Accelerating Polynomial Multiplication for RLWE using Pipelined FFT

Neil Thanawala
University of California, Irvine
nthanawa@uci.edu

Hamid Nejatollahi
University of California, Irvine
hnejatol@uci.edu

Nikil Dutt
University of California, Irvine
dutt@ics.uci.edu

Abstract

The evolution of quantum algorithms threatens to break public key cryptography in polynomial time. The development of quantum-resistant algorithms for the post-quantum era has seen a significant growth in the field of post quantum cryptography (PQC). Polynomial multiplication is the core of Ring Learning with Error (RLWE) lattice based cryptography (LBC) which is one of the most promising PQC candidates. In this work, we present the design of fast and energy-efficient pipelined Number Theoretic Transform (NTT) based polynomial multipliers and present synthesis results on a Field Programmable Gate Array (FPGA) to evaluate their efficacy.

NTT is performed using the pipelined R2SDF and R2^2SDF Fast Fourier Transform (FFT) architectures. In addition, we propose an energy efficient modified architecture (Modr2). The NTT-based designed polynomial multipliers employs the Modr2 architecture that achieve on average 2× better performance over the R2SDF FFT and 2.4× over the R2^2SDF FFT with similar levels of energy consumption. The proposed polynomial multiplier with Modr2 architecture reaches 12.5× energy efficiency over the state-of-the-art convolution-based polynomial multiplier and 4× speedup over the systolic array NTT based polynomial multiplier for polynomial degrees of 1024, demonstrating its potential for practical deployment in future designs.

1 Introduction

Public-key cryptography schemes such as RSA heavily rely on the fact that classical computers are incredibly slow, since they take exponential time to find prime factors (p,q) of a very large number $n = p \times q$. However, Shor's Algorithm[29] shows that quantum computers can perform prime factorization in polynomial time which makes public-key obsolete. The National Institute of Standards and Technology (NIST) started an initiative in 2017 to encourage research on Post-quantum cryptography (PQC)[19] that promise to yield quantum-resistant cryptographic schemes [18]. Among the submissions to NIST, Lattice-based Cryptography (LBC) is most common, and is rising as a popular family of PQC algorithms. Lattice-based cryptosystems are promising PQC candidates because some of them combine strong security guarantees in the form of a worst-to-average case reduction with high efficiency and small key and ciphertext/signature sizes. Examples of lattice-based cryptosystems include key exchange mechanism (KEX), key encapsulation mechanism and Digital signature schemes built on the hardness of the Learning With Errors (LWE) problem and its ring variant, the RLWE problem[13]. An attractive property of the LWE problem shown by Regev [27] is that to solve the average-case LWE problem is at least as hard as to (quantumly) solve some worst-case hard lattice problems[8]. Ring Learning with Error (RLWE) is one of the most promising encryption scheme in LBC [9]. The most computationally-intensive kernel of RLWE is polynomial multiplication. In this work, we accelerate polynomial multiplication using Number Theoretic Transform (NTT). NTT is Fast Fourier Transform (FFT) over a finite field. We use Pipelined FFT processors to develop polynomial multipliers on FPGA [11] to demonstrate both improved performance, as well as improved energy efficiency.

1.1 Trends and Challenges

The modern trend to develop sensitive software both at the datacenter level and on the edge has created an urgent need to make the software secure. This requires the implementation of cryptographic algorithms on such devices. Modern day datacenters demand high performance of applications. Developing a low-power secure encryption engine for internet of things (IoT) edge devices is a key challenge. This calls for the need to develop programmable hardware accelerators that achieve a high performance and energy efficiency while also are capable of being reconfigurable to adapt to changing strandards. Developing the newest standards of cryptography on the spectrum of devices along with their constraints poses a huge challenge [19].

1.2 Previous Works

Previous research has made an effort to accelerate RLWE on other platforms such as CPU and GPU. Nejatollahi et. al [22] have implemented NTT based polynomial multipliers on a GPU. Their approach involves accelerating NTT and inverse NTT operations on GPU while the point-wise multiplication and bit-reversal operations run on a CPU with four ARM cortex A-57 cores.

Previous works for accelerating polynomial multiplication for LBC on FPGA have mainly focused on the performance of the accelerators while ignoring the energy efficiency [28][26][5]. However, some efforts have evaluated the energy as well as the area and performance of NTT accelerators [16][2][17]. Nejatollahi et. al [21, 22] explore NTT and convolution polynomial multiplication using systolic array architecture. To the best of our knowledge, there has not been any notable work in the literature on synthesizing NTT cores using pipelined FFT architectures to accelerate polynomial multiplication while considering energy and evaluating its tradeoffs with performance.

1.3 Contributions

In this paper we focus on the design of programmable hardware accelerators on FPGA for polynomial multiplication. We implement NTT-based polynomial multipliers using pipelined-FFT architectures - R2SDF and R2^2SDF for the NTT cores and evaluate the performance, area and energy efficiency. Modr2, a new, modified architecture for the NTT core is proposed and evaluated.

2 Background Work

The learning with error problem requires to find a secret key $s \in \mathbb{Z}_q^n$ from a given sequence of random linear equations with an error e [27]. The difficulty of solving ring learning with error problem even on a quantum computer has made Ring Learning with Error (RLWE) based cryptography a popular choice for post quantum cryptography. Lyubashevsky [14] later showed that applications of LWE can be made more efficient through the use of ring-LWE. RLWE is the LWE problem specialized to polynomial rings over a finite field.

The RLWE problem deals with the arithmetic of polynomials with coefficients from a finite field. Let the ring of polynomials be $\mathcal{R} \equiv \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where n is a power of 2 and q is a relatively large prime number such that $q = 1 \ mod \ 2n$. Polynomial multiplication becomes the most compute intensive operation in this problem.

Let two polynomials of length n be $a(x)$ and $b(x)$ and their product $c(x)$ such that

$$c(x) = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} a_i \cdot b_j x^{i+j}$$

This simplest way to compute this product is using the Schoolbook method in time complexity $\mathcal{O}(n^2)$. Fast Fourier Transform (FFT) however, computes the product in $\mathcal{O}(nlog(n))$. In this work, we implement polynomial multipliers using FFT architectures on FPGA.

2.1 Convolution based Polynomial Multipliers

The Schoolbook method of polynomial multiplication computes the product in a time complexity of $\mathcal{O}(n^2)$. Nejatollahi et al. [21] have used a systolic array architecture to perform convolution in $\mathcal{O}(n)$ where a convolution based multiplier as described in Algorithm 1 is seen as a discrete feed-forward finite impulse response (FIR) over the polynomials in $\mathcal{R} \equiv \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. The speedup is obtained by cascading n multiply-accumulators (MACs) in the systolic architecture. This comes at the cost of higher area and energy which is discussed later.

Algorithm 1 Convolution (Schoolbook)-based Polynomial Multiplier [21]

```
Initialization: Let $a = \{a_0, a_1, a_2, ..., a_{n-1}\}$ and $b = \{b_0, b_1, b_2, ..., b_{n-1}\} \in \mathbb{Z}_q[x]/<f(x)>$ be two polynomials with the length of $n$, where $f(x) = x^n + 1$ is an irreducible polynomial with $n$ a power of 2, and $q \equiv 1\, mod\, 2n$ is a large prime number.
$c \leftarrow 0$
for $i = 0$ to $n-1$ do
    for $j = 0$ to $n-1$ do
        $sign \leftarrow (-1)^{\lfloor (i+j)/n \rfloor}$
        $index \leftarrow (i+j)\ mod\ n$
        $coeff \leftarrow a_i b_i\ mod\ q$
        $c_{index} \leftarrow integer(c_{index} + sign * coeff)\ mod\ q$
    end for
end for
Return c
```

2.2 Number Theoretic Transform

Number theoretic transform (NTT) is discrete Fourier transform performed over a ring. Let n be a power of 2, q is a prime such that $q \equiv 1\ mod\ 2n$, ω is a primitive n-th root of unity such that $\omega^n \equiv 1 mod q$ and $a(x)$ be polynomials of degree n whose coefficients $\in \mathbb{Z}_q^n$ then the NTT of a is defined as:

$$A_i = NTT(a) = \sum_{j=0}^{n-1} a_j \omega^{ij}\ mod\ q$$

where $i = 0, 1, ...n-1$. To compute the inverse NTT, ω is replaced by ω^{-1} and n is replaced by n^{-1} such that $n^{-1} = 1 \ mod \ q$. NTT can be computed using FFT [24].

2.3 Modular Reduction

Addition of two numbers each of bit width N generates a sum of bit width $(N+1)$ whereas their multiplication generates a product of bit width $2N$. To maintain a fixed bit width of N without changing the algorithm, a modulo operation is performed after each addition and multiplication. The modulo operation calculates the remainder which involves a division operation which is quite expensive in terms of latency on hardware. We use Barrett Reduction [3] and Montgomery Reduction [15] to perform the modulo operation. In NTT, given a polynomial length, the modulo factor q is fixed. For example, for $n = 256$ and lesser, $q = 7681$ [1] and for $n = 512$ and $n = 1024$, $q = 12289$ [25].

2.4 NTT based Polynomial Multipliers

NTT based polynomial multiplication is described in Algorithm 2. NTT reduces the time complexity of polynomial multiplication to $\mathcal{O}(nlog(n))$ as compared to $\mathcal{O}(n^2)$ for DFT and Schoolbook methods. The two polynomials $a(x) = a_0 + a_1x + ...a_nx^n$ and $b = b_0 + b_1x + ...b_nx^n$ are first converted to their NTT representation and their multiplication is performed point wise. After the point wise multiplication, an inverse NTT operation is performed to get the desired product of the two polynomials.

$c = INTT(NTT(a) \cdot NTT(b))$

3 Fast Fourier Transform

Fast Fourier Transform (FFT) is a faster method to compute the discrete Fourier transform (DFT) [4]. A N-point Fourier transform takes a time complexity of $\mathcal{O}(n^2)$ using the direct method DFT and $\mathcal{O}(nlogn)$ using FFT. A Fourier transform is popular in the digital signal processing domain as it is used to convert a signal from the time domain to frequency domain. The Fourier transform is heavily used in computer vision, image processing and audio processing. For example, in image processing, eliminating low frequencies renders the edges of the image, where as filtering the higher frequencies blurs the image [7].

Algorithm 2 NTT-based Polynomial Multiplier

```
Initialization: Let $a = \{a_0, a_1, a_2, ..., a_{n-1}\}$ and $b = \{b_0, b_1, b_2, ..., b_{n-1}\} \in \mathbb{Z}_q[x]/<f(x)>$ be two polynomials with length of $n$, where $f(x) = x^n + 1$ is an irreducible polynomial with $n$ a power of 2, and $q \equiv 1\,mod\,2n$ is a large prime number. $w$ is the $n$-th root of unity and $\phi$ is the $2n$-th root of unity ($\phi^2 = w$ mod $q$); $w^{-1}$ and $\phi^{-1}$ are the inverse of $w$ mod$q$ and $\phi$ mod$q$, respectively.
Precompute: $\{w^i, w^{-i}, \phi^i, \phi^{-i}\}$ for $i \in [0, n-1]$
for $i = 0$ to $n-1$ do
    $\bar{a_i} \leftarrow a_i\phi^i$
    $\bar{b_i} \leftarrow b_i\phi^i$
end for
$\bar{A} \leftarrow NTT_w^n(\bar{a})$
$\bar{B} \leftarrow NTT_w^n(\bar{b})$
$\bar{C} = \bar{A}.\bar{B}$
$\bar{c} \leftarrow iNTT_w^n(\bar{C})$
for $i = 0$ to $n-1$ do
    $c_i \leftarrow \bar{c}_i\phi^{-i}$
end for
Return C
```

3.1 Algorithm

FFT is an efficient method to compute the DFT [6]. The DFT problem is defined by

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi jnk/N} \; ; n = 0, 1, ...N-1 \tag{1}$$

where x_n is the n^{th} sample of the input polynomial and X_k is k^{th} coefficient of DFT. $j = \sqrt{-1}$. The equation can be rewritten as

$$X_k = \sum_{k=0}^{N-1} x_n \omega^{nk}$$

where ω is the n^{th} square root of unity given by

$$\omega = e^{-2\pi j/N}$$

.

The FFT algorithm takes advantage of the cyclic nature of ω to efficiently compute the DFT. There are two methods to compute FFT from the given DFT equation. The two methods are discussed below.

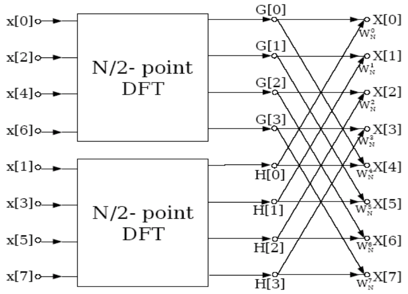

Figure 1: Recursive 8-point DIT FFT [12]

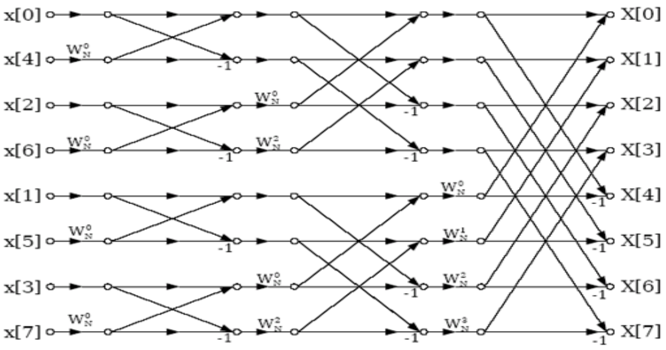

Figure 2: Signal flow graph from 8-point DIT FFT [12]

3.1.1 Decimation in Time

The decimation in time algorithm splits the sum for X_k into even $n = [0, 2, 4...]$ and odd $n = [1, 3, 5...]$ numbered indices given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi nk/N}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j2\pi k(2n)/N} + \sum_{n=0}^{\frac{N}{2}-1} x_{(2n+1)} e^{-j2\pi k(2n+1)/N}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j2\pi kn/\frac{N}{2}} + e^{-j2\pi kn/N} \cdot \sum_{n=0}^{\frac{N}{2}-1} x_{(2n+1)} e^{-j2\pi kn/\frac{N}{2}}$$

$$= DFT_{\frac{N}{2}}[x_{even}] + \omega_N^k DFT_{\frac{N}{2}}[x_{odd}] \tag{2}$$

From Equation 2, it can be seen that the FFT equation is a recursive form of DFT. Figure 1 shows the recursive nature of the FFT. The even indexed samples

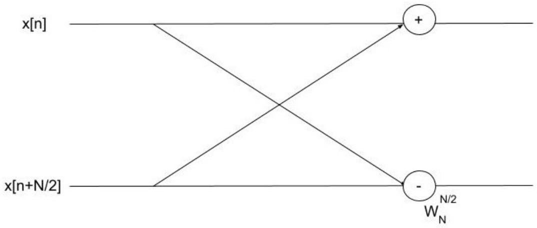

Figure 3: Butterfly operation

are grouped together and the odd indexed samples are indexed together to perform a $\frac{N}{2}$ point DFT of the samples followed by multiplication with ω_k^n also known as twiddle factors. The $\frac{N}{2}$ point DFT can be further split into a $\frac{N}{4}$ point DFT which can be further split into $\frac{N}{8}$ point DFT and so on until there are only two elements which can't be further split. Figure 2 is the signal flow graph for a 8-point FFT that is generated when the all the recursions are simplified.

Figure 3 illustrates the DFT between two inputs known as the butterfly operation. The result of the butterfly is given in the equation below:

$$X[n] = x[n] + x[n+N/2]$$

$$X[n+N/2] = \omega_N^{N/2} \cdot (x[n] - x[n+N/2])$$

An important observation in the DIT FFT is that the inputs are arranged bit reversed order while the outputs are generated in their correct order.

3.1.2 Decimation in frequency

The decimation in frequency algorithm splits the sum for X_k into even $k = [0,2,4...]$ and odd $k = [1,3,5...]$ numbered indices given by $X_k = X_{2r} + X_{2r+1}$:

$$X_{2r} = \sum_{n=0}^{N-1} x_n \omega_N^{2rn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x_n \omega_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x_{(n+\frac{N}{2})} \omega_N^{2r(n+\frac{N}{2})}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x_n \omega_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x_{(n+\frac{N}{2})} \omega_N^{2rn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{(n+\frac{N}{2})}) \cdot \omega_{\frac{N}{2}}^{rn}$$

$$= DFT_{\frac{N}{2}}[x_n + x_{(n+\frac{N}{2})}] \tag{3}$$

$$X_{(2r+1)} = \sum_{n=0}^{N-1} x_n \omega_N^{(2r+1)n}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{(n+\frac{N}{2})} \cdot \omega_N^{\frac{N}{2}}) \cdot \omega_N^{(2r+1)n}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} \left(x_n - x_{(n+\frac{N}{2})} \cdot \omega_N^n\right) \cdot \omega_{\frac{N}{2}}^{rn}$$

$$= DFT_{\frac{N}{2}}[(x_n - x_{(n+\frac{N}{2})}) \cdot \omega_N^n] \tag{4}$$

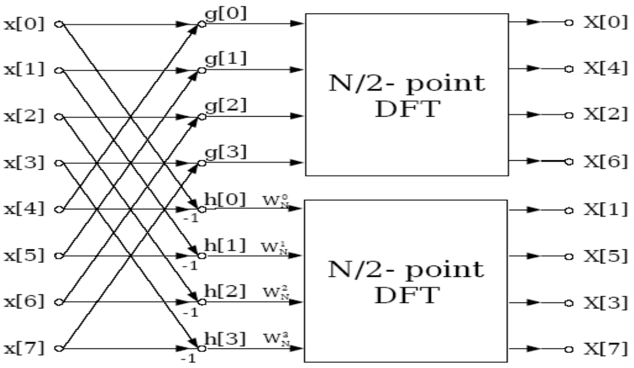

Figure 4: Recursive 8-point DIF FFT [12]

Equations 3 and 4 prove that similar to the DIT FFT, the DIF FFT takes advantage of the recursive nature of the twiddle factors and is seen in Figure 4. Figure 5 shows the signal flow graph of a 8 point DIF FFT. The key difference in this case is that the inputs are in order while the outputs generated are in bit-reversed order.

In the sections below we explore two architectures that perform radix-2 FFT namely radix-2 single delay feedback (R2SDF) and radix-2^2 single delay feedback (R2^2SDF)

3.2 R2SDF FFT

The radix-2 single delay feedback (R2SDF) FFT architecture uses a pipeline of processing units to compute the FFT [23]. This architecture implements the radix-2 DIF FFT algorithm explained above. Inputs are in order while the outputs are in

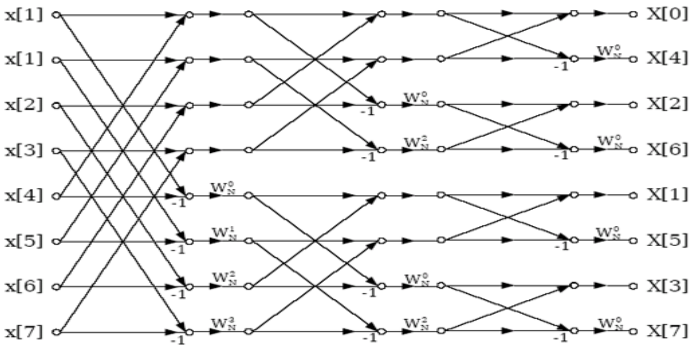

Figure 5: Signal flow graph from 8-point DIF FFT [12]

bit-reversed order. At the end, a bit-reversal operation is necessary to rearrange the outputs.

Algorithm 3 describes the pipelined FFT algorithm for each Stage s in the pipeline. An important note is that the loop iterations for the outer loop of j and inner loop k varies in each stage depending on the value of s. However, the total number of loop iterations $j \times k$ remains the same for each stage.

R2SDF uses $\log_2 N$ stages to compute the Fast Fourier Transform of a signal of length N [23]. Each stage contains a butterfly unit and a FIFO. The microarchitecture of the 'i^{th} stage' is described in Figure 6:

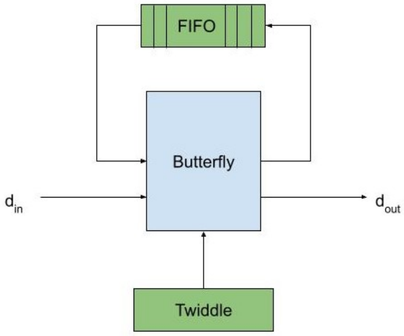

Figure 6: Processing element of R2SDF FFT. The input data is d_{in} and the output is d_{out}.ω or twiddle factors are the n^{th} roots of unity. Butterfly operation consists of addition and subtraction of the two inputs followed by multiplication with the twiddle factors.

In each stage, the initial $\frac{N}{2}i$ inputs are serially stored in the FIFO. A butterfly operation consists of a multiplication and addition. The butterfly operation is per-

Algorithm 3 FFT using R2SDF architecture

```
Initialization: Let $a = \{a_0, a_1, a_2, ..., a_{n-1}\} \in \mathbb{Z}_q[x]/<f(x)>$ be the input signal of length $n$, with $n$ a power of 2, and $q \equiv 1 \, mod \, 2n$ is a large prime number. $\omega$ is the $n$-th root of unity.
Precompute: $\{\omega^i\}$ for $i \in [0, n-1]$
Output: A
Stage=s
depth = $\frac{1024}{2^s}$
num_iterations = $(2^{s-1})$
for $j = 0$ to $j = num\_iterations$ do
    for $k = 0$ to $k = depth$ do
        fifo.write($a_i$)
    end for
    for $k = 0$ to $depth$ do
        $A_k \leftarrow (a_{(k+depth)} +$ fifo.read($a_k$)) mod q
        fifo.write(fifo.read($a_k$) - $a_{(k+depth)}$)
    end for
    for $k = 0$ to $depth$ do
        $A_{(k+depth)} \leftarrow (\omega_i \times$ fifo.read($a_k$)) mod q
    end for
end for
for $i = 0$ to $n-1$ do
    $A_i \leftarrow$bit_reverse($A_i$)
end for
Return A
```

formed between the $\frac{N}{2}i$ data stored and the next $\frac{N}{2}i$ inputs and the results computed are stored back into the FIFO. The next step involves multiplication of the butterfly output with the twiddle factors stored in the twiddle ROM. After the first $\frac{N}{2}i$ data points are processed, the next $\frac{N}{2}i$ data points enter the FIFO and thus all points in the signal are processed. The 4 stages of a 16 point R2SDF FFT are pipelined as shown in the Figure 7. When N=16, $log_2 N = 4$. Stage 1 has a FIFO of depth $\frac{16}{2} = 8$. Similarly, stage 2 has a FIFO depth of $\frac{16}{2^2} = 4$ and so on.

3.3 R2^{2}SDF FFT

The radix-2^2 single delay feedback (R2^{2}SDF) FFT has the same butterfly structure as a radix-2 FFT and the same number of multiplications as radix-4 FFT. Compared to the R2SDF architecture, the R2^{2}SDF architecture tries to improve performance

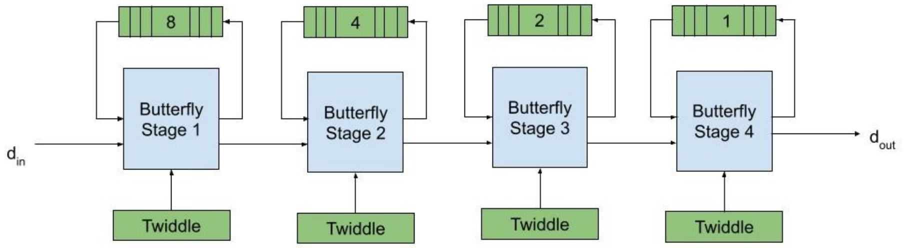

Figure 7: Dataflow of a 16-point FFT performed with a 4-stage pipelined R2SDF architecture.

by reducing the number of non-trivial multiplications [11]. The signal flow graph for a 16 point R2^2SDF FFT algorithm is shown in the Figure 8. In the signal flow graph above describing the 16 point R2^2SDF algorithm, it can be seen that the non-trivial multiplications with the twiddle factors ω occur only in stage 2 and stage 4; in every alternate stage. In stage 1 and stage 3, the multiplications are trivial and require smaller multipliers.

3.4 Modified R2SDF FFT (Modr2)

Taking a close look at the R2SDF architecture, it can be seen that there are three main tasks being performed - (i) store the inputs into the FIFO, (ii) perform the butterfly, (iii) multiply the twiddle factors. The three tasks utilize the same FIFO. This restricts parallelism and forces the three tasks to perform serially creating a major bottleneck. We propose the Modified R2SDF FFT (Modr2) architecture that increases task level parallelism and gets a higher throughput by modifying the existing R2SDF architecture. It is observed that in each stage of the FFT, a butterfly operation is performed between two inputs. Hence, we split the input array into two such that the i^{th} element of the first FIFO performs a butterfly with the i^{th} element of the second FIFO and each FIFO is of size $\frac{N}{2}$ as shown in Figure 9.

Another observation that is made is that in each stage 'i', the butterfly operation is performed between the n^{th} and the $[n+\frac{N}{2}i]^{th}$ input. Hence, it becomes essential that the output of stage 'i' is stored in the two FIFO's in an order such that the n^{th} element of the first FIFO performs a butterfly with the n^{th} element of the second FIFO in stage $i+1$. This can be easily achieved due to the recursive nature of the FFT. The dataflow for the Modr2 architecture is shown in Figure 10.

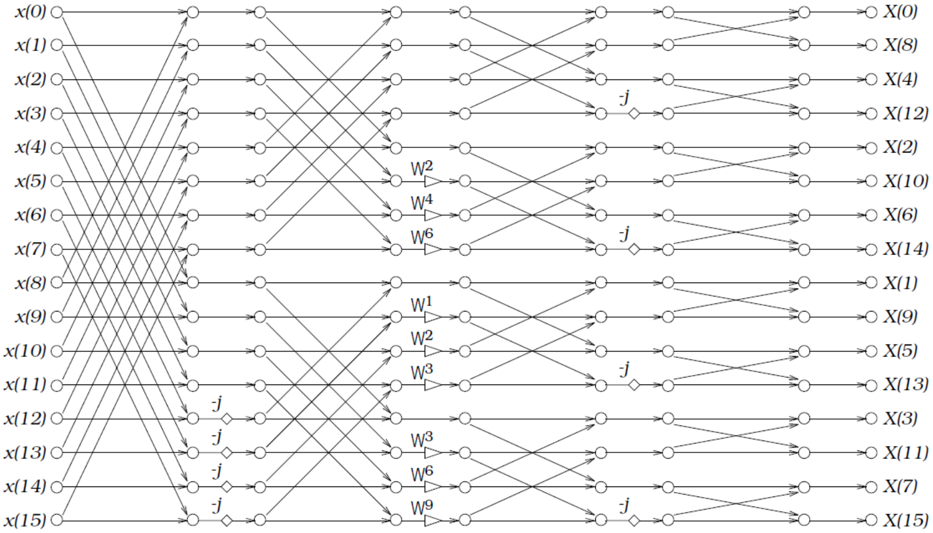

Figure 8: Signal flow graph for a 16 point R2^2SDF FFT

4 Evaluation

The evaluation of different schemes was performed by synthesizing the algorithms on the Artix-7 FPGA platform using Vivado HLS 2018.2. The key metrics we used to evaluate the results are resource utilization - (BRAM, LUT, DSP, FF), frequency, post implementation energy. The microarchitecture of the FFT designs were described using high-level synthesis. The Vivado HLS tool compiles and synthesizes the HLS files to generate the RTL.

Table 1 illustrates the results obtained to perform NTT for different polynomial

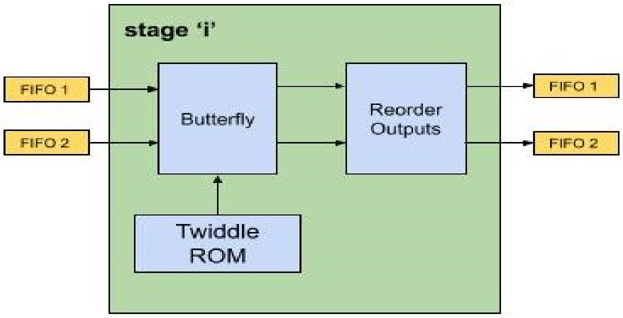

Figure 9: Processing element of each stage 'i' in the Modr2 architecture

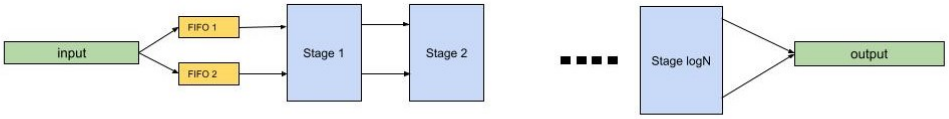

Figure 10: Dataflow of a N-point FFT performed with a $logN$-stage pipelined Modr2 architecture

lengths using R2SDF, R2^2SDF and the newly proposed Modr2 Modified R2SDF FFT architecture.

Table 1: Table describing the post implementation results of the R2SDF, R2^2SDF and the Modr2 FFTs simulated on Vivado HLS 2018.2 for N=64,128,256,512,1024.

Design	N	Cycles	Latency (μS)	Energy (μJ)	BRAM	CLB	DSP	FF	LUT	Freq. (MHz.)
R2SDF	64	347	2.53	0.16	15	1024	35	1024	2820	136.96
	128	673	5.10	0.36	17	1132	40	2206	3110	131.87
	256	1318	10.14	0.81	19	1336	45	2585	3572	129.98
	512	2603	19.43	1.75	21	1509	50	2969	4086	133.95
	1024	5168	38.63	3.59	23	1583	55	3095	4508	133.77
R2^2SDF	64	329	3.61	0.24	15	1111	41	1784	3001	91.12
	128	772	6.57	0.60	32	1196	44	2834	3316	117.5
	256	1260	11.28	1.25	36	1449	57	3408	4014	111.69
	512	2992	25.32	3.09	42	1750	60	4074	4732	118.19
	1024	4920	41.26	5.57	44	1910	66	4396	5160	119.23
Modr2	64	212	1.54	0.13	4	1840	35	3389	4440	137.93
	128	401	2.62	0.44	41	2027	40	5650	4861	152.93
	256	731	4.75	0.91	46	2315	45	6586	5771	154.04
	512	1381	9.49	1.95	52	2713	54	7559	6789	145.48
	1024	2671	17.89	4.58	57	3069	55	8566	7760	149.34

Taking a look at $N = 1024$, the highly parallel modified R2SDF architecture gives a slightly better than 2× performance to compute NTT than the R2SDF and R2^2SDF architectures in terms of latency and clock cycles at a better frequency. This comes at the cost of higher resource utilization resulting in a larger area. The comparison of performance is described in Figure 11.

Energy consumption is another important factor in the design space. Figure 12 illustrates the plot between energy and log_2N, obtained from Table 1. For $N = 64, 128, 256, 512$, the Modr2 architecture and R2SDF have an almost identical energy consumption, R2SDF being slightly better. The lower energy for R2SDF can be accounted for by the low resource utilization.

We further analyze the performance and energy consumption of the Modr2 architecture at different frequencies. A frequency sweep was performed on the

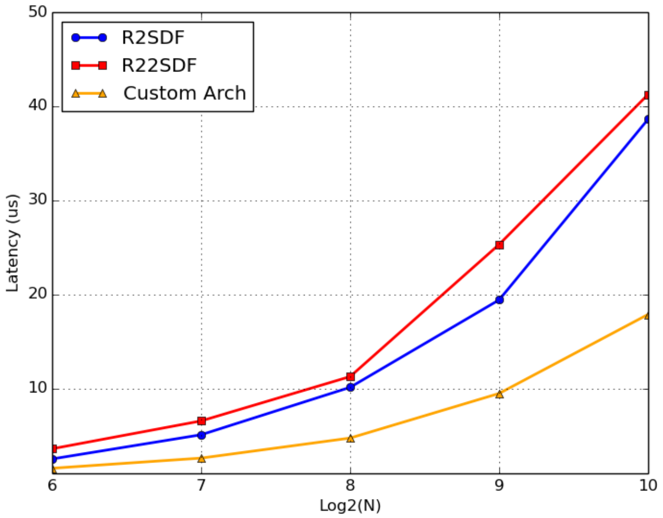

Figure 11: Plot of latency vs $\log_2$N for R2SDF, R2^2SDF, Modr2 architectures.

design. A frequency sweep synthesizes the HLS design using different resources to meet the timing constraints. We perform the frequency sweep starting with a target clock period of 1ns upto a clock period of $20ns$. The results obtained can be categorized into three buckets: high target frequency, medium target frequency and low target frequency. The results are summarized in the Table 2:

According to Table 2, as the frequency increases from $110MHz$ to $200MHz$, the clock cycles also increase. For instance when $N = 256$, the low frequency implementation requires 709 clock cycles while the high frequency implementation takes 825 clock cycles which is 16% higher. This can be justified by the fact that in a pipelined architecture, each stage of the pipeline takes more clock cycles due to the combinational delay of processing elements as frequency increases and clock period decreases. Another observation from the results is that, with increasing frequency a higher number of flip flops are used.

For $N = 64$, 128, $2\times$ energy is consumed at higher target frequencies as compared to the medium and low frequencies. For all values of N, the medium and low frequency implementation have identical energy requirements. For $N = 512$, the higher frequency design takes $2.24\mu s$, 23%less than the medium frequency but at a cost of 53% higher energy.

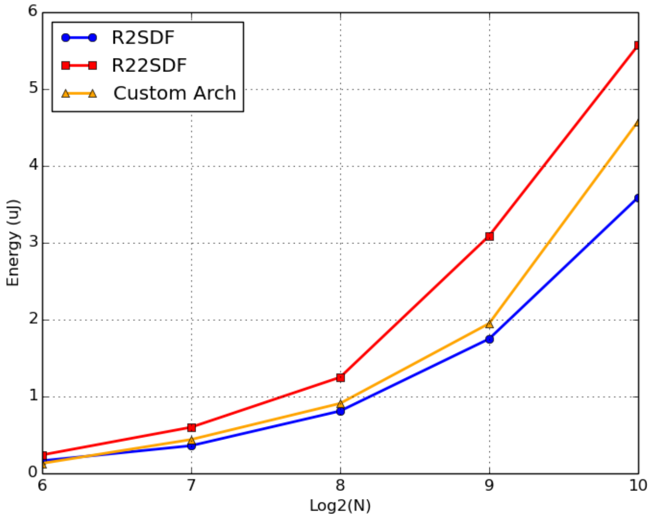

Figure 12: Plot of energy vs $\log_2$N for R2SDF, R2^2SDF, Modr2 architectures.

5 Polynomial Multiplication

As seen from Algorithm 2, polynomial multiplication consists of three major steps: (i) forward NTT of the 2 polynomials (ii) element wise multiplication of the computed NTT (iii) inverse NTT.

A NTT Multiplier requires 2 forward NTT blocks and 1 inverse NTT block. The two task level pipelines are shown in Figures 13 and 14.

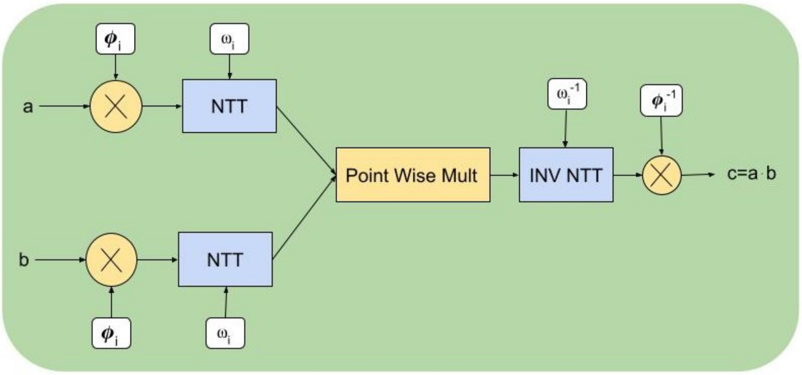

Figure 13: Pipeline for polynomial multiplication using two parallel NTT blocks.

In Figure 13, the forward NTT blocks for the two polynomials are executed in parallel while being pipelined with the element wise multiply and inverse NTT

Table 2: Table describing the post implementation results of the frequency sweep performed on the Modr2 FFT architecture categorized into high, medium and low frequency simulated on Vivado HLS 2018.2 for N=64, 128, 256, 512, 1024.

Target Freq.	N	Cycles	Latency (μS)	Energy (μJ)	BRAM	CLB	DSP	FF	LUT	Freq. (MHz.)
High	64	292	1.42	0.26	4	1987	35	4564	4583	206.31
	128	488	2.36	0.87	41	2235	40	6996	5126	206.78
	256	825	3.83	1.53	46	2552	45	8101	6067	215.38
	512	1482	7.25	4.13	52	2885	50	9244	6933	204.33
	1024	2779	13.36	8.06	58	3211	55	10233	7810	208.03
Medium	64	212	1.54	0.13	4	1840	35	3389	4440	137.93
	128	401	2.62	0.44	41	2027	40	5650	4861	152.93
	256	731	4.75	0.91	46	2315	45	6586	5771	154.04
	512	1381	9.49	1.95	52	2713	54	7559	6789	145.48
	1024	2671	17.89	4.58	57	3069	55	8566	7760	149.34
Low	64	194	2.13	0.13	2	1971	35	2977	5080	91
	128	381	3.23	0.41	38	2249	40	5169	5505	117.87
	256	709	6.18	0.83	43	2457	45	6036	6418	114.77
	512	1357	11.88	1.98	48	2815	50	6940	7445	114.23
	1024	2645	22.87	4.14	53	3239	55	7878	8414	115.66

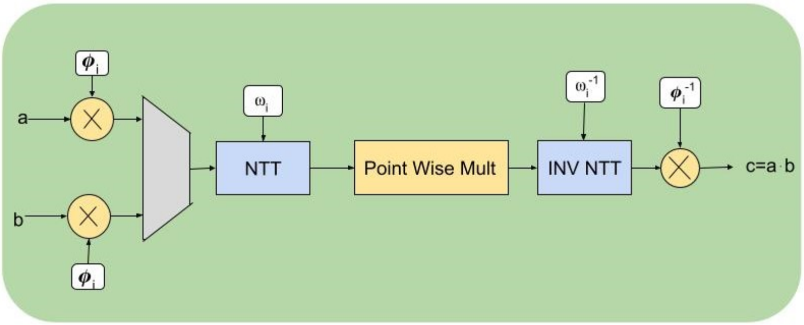

Figure 14: Pipeline for polynomial multiplication using one NTT block.

blocks. Two forward NTT blocks are required for this case. While in Figure 14, the forward NTT of the two polynomials is serial, enabling the reuse of the forward NTT block for the second polynomial. Hence only one forward NTT block is required. This implementation would be useful in energy and area constrained applications. Table 3 compares the two implementations of the NTT based polynomial multiplier simulated on Xilinx Artix 7 FPGA using Vivado HLS 2018.2. The Modr2 architecture is used for the NTT blocks.

Looking at the results, for example, when N=256, the parallel implementation takes a lower number of clock cycles and has a latency improvement of 46% at the cost of 1.98 μJ energy which is about 34% higher than the serial implementation.

Table 3: Table describing the post implementation results of the serial and parallel polynomial multiplier using the Modr2 architecture for NTT core simulated on Vivado HLS 2018.2 for N=64,128,256,512,1024.

Impl.	N	Cycles	Latency (μS)	Energy (μJ)	Freq. (MHz.)
Serial	64	636	4.61	0.4	137.93
	128	1203	7.87	1.33	152.93
	256	2193	14.24	2.72	154.04
	512	4143	28.48	5.87	145.48
	1024	8013	53.66	13.74	149.34
Parallel	64	354	2.51	0.66	141.3
	128	656	4.42	1.69	148.43
	256	1184	8.54	3.89	138.6
	512	2224	15.62	9.61	142.37
	1024	4288	29.13	20.22	147.19

5.1 Comparison with existing implementations

The existing implementations to accelerate polynomial multiplication focus mainly on the latency while not considering the energy consumption. Figure 15 and Figure 16 compare the results of serial and parallel Modr2 based polynomial multiplication from this work to existing implementations on FPGA.

[htbp]

Implementations [26][10][5] have only highlighted the performance metrics of their designs without any mention of power or energy consumption. Of these implementations, the design of Chen et. al has an average 3× less latency than Poppleman et. al [26] and 2.5× better performance than Du et. al [10]. The convolution based multiplier using systolic architectures proposed by Nejatollahi et. al [21] performs the best in terms of latency bettering the parallel Modr2 architecture by 1.5×.

Although the convolution implementation outperforms the parallel Modr2 architecture, it consumes significantly higher energy which might not be suitable in energy constrained devices. The Modr2 architecture consumes 2.32× for $N = 256$, 6.65× for $N = 512$, 12.71× for $N = 1024$ less energy. This difference is considerable compared to the performance and an infavorable tradeoff in most circumstances. The parallel Modr2 FFT consumes approximately 1.8× more energy than the CrytoPIM [20] and the systolic array SA_NTT [21] implementation however having a significantly better performance.

In the design space of latency, area and energy it becomes essential to pick

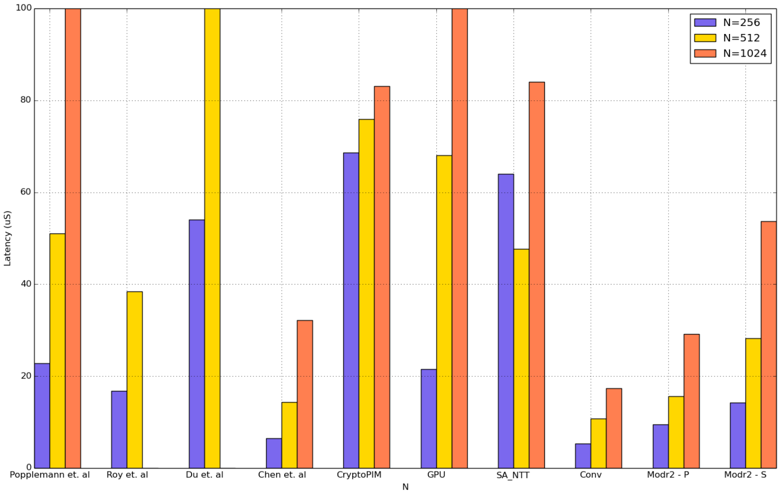

Figure 15: Bar graph comparing the latency of different implementations of polynomial multiplication for $N = 256, 512, 1024$ with Modr2-S being the serial Modr2 and Modr2-P being the parallel Modr2. Conv, SA_NTT, CryptoPIM and GPU are implementations of Nejatollahi et. al [22]

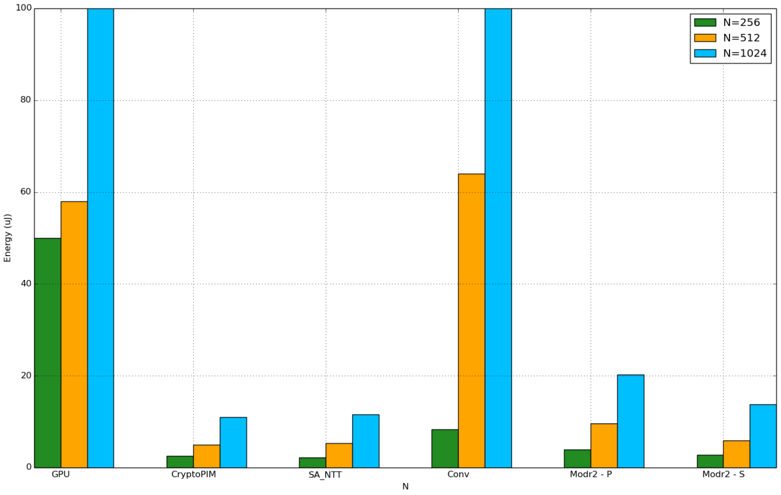

Figure 16: Bar graph comparing the energy of different implementations of polynomial multiplication for $N = 256, 512, 1024$ with Modr2-S being the serial Modr2 and Modr2-P being the parallel Modr2. Conv, SA_NTT, CryptoPIM and GPU are implementations of Nejatollahi et. al [22]

the right implementation given the constraints of the system. In low-powered IoT devices which work on real time data, high performance at a low energy cost is required for which the Modr2 implementation can be ideal. In datacenters, the high frequency implementation of the Modr2 architecture is suitable given that the server farms can afford higher energy consumption at the cost of high frequency and a very low latency.

6 Conclusion and Future Work

The advent of quantum computer threatens to break public key cryptography and make our secure data public. PQC algorithms, and in particular the LBC family of algorithms appear to be promising prospects, but suffer from high computational requirements with heavy energy budgets that may not be feasible for implementation across the cloud-edge-device spectrum. Towards this end, in this paper we explored the implementation of polynomial multiplication on programmable hardware accelerators, the most compute intensive operation in RLWE LBC. We demonstrated improved performance, and further studied energy efficiency of these kernels as well as the design space tradeoffs that were not often considered in previous literature. Indeed, our work for the first time evaluates the tradeoffs between energy and latency in the design pipelined NTT processors for polynomial multiplication. The results are promising, showing that our Modr2 architecture achieves $12.5\times$ better energy efficiency over the state-of-the-art implementations. Future work will investigate further improvements in energy efficiency and performance of the Modr2 architecture. While the performance of an accelerator increases with higher number of resources, the resources themselves may be limited. Thus another future work will consider the dimension of resource utilization in design space exploration, with the overall goal of achieving similar-or-better performance with lower and more efficient resource utilization.

References

[1] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J.M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. 2017. Crystals-kyber. *NIST, Tech. Rep* (2017).

[2] U. Banerjee, T.S. Ukyab, and A.P. Chandrakasan. 2019. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version). Cryptology ePrint Archive, Report 2019/1140. (2019). `https://eprint.iacr.org/2019/1140`.

[3] P. Barrett. 1987. 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.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.

[4] E.O. Brigham and R.E. Morrow. 1967. The fast Fourier transform. *IEEE Spectrum* 4, 12 (1967), 63–70. `https://doi.org/10.1109/MSPEC.1967.5217220`

[5] D.D. Chen, N. Mentens, F. Vercauteren, S.S. Roy, R.C. Cheung, D. Pao, and I. Verbauwhede. 2015. High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems. *TCS* (2015).

[6] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, D.E. Nelson, C.M. Rader, and P.D. Welch. 1967. What is the fast Fourier transform? *Proc. IEEE* 55, 10 (1967), 1664–1674. `https://doi.org/10.1109/PROC.1967.5957`

[7] A. Dhuriya. 2021. Why Fourier Transform is so important? (Jan 2021). `https://medium.com/analytics-vidhya/why-fourier-transform-is-so-important-cb7841733bb8`

[8] J. Ding, X. Xie, and X. Lin. 2012. A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem. Cryptology ePrint Archive, Report 2012/688. (2012). `https://eprint.iacr.org/2012/688`.

[9] C. Du and G. Bai. 2016. Towards efficient polynoFmial multiplication for lattice-based cryptography. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1178–1181. `https://doi.org/10.1109/ISCAS.2016.7527456`

[10] C. Du and G. Bai. 2016. Towards Efficient Polynomial Multiplication for Battice-Based Cryptography. In *ISCAS*.

[11] S. He and M. Torkelson. 1996. A new approach to pipeline FFT processor. In *Proceedings of International Conference on Parallel Processing*. IEEE, 766–770.

[12] D.L. Jones. 2014. Digital Signal Processing: A User's Guide. `https://oers.taiwanmooc.org/jspui/handle/123456789/129699`. (2014).

[13] Z. Liu, K.-K.R. Choo, and J. Grossschadl. 2018. Securing Edge Devices in the Post-Quantum Internet of Things Using Lattice-Based Cryptography. *IEEE Communications Magazine* 56, 2 (2018), 158–162. `https://doi.org/10.1109/MCOM.2018.1700330`

[14] V. Lyubashevsky et al. 2010. On Ideal Lattices and Learning with Errors over Rings *(EUROCRYPT'10)*.

[15] P.L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44 (1985), 519–521.

[16] H. Nejatollahi et al. 2020. Synthesis of Flexible Accelerators for Early Adoption of Ring-LWE Post-quantum Cryptography. *TECS* (2020).

[17] H. Nejatollahi, R. Cammarota, and N. Dutt. 2019. Flexible NTT Accelerators for RLWE Lattice-based Cryptography. *ICCD* (2019).

[18] H. Nejatollahi, N. Dutt, I. Banerjee, and R. Cammarota. 2019. Post-Quantum Lattice-Based Cryptography Implementations: A Survey. *ACM CSUR* (2019).

[19] H. Nejatollahi, N. Dutt, and R. Cammarota. 2017. Trends, Challenges and Needs for Lattice-based Cryptography Implementations: Special Session. In *CODES*.

[20] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt. 2020. CryptoPIM: In-memory Acceleration for Lattice-based Cryptographic Hardware. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*.

[21] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt. 2020. Exploring Energy Efficient Quantum-resistant Signal Processing Using Array Processors. ICASSP. (2020).

[22] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt. 2021. Exploring Energy Efficient Architectures for RLWE Lattice-Based Cryptography. In *Journal of Signal Processing Systems*.

[23] R. Netto and J.L. Güntzel. 2012. A high throughput configurable FFT processor for WLAN and WiMax protocols. In *2012 VIII Southern Conference on Programmable Logic*. 1–5. `https://doi.org/10.1109/SPL.2012.6211793`

[24] J.M. Pollard. 1971. The Fast Fourier Transform in a Finite Field. Mathematics of Computation. (1971).

[25] T. Poppelmann, E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, P. Schwabe, D. Stebila, M.R. Albrecht, E. Orsini, et al. 2017. Newhope. *NIST, Tech. Rep* (2017).

[26] T. Pö ppelmann and T. Güneysu. 2012. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In *LATINCRYPT*.

[27] O. Regev. 2005. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. (2005).

[28] S.S. Roy, F. Vercauteren, N. Mentens, D.D. Chen, and I. Verbauwhede. 2014. Compact Ring-LWE Cryptoprocessor. In *CHES'14*.

[29] P.W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct 1997), 1484–1509. `https://doi.org/10.1137/s0097539795293172`