted data was introduced by Craig Gentry in 2009 [13]. This breakthrough was achieved through a scheme based on bootstrapping, the principle by which an encryption scheme evaluates its own decryption circuit, and operates over ideal lattices. Gentry later assisted in the development of a more efficient FHE scheme called Brakerski-Gentry-Vaikuntanathan (BGV) without the need of bootstrapping by utilizing a novel modulus switching technique [14, 15]. The BGV scheme is much more efficient than previous implementations and can operate over the Learning with Errors (LWE) or Ring Learning with Errors (RLWE) hardness assumptions.

2.3.1 Learning with Errors

The LWE problem was first proposed by Regev in [16]. Select a dimension $n \geq 1$, a prime modulus $q \geq 2$, and an error distribution χ over $\mathbb{Z}_q$. Establish a uniformly random secret vector $\mathbf{s} \in \mathbb{Z}_q^n$. Choose a vector $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random, select $e \in \mathbb{Z}_q$ based on the error distribution χ, and output $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$. The LWE problem thus states that for an arbitrary number of samples of the form $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$, it is computationally infeasible to determine $\mathbf{s}$ [16, 17].

2.3.2 Ring Learning with Errors

The RLWE problem is an extension of the LWE problem over algebraic rings first introduced in [18]. This allows for simpler computations as it reduces the inherent quadratic overhead associated with the basic implementation of LWE. An informal definition of the RLWE problem is presented here for simplicity. Select a dimension

$n \geq 1$ where n is a power of 2, a prime modulus $q \geq 2$ such that $1 = q \mod 2n$, and $f(x) = x^n + 1 \in \mathbb{Z}[x]$. Let $\mathbb{R} = \mathbb{Z}[x]/\langle f(x) \rangle$, $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$, and χ be an error distribution over $\mathbb{R}$. Establish a uniformly random secret polynomial $\mathbf{s} = \mathbf{s}(x) \in \mathbb{R}_q$. Choose a polynomial $\mathbf{a} = \mathbf{a}(x) \in \mathbb{R}_q$ uniformly at random, generate $e = e(x) \in \mathbb{R}$ based on the error distribution χ, and output$(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$. Like LWE, the RLWE problem thus states that for an arbitrary number of samples of the form $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$, it is computationally infeasible to determine $\mathbf{s}$ [17, 18].

Security

The security of RLWE is currently an area of research. It is heavily dependent upon the noise e added to the coefficients along with the selection of n and the prime modulus q. An ongoing effort is being made to standardize HE schemes based on RLWE for power 2 cyclotomic rings. The security analysis in [19] presents recommended parameters n and q for various levels of the security parameter λ. Though separate parameters are provided for uniform, ternary, and error distributions, the parameters based on the error distribution will be used here as shown in Table 2.1. These recommendations are made based on the LWE-estimator tool introduced in [20] which determines parameters for a given λ based on the estimated complexities of currently known attacks on RLWE. These attacks include the unique shortest vector attack (uSVP) [21], the decoding attack (dec) [22, 20], and the dual attack (dual) [23]. The total size in bits of a single plaintext is calculated by multiplying the number of polynomial coefficients n and the size of the ciphertext modulus q in bits. Note that increasing the polynomial degree allows for larger coefficients while achieving the same level of security. The typical plaintext space is equal to n with a single plaintext bit encoded within each coefficient, but other packing techniques may be used.

Table 2.1: Recommended Security Parameters for RLWE-based HE Schemes [19]

n	λ	$\log q$	Size (bits)	uSVP	dec	dual
1,024	256	19	19,456	269.9	280.5	259.5
1,024	192	22	22,528	203.6	211.2	200.8
1,024	128	31	31,744	130.6	133.8	129.6
2,048	256	33	67,584	263.8	270.7	258.1
2,048	192	42	86,016	194.0	197.6	190.6
2,048	128	58	118,784	132.1	132.4	130.2
4,096	256	62	253,952	266.0	268.9	259.3
4,096	192	80	327,680	195.6	196.1	192.2
4,096	128	113	462,848	131.9	129.4	128.8
8,192	256	123	1,007,616	259.6	259.4	256.2
8,192	192	157	1,286,144	195.4	192.8	192.3
8,192	128	223	1,826,816	132.3	128.3	128.0
16,384	256	243	3,981,312	259.5	256.6	259.3
16,384	192	310	5,079,040	196.4	192.4	193.9
16,384	128	443	7,258,112	133.0	128.1	129.3
32,768	256	481	15,761,408	261.0	257.0	256.2
32,768	192	616	20,185,088	197.4	192.5	192.2
32,768	128	886	29,032,448	133.4	128.2	129.4

2.3.3 Basic Scheme

The following is a basic outline of the BGV scheme based on RLWE [14, 15, 19].

- ParamGen(λ, L) $\rightarrow q, n, \chi$: Given security parameter λ (e.g. 128, 192, or 256) and the maximum multiplicative depth of the circuit L, the parameters q, n, and χ are generated where q is the ciphertext modulus, n is the message

polynomial degree, and χ is a discrete Gaussian distribution.

- SecKeygen(q, n, and χ) $\rightarrow sk$: The secret key sk is generated as a polynomial $\mathbf{s}(x)$ generated by selecting random elements from the error distribution χ.

- PubKeygen(q, n, and χ) $\rightarrow pk$: The public key pk is generated as a pair of polynomials $(\mathbf{a}, \mathbf{b}) = (\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$ where e is a polynomial sampled from the error distribution χ.

- SecEncrypt(sk, P) $\rightarrow C$: The secret key encryption function first maps the plaintext P over the ring $R_p = \mathbb{Z}_p[x]/\langle f(x) \rangle$ where $f(x) = x^n + 1 \in \mathbb{Z}[x]$ and p is the plaintext modulus. A random polynomial $\mathbf{a}(x)$ is sampled from the ring R_p and the ciphertext is output as a pair of polynomials $(c_0, c_1) = (a, a \cdot \mathbf{s} + p \cdot e + P)$ where e is a polynomial sampled from the error distribution χ.

- PubEncrypt(pk, P) $\rightarrow C$: The public key encryption function first maps the plaintext P over the ring $R_p = \mathbb{Z}_p[x]/\langle f(x) \rangle$ where $f(x) = x^n + 1 \in \mathbb{Z}[x]$ and p is the plaintext modulus. Three random polynomials $\mathbf{r}(x)$, $\mathbf{f}(x)$, and $\mathbf{f}'(x)$ are sampled from the ring R_p, and the ciphertext is output as a pair of polynomials $(c_0, c_1) = (a \cdot \mathbf{r} + \mathbf{f}, b \cdot \mathbf{r} + p \cdot \mathbf{f}' + P)$.

- Decrypt(sk, C) $\rightarrow P$: The decryption function reduces the ciphertext C by computing $c' = c_0 \cdot \mathbf{s} + c_1$ over R_q and calculating $c' \mod p$.

- EvalAdd(C_0, C_1) $\rightarrow C_2$: Addition is performed homomorphically between two ciphertexts $C_0 = (c_{0,0}, c_{0,1})$ and $C_1 = (c_{1,0},\ c_{1,1})$ by producing the output $C_2 = (c_{0,0} + c_{1,0},\ c_{0,1} + c_{1,1})$.

- EvalMult(C_0, C_1) $\rightarrow C_2$: Multiplication is performed homomorphically between two ciphertexts $C_0 = (c_{0,0}, c_{0,1})$ and $C_1 = (c_{1,0}, c_{1,1})$ by producing the output $C_2 = (c_{0,0} \cdot c_{1,0},\ c_{0,0} \cdot c_{1,1} + c_{0,1} \cdot c_{1,0},\ c_{0,1} \cdot c_{1,1})$.

2.4 Arithmetic Software Libraries

This section contains summaries of a subset of mathematic software libraries implemented in C. These libraries contain efficient algorithms that enable fast computations for software implementations of HE. Analysis of these libraries will allow for insight into the design of algorithms for arithmetic libraries and a benchmark by which to compare the final hardware accelerated algorithm developed in this work.

GMP

The GNU Multiple Precision (GMP) arithmetic library is a portable library written in C allowing for multiple precision arithmetic on integers, rational numbers, and floating-point numbers [24]. The goal of GMP is to provide high speed arithmetic for high precision data types that are outside of the basic C data types. The library is developed with a general emphasis on speed with optimized assembly code and full word arithmetic.

MPIR

The Multiple Precision Integers and Rationals (MPIR) library is a fork of the GMP library written in assembly language and C [25]. The goal of the library is the same as GMP, emphasizing speed for higher precision data types than directly supported by C.

MPFR

The Multiple Precision Floating-Point Reliable (MPFR) library is a portable library written in C for arbitrary precision arithmetic on floating-point numbers based on GMP [26, 27]. MPFR is unique in that the code is fully portable and independent of machine word size, the precision of bits can be set exactly for each variable, and

advanced rounding modes are supported on top of the four rounding modes specified in the IEEE 754-1985 standard.

NTL

The Number Theory Library (NTL) is a portable, high-performance library written in C++ to allow for arithmetic operations over finite fields [28]. NTL is compatible with Unix, MacOS, and Windows and can be built with GMP to improve performance.

PARI/GP

The PARI library is a computer algebra library written in C for number theory operations [29]. The three main benefits of the PARI library are its speed, mathematician friendly data types, and extensive support of number theory operations. Another benefit of PARI is gp, a user shell with access to PARI functions that can be used as a programmable number theory calculator. The interactive shell gp operates on the scripting language GP.

FLINT

The FLINT library was first introduced in [30, 31] as an arithmetic library for number theory computations. The initial goal of FLINT was to match the functionality of NTL and eventually act as an alternative to PARI. FLINT is written in C with assembly optimizations and is threadsafe.

2.5 Multiplication Algorithms

Most arithmetic software libraries contain implementations of several multiplication algorithms of varying complexity. These methods introduce benefits and drawbacks stemming from the overall asymptotic complexity and inherent overhead associated with each of them. The most common multiplication algorithms and their respective

computational complexities are shown in Table 2.2. The Schönhage-Strassen algorithm was the asymptotically fastest known integer multiplication algorithm until the introduction of Fürer's algorithm in 2007 [32]. Implementation of Fürer's algorithm is currently impractical, however, as it does not achieve performance benefits over Schönhage-Strassen until very large values of n are reached.

Table 2.2: Computational complexities of common multiplication algorithms

Algorithm	Complexity
Schoolbook	$O(n^2)$
Karatsuba [33]	$O(n^{\log_2 3})$
k-way Toom-Cook [34, 35]	$O(n^{\log(2k-1)/\log k})$
Schönhage-Strassen [36, 37]	$O(n \log n \log \log n)$
Fürer [32]	$O(n \log n \; 2^{3 \log^* n})$

Because of the differences between multiplication algorithms, each library selects the best implementation for a specific application through a method generally based upon both the size of the operands and the target computer architecture. Table 2.3 shows the operand size threshold for each multiplication algorithm in the MPIR library. Though these values are strictly for MPIR, similar thresholding methods are used by other libraries.

Table 2.3: Operand size thresholds for multiplication using MPIR [25]

Algorithm	Threshold for Generic Architecture with 32-bit Limbs (bits)	Threshold for x86_64 Haswell with 64-bit Limbs (bits)
Basecase (Schoolbook)	N/A	N/A
Karatsuba	1,024	1,024
Toom-3	4,096	6,720
Toom-4	9,600	15,744
Toom-8.5	12,832	19,392
Schönhage-Strassen	128,320	249,856

2.6 Polynomial vs. Integer Multiplication

The link between polynomial and integer multiplication can be illustrated by the Kronecker-Schönhage trick. This trick (also known as Kronecker Substitution or segmentation) is a method that converts the multiplication of integers to the multiplication of polynomials and vice versa [38]. Let A be an integer represented in base β. Then A can be represented as a polynomial of the form

$$A = a(\beta) = \sum_{i=0}^{n-1} a_i \beta^i$$

For example, let $A = 147$ and $B = 239$ be the integers to be multiplied with base $\beta = 10$. Then

$$a(\beta) = 1\beta^2 + 4\beta + 7$$

$$b(\beta) = 2\beta^2 + 3\beta + 9$$

with multiplication of the two polynomials resulting in

$$c(\beta) = a(\beta) \cdot b(\beta) = 2\beta^4 + 11\beta^3 + 35\beta^2 + 57\beta + 63$$

The integer result can then be recovered by evaluating $c(\beta)$ for $\beta = 10$ resulting in $c(10) = A \cdot B = 35133$. This method can also be performed in reverse to multiply polynomials as integers. Let $a(x)$ and $b(x)$ be polynomials of degree less than n with positive coefficients bounded by p. The desired computation is thus $c(x) = a(x) \cdot b(x)$. Select $X = \beta^k > np^2$ where β is the base of the coefficients. Solving for $c(X) = a(X) \cdot b(X)$ results in $c(X) = \sum_{i=0}^{n-1} c_i X^i$ where c_i is a coefficient of $C(x)$. Because each c_i is bound by a fixed multiple of the base, the coefficients of $c(x)$ can simply be extracted. For example, suppose we want to perform the multiplication

$$c(x) = (6x^2 + 5x + 1) \cdot (9x^2 + 3x + 7)$$

with base $\beta = 10$ where the degree of the polynomials is less than $n = 3$ and each coefficient is bounded by $p = 9$. Then $X = 10^3 > 3 \cdot 9^2$ and the multiplication of the polynomials evaluated at X results in

$$c(X) = 006\ 005\ 001 \cdot 009\ 003\ 007$$
$$c(X) = 54\ 063\ 066\ 038\ 007$$

from which the polynomial product can be extracted as

$$c(x) = 54x^4 + 63x^3 + 66x^2 + 38x + 7$$

2.7 Basic Convolution

Let R be an arbitrary ring with vectors a and b of length n composed of elements $a_i, b_i \in R$ for $i = 0, 1, \ldots, n-1$. Then the convolution between vectors a and b results in a vector c of length $2n$ where

$$c_i = \sum_{j=0}^{n-1} a_j b_{i-j}, \; i = 0, 1, \ldots, 2n-1 \tag{2.1}$$

and $a_i = b_i = 0$ for $i < 0$ or $i \geq n$ [39]. Note that if a and b are composed of polynomial coefficients, then the convolution of a and b is equivalent to polynomial multiplication. For example if $a = [5, 3, 1]$ and $b = [4, 2, 3]$ represent the polynomials $a(x) = 5x^2 + 3x + 1$ and $b(x) = 4x^2 + 2x + 3$, then the convolution between the vectors results in

$$c_0 = a_0 \cdot b_0 = 5 \cdot 4 = 20$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 = 5 \cdot 2 + 3 \cdot 4 = 22$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 = 5 \cdot 3 + 3 \cdot 2 + 1 \cdot 4 = 25$$

$$c_3 = a_1 \cdot b_2 + a_2 \cdot b_1 = 3 \cdot 3 + 1 \cdot 2 = 11$$

$$c_4 = a_2 \cdot b_2 = 1 \cdot 3 = 3$$

$$c_5 = 0$$

representing the product polynomial $c(x) = 20x^4 + 22x^3 + 25x^2 + 11x + 3$.

2.8 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) has the unique property of converting convolution to a simple point-wise product [40]. This feature is known as the convolution theorem and forms the basis for many useful applications of the DFT, and can be

applied to any vector x of length n within a field or ring in which n^{-1} exists. Let ω be a primitive n-th root of unity within the target domain, hence $\omega^n = 1$ and $\omega^k \neq 1$, for $1 \leq k < n$. The DFT can then be defined as

$$X_i = \sum_{j=0}^{n-1} x_j \omega^{ji} \tag{2.2}$$

By extension, the inverse DFT can be defined as

$$x_j = \frac{1}{n} \sum_{i=0}^{n-1} X_i \omega^{-ji} \tag{2.3}$$

A typical signal processing application of the DFT is to transform samples of a signal measured in the time domain to their frequency domain counterparts through application of $\omega = e^{2\pi i/n}$ in C, the complex numbers field, in which $i = \sqrt{-1}$. However, the transform can be applied to a vector in any field or ring with an appropriately selected primitive root of unity. The application of a DFT over a ring or field is sometimes referred to as the Number Theoretic Transform (NTT). Naive computation of the DFT yields complexity $O(n^2)$, but many algorithms exist that allow for faster computation.

2.9 Fast Fourier Transform

The FFT is a classifier for any algorithm that resolves the DFT with a computational complexity of $O(n \log n)$. These algorithms apply a divide and conquer technique to reduce the computation of the DFT into two smaller problems.

$$X_i = \sum_{j=0}^{n-1} x_j \omega^{ji} = \sum_{j=0}^{\frac{n}{2}-1} x_{2j} \omega^{2ji} + \sum_{j=0}^{\frac{n}{2}-1} x_{2j+1} \omega^{(2j+1)i} \tag{2.4}$$

These methods require the length of the input vector to be of the form $n = 2^k$, but the FFT can be applied to vectors of arbitrary length by zero padding until a length of this form is achieved. The most popular iterative forms of the FFT exploit a reordering technique in which each element is swapped with its reverse-binary index. This allows for in place computations to be performed where the elements of the original vector are directly replaced with the calculated DFT values [40]. Depending on implementation, the FFT can be of the form Decimation In Time (DIT) or Decimation In Frequency (DIF) stemming from the typical application of the DFT for Digital Signal Processing (DSP) applications. The Cooley-Tukey DIT algorithm performs the scrambling operation prior to the DFT calculations as shown in Algorithm 1.

Algorithm 1 Cooley-Tukey FFT Algorithm (DIT) [40]

Input: vector x of length n, primitive n-th root of unity ω
Output: transformed vector x

1: REORDER(x)
2: **for** $m \leftarrow 1$ **to** n **by** $2m$ **do**
3:     **for** $j \leftarrow 0$ **to** m **do**
4:         $a \leftarrow \omega^{jn/2m}$
5:         **for** $i \leftarrow j$ **to** n **by** $2m$ **do**
6:             $(x_i,\ x_{i+m}) \leftarrow (x_i + ax_{i+m},\ x_i - ax_{i+m})$

Conversely, the Gentleman-Sande DIF algorithm performs the scrambling operation after computation of the DFT as shown in Algorithm 2.

Algorithm 2 Gentleman-Sande FFT Algorithm (DIF)[40]

Input: vector x of length n, primitive n-th root of unity ω
Output: transformed vector x

1: **for** $m \leftarrow n/2$ **to** 1 **by** $m/2$ **do**
2:     **for** $j \leftarrow 0$ **to** m **do**
3:         $a \leftarrow \omega^{jn/2m}$
4:         **for** $i \leftarrow$ j **to** n **by** $2m$ **do**
5:             $(x_i,\ x_{i+m}) \leftarrow (x_i + x_{i+m},\ a(x_i - x_{i+m}))$
6: REORDER(x)

The Inverse Fast Fourier Transform (IFFT) can be computed using a forward FFT algorithm by replacing ω with ω^{-1} and multiplying each element of the resulting vector by n^{-1}. Because the DIF and DIT algorithms perform the reordering operation before and after the DFT calculations, respectively, a DIF algorithm can be used to calculate the FFT and a DIT method can be used to compute the IFFT eliminating the need for the reordering procedure to be implemented and performed discretely.

2.10 FFT Polynomial Multiplication

The FFT can be used to simplify the operation of polynomial multiplication through application of the convolution theorem. Let $\mathbb{R}$ be an arbitrary ring with polynomials $a(x)$ and $b(x)$ of degree $n-1$ with vectors of coefficients $a = a_i \in \mathbb{R}$ and $b = b_i \in \mathbb{R}$ for $i = 0, 1, \ldots, n-1$. Then the convolution of the coefficient vectors results in a vector c with $c_i \in \mathbb{R}$ for $i = 0, 1, \ldots, 2n-2$ representing the product polynomial $c(x)$ of degree $2n-2$. Performing this convolution directly results in complexity $O(n^2)$, but this can be reduced by instead computing the FFT of the vectors a and b, performing point-wise multiplication of the vector elements, and computing the IFFT of the resulting vector to retrieve the product vector c. Therefore, convolution, and thus polynomial multiplication, becomes

$$c = IFFT^{-1}(FFT(a) \cdot FFT(b)) \tag{2.5}$$

with reduced complexity $O(n \log n)$. Note that because the result vector c has a resulting length of $2n$, the input vectors a and b must be zero padded to a length greater than or equal to $2n$ of the form 2^k in order to produce the correct result through application of the FFT. For example, if $a = [5,\ 3,\ 1]$ and $b = [4,\ 2,\ 3]$ represent the polynomials $a(x) = 5x^2 + 3x + 1$ and $b(x) = 4x^2 + 2x + 3$, the vectors must be of length greater than or equal to length $2n = 2 \cdot 3 = 6$ of the form 2^k resulting

in $2^3 = 8$. Zero padding the polynomial vectors and rearranging them for the FFT results in $a = [1, 3, 5, 0, 0, 0, 0, 0]$ and $b = [3, 2, 4, 0, 0, 0, 0, 0]$. Then, operating modulo 257 with $\omega = 4$, the FFT can be performed on each vector producing

$$FFT(a) = [9, 3, 44, 205, 93, 69, 113, 243]$$

$$FFT(b) = [9, 5, 31, 224, 75, 59, 67, 68]$$

Point-wise multiplication is then performed between the two transformed vectors resulting in

$$FFT(a) \cdot FFT(b) = [81, 15, 79, 174, 36, 216, 118, 76]$$

Finally, the product polynomial can be retrieved by performing the IFFT with $w^{-1} = 193$ yielding

$$IFFT^{-1}(FFT(a) \cdot FFT(b)) = [3, 11, 25, 22, 20, 0, 0, 0]$$

representing the product polynomial $c(x) = 20x^4 + 22x^3 + 25x^2 + 11x + 3$.

2.11 FFT Modular Polynomial Multiplication

Let a, b, and c be vectors of length n where c is the negative wrapped convolution of a and b. Then each element c_i can be calculated as

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \tag{2.6}$$

This computation is equivalent to polynomial multiplication modulo $x^n + 1$. Let ω be a primitive n-th root of unity and $\theta^2 = \omega$, then with

$$\hat{a} = (a_0, \theta a_1, \ldots, \theta^{n-1} a_{n-1})$$
$$\hat{b} = (b_0, \theta b_1, \ldots, \theta^{n-1} b_{n-1})$$

the negative wrapped convolution can be computed as

$$c = IFFT^{-1}(FFT(\hat{a}) \cdot FFT(\hat{b}))$$

Polynomial multiplication can also be performed through negative wrapped convolution with coefficients modulo a prime p of the form $p = 1 \mod 2n$ by selecting θ such that $\theta^2 = \omega \mod p$ and reducing the coefficient calculations at each stage. The application of negative wrapped convolution to perform polynomial multiplication with coefficients modulo p and the result modulo $x^n + 1$ is shown in Algorithm 3.

Algorithm 3 FFT Based Modular Polynomial Multiplication

Input: Polynomials $a(x)$ and $b(x)$ of maximum degree n with coefficients $a_i, b_i \in \mathbb{Z}_p$ for $i = 0, 1, \ldots, n-1$
Output: $c(x) = a(x) \cdot b(x) \mod (x^n + 1)$
1: Select primitive n-th-root of unity ω
2: Select θ such that $\theta^2 = \omega \mod p$
3: **for** $i \leftarrow 0$ **to** n **do**
4: $\quad (a_i, b_i) \leftarrow (\theta^i a_i, \theta^i b_i) \mod p$
5: $a \leftarrow \text{FFT}(a, \omega)$
6: $b \leftarrow \text{FFT}(b, \omega)$
7: **for** $i \leftarrow 0$ **to** n **do**
8: $\quad c_i \leftarrow a_i \cdot b_i \mod p$
9: $c \leftarrow \text{IFFT}(c, \omega^{-1})$
10: **for** $i \leftarrow 0$ **to** n **do**
11: $\quad c_i \leftarrow c_i \cdot \theta^{-i} \mod p$

2.12 FFT Integer Multiplication

The Schönhage-Strassen algorithm first introduced multiplication of large integers through an application of the FFT with a complexity of $O(n \log n \log \log n)$ [36]. This algorithm was later improved by Schönhage in 1982 to remove some of the clumsiness of the original implementation [37]. The algorithm multiplies two integers modulo 2^n+1, but standard multiplication can be achieved through selection of n large enough such that all desired operands are less than $n/2$ bits in length. Let $A, B < 2^n + 1$ be the two integers to be multiplied. Select an integer k such that $n = MK$ and $K = 2^k$. This value is necessary to decompose each input integer into K segments of M bits. The integer $A \in [0, 2^n]$ can be represented in the form $A = \sum_{i=0}^{K-1} a_i 2^{iM}$. Note that because the value 2^n requires $n + 1$ bits for representation, $0 \leq a_i < 2^M$ for $i < K - 1$ and $0 \leq a_{K-1} \leq 2^M$. Integer B is decomposed similarly, and the product of A and B can be represented as $C = \sum_{i=0}^{K-1} c_i 2^{iM} = A \cdot B \mod (2^n + 1)$. The FFT modular polynomial multiplication algorithm can then be applied through negative wrapped convolution of the decomposed integers with coefficient modulus $2^N + 1$. This requires selection of $N \geq \frac{2n}{K} + k$ to ensure that no precision is lost. If an element of the resulting vector exceeds its initial interval, the value requires an adjustment step to return the value to the correct interval by subtracting $2^N + 1$. The final integer result C can then be obtained by evaluating the resulting polynomial at 2^M. The general form of the Schönhage-Strassen integer multiplication is presented in Algorithm 4.

Algorithm 4 FFT Based Integer Multiplication (Schönhage-Strassen) [38]

Input: Integers A, B with $0 \leq A, B < 2^n + 1$; Integer K $= 2^k$ such that $n = MK$
Output: $C = A \cdot B \mod (2^n + 1)$

1: $A = \sum_{i=0}^{K-1} a_i 2^{iM}$ ▷ Decompose A
2: $B = \sum_{i=0}^{K-1} b_i 2^{iM}$ ▷ Decompose B
3: Select $N \geq \frac{2n}{K} + 1$, where N is a multiple of K; Let $\theta = 2^{\frac{N}{K}}, \omega = \theta^2$
4: **for** $i \leftarrow 0$ **to** K **do**
5: $\quad (a_i, b_i) \leftarrow (\theta^i a_i, \theta^i b_i) \mod (2^N + 1)$ ▷ Weight the inputs
6: $a \leftarrow \textsc{fft}(a, \omega)$
7: $b \leftarrow \textsc{fft}(b, \omega)$
8: **for** $i \leftarrow 0$ **to** K **do**
9: $\quad c_i \leftarrow a_i \cdot b_i \mod (2^N + 1)$ ▷ Any multiplication algorithm can be used
10: $c \leftarrow \textsc{ifft}(c, \omega^{-1})$
11: **for** $i \leftarrow 0$ **to** K **do**
12: $\quad c_i \leftarrow c_i \cdot \theta^{-i} \mod (2^N + 1)$
13: $\quad$ **if** $c_i \geq (i + 1)^{2M}$ **then**
14: $\quad\quad c_i \leftarrow c_i - (2^N + 1)$
15: $C = \sum_{i=0}^{K-1} c_i 2^{iM}$

2.13 Modular Reduction Algorithms

Many algorithms exist to perform efficient general modular reduction. Though optimizations can be performed based on the specific modulus, these implementations are restrictive and do not allow for arbitrary selection of the modulus value. The two most common modular reduction algorithms are the Montgomery and Barret's reduction techniques.

2.13.1 Barret Reduction

The Barret reduction algorithm was first introduced in [41] and operates over the basic idea that $c = a \mod b$ is equivalent to

$$c = a - b \cdot \lfloor a/b \rfloor \tag{2.7}$$

A general version of Barrett reduction is presented in Algorithm 5. This algorithm has been slightly modified as it was presented in [42] to substitute a slightly larger initial computation for a reduction in stages and a singular correction step.

Algorithm 5 Barrett Modular Reduction [42]

Input: Integer a, b with $0 \leq a < b^2$, $b > 1$; $\mu = \lfloor \beta^{2m}/b \rfloor$; $m = \lceil \log_\beta b \rceil$
Output: $c = a \mod b$
1: $q \leftarrow a \cdot \mu$
2: $q \leftarrow \lfloor q/\beta^m \rfloor$
3: $q \leftarrow q \cdot b$
4: $c \leftarrow a - q$
5: **if** $c < 0$ **then**
6: $\quad c \leftarrow c + b$
7: **return** c

2.13.2 Montgomery Reduction

Montgomery reduction was first introduced in [43] and reduces a modulo b where the modulus value b is odd and $0 \leq a < b^2$. This algorithm does not compute the residue of the input directly and instead computes an equivalent residue of the input multiplied by a constant. A general version of Montgomery reduction is presented in Algorithm 6.

Algorithm 6 Montgomery Modular Reduction [42]

Input: Integer a, b with $0 \leq a < b^2$, $b > 1$; $\rho = -1/n_0 \mod b$
Output: $c = \beta^{-k}a \mod b$

1: **for** $i \leftarrow 0$ **to** k **do**
2: $\quad \mu_i \leftarrow a_i \cdot \rho \mod \beta$
3: $\quad u \leftarrow 0$
4: $\quad$ **for** $j \leftarrow 0$ **to** k **do**
5: $\quad\quad \hat{r} \leftarrow \mu b_j + a_{i+j} + u$
6: $\quad\quad a_{i+j} \leftarrow \hat{r} \mod \beta$
7: $\quad\quad u \leftarrow \lfloor \hat{r}/\beta \rfloor$
8: $\quad$ **while** $u > 0$ **do**
9: $\quad\quad j \leftarrow j + 1$
10: $\quad\quad a_{i+j} \leftarrow a_{i+j} + u$
11: $\quad\quad u \leftarrow \lfloor a_{i+j}/\beta \rfloor$
12: $\quad\quad a_{i+j} \leftarrow a_{i+j} \mod \beta$
13: $c \leftarrow \lfloor a/\beta^k \rfloor$
14: **if** $c \geq b$ **then**
15: $\quad c \leftarrow c - b$
16: **return** c

Chapter 3

FFT Based Integer Multiplier

The Schönhage-Strassen FFT multiplication algorithm was initially chosen as the target for HLS design of the flexible FFT multiplier. This algorithm was chosen because it performs integer multiplication with a computational complexity of $O(n \log n \log \log n)$ allowing for flexibility in hardware for high-precision integers. This algorithm is implemented within software libraries such as GMP and FLINT for integers of very large size. The algorithm is also naturally parallelizable lending itself to be implemented efficiently in hardware. Through the application of the Kronecker-Schönhage technique presented in Section 2.6, a large scale integer multiplier is also capable of performing polynomial multiplications for polynomials of varying degree and coefficient sizes. To maintain flexibility, this method would require that all modular reductions be performed in software after the result has been received.

3.1 Initial Implementation

Initial design of the high-precision multiplier began with selection of the target language as Vivado HLS supports synthesis of both C and C++ designs. Though both languages natively support integers up to only 64 bits in size, Vivado HLS provides libraries with arbitrary precision data types allowing for data types up to 1024 bits to be utilized in C designs and data types up to 32,768 bits to be utilized in C++ designs [12]. Because of the larger maximum data type size, C++ was chosen as

the target language for the design. Though a maximum size of 32,768 bits is smaller than the typical use case of the Schönhage-Strassen algorithm, the selection of various sizes up to this maximum should provide a semi-accurate model by which to estimate the scalability of the design. The initial implementation of the Schönhage-Strassen algorithm was directly modeled after Algorithm 4 and written in Python. The design targeted standard multiplication with operands of size $2^{n/2}$ and an output size of 2^n allowing for removal of the correction step on lines 13 and 14. Note that this effectively removed the modular reduction by $2^n + 1$ on the output of the algorithm as all results were bounded by 2^n. This design was tested for integers up to 32,768 bits in size against the built-in multiplication operation to verify correct functionality.

Once completed, the design was ported to C++ to allow for synthesis through Vivado HLS, starting with a base operand size of 128 bits. A simple testbench was created to ensure the design functioned correctly. This was achieved though the generation of 100 test vectors and verification of the experimental result against a calculated value using the standard multiplication operator. The target device for this work is the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (xczu9eg-ffvb1156-2-i-es2). The available resources for the target device are listed in Table 3.1. This basic

Table 3.1: Available resources for part xczu9eg-ffvb1156-2-i-es2

BRAM_18K	DSP48E	FF	LUT
1,824	2,520	548,160	274,080

design was run through Vivado HLS 2018.3 on the default settings targeting the same device with a clock period of 10 ns resulting in the output shown in Table 3.2.

These results are far from desired with 20% of the available device LUTs accounted for with only a 128-bit multiplier. These base synthesis results show that the design requires optimizations to improve device utilization through the application of synthesis directives and improved coding style techniques. Because hardware designs

Table 3.2: Synthesis results of the base integer multiplier

	BRAM_18K	DSP48E	FF	LUT
Required	38	361	60,499	56,540
Available	1,824	2,520	548,160	274,080
Utilization (%)	2	14	11	20

have an inherent trade-off between speed and area, initial optimizations of the design targeted area reduction by improving the basic structure of the algorithm to allow for further flexibility in decreasing the latency of the final design.

3.2 Arbitrary Precision Data Types

Analysis of the design for possible areas of improvement revealed that all loop counter variables were implemented as 32-bit integers. Additionally, 256-bit data types were allocated for the operands despite requiring only 128 bits, and when decomposed, they were constructed as an array of 256-bit values regardless of the number of bits required by the design. The use of generic data types for these variables led to the synthesis and inclusion of signals with larger data widths than necessary within the design. For example, when multiplying 128-bit operands with the selection of $k = 4$, the input operands are both decomposed into arrays containing 16 elements each with a maximum size of 2^{48}. In the current state of the design, each array element was represented with 256-bits resulting in a large waste of 207 bits for each element. Similarly, the loop counter variables iterating over the elements of the array were instantiated as standard 32-bit integers whereas only 4 bits were actually necessary to address the 16 elements of each array. Each variable in the design was therefore analyzed to determine the maximum number of bits required and declared with an arbitrary precision data type of the correct size.

The design at this stage was also constructed such that the main multiplication

function took as input the decomposition parameter k and the output operand size n. Though these values are required for the algorithm to function properly, allowing these inputs to be variable forced the HLS tools to construct hardware such that any value could be passed into the multiplication function. These parameters were therefore changed to be defined as constants in a header file to ensure that the design was synthesized strictly for a fixed output size n, a fixed input operand size of $n/2$, and a fixed decomposition parameter k. Selection of these parameters heavily impact the size and speed of the design and will be discussed further in Section 3.4. The design was then synthesized using the same settings as the base implementation producing the results shown in Table 3.3.

Table 3.3: Synthesis results of the integer multiplier with arbitrary precision data types

	BRAM_18K	DSP48E	FF	LUT
Required	15	36	18,636	19,803
Available	1,824	2,520	548,160	274,080
Utilization (%)	∼0	1	3	7

These results show a significant improvement over the base implementation with the resource utilization reduced by more than half. Despite this decrease in area, the design continued to exhibit major areas of potential optimization.

3.3 Arithmetic Modulo $2^N + 1$

The heart of the Schönhage-Strassen algorithm requires arithmetic operations modulo $2^N + 1$. As the selected primitive roots of unity are always powers of 2, the required operations are addition, subtraction, multiplication by 2^k, and division by 2^k. Note that the smallest representation for all values modulo $2^N + 1$ requires $N + 1$ bits. Though reductions modulo $2^N + 1$ are relatively fast, arithmetic operations between values represented with $N + 1$ bits would require modulo reduction after each step.

By selecting an operating size of $2N$-bits, all operations can be performed through a series of shifts and additions with a single reduction modulo $2^N + 1$ applied only at the final stage to represent the value with $N + 1$ bits [2]. Further optimizations can then be applied through application of the following identities:

$$\begin{cases} -1 \equiv (2^N)^k \bmod (2^N+1), & \text{k is odd} \\ 1 \equiv (2^N)^k \bmod (2^N+1), & \text{k is even} \end{cases}$$

3.3.1 Addition

Let a, b, and r be integers modulo $2^N + 1$ represented with $2N$ bits. Then the computation of $r = a + b$ may result in a carry at the most significant bit with value $(2^N)^2$. Because it is known that $1 \equiv (2^N)^2 \mod (2^N + 1)$, if a carry occurs, it is sufficient to simply add 1 to the result. This method of performing addition modulo $2^N + 1$ is presented in Algorithm 7.

Algorithm 7 Addition Modulo $2^N + 1$

Input: Integers $a, b \bmod (2^N + 1)$ represented with $2N$ bits
Output: $c = a + b \bmod (2^N + 1)$ represented with $2N$ bits
1: $(c, r) \leftarrow a + b$ ▷ c is the carry bit, r is the $2N$-bit result
2: **if** c **then**
3: **return** r
4: **else**
5: **return** $r + 1$

3.3.2 Multiplication by 2^k

Let a and r be integers modulo $2^N + 1$ represented with $2N$ bits with desired computation $r = a \cdot 2^k$. Multiplication by 2^k is typically performed by a left shift operation in hardware. This is also the case here, however, special care must be taken to ensure that the resulting value is correct modulo $2^N + 1$. The simple case of multiplication by 2 results in a single left shift. If the most significant bit of a is '1', then the shift

results in the value $(2^N)^2$ requiring that 1 is added to the result. If the most significant bit of a is '0', then the shift does not result in this overflow. Both of these cases can be represented through a single circular left shift. Therefore, all multiplications by 2^k can be computed through a circular left shift of k bits.

3.3.3 Subtraction

Let a, b, and r be integers modulo $2^N + 1$ represented with $2N$ bits with desired computation $r = a - b$. Because it is known that $-1 \equiv (2^N) \mod (2^N + 1)$, this operation can be represented as $r = a + b \cdot 2^N$. This allows for subtraction to be performed between any two values by utilizing a constant circular left shift of N bits and addition as in Algorithm 7.

3.3.4 Division by 2^k

Let a and r be integers modulo $2^N + 1$ represented with $2N$ bits with desired computation $r = a/2^k$. Division by 2^k is typically performed by a right shift operation in hardware, however, division modulo $2^N + 1$ requires division to be implemented through multiplication by the multiplicative inverse. Because $1 \equiv (2^N)^2 \mod (2^N + 1)$, the multiplicative inverse x of 2^k can be calculated as follows:

$$
\begin{aligned}
x \cdot 2^k \bmod (2^N + 1) &= 1 \bmod (2^N + 1) \\
\implies x \cdot 2^k \bmod (2^N + 1) &= 2^{2N} \bmod (2^N + 1) \\
\implies x \cdot 2^k \cdot 2^{-k} \bmod (2^N + 1) &= 2^{2N} \cdot 2^{-k} \bmod (2^N + 1) \\
\implies x \bmod (2^N + 1) &= 2^{2N-k} \bmod (2^N + 1)
\end{aligned}
$$

The resulting operation thus becomes $r = a \cdot 2^{2N-k}$. Therefore, division by 2^k can be performed by applying a circular left shift of $2N - k$ bits.

3.3.5 Modular Reduction

This operation reduces integers modulo 2^N+1 represented with $2N$ bits down to their N-bit representation. Let $a < 2^N + 1$ be an integer represented with $2N$ bits. This integer can be represented in the form $a = b + c \cdot 2^N$ where b and c are each N bits. Because it is known that $-1 \equiv (2^N) \mod (2^N + 1)$, the value can be represented as $a = b - c$. The only step necessary to complete the reduction is then to add the value $2^N + 1$ to this result if it is negative. This method of reducing a value with $2N$ bits modulo $2^N + 1$ is presented in Algorithm 8.

Algorithm 8 Reduction Modulo $2^N + 1$

Input: Integer $a \mod (2^N + 1)$ represented with $2N$ bits
Output: Integer $a \mod (2^N + 1)$ represented with $N + 1$ bits
1: $(c, b) \leftarrow a$ ▷ c, b are the upper and lower N bits of a, respectively
2: $a \leftarrow b - c$
3: **if** $a < 0$ **then**
4:     **return** $a + 2^N + 1$
5: **else**
6:     **return** a

3.3.6 Results

The design was modified to perform the modular arithmetic operations as described above. The modular reduction step was left unchanged leaving the standard modulus operator to observe the direct impact of these optimizations. The design was then synthesized using the same settings as the base implementation producing the results shown in Table 3.4.

The improvements to the modular arithmetic operations led to a significant reduction in both FF and LUT utilization. There was, however, an increase in the number of BRAM_18K resources from 15 to 18 instances. This rise in memory usage is a direct result of the change from the working values represented with $N + 1$ bits to $2N$ bits as the elements are stored in arrays. Despite this, the trade-off is

Table 3.4: Synthesis results of the integer multiplier with efficient arithmetic modulo $2^N + 1$

	BRAM_18K	DSP48E	FF	LUT
Required	18	36	5,765	14,015
Available	1,824	2,520	548,160	274,080
Utilization (%)	∼0	1	1	5

acceptable because the overall BRAM_18K utilization remains less than 1%. The modular reduction algorithm described above was then implemented and the design was synthesized yielding the results shown in Table 3.5.

Table 3.5: Synthesis results of the integer multiplier with efficient modular reduction by $2^N + 1$

	BRAM_18K	DSP48E	FF	LUT
Required	18	9	4,016	12,160
Available	1,824	2,520	5,48160	274,080
Utilization (%)	∼0	∼0	∼0	4

Implementation of the efficient modular reduction technique resulted in a significant decrease in both FF and LUT usage as well as a 75% reduction in the number of DSP48E instances. The standard modulus operator results in the instantiation of parameterized Xilinx LogiCORE divider cores in the design resulting in a large allocation of DSP resources [12]. These DSP resources will be critical in performing the large scale multiplication operations necessary to achieve high-precision multiplication and will likely be the resource bottleneck of the design. Therefore, the resource savings obtained by the implementation of efficient modular reduction through the increased data width of $2N$ bits further outweighs the increase in BRAM_18K instances. Final steps were taken to refactor the design and perform the optimizations described above in areas that were overlooked producing the synthesis results shown in Table 3.6.

Table 3.6: Final synthesis results of the integer multiplier

	BRAM_18K	DSP48E	FF	LUT
Required	18	9	1,972	10,869
Available	1,824	2,520	5,48160	274,080
Utilization (%)	∼0	∼0	∼0	3

3.4 Parameter Selection

To perform integer multiplication through application of the FFT, the decomposed modulus $2^N + 1$ must be chosen such that $N \geq \frac{2n}{K} + k$ to prevent the loss of precision in the result. Furthermore, N must be a multiple of 2^k to ensure that the weight parameter $\theta = 2^{\frac{N}{K}}$ and that ω is a primitive n-th root of unity such that $\omega = \theta^2$ allowing for all multiplications to be by a power of 2. Therefore, the minimum size for the working modulus is achieved when $N = \frac{2n}{K} + k$ with the efficiency of the design for a selected k defined as

$$\frac{2n/K + k}{N} \text{ [44].}$$

The efficiency of the algorithm for various operand sizes and selected values of k were calculated as shown in Table 3.7. These calculations show that typically a lower value of k results in higher efficiency. However, when selecting a value of k for the design, there is an expected trade-off between latency and the number of resources required as a lower k is expected to result in lower latency but increased resource utilization whereas a higher k is expected to result in increased latency but reduced resource utilization. This is because the value of k specifies the amount by which the integer operands are decomposed. Decomposition of the operands into more pieces results in smaller individual operations, but the number of loop iterations and the resulting latency are increased because each decomposed portion must be read from

Table 3.7: Efficiency of the parameter k for various operand sizes

n	k	N	Efficiency
2,048	2	1,028	0.998
2,048	3	520	0.990
2,048	4	272	0.956
2,048	5	160	0.831
2,048	6	128	0.547
4,096	3	1,032	0.995
4,096	4	528	0.977
4,096	5	288	0.906
4,096	6	192	0.698
8,192	4	1,040	0.989
8,192	5	544	0.950
8,192	6	320	0.819
8,192	7	256	0.527
16,384	5	1,056	0.974
16,384	6	576	0.899
16,384	7	384	0.685
32,768	6	1,088	0.947
32,768	7	640	0.811
32,768	8	512	0.516

BRAM and operated on. In contrast, a smaller value of k results in fewer decomposed portions of the original operands requiring fewer loop iterations but more resources to operate on each part.

3.5 Results

The integer multiplier was synthesized with Vivado HLS 2018.3 on the default settings targeting the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (xczu9eg-ffvb1156-2-i-es2) with a clock period of 5 ns resulting in the output shown in Table 3.8. Multiple configurations were generated for operands ranging from 1 kB to 16 kB and the values of k presented in Table 3.7. As expected, the smaller values of k resulted in an increase in the number of required DSP resources with the smallest configurations

Table 3.8: Synthesis results of the final integer multiplier, values with * indicate that they do not fit on the target device

n	k	N	BRAM_18K	DSP48E	FF	LUT
2,048	2	1,028	0	3,601*	150,760	85,707
2,048	3	520	145	961	36,502	53,896
2,048	4	272	78	256	18,561	52,616
2,048	5	160	45	82	11,906	33,963
2,048	6	128	38	50	10,007	25,868
4,096	3	1,032	288	3,721*	60,497	68,939
4,096	4	528	148	961	40,972	54,262
4,096	5	288	80	289	23,443	53,707
4,096	6	192	55	122	17,190	39,652
8,192	4	1,040	290	3,722*	69,039	69,193
8,192	5	544	153	1,024	49,838	54,873
8,192	6	320	90	361	36,981	56,026
8,192	7	256	73	225	29,677	51,086
16,384	5	1,056	295	3,845*	86,048	69,586
16,384	6	576	160	1,156	67,503	56,044
16,384	7	384	108	529	59,384	60,540
32,768	6	1,088	303	4,096*	120,003	70,307
32,768	7	640	180	1,444	105,329	58,740
32,768	8	512	143	901	97,273	53,278

for each value of n exceeding the available DSP resources. To benchmark the speed of the design over a proven software solution, the performance of the synthesized integer multiplier was compared against the polynomial multiplication function within the NTL software library as it is used within HELib, a software library for FHE [45]. This was achieved by generating and performing 100 random multiplications for each operand size through the NTL software library on a 4-core/4-thread 3.7 GHz AMD A10-7850k CPU with 16 GB of RAM and measuring the average computation time. From these results, the speedup of the design over the integer multiplication function within NTL was calculated as shown in Table 3.9. The synthesized configurations unfortunately resulted in speedups less than 1.000 for all values of k as the hardware design required more time to perform the various multiplications than

Table 3.9: Timing results of the final integer multiplier

n	k	Latency	Time (µs)	NTL (µs)	Speedup
2,048	2	244	1.220	1.157	0.948
2,048	3	665	3.325	1.157	0.348
2,048	4	1,706	8.530	1.157	0.136
2,048	5	4,399	21.995	1.157	0.053
2,048	6	10,912	54.560	1.157	0.021
4,096	3	687	3.435	2.156	0.628
4,096	4	1,720	8.600	2.156	0.251
4,096	5	4,701	23.505	2.156	0.092
4,096	6	13,506	67.530	2.156	0.032
8,192	4	1,766	8.830	5.274	0.597
8,192	5	4,715	23.575	5.274	0.224
8,192	6	14,208	71.040	5.274	0.074
8,192	7	40,356	201.780	5.274	0.026
16,384	5	4,809	24.045	1.575	0.066
16,384	6	14,014	70.070	1.575	0.023
16,384	7	45,891	229.455	1.575	0.007
32,768	6	14,204	71.020	40.556	0.571
32,768	7	45,505	227.525	40.556	0.178
32,768	8	146,984	734.920	40.556	0.055

the NTL software library. This is not a fair comparison, however, as the NTL software library does not utilize the Schönhage-Strassen multiplication algorithm for the small operand sizes tested here. Though it is possible that positive speedup would be achieved for larger operand sizes, the arbitrary precision integer library provided through Vivado HLS limits the maximum size of a variable to 32,768 bits making it impossible to synthesize larger designs. For these reasons, the integer multiplier was not improved further and the development of an FFT based polynomial multiplier was explored.

Chapter 4

FFT Based Polynomial Multiplier

The FFT based polynomial multiplication algorithm was chosen as the next design target for HLS. The algorithm was chosen because application of the FFT on the input polynomials reduces convolution to a point-wise multiplication of the transformed elements reducing the computational complexity of polynomial multiplication to $O(n \log n)$. Many steps of the algorithm are also naturally parallelizable accommodating efficient hardware implementation. FFT polynomial multiplication is implemented within software libraries such as NTL and FLINT for polynomials of very large degree. The design includes the pre- and post-processing steps described in Section 2.10 to perform negative-wrapped convolution. The addition of these steps allow for in place modular reduction by a polynomial of the form $x^n + 1$. This is greatly beneficial as it enables the multiplier to be used directly in RLWE base HE schemes. Modular reduction of the coefficients is not inherent to the algorithm and requires the design and inclusion of a generic reduction technique.

4.1 Initial Implementation

Implementation of the FFT based polynomial multiplication algorithm borrowed heavily from the previously implemented integer multiplier discussed in Chapter 3 as the Schönhage-Strassen algorithm contains polynomial multiplication at its core. Like the integer multiplier, the algorithm was implemented in C++ and utilized the

arbitrary precision types available through Vivado HLS. Because the coefficient sizes required by secure RLWE schemes as outlined in Table 2.1 are far below the 32,768 bit limit of the arbitrary precision type library, the design of the polynomial multiplier is not limited in its scope like the integer multiplier. The basic structure of the polynomial multiplier was based on Algorithm 3. The basic block diagram of the targeted design for the polynomial multiplier is displayed in Figure 4.1. This image shows the

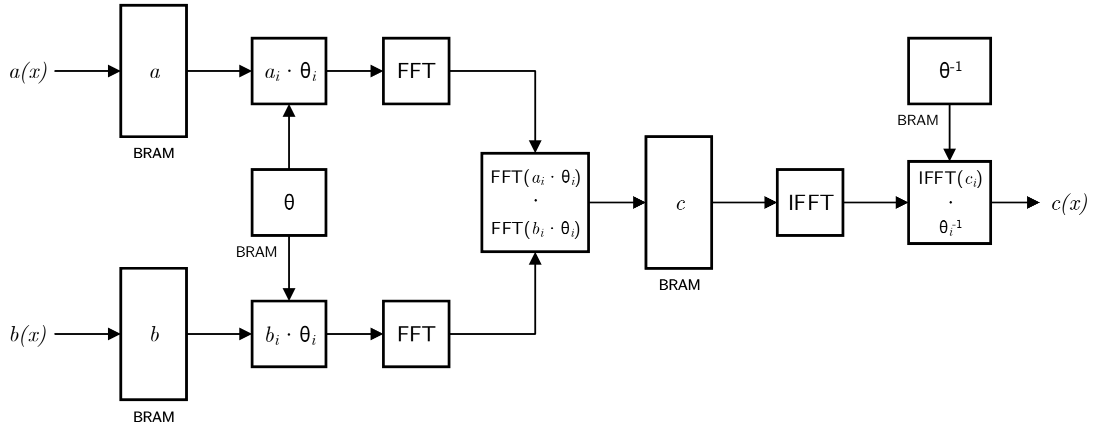

Figure 4.1: Basic block diagram of the FFT based polynomial multiplier

storage of the input polynomials a and b into Block RAMs (BRAMs) and the operations necessary to produce the resulting product polynomial c. Unlike the integer multiplier, the primitive n-th root of unity ω and the weighting parameter θ were not guaranteed to be powers of 2. The coefficients are also reduced by some prime p that is not guaranteed to be of the form $2^n + 1$ making the efficient arithmetic operations modulo $2^n + 1$ discussed in Section 3.3 unable to be used. Therefore, modular reduction of the coefficients was modeled after Barrett's reduction algorithm outlined in Algorithm 5 through the creation of a reduce function. Furthermore, because the algorithm requires consecutive powers of ω and θ, and their inverses ω^{-1} and θ^{-1}, by which to multiply the coefficients, Lookup Tables (LUTs) were generated with all pre-calculated powers of these values to prevent the expensive task of computing

them in real time. These LUTs provided further benefits as well as the backward FFT typically requires an additional step in which each of the vector elements is multiplied by n^{-1}. This would require an additional loop iterating through each element and performing this computation. An optimization can be performed to remove this step, however, by instead pre-calculating the multiplicative inverse of n modulo p and multiply each power of θ^{-1} by this value. The reduction of each resulting value modulo p can then instead be stored in an array and used in the unweighting step to perform both operations at once and retrieve the final result. The initial design of the FFT polynomial multiplier is shown in Algorithm 9.

Algorithm 9 Initial Implementation of the FFT Polynomial Multiplier

Input: Polynomials $a(x)$ and $b(x)$ of maximum degree n with coefficients $a_i, b_i \in \mathbb{Z}_p$ for $i = 0, 1, \ldots, n-1$
Output: $c(x) = a(x) \cdot b(x) \mod (x^n + 1)$

1: Pre-calculate arrays for consecutive powers of θ, ω, θ^{-1}, and ω^{-1}
2: weight_coeff : **for** $i \leftarrow 0$ **to** n **do**
3: $\quad a_i \leftarrow \text{REDUCE}(a_i \cdot \theta^i,\ p)$
4: $\quad b_i \leftarrow \text{REDUCE}(b_i \cdot \theta^i,\ p)$
5: $a \leftarrow \text{FFT}(a,\ \omega)$
6: $b \leftarrow \text{FFT}(b,\ \omega)$
7: mult_coeff : **for** $i \leftarrow 0$ **to** n **do**
8: $\quad c_i \leftarrow \text{REDUCE}(a_i \cdot b_i,\ p)$
9: $c \leftarrow \text{IFFT}(c,\ \omega^{-1})$
10: unweight_coeff : **for** $i \leftarrow 0$ **to** n **do**
11: $\quad c_i \leftarrow \text{REDUCE}(c_i \cdot (\theta^{-1} \cdot n^{-1} \mod p)_i,\ p)$

A simple testbench was created to test the functionality of the base design supporting a small polynomial with $n = 512$ and $\log p = 17$. These basic design parameters were used throughout the optimization process to benchmark the progress of the current design in both area and latency. The testbench generated 100 random test vectors through the NTL software library and verified the produced result against a calculated value. This basic, unoptimized design was run through Vivado HLS 2018.3 on the default settings targeting the Zynq UltraScale+ MPSoC ZCU102 Evaluation

Kit (xczu9eg-ffvb1156-2-i-es2) with a clock period of 4 ns resulting in the output shown in Table 4.1.

Table 4.1: Synthesis results of the initial polynomial multiplier

	BRAM_18K	DSP48E	FF	LUT
Total	5	7	1,560	3,750
Available	1,824	2,520	548,160	274,080
Utilization (%)	∼0	∼0	∼0	1

These results show that the initial design does not utilize more than 1% of the resources on the target device. This suggests that the area of the design has already been sufficiently minimized based on the techniques borrowed from the design of the integer multiplier. Because of this, attention was turned to reducing the overall latency of the design, and the initial timing results are shown in Table 4.2.

Table 4.2: Timing results of the initial polynomial multiplier

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
weight_coeff	2,560	5	2,560	512
FFT	N/A	-	N/A	-
mult_coeff	2,560	5	2,560	512
IFFT	N/A	-	N/A	-
unweight_coeff	2,560	5	2,560	512
Total	37,397	-	37,398	-

This table shows the latency, iteration latency, Initiation Interval (II), and trip count for each of the major sections within the design. The latency of a section is the number of clock cycles required for the input data to be fully operated on within that section whereas the iteration latency is the number of clock cycles required for each iteration. The initiation interval is the number of clock cycles required before new data can be operated on within the section, and the trip count is the number of iterations required for the section. Note that both the FFT and IFFT sections have

unknown latency. This is because it was not possible for the tool to report the exact latency of the design as the inner loops of these sections did not have fixed bounds at compile time. The table also shows that each of the three major loops in the design have a latency of 2,560 clock cycles, an iteration latency of 5 clock cycles, and a trip count of 512. The HLS PIPELINE directive was applied to each of the three main loops in the design to exemplify the impact that the inclusion of HLS directives has on the synthesis of the design producing the timing results shown in Table 4.3.

Table 4.3: Timing results of the polynomial multiplier with main loops pipelined

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
weight_coeff	515	5	1	512
FFT	N/A	-	N/A	-
mult_coeff	515	5	1	512
IFFT	N/A	-	N/A	-
unweight_coeff	515	5	1	512
Total	31,265	-	31,266	-

The addition of these directives reduce the overall latency from 37,397 clock cycles to 31,265. An iteration interval of 1 was achieved by each loop resulting in an overall latency of 515 clock cycles, just above the trip count of 512. These loops were pipelined without issue because the arrays accessed in each of the loops are composed of dual port BRAMs allowing for two concurrent memory accesses from each. This allows for the scheduling of both a read and a write operation of the same memory during each loop iteration. The further addition of HLS directives to the design requires careful consideration and analysis of the scheduling and inclusion of hardware primitives. For example, unrolling the *weight_coeff* loop by any factor to parallelize the operations within it would not be possible because this would require more than two concurrent memory accesses. In contrast, the *mult_coeff* loop performs read and write operations between three different memories requiring the use of only a single port from each memory on each iteration and could potentially be unrolled by a factor of two.

4.2 Loop Structure of the FFT

The loop structures of both the backward and forward FFTs up until this point were based on Algorithm 1 and Algorithm 2, respectively. Though these implementations are effective in software because they ensure that the same power of ω is used in each inner loop iteration to prevent its calculation at each step, the triple loop structure utilized in each is nonoptimal. This is because the entrance and exit of each loop requires an extra cycle, increasing latency over time especially for a large number of coefficients. Furthermore, because the consecutive powers of ω are stored in a array and read from memory when needed, it is not necessary to ensure that every computation requiring the same power of ω is performed together. In order to improve the loop structure of the implemented FFT algorithms, it was thus necessary to investigate the overall construction of the FFT. Focus began with the design of the forward FFT implemented in the Gentleman-Sande DIF form. The basic structure of the FFT is composed of $\log n$ stages with each stage requiring the computations over the n points contained within the vector, or in this case, the polynomial coefficients. Note that the combination of these stages attains the computational complexity of $O(n \log n)$ for which the FFT is known. As an example, a diagram of the basic structure of an 8-point FFT of the DIF form is shown in Figure 4.2.

The fundamental building block of an n-point FFT is the 2-point FFT in which a simple computation is performed between two elements of the input vector. This 2-point FFT is commonly referred to as a "butterfly" operation. The typical butterfly for a decimation-in-frequency FFT is shown in Figure 4.3.

Note that n calculations are performed at each of the $log(n)$ stages between elements within the array from decreasing distances apart, with each stage decomposed into twice as many smaller groups. If the size of the FFT were increased to be a 16-point computation, then after the initial stage the design would be split into two

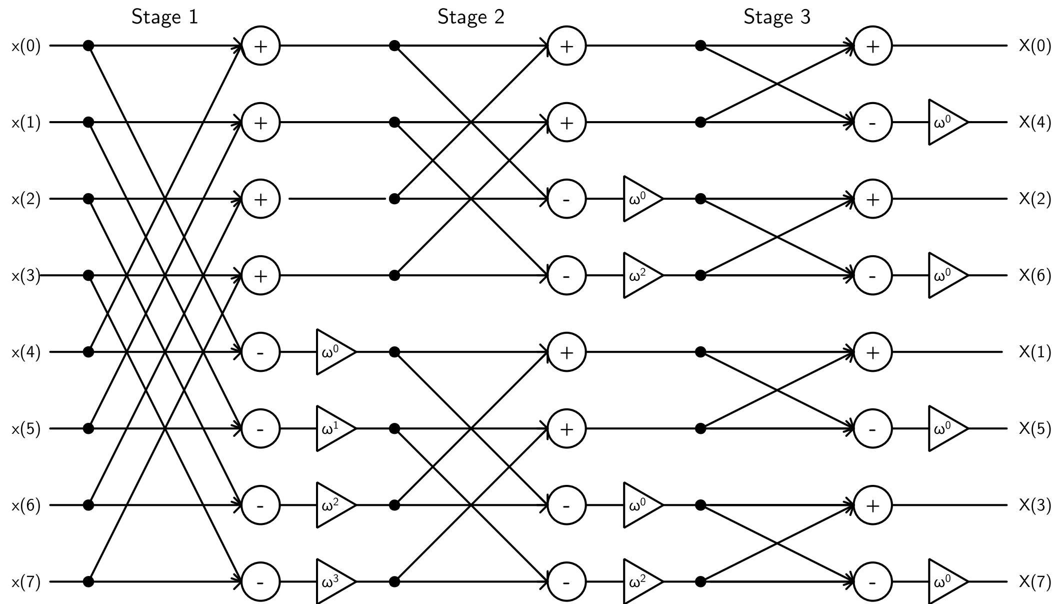

Figure 4.2: Structure of an 8-point decimation-in-frequency FFT

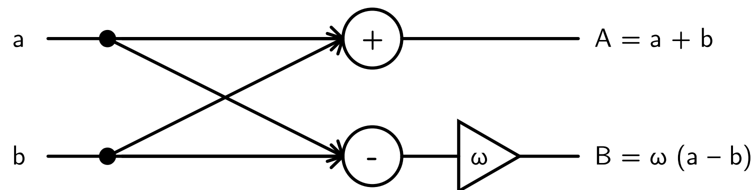

Figure 4.3: Butterfly computation of a decimation-in-frequency FFT

8-point FFTs. This trend continues as the size of the n-point FFT increases. Therefore, the improved design of the FFT was modeled after the basic structure of the FFT. This design contained only two nested loops with the outer loop iterating over each of the $\log n$ stages and the inner loop performing the butterfly computations between the n elements. Note that because the butterfly operation operates on two elements at once, the inner loop requires only an iteration depth of $n/2$. Pseudocode of the improved forward FFT in the Gentleman-Sande DIF form is described in Algorithm 10.

Algorithm 10 Improved Forward FFT Algorithm (DIF)

Input: vector x of length n, pre-computed powers of ω
Output: transformed vector x

1: $m = n/2$
2: $j = \log n - 1$
3: fft_stage : **for** $s \leftarrow 0$ **to** $\log n$ **do**
4:     fft_group : **for** $k \leftarrow 0$ **to** $n/2$ **do**
5:         $i \leftarrow k + \lfloor k/2^j \rfloor \cdot 2^j$
6:         $a \leftarrow k \cdot 2^s$
7:         $(x_i,\ x_{i+m}) \leftarrow (x_i + x_{i+m},\ \omega^a(x_i - x_{i+m}))$
8:     $m \leftarrow m/2$
9:     $j \leftarrow j - 1$

Similarly, the backward FFT was implemented in the Cooley-Tukey DIT form. This form of the FFT works in reverse from the DIF version with the first stage starting with $n/2$ 2-point FFTs and each of the next $\log n$ stages combines the results of the previous stage in groups of two. A diagram of the basic structure of an 8-point FFT of the DIF form is shown in Figure 4.4.

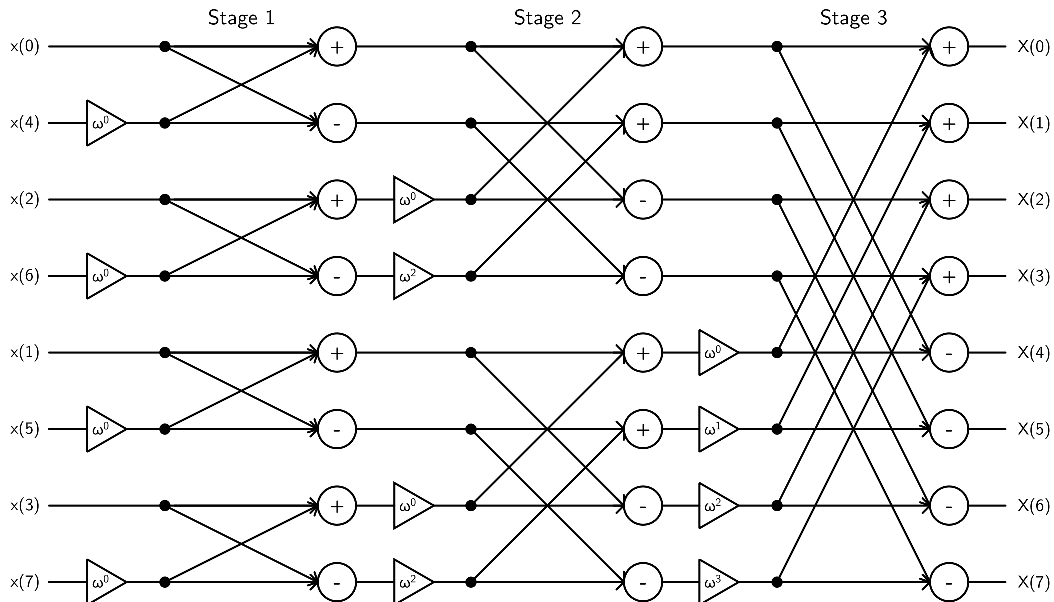

Figure 4.4: Structure of an 8-point decimation-in-time FFT

Note that the DIF form begins with the components in typical binary order and

ends with the elements in reverse binary order whereas the DIT form begins with the components in reverse index order and returns them to typical binary order. It is this property that makes these forms best suited for the forward and backward FFTs, respectively. The standard butterfly for a decimation-in-time FFT is shown in Figure 4.5.

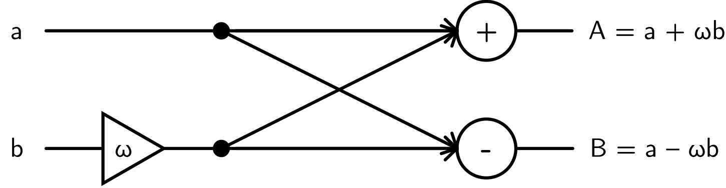

Figure 4.5: Butterfly computation of a decimation-in-time FFT

Like the forward FFT, the backward FFT was implemented using only two nested loops with the outer loop iterating over each of the $\log n$ stages and the inner loop performing the butterfly computations between the n elements. Pseudocode of the improved backward FFT of the DIT form is described in Algorithm 11.

Algorithm 11 Improved Backward FFT Algorithm (DIT)

Input: vector x of length n, pre-computed powers of ω^{-1}
Output: transformed vector x

1: $m = 1$
2: $j = \log n - 1$
3: ifft_stage : **for** $s \leftarrow 0$ **to** $\log n$ **do**
4:     ifft_group : **for** $k \leftarrow 0$ **to** $n/2$ **do**
5:         $i \leftarrow k + \lfloor k/2^j \rfloor \cdot 2^j$
6:         $a \leftarrow k \cdot 2^j$
7:         $(x_i,\ x_{i+m}) \leftarrow (x_i + \omega_a^{-1} x_{i+m},\ x_i - \omega_a^{-1} x_{i+m})$
8:     $m \leftarrow 2m$
9:     $j \leftarrow j - 1$

With the FFT and IFFT of the polynomial multiplier design replaced with these modified versions, it became possible for the HLS tools to determine their latency. The design was then synthesized producing the results shown in Table 4.4.

Table 4.4: Timing results of the polynomial multiplier with improved FFT loop structure

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
weight_coeff	515	5	1	512
FFT	11,539	-	11,539	-
mult_coeff	515	5	1	512
IFFT	16,147	-	16,147	-
unweight_coeff	515	5	1	512
Total	29,240	-	29,240	-

These results show that the FFT and IFFT operations within the design comprise the majority of the latency with each requiring 11539 and 16147 clock cycles, respectively. Optimization of these sections was critical in reducing the overall latency of the design.

4.3 Pipelining

To further optimize the design, it was necessary to reduce the latency of both the FFT and IFFT functions. This was done through analysis and modification of the FFT algorithm with successful improvements carried over to the IFFT algorithm due to their nearly identical structure. The timing results of the internal loops of the FFTs are shown in Table 4.5.

Table 4.5: Timing results of the improved FFT

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
FFT	16,147	-	16,147	-
fft_stage	11,538	1,282	-	9
fft_group	1,280	5	-	256
IFFT	16,147	-	16,147	-
ifft_stage	16,146	1,794	-	9
ifft_group	1,792	7	-	256

These results show that the *fft_group* loop would benefit greatly from pipelining

as each iteration currently has a latency of 5 cycles with the corresponding *ifft_group* loop requiring 7 cycles per loop iteration. Direct application of the HLS PIPELINE directive to each of these loops with a target II of 1 resulted in synthesis errors because the tool was unable to ensure that there were no data dependencies between loop iterations. It is known that the inner loop of the FFT does not contain data dependencies between loop iterations because it performs all computations in place, iterating through each element within the array. Therefore, an HLS DEPENDENCE directive was added to each of these loops to inform the tool that they could be pipelined without the risk of compromising the data dependencies between loop iterations. Because the *fft_group* loop both reads and writes to the same array, the scheduling of store operations became an issue when attempting to reach an II of 1 and a minimum interval of 2 was the best that could be achieved. Unfortunately, though the inclusion of the HLS DEPENDENCE directive permitted the tool to continue pipelining the design, the C/RTL co-simulation of the synthesized design returned dependence errors. This was caused by the fact that when the HLS PIPELINE directive is applied to a nested loop, it automatically flattens the outer loops by default. This became an issue because, despite the fact that no data dependencies existed between groups, the entire *ifft_group* must be executed and the pipeline flushed before the next iteration of the *fft_stage* loop due to dependencies between each stage. To combat this, applying an HLS LOOP_FLATTEN directive to the *fft_stage* with the 'off' option was attempted to prevent the tool from flattening the outer loop. Though this directive performed correctly and the outer loop was not flattened, co-simulation of the design continued to report dependency errors as the inner loop pipeline must not be guaranteed to flush. The next attempt to combat the data dependency issue was the application of an HLS UNROLL directive to the *fft_stage* loop. The idea was that the inclusion of this option would ensure that each stage would perform the *fft_group* loop separately with their own pipelined logic. Unfortunately, this option also did not solve the

problem with the loop dependency error still appearing during co-simulation of the design. The application of HLS directives did not appear to be a sufficient solution to this problem and the design was analyzed to determine areas of possible architectural improvement.

A new design was conceived utilizing a "ping-pong" buffer structure to prevent data dependencies between loop iterations and to allow for full pipelining of the *fft_group* loop with an II of 1. This was achieved through the addition of two memories to the design with each loop iteration reading from one memory and writing to the other. The memories would then swap in functionality each stage, ensuring that all read and write operations could be scheduled without contention. The basic hardware layout of the FFT with the "ping-pong" buffer is shown in Figure 4.6.

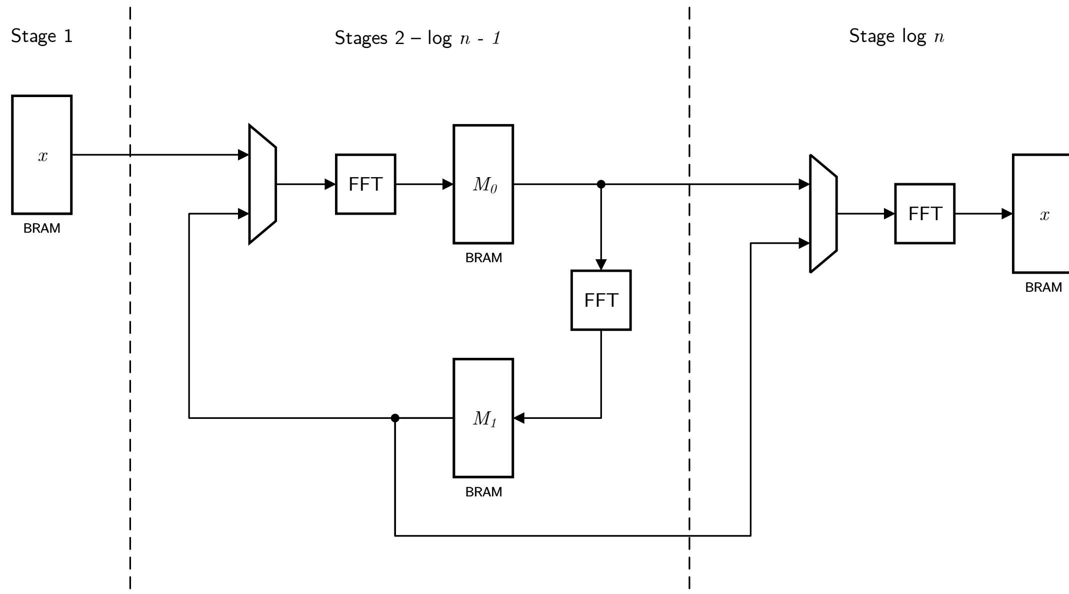

Figure 4.6: Basic block diagram of the FFT with "ping-pong" memory buffer

This block diagram shows the first iteration operating over the data in the input memory and writing the result into the first 'ping' memory. The next stage then reads from the 'ping' memory and writes the result to the 'pong' memory. The following stage then reads the previous result from the 'pong' memory and stores the results in

the 'ping' memory. This pattern continues until the last stage of the FFT at which point the last memory to have been written is read from and the results stored back into the input memory for further operation. Though the array containing the initial data could be used as one of the two required memories, an odd number of stages would require an entire memory copy between the second and initial memory to allow the next operations access to the result. Because of this, it was decided to use the input array only in the initial and final iterations of the FFT. The goal then became to design the software in such a way that the HLS tool would generate this desired hardware design. The first step was modification of the FFT functions to include the "ping-pong" memory structure. A 2D array was added to the design with dimension 1 of size 2 and dimension 2 of size n. This made differentiating between the 'ping' and 'pong' memories simple as a single bit could be used as the address to select the desired memory for access. By default, HLS does not implement these as two separate memories because they are declared under the same array structure. To ensure that each memory was implemented as its own BRAM, the HLS ARRAY_PARTITION directive was applied on the first dimension with the option 'complete'. The first iteration of the *fft_stage* was then manually unrolled with the data read from the input array and the result stored in the first 'ping' memory by setting the array index to '0'. The main *fft_stage* loop was largely unchanged with only the loop bounds reduced to allow for the unrolled stages and the address bit of the "ping-pong" array negated to swap the role of each memory. The main *fft_group* loop was then modified to read from and write to the selected memories based on this address bit. The final stage was also manually unrolled to read the data from the last written memory and write the result to the input array. Pseudocode for the FFT with the "ping-pong" memory structure is shown in Algorithm 12.

The HLS PIPELINE directive was then added to the *fft_stage_1*, *fft_group*, and *fft_stage_log_n* loops with a target II of 1. The IFFT was also modified through the

Algorithm 12 Forward FFT Algorithm with "Ping-Pong" Memory Structure

Input: vector x of length n, pre-computed powers of ω
Output: transformed vector x

1: $m = n/2$
2: $j = \log n - 1$
3: $addr = 0$
4: fft_stage_1 : **for** $k \leftarrow 0$ **to** $n/2$ **do**
5: $i \leftarrow k$
6: $a \leftarrow k$
7: $M_{addr,i} \leftarrow x_i + x_{i+m}$
8: $M_{addr,i+m} \leftarrow \omega^a(x_i - x_{i+m})$
9: $m \leftarrow m/2$
10: $j \leftarrow j - 1$
11: fft_stage : **for** $s \leftarrow 1$ **to** $\log n - 1$ **do**
12: fft_group : **for** $k \leftarrow 0$ **to** $n/2$ **do**
13: $i \leftarrow k + \lfloor k/2^j \rfloor \cdot 2^j$
14: $a \leftarrow k \cdot 2^s$
15: $M_{\neg addr,i} \leftarrow M_{addr,i} + M_{addr,i+m}$
16: $M_{\neg addr,i+m} \leftarrow \omega^a(M_{addr,i} - M_{addr,i+m})$
17: $m \leftarrow m/2$
18: $j \leftarrow j - 1$
19: $addr \leftarrow \neg addr$
20: fft_stage_log_n : **for** $k \leftarrow 0$ **to** $n/2$ **do**
21: $i \leftarrow k \cdot 2$
22: $x_i \leftarrow M_{addr,i} + M_{addr,i+m}$
23: $x_{i+m} \leftarrow M_{addr,i} - M_{addr,i+m}$

addition of the "'ping-pong" memory structure in the same manner as the FFT. Synthesis of the design failed because the HLS tool continued to detect data dependencies between loop iterations. An HLS UNROLL directive was applied to the *fft_stage* loop to ensure that the HLS tool would pipeline each trip of the *fft_group* loop separately. Because of the "ping-pong" memory structure, each *fft_group* contained explicit read and write operations between separate memories. Synthesis of the design was successful with all loops achieving the desired II of 1 clock cycle. The timing results of the pipelined design are shown in Table 4.6.

These results show significant reduction in latency over the non-pipelined design of the FFT. Note that the because the *fft_stage* loop is completely unrolled, the latency

Table 4.6: Timing results of the FFT with pipelined loops

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
FFT	2,363	-	2,363	-
fft_stage_1	261	7	1	256
fft_group	261	7	1	256
fft_stage_log_n	257	3	1	256
IFFT	2,371	-	2,371	-
ifft_stage_1	257	3	1	256
ifft_group	262	8	1	256
ifft_stage_log_n	262	8	1	256

of the *fft_group* loop is reported for only a single instance but is performed $\log n - 2$ times. The timing results for the polynomial multiplier with pipelined FFT and IFFT functions are shown in Table 4.7.

Table 4.7: Timing results of the polynomial multiplier with pipelined FFT

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
weight_coeff	516	6	1	512
FFT	2,363	-	2,363	-
mult_coeff	516	6	1	512
IFFT	2,371	-	2,371	-
unweight_coeff	516	6	1	512
Total	6,290	-	6,291	-

The overall latency of the design was significantly reduced to 6,290 clock cycles from the 29,240 clock cycles required by the base implementation through the addition of pipelined loops within the FFT and IFFT functions. Despite this, these functions still comprise over 75% of the total latency of the design.

4.4 Loop Unrolling

The unrolling of loops was explored to further increase the parallelism of the design. All of the inner loop operations within the design perform operations between elements within the array in place and could potentially be performed in parallel. There is a large inherent trade-off between loop unrolling and the area of the design. Though unrolling a loop divides the latency of that loop by the factor it is unrolled, it also multiplies the number of operations by the same factor. As a result, the number of hardware primitives required is also nearly multiplied by the loop unrolling factor. Because of this, it is not possible to simply unroll every loop in the design by the maximum amount possible and careful selection of both the loops to unroll and the factor by which to unroll them must be considered. Because the *fft_group* loop is a nested loop with $\log n - 2$ occurrences, unrolling this loop would result in the largest reduction in latency. The memories within this loop already contain two concurrent read and write operations per iteration making it impossible to unroll the loop and achieve further parallelization without increasing the number of available read and write ports. Though the loop would be unrolled and the trip count of the loop unrolled by the unrolling factor, the iteration latency would be multiplied by the same factor resulting in the same overall latency. Thus, partitioning of the memories was necessary to create more ports from which to access the memories concurrently. The HLS PARTITION directive allows for the partitioning of arrays in multiple ways. The cyclical option will partition the array into n segments with the data at each index cyclically allocated to the n arrays in numeric order. For example, cyclic partitioning by a factor of two would result in two arrays with one containing all data from the even indices and the other containing all data from the odd indices of the original array. Alternatively, block partitioning can be applied in which the array is directly divided into even segments. For example, an array containing 100 elements that is

block partitioned by a factor of two would result in 2 arrays with the first array containing the first 50 elements and the second array containing the last 50 elements. To accurately determine the best partitioning scheme necessary to unroll the *fft_group* loop, a map of the indices necessary at each stage of a 16-point FFT was analyzed as shown in Figure 4.7.

	Stage 1		Stage 2		Stage 3		Stage 4	
k	i	$i+m$	i	$i+m$	i	$i+m$	i	$i+m$
0	0	8	0	4	0	2	0	1
1	1	9	1	5	1	3	2	3
2	2	10	2	6	4	6	4	5
3	3	11	3	7	5	7	6	7
4	4	12	8	12	8	10	8	9
5	5	13	9	13	9	11	10	11
6	6	14	10	14	12	14	12	13
7	7	15	11	15	13	15	14	15
	$m=8$		$m=4$		$m=2$		$m=1$	

Figure 4.7: Index mapping of a 16-point FFT

It should be noted that in the case of the IFFT, the stage order is simply reversed while all index values remain the same. This image shows that the FFT alternates between operating on two even indices and operating on two odd indices. This is true for all stages except for the last in which the first index is always even and the second index is always odd. Analysis of the FFT structure shows that this will be the case for all n-point FFTs because each stage forms groups of consecutive powers of two with $m = 2^0 = 1$ resulting in the only group that operates on two points in an even/odd pair. These results suggest that cyclically partitioning the input arrays and unrolling the *fft_group* by a factor of two would be optimal if not for the deviation in indexing on the last stage. Fortunately, the stage resulting in $m = 1$ has already been manually unrolled from the main *fft_group* loop permitting the HLS UNROLL directive to be

applied with an unrolling factor of two along with the HLS PARTITION array with cyclic partitioning by a factor of two on the input arrays. Synthesis of the design with these directives applied did not result in the desired output as the HLS tool was unable to determine the static nature of the indexing between the odd and even partitions within the loop structure. This resulted in the tool scheduling two read and write operations for each partitioned array and a multiplexer to dynamically select which partition was necessary for the current iteration. This effectively nullified the unrolling of the loop as no parallelization was achieved due to the incorrect scheduling of the read and write operations. To remedy this, the input arrays were manually partitioned into two arrays of half the size containing the even and odd data. To improve the layout of the code, the butterfly operation was placed in a function. The butterfly function was designed with a C++ function template to ensure that each iteration of the *fft_stage* loop statically selected which of the "ping-pong" memories from which to read and write. This was necessary because the HLS tools cannot resolve dependencies between function calls within a loop that operate on the same array and will always schedule them sequentially. Use of the templated function ensures that only a single version of the templated function will be called in each loop iteration with static memory addressing allowing for parallel execution. The *fft_group* loop was then manually unrolled by a factor of two as shown in Algorithm 13 with the unrolled initial and final stages omitted for clarity. The design was then synthesized producing the timing results shown in Table 4.8. These results show that unrolling the *fft_group* loop resulted in the expected latency reduction of both the FFT and IFFT functions by nearly half. Because the input arrays of the design were partitioned, the *weight_coeff*, *mult_coeff*, and *unweight_coeff* were also unrolled through application of the HLS UNROLL directive with an unrolling factor of two producing the timing results shown in Table 4.9.

Though a large reduction in latency is achieved, these reductions in latency result

Algorithm 13 Forward FFT Algorithm with Partially Unrolled *fft_group* Loop

Input: vector x of length n, pre-computed powers of ω
Output: transformed vector x

1: … *stage 1 is omitted* …
2: $m \leftarrow m/4$
3: $j \leftarrow j - 2$
4: fft_stage : **for** $s \leftarrow 1$ **to** $\log n - 1$ **do**
5: fft_group : **for** $k \leftarrow 0$ **to** $n/2$ **by** 2 **do**
6: $base \leftarrow k + \lfloor k/2^j \rfloor \cdot 2^j$
7: $idx0 \leftarrow base/2$
8: $idx1 \leftarrow (base + m)/2$
9: $a \leftarrow k \cdot 2^s$
10: $b \leftarrow (k+1) \cdot 2^s$
11: **if** s_0 **then**
12: $addr = 0$
13: **else**
14: $addr = 1$
15: FFT_BUTTERFLY$(M, addr, idx0, idx1, a, b)$
16: $m \leftarrow m/2$
17: $j \leftarrow j - 1$
18: $addr \leftarrow \neg addr$
19: … *stage* $\log n$ *is omitted* …

in increased resource utilization. This trade-off between latency and resource utilization required further analysis of the synthesized design for various polynomial degrees and coefficient sizes.

Table 4.8: Timing results of the FFT with partially unrolled *fft_group* loop

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
FFT	1,202	-	1,202	-
fft_stage_1	131	5	1	128
fft_group	132	6	1	128
fft_stage_log_n	129	3	1	128
IFFT	1,210	-	1,210	-
ifft_stage_1	129	3	1	128
ifft_group	133	7	1	128
ifft_stage_log_n	132	6	1	128

Table 4.9: Timing results of the polynomial multiplier with partially unrolled FFT loops

Section	Latency	Iteration Latency	Initiation Interval	Trip Count
weight_coeff	132	6	1	128
FFT	1,202	-	1,202	-
mult_coeff	132	6	1	128
IFFT	1,210	-	1,210	-
unweight_coeff	132	6	1	128
Total	2,816	-	2,817	-

4.5 Design Parameters

A python script was created to generate various design parameters based on the security recommendations in Table 2.1. This table presents the maximum size of the coefficients for polynomials of varying degree achieving 128, 192, and 256-bit security levels. The script generated primes of the maximum coefficient size for each of the configurations along with the primitive n-th root of unity ω and the weighting factor θ associated with each. The memories containing the exponentiation of these values were also generated for each of these configurations with the script automatically creating a header file containing the initialization of constant arrays to be included in the synthesized design. The script allows for the design to be flexible for polynomials

of any degree and coefficient size requiring only the specification of n and $\log p$ to obtain the necessary constants to synthesize a specific design configuration.

4.6 Initial Results

The basic and modified polynomial multipliers were synthesized with Vivado HLS 2018.3 using the default settings and targeted the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (xczu9eg-ffvb1156-2-i-es2) with a clock period of 4 ns for each of the configurations outlined in Table 2.1. The synthesis results of the base multiplier are shown in Table 4.10. The resource utilization is best put in perspective of the target

Table 4.10: Synthesis results of the base polynomial multiplier for each security configuration

n	$\log p$	BRAM_18K	DSP48E	FF	LUT
1,024	19	18	42	3,705	4,286
1,024	22	18	49	3,872	4,265
1,024	31	18	112	5,454	5,277
2,048	33	28	112	5,681	5,605
2,048	42	37	133	9,133	6,938
2,048	58	51	266	10,569	8,000
4,096	62	86	399	12,484	9,281
4,096	80	110	623	13,895	10,402
4,096	113	158	1,218	15,974	12,108
8,192	123	303	1,281	18,494	12,499
8,192	157	386	2,086	21,146	14,380
8,192	223	552	4,277	28,307	17,898
16,384	243	1,136	4,956	30,098	18,854
16,384	310	1,452	8,134	35,927	22,137
16,384	443	2,070	16,828	47,143	23,850

device with the percentage of available resources that would be required to implement the design. The utilization percentage of each resource on the target device for the base design was calculated as shown in Table 4.11. These results show that configurations up to and including $n = 8192$ and $\log p = 157$ fit on the device without

Table 4.11: Resource utilization of the base polynomial multiplier for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)
1,024	19	1	2	1	2
1,024	22	1	2	1	2
1,024	31	1	4	1	2
2,048	33	2	4	1	2
2,048	42	2	5	2	3
2,048	58	3	11	2	3
4,096	62	5	16	2	3
4,096	80	6	25	3	4
4,096	113	9	48	3	4
8,192	123	17	51	3	5
8,192	157	21	83	4	5
8,192	223	30	170	5	7
16,384	243	62	197	5	7
16,384	310	80	323	7	8
16,384	443	113	668	9	9

issue whereas larger configurations run out of DSP resources. Because the amount of DSP resources required are only dependent upon the size of the coefficients and the maximum coefficient size grows for polynomials of higher degree, larger polynomials could be utilized for the resource constrained maximum coefficient size without compromising security. For example, it would be possible to have a configuration with $n = 16384$ and $\log p = 157$ that would both fit on the target device and allow for double the amount of data with at least a 256-bit level of security.

The latency of the design was also reported by the HLS tool and recorded in Table 4.12. Because the design was synthesized with a target clock period of 4 ns, the time required to produce a polynomial product was calculated by multiplying the clock period by the latency in clock cycles. The goal of this work is to improve the performance of purely software based HE solutions through the design of hardware accelerators with HLS. To benchmark this achievement, the performance of the syn-

Table 4.12: Timing results of the base polynomial multiplier for each security configuration

n	$\log p$	Latency	Time (ms)	NTL (ms)	Speedup
1,024	19	94,251	0.377	1.231	3.265
1,024	22	94,251	0.377	1.211	3.212
1,024	31	114,734	0.459	1.211	2.639
2,048	33	250,928	1.004	2.486	2.477
2,048	42	284,726	1.139	2.337	2.052
2,048	58	284,726	1.139	3.248	2.852
4,096	62	741,438	2.966	7.301	2.462
4,096	80	741,438	2.966	10.436	3.519
4,096	113	741,438	2.966	14.582	4.917
8,192	123	1,654,848	6.619	25.367	3.832
8,192	157	1,654,848	6.619	30.280	4.574
8,192	223	1,761,347	7.045	40.644	5.769
16,384	243	3,784,773	15.139	98.995	6.539
16,384	310	3,784,773	15.139	121.281	8.011
16,384	443	4,128,840	16.515	228.205	13.818

thesized design was compared against the polynomial multiplication function within the NTL software library as it is used within HELib, a software library for FHE [45]. This was achieved by generating and performing 100 random multiplications for each configuration through the NTL software library on a 4-core/4-thread 3.7 GHz AMD A10-7850k CPU with 16 GB of RAM and measuring the average computation time. With these results, it was possible to calculate the speedup of the design synthesized through HLS over a proven software solution. The results were promising as even the basic implementation produced a speedup greater than 1 with a maximum speedup of 4.574 achieved for the largest configuration fitting on the target device with $n = 8192$ and $\log p = 157$. The synthesis results of the pipelined and loop unrolled polynomial multiplier are shown in Table 4.13 and the resource utilization percentage for the target device is shown in Table 4.14. As expected, these results show an increase in utilization across all resources, but both FF and LUT totals remain below the available device total for all configurations. The DSP resources

Table 4.13: Synthesis results of the pipelined and loop unrolled polynomial multiplier for each security configuration

n	$\log p$	BRAM_18K	DSP48E	FF	LUT
1,024	19	114	156	15,883	20,158
1,024	22	114	182	16,697	20,350
1,024	31	114	416	23,432	24,852
2,048	33	140	416	24,718	26,907
2,048	42	197	494	38,090	32,441
2,048	58	267	988	44,530	37,967
4,096	62	412	1,482	52,407	43,294
4,096	80	526	2,314	59,921	49,322
4,096	113	754	4,524	70,349	58,500
8,192	123	1,401	4,758	81,179	61,183
8,192	157	1,774	7,748	93,895	71,371
8,192	223	2,538	15,886	132,084	90,935
16,384	243	5,116	18,408	141,242	96,662
16,384	310	6,540	30,212	170,052	115,156
16,384	443	9,318	62,504	238,129	134,520
32,768	481	19,534	72,462	254,683	143,607
32,768	616	25,022	119,366	368,677	174,071
32,768	886	35,972	248,092	396,001	122,653

continue to be the limiting factor as the design achieves a maximum configuration of $n = 4096$ and $\log p = 80$ before more than 100% of the available DSP resources are required. The timing results of the design were recorded and the speedup over NTL was calculated as shown in Table 4.15. These results show a significant improvement over the base design achieving a minimum speedup over the NTL software library of 44.626 for $n = 2048$ and $\log p = 42$ and a maximum speedup of 93.092 for $n = 4096$ and $\log p = 80$ for configurations fitting the target device. Though the design achieves significant speedup over NTL, the explosion of DSP and BRAM resources for large polynomials becomes problematic as the design is very restrictive in terms of possible configurations on the target device. As flexibility is another primary goal of the work, the design was analyzed to reduce the number of required resources.

Table 4.14: Resource utilization of the pipelined and loop unrolled polynomial multiplier for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)
1,024	19	6	6	3	7
1,024	22	6	7	3	7
1,024	31	6	17	4	9
2,048	33	8	17	5	10
2,048	42	11	20	7	12
2,048	58	15	39	8	14
4,096	62	23	59	10	16
4,096	80	29	92	11	18
4,096	113	41	180	13	21
8,192	123	77	189	15	22
8,192	157	97	307	17	26
8,192	223	139	630	24	33
16,384	243	280	730	26	35
16,384	310	359	1,199	31	42
16,384	443	511	2,480	43	49
32,768	481	1,071	2,875	46	52
32,768	616	1,372	4,737	67	64
32,768	886	1,972	9,845	72	45

Table 4.15: Timing results of the pipelined and loop unrolled polynomial multiplier

n	$\log p$	Latency	FPGA (ms)	NTL (ms)	Speedup
1,024	19	6,083	0.024	1.231	50.592
1,024	22	6,083	0.024	1.211	49.770
1,024	31	6,122	0.024	1.211	49.453
2,048	33	13,056	0.052	2.486	47.603
2,048	42	13,092	0.052	2.337	44.626
2,048	58	13,092	0.052	3.248	62.023
4,096	62	28,026	0.112	7.301	65.127
4,096	80	28,026	0.112	10.436	93.092
4,096	113	28,026	0.112	14.582	130.076
8,192	123	59,812	0.239	25.367	106.028
8,192	157	59,812	0.239	30.280	126.563
8,192	223	59,841	0.239	40.644	169.800
16,384	243	127,458	0.510	98.995	194.172
16,384	310	127,458	0.510	121.281	237.884
16,384	443	127,502	0.510	228.205	447.454
32,768	481	270,898	1.084	470.785	434.467
32,768	616	270,926	1.084	570.413	526.355
32,768	886	270,865	1.083	901.426	831.988

4.7 Resource Allocation

The DSP resources were primarily utilized for the many multiplication operations within the polynomial multiplier. Because the multiplication operation was performed at many different stages within the design, these they were implemented as separate multiplier primitives in hardware utilizing different DSP resources. Vivado HLS supports resource sharing in which the number of instances of specific hardware primitives or functions can be allocated. This allows for a single hardware instance to be utilized in multiple locations within the design. Though resource sharing is beneficial in that it can reduce the number of required resources, it does have drawbacks as the sharing of hardware resources results in more congested routing and can lower the maximum clock frequency achievable by the design. The first step taken was to move the multiplication operation within the modular reduction function and target this as the limited resource. The current design contained HLS UNROLL directives with a factor of two applied to the *weight_coeff*, *mult_coeff*, and *unweight_coeff* loops resulting in a minimum of four concurrent modular multiplication functions as each array was manually partitioned by two and operated in parallel. To reduce the number of required function calls, the HLS UNROLL directives were removed from each of these loops resulting in a reduced minimum of two required instances. To ensure that the tool shared these resources in the synthesized design, the HLS ALLOCATION directive was applied to the top-level function limiting the total number of instances of the modular multiplication to two. A second version of the design was also created removing the manual partitioning of the data arrays and the manual loop unrolling of the FFT and IFFT functions for comparison. With these changes, the design required a minimum of only a single shared modular multiplication function and the HLS ALLOCATION directive was applied to the top-level function accordingly.

4.8 Synthesis Results

The final polynomial multiplier designs with resource sharing were synthesized with Vivado HLS 2018.3 using the default settings targeting the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (xczu9eg-ffvb1156-2-i-es2) with a clock period of 4 ns for each of the configurations outlined in Table 2.1. The synthesis results of the pipelined and loop unrolled polynomial multiplier with resource sharing are shown in Table 4.16. These results show a significant reduction in the number of required BRAM and DSP

Table 4.16: Synthesis results of the pipelined and loop unrolled polynomial multiplier with resource sharing for each security configuration

n	$\log p$	BRAM_18K	DSP48E	FF	LUT
1,024	19	59	24	10,568	20,172
1,024	22	59	48	11,225	21,924
1,024	31	59	96	14,363	25,420
2,048	33	66	96	17,213	23,044
2,048	42	97	120	24,490	26,724
2,048	58	130	240	29,338	31,842
4,096	62	207	384	35,009	36,743
4,096	80	265	600	40,235	42,553
4,096	113	381	1,176	49,143	52,555
8,192	123	754	1,200	44,847	31,264
8,192	157	947	1,968	51,613	35,318
8,192	223	1,354	4,056	71,926	43,426
16,384	243	2,812	4,728	79,519	46,395
16,384	310	3,594	7,800	98,020	53,599
16,384	443	5,125	16,224	129,570	56,349
32,768	481	10,916	18,840	140,062	60,114
32,768	616	13,980	31,128	189,248	70,952
32,768	886	20,100	64,896	229,909	62,180

resources. For example, the largest achieved configuration of the pipelined and loop unrolled polynomial multiplier without resource sharing for the target device was $n = 4096$ and $\log p = 80$ requiring 526 BRAM_18K and 2,314 DSP48E resources whereas this same configuration with resource sharing requires only 265 BRAM_18K

and 600 DSP48E resources. The utilization percentage of each resource on the target device for the pipelined and loop unrolled polynomial multiplier with resource sharing was calculated as shown in Table 4.17. The largest configuration achieved without

Table 4.17: Resource utilization of the pipelined and loop unrolled polynomial multiplier with resource sharing for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)
1,024	19	3	1	2	7
1,024	22	3	2	2	8
1,024	31	3	4	3	9
2,048	33	4	4	3	8
2,048	42	5	5	4	10
2,048	58	7	10	5	12
4,096	62	11	15	6	13
4,096	80	15	24	7	16
4,096	113	21	47	9	19
8,192	123	41	48	8	11
8,192	157	52	78	9	13
8,192	223	74	161	13	16
16,384	243	154	188	15	17
16,384	310	197	310	18	20
16,384	443	281	644	24	21
32,768	481	598	748	26	22
32,768	616	766	1,235	35	26
32,768	886	1,102	2,575	42	23

over utilization of DSP resources became $n = 8192$ and $\log p = 157$, a significant improvement over the implementation without resource sharing. Despite this, these changes were expected to come at a cost as the removal of unrolled loops will increase the latency of the design to some degree. The timing results of the pipelined and loop unrolled polynomial multiplier with resource sharing are shown in Table 4.18. The speedup of the configuration with $n = 4096$ and $\log p = 80$ for the pipelined and loop unrolled polynomial multiplier with resource sharing is 78.587 versus the speedup of 93.092 achieved without resource sharing. Though reduced, the speedup

Table 4.18: Timing results of the pipelined and loop unrolled polynomial multiplier with resource sharing for each security configuration

n	$\log p$	Latency	FPGA (ms)	NTL (ms)	Speedup
1,024	19	7,390	0.030	1.231	41.644
1,024	22	7,390	0.030	1.211	40.968
1,024	31	7,399	0.030	1.211	40.918
2,048	33	15,614	0.062	2.486	39.804
2,048	42	15,686	0.063	2.337	37.247
2,048	58	15,686	0.063	3.248	51.766
4,096	62	33,184	0.133	7.301	55.004
4,096	80	33,199	0.133	10.436	78.587
4,096	113	33,210	0.133	14.582	109.771
8,192	123	70,120	0.280	25.367	90.441
8,192	157	70,120	0.280	30.280	107.958
8,192	223	70,150	0.281	40.644	144.847
16,384	243	148,011	0.592	98.995	167.209
16,384	310	148,011	0.592	121.281	204.851
16,384	443	148,041	0.592	228.205	385.375
32,768	481	311,920	1.248	470.785	377.328
32,768	616	311,978	1.248	570.413	457.094
32,768	886	311,948	1.248	901.426	722.417

remains significantly above 1 showcasing the speed of the accelerated multiplier over the polynomial multiplication function within the NTL software library. Additionally, the largest achieved configuration of $n = 8192$ and $\log p = 157$ for the design with resource sharing results in a maximum speedup of 107.958 over the NTL software library. The full speedup results for this design are displayed in Figure 4.8.

The synthesis results of the pipelined polynomial multiplier with resource sharing and no unrolled loops are shown in Table 4.19. These results also show a significant reduction in the number of required BRAM and DSP resources as the pipelined and loop unrolled design with resource sharing with $n = 4096$ and $\log p = 80$ required only 265 BRAM_18K and 600 DSP48E resources whereas the pipelined only version with resource sharing required 191 BRAM_18K and 300 DSP48E resources. The number

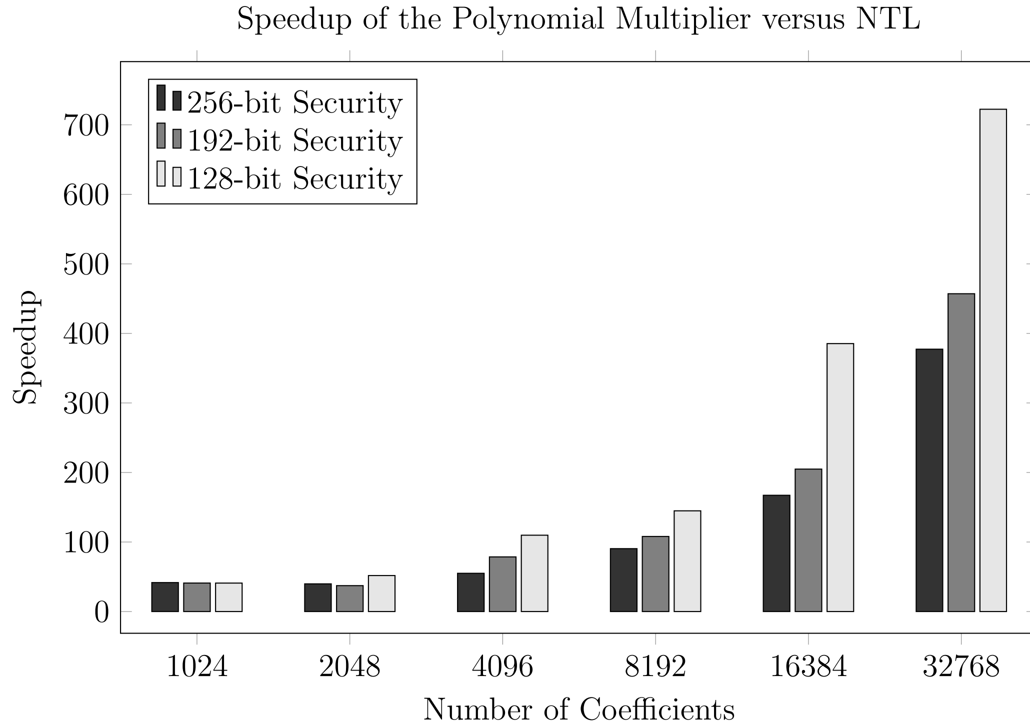

Figure 4.8: Speedup of the pipelined and loop unrolled polynomial multiplier with resource sharing versus NTL

of required DSP resource were reduced by half, directly proportional to the removal of the loops unrolled by a factor of two. The utilization percentage of each resource on the target device for the pipelined polynomial multiplier with resource sharing and no unrolled loops was calculated as shown in Table 4.20. The largest configuration achieved by the pipelined design without unrolled loops became $n = 8192$ and $\log p = 223$ with the number of available BRAMs becoming the new limiting factor as $n = 16384$ and $\log p = 243$ would require only 93.810% of the DSP resources but 115.170% of the available BRAM resources. The timing results of the pipelined polynomial multiplier with resource sharing and no unrolled loops are shown in Table 4.21. As anticipated, the removal of unrolled loops by a factor of two doubled the latency of the design as the configuration with $n = 4096$ and $\log p = 80$ yielding a speedup of 39.557 versus the speedup of 78.587 achieved previously. The largest achievable configuration of $n = 8192$ and $\log p = 223$ for this design resulted in a maximum

Table 4.19: Synthesis results of the pipelined polynomial multiplier with resource sharing for each security configuration

n	$\log p$	BRAM_18K	DSP48E	FF	LUT
1,024	19	27	12	8,744	13,357
1,024	22	27	24	9,041	14,227
1,024	31	27	48	10,182	15,745
2,048	33	46	48	11,379	17,965
2,048	42	61	60	15,100	20,417
2,048	58	84	120	17,580	24,464
4,096	62	149	192	21,503	27,981
4,096	80	191	300	26,482	33,073
4,096	113	275	588	32,278	40,681
8,192	123	552	600	35,863	44,922
8,192	157	701	984	42,017	54,077
8,192	223	1,002	2,028	69,054	72,665
16,384	243	2,108	2,364	77,740	81,999
16,384	310	2,694	3,900	95,734	100,511
16,384	443	3,843	8,112	136,186	134,169
32,768	481	8,240	9,420	152,827	152,290
32,768	616	10,552	15,564	254,252	189,655
32,768	886	15,172	32,448	266,559	137,797

speedup of 72.704 over the NTL software library. The full speedup results for this design are displayed in Figure 4.9.

The developed polynomial multiplier designs expose the trade-offs between speed and area that can be achieved and allow for further flexibility depending upon the resource and security constraints of the target device. For example, the fully pipelined and loop unrolled design provides improved speed at the cost of increased resource utilization, but it may be suitable in an application that does not require polynomials with large coefficients. Furthermore, the xczu9eg device targeted in this work was used to benchmark the design, but there are many devices available that contain a larger number of hardware resources. For example, the Virtex Ultrascale+ devices utilized by AWS F1 instances in the cloud (xcvu9p) [1] and the Xilinx Alveo family

Table 4.20: Resource utilization of the pipelined polynomial multiplier with resource sharing for each security configuration targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)
1,024	19	1	0	2	5
1,024	22	1	1	2	5
1,024	31	1	2	2	6
2,048	33	3	2	2	7
2,048	42	3	2	3	7
2,048	58	5	5	3	9
4,096	62	8	8	4	10
4,096	80	10	12	5	12
4,096	113	15	23	6	15
8,192	123	30	24	7	16
8,192	157	38	39	8	20
8,192	223	55	80	13	27
16,384	243	116	94	14	30
16,384	310	148	155	17	37
16,384	443	211	322	25	49
32,768	481	452	374	28	56
32,768	616	579	618	46	69
32,768	886	832	1,288	49	50

of data center workload accelerator cards (xcu250) [46] contain vast quantities of available resources for hardware acceleration of large-scale computing tasks. The available resources of each of these parts are outlined in Table 4.22. Both devices provide significantly more resources over the target xczu9eg device with the xcvu9p devices attaining a maximum configuration of $n = 16,384$ and $\log p = 310$ and the xcu250 devices reaching a maximum configuration of $n = 32,768$ and $\log p = 481$ before running out of DSP resources for the pipelined polynomial multiplier with resource sharing and no unrolled loops. The developed polynomial multiplier is thus configurable for a wide range of security configurations that can be selected depending on the required latency and resource constraints for a particular application.

Table 4.21: Timing results of the pipelined polynomial multiplier with resource sharing for each security configuration

n	$\log p$	Latency	FPGA (ms)	NTL (ms)	Speedup
1,024	19	14,551	0.058	1.231	21.150
1,024	22	14,551	0.058	1.211	20.806
1,024	31	14,560	0.058	1.211	20.793
2,048	33	30,964	0.124	2.486	20.072
2,048	42	31,036	0.124	2.337	18.825
2,048	58	31,036	0.124	3.248	26.163
4,096	62	65,930	0.264	7.301	27.685
4,096	80	65,956	0.264	10.436	39.557
4,096	113	65,967	0.264	14.582	55.262
8,192	123	139,728	0.559	25.367	45.386
8,192	157	139,728	0.559	30.280	54.177
8,192	223	139,758	0.559	40.644	72.704
16,384	243	295,441	1.182	98.995	83.769
16,384	310	295,441	1.182	121.281	102.627
16,384	443	295,486	1.182	228.205	193.076
32,768	481	623,204	2.493	470.785	188.857
32,768	616	623,334	2.493	570.413	228.775
32,768	886	623,290	2.493	901.426	361.560

Table 4.22: Available resources for various Xilinx FPGAs

Part	URAM_288K	BRAM_36K	BRAM_18K	DSP48E	FF	LUT
xczu9eg	-	-	1,824	2,520	548,160	274,080
xcvu9p	-	-	4,320	6,840	2,364,480	1,182,240
xcu250	1,280	2,000	-	11,508	2,749,000	1,341,000

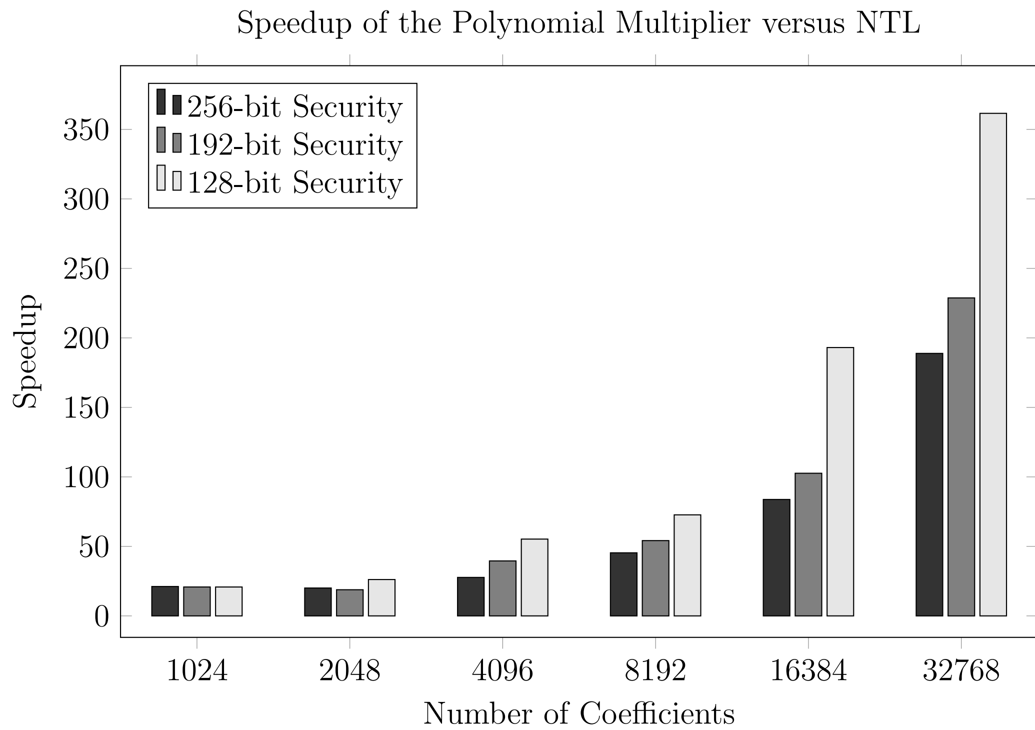

Figure 4.9: Speedup of the pipelined polynomial multiplier with resource sharing versus NTL

4.9 Implementation Results

The synthesized polynomial multiplier designs with resource sharing were exported to and implemented with Vivado 2018.3 using the default settings targeting the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (xczu9eg-ffvb1156-2-i-es2). The initial target clock period was 4 ns for a subset of the 256-bit security configurations outlined in Table 2.1 up to a maximum of 8,192 coefficients. The implementation results of the pipelined and loop unrolled polynomial multiplier with resource sharing are shown in Table 4.23. This table shows that the actual number of resources required

Table 4.23: Implementation results of the pipelined and loop unrolled polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K	DSP48E	FF	LUT
1,024	19	30	24	8,177	6,485
2,048	33	36	96	10,160	10,789
4,096	62	120	384	20,228	18,956
8,192	123	483	1,192	41,645	28,407

for each configuration is less than reported by synthesis for all types. For example, for $n = 8192$ and $\log p = 123$, synthesis reported utilizations of 754 BRAMs, 1,200 DSPs, 44,847 FFs, and 31,264 LUTs whereas implementation of the design actually required 483 BRAMs, 1,192 DSPs, 41,645 FFs, and 28,407 LUTs. The utilization percentage of each resource on the target device for the pipelined and loop unrolled polynomial multiplier with resource sharing was also calculated for the implementation results as shown in Table 4.24. Because of the reduced resource utilization, the percentage of resources required for the target device has been decreased as well. The BRAM utilization has been significantly reduced as, with $n = 8192$ and $\log p = 123$, 41% of the BRAM resources are required for synthesis whereas only 26% of the BRAM resources are actually required for implementation. The implementation timing results of the pipelined and loop unrolled polynomial multiplier with resource sharing are shown in

Table 4.24: Implementation resource utilization of the pipelined and loop unrolled polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)
1,024	19	2	1	1	2
2,048	33	2	4	2	4
4,096	62	7	15	4	7
8,192	123	26	47	8	10

Table 4.25. Note that the target clock period was 4 ns for all configurations and was

Table 4.25: Implementation timing results of the pipelined and loop unrolled polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	WNS (ns)	f_{max} (MHz)	Latency	FPGA (ms)	NTL (ms)	Speedup
1,024	19	0.658	299.222	7,390	0.025	1.231	49.843
2,048	33	0.317	271.518	15,614	0.058	2.486	43.230
4,096	62	0.099	256.345	33,184	0.129	7.301	56.400
8,192	123	0.002	200.080	70,120	0.350	25.367	72.382

achieved for $n = 1024,\ 2048,\ 4096$, but had to be increased to 5 ns for $n = 8192$ as setup times were violated and timing was not met. The maximum achievable clock frequency for each configuration was calculated based on the Worst Negative Slack (WNS) achieved from the place and route of the design on the target device. The goal clock period of 4 ns was used as the basis for the timing calculations from the synthesis results, but the actual timing results for the target device were obtained from the implementation. A maximum clock frequency of 299.222 MHz was achieved for $n = 1024$ and $\log q = 19$ resulting in a speedup of 49.843 versus the synthesis speedup of 41.644. As the synthesis speedup results were based on a 250 MHz clock, the actual speedup that can be achieved is greater once implemented on the target device. Conversely, a minimum clock frequency of 200.080 MHz was achieved for $n = 8192$ and $\log q = 123$ resulting in a speedup of 72.382, significantly lower than the speedup of 90.441 achieved by the synthesis target. The speedup results using the

achieved frequencies from implementation versus the target frequencies from synthesis for a clock frequency of 250 MHz are displayed in Figure 4.10.

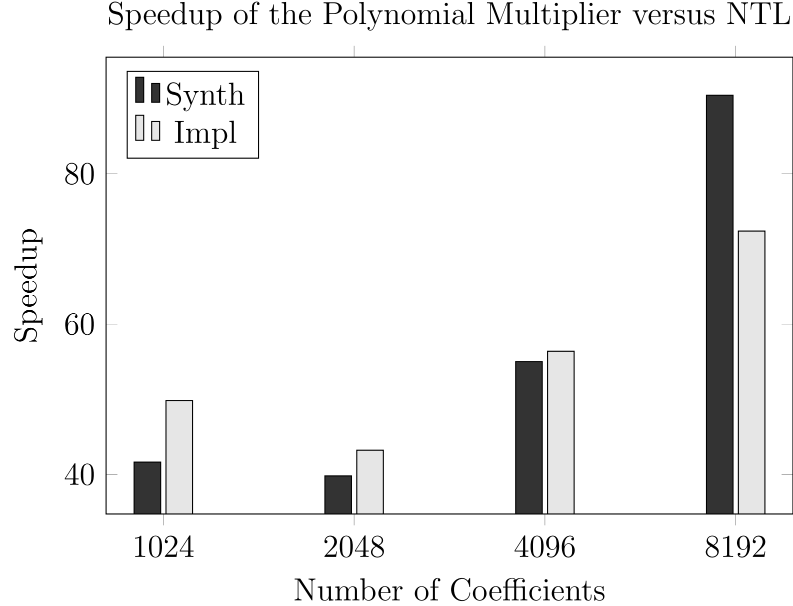

Figure 4.10: Implementation speedup of the pipelined and loop unrolled polynomial multiplier with resource sharing versus NTL for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

The implementation results of the pipelined polynomial multiplier with resource sharing are shown in Table 4.26. Additionally, this table shows that the actual num-

Table 4.26: Implementation results of the pipelined polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K	DSP48E	FF	LUT
1,024	19	15.5	12	5,282	5,154
2,048	33	32.5	48	7,412	7,570
4,096	62	113	192	14,144	13,860
8,192	123	479	596	35,665	31,993

ber of resources required for each configuration is also less than reported by synthesis for each type. For $n = 8192$ and $\log p = 123$, synthesis reported utilizations of 552 BRAMs, 600 DSPs, 35,863 FFs, and 44,922 LUTs whereas implementation of the design actually required 479 BRAMs, 596 DSPs, 35,665 FFs, and 31,993 LUTs. Inter-

estingly, the BRAMs necessary for synthesis of the design without loop unrolling was much closer to the amount required for implementation than the design with loop unrolling. The utilization percentage of each resource on the target device for the pipelined and loop unrolled polynomial multiplier with resource sharing was also calculated for the implementation results as shown in Table 4.27. Though the amount of

Table 4.27: Implementation resource utilization of the pipelined polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	BRAM_18K (%)	DSP48E (%)	FF (%)	LUT (%)
1,024	19	1	0	1	2
2,048	33	2	2	1	3
4,096	62	6	8	3	5
8,192	123	26	24	7	12

required resources has been reduced compared to synthesis, because the design without loop unrolling requires fewer resources overall, there was not a large reduction in the utilization percentage. For example, the BRAM utilization has been not significantly reduced as, with $n = 8192$ and $\log p = 123$, 30% of the BRAM resources were estimated for synthesis whereas 26% of the BRAM resources are actually required for implementation. The implementation timing results of the pipelined multiplier with resource sharing are shown in Table 4.28. As before, the target clock period was

Table 4.28: Implementation timing results of the pipelined polynomial multiplier with resource sharing for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

n	$\log p$	WNS (ns)	f_{max} (MHz)	Latency	FPGA (ms)	NTL (ms)	Speedup
1,024	19	0.692	302.297	14,551	0.048	1.231	25.574
2,048	33	0.246	266.383	30,964	0.116	2.486	21.387
4,096	62	0.079	255.037	65,930	0.259	7.301	28.242
8,192	123	0.084	203.417	139,728	0.687	25.367	36.930

4 ns for all configurations and was achieved for $n = 1024,\ 2048,\ 4096$, but had to be increased to 5 ns for $n = 8192$ as setup times were violated and timing was not met. The maximum achievable clock frequency for each configuration was again calculated

based on the WNS achieved from the place and route of the design on the target device. The goal clock period of 4 ns was used as the basis for the timing calculations from the synthesis results, but the actual timing results for the target device were obtained from the implementation. A maximum clock frequency of 302.297 MHz was achieved for $n = 1024$ and $\log q = 19$ resulting in a speedup of 25.574 versus the synthesis speedup of 21.150. As the synthesis speedup results were based on a 250 MHz clock, the actual speedup that can be achieved is greater once implemented on the target device. Conversely, a minimum clock frequency of 203.417 MHz was achieved for $n = 8192$ and $\log q = 123$ resulting in a speedup of 36.930, much lower than the speedup of 45.386 achieved by the synthesis target. The speedup results using the achieved frequencies from implementation versus the target frequencies from synthesis for a clock frequency of 250 MHz are displayed in Figure 4.11.

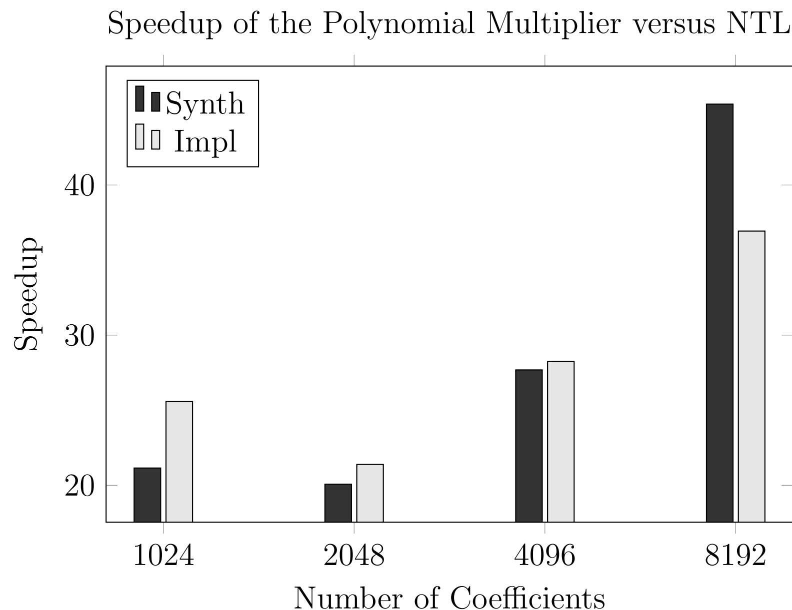

Figure 4.11: Implementation speedup of the pipelined polynomial multiplier with resource sharing versus NTL for 256-bit security configurations targeting part xczu9eg-ffvb1156-2-i-es2

These implementation results show that the smaller configurations are able to achieve higher clock frequencies than the larger configurations for both designs. This

allows for the smaller configurations to achieve greater speedups than the target results from synthesis. The larger designs required slower clock frequencies than the target resulting in lower speedups than the synthesis results. This is likely due to more congested routing on the FPGA because of the higher resource utilization and larger coefficient sizes. It is suggested that the coefficient size is the largest bottleneck because the design without loop unrolling does not reach significantly higher clock frequencies than the design with loop unrolling for the same configuration despite much higher resource utilization. Further speedup could potentially be achieved by dividing the integer multiplications for these large coefficients into multiple operations with smaller operands. Though this method would result in higher latency, this drawback may be overshadowed by an increase in clock frequency and requires further investigation. Despite this, the overall results continue to show a positive increase in speedup over NTL. This is shown by the implementation results in Figures 4.10 and 4.11 in which an increase in the configuration size continues to result in an increase in achieved speedup despite the decrease in achieved clock frequency over the synthesis results. It should be noted that these implementation results are dependent upon the target device and larger devices with more resources may be able to achieve the target frequency of 250 MHz or more. Therefore, the best design and configuration for a specific application will be heavily dependent on the overall timing requirements and the selected target device.

Chapter 5

Conclusion

The security benefits of HE are significant, especially in regards to cloud computing, as secure data can be operated on without revealing the underlying plain text to untrusted parties. The acceleration of the computationally intensive high-precision, high-degree polynomial arithmetic operations within FHE schemes is of the utmost importance to enable their widespread use. The hardware accelerated FFT based polynomial multiplier developed in this work through HLS shows promise in achieving this reality with significant speedup over the modular polynomial multiplication operations performed by the NTL software library for various security configurations. Although the design does not exceed the performance of dedicated hardware solutions, the multiplier exhibits flexibility in the selection of both the polynomial degree and coefficient size allowing for it to be configured for the security level and target device required for a specific application. This would not be possible for a typical hardware design flow without significant effort. Future work will be necessary to implement the remaining functions required by FHE schemes and to test a full system within a cloud environment.

Bibliography

[1] Amazon. Amazon Web Services (AWS) - cloud computing services. [Online]. Available: https://aws.amazon.com

[2] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI design of a large-number multiplier for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 1879–1887, Sept 2014.

[3] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, Jan 2015.

[4] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, June 2015.

[5] M. J. Foster, "Accelerating homomorphic encryption in the cloud environment through high-level synthesis and reconfigurable resources," Master's thesis, Rochester Institute of Technology, 2017.

[6] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using karatsuba algorithm," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 335–347, March 2018.

[7] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[8] K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast number theoretic transform," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, March 2018, pp. 106–111.

[9] A. Mkhinini, P. Maistri, R. Leveugle, R. Tourki, and M. Machhout, "A flexible RNS-based large polynomial multiplier for fully homomorphic encryption," in *2016 11th International Design Test Symposium (IDT)*, Dec 2016, pp. 131–136.

[10] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, "HLS design of a hardware accelerator for homomorphic encryption," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2017, pp. 178–183.

[11] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, “Co-designed accelerator for homomorphic encryption applications,” *Advances in Science, Technology and Engineering Systems Journal*, vol. 3, no. 1, pp. 426–433, 2018.

[12] *Vivado Design Suite User Guide: High-Level Synthesis*, Xilinx, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

[13] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC ‘09*. ACM Press, 2009.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “Fully homomorphic encryption without bootstrapping,” Cryptology ePrint Archive, Report 2011/277, 2011, https://eprint.iacr.org/2011/277.

[15] ——, “(Leveled) fully homomorphic encryption without bootstrapping,” in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference on - ITCS ‘12*. ACM Press, 2012.

[16] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC ’05. New York, NY, USA: ACM, 2005, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/1060590.1060603

[17] O. Regev, “The learning with errors problem (invited survey),” in *2010 IEEE 25th Annual Conference on Computational Complexity*, June 2010, pp. 191–204.

[18] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[19] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, “Security of homomorphic encryption,” HomomorphicEncryption.org, Redmond WA, USA, Tech. Rep., July 2017.

[20] M. Albrecht, R. Player, and S. Scott, “On the concrete hardness of learning with errors,” *Journal of Mathematical Cryptology*, vol. 9, 10 2015.

[21] M. R. Albrecht, R. Fitzpatrick, and F. Göpfert, “On the efficacy of solving LWE by reduction to unique-SVP,” in *Information Security and Cryptology – ICISC 2013*, H.-S. Lee and D.-G. Han, Eds. Cham: Springer International Publishing, 2014, pp. 293–310.

[22] R. Lindner and C. Peikert, “Better key sizes (and attacks) for LWE-based encryption,” in *Topics in Cryptology – CT-RSA 2011*, A. Kiayias, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 319–339.

[23] H. Chen, K. E. Lauter, and K. E. Stange, "Attacks on the search-RLWE problem with small errors," *CoRR*, vol. abs/1710.03739, 2017. [Online]. Available: http://arxiv.org/abs/1710.03739

[24] T. Granlund and the GMP development team, "GNU multiple precision arithmetic library," 2016, version 2.5.2,, https://gmplib.org/.

[25] B. Gladman, W. Hart, J. Moxham *et al.*, "MPIR: Multiple Precision Integers and Rationals," 2017, version 3.0.0, http://mpir.org.

[26] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[27] W. Hart, F. Johansson, and S. Pancratz, "The GNU MPFR library," 2018, version 4.0.1, https://www.mpfr.org/.

[28] V. Shoup, "NTL: A library for doing number theory," 2018, version 11.3.2, https://www.shoup.net/ntl/.

[29] *PARI/GP* , The PARI Group, Univ. Bordeaux, 2018, version 2.11.0, http://pari.math.u-bordeaux.fr/.

[30] W. B. Hart, "Fast library for number theory: An introduction," in *Proceedings of the Third International Congress on Mathematical Software*, ser. ICMS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 88–91, http://flintlib.org.

[31] W. Hart, F. Johansson, and S. Pancratz, "FLINT: Fast Library for Number Theory," 2015, version 2.5.2, http://flintlib.org.

[32] M. Fürer, "Faster integer multiplication," *SIAM Journal on Computing*, vol. 39, no. 3, pp. 979–1005, 2009. [Online]. Available: https://ezproxy.rit.edu/login?url=https://search.proquest.com/docview/880199851?accountid=108

[33] A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," 1963.

[34] A. L. den Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," 1963.

[35] S. A. Cook and S. O. Aanderaa, "On the minimum computation time of functions," 1969.

[36] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, sep 1971.

[37] A. Schönhage, "Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients," in *Computer Algebra*, J. Calmet, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 3–15.

[38] R. P. Brent and P. Zimmerman, *Modern Computer Arithmetic*. Cambridge University Press, 2010. [Online]. Available: https://www.ebook.de/de/product/12890013/richard_p_brent_modern_computer_arithmetic.html

[39] S. Benz, "Fast multiplication of multiple-precision integers," Master's thesis, Rochester Institute of Technology, 1991.

[40] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, 2nd ed. Springer-Verlag, 2005.

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

[42] T. Saint Denis and G. Rose, *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic*. Syngress Publishing Inc., 01 2006.

[43] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation - Math. Comput.*, vol. 44, pp. 519–519, 04 1985.

[44] P. Gaudry, A. Kruppa, and P. Zimmermann, "A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm," in *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '07. New York, NY, USA: ACM, 2007, pp. 167–174. [Online]. Available: http://doi.acm.org/10.1145/1277548.1277572

[45] S. Halevi and V. Shoup, "HElib - an implementation of homomorphic encryption," Online, 2013. [Online]. Available: https://github.com/shaih/HElib/

[46] *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*, Xilinx, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf