Proteus: A Pipelined NTT Architecture Generator

Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy

***Abstract*—Number Theoretic Transform (NTT) is a fundamental building block in emerging cryptographic constructions like fully homomorphic encryption, post-quantum cryptography and zero-knowledge proof. In this work, we introduce Proteus, an open-source parametric hardware to generate pipelined architectures for the NTT. For a given parameter set including the polynomial degree and size of the coefficient modulus, Proteus can generate Radix-2 NTT architectures using Single-path Delay Feedback (SDF) and Multi-path Delay Commutator (MDC) approaches. We also present a detailed analysis of NTT implementation approaches and use several optimizations to achieve the best NTT configuration. Our evaluations demonstrate performance gain up to $1.8\times$ compared to SDF and MDC-based NTT implementations in the literature. Our SDF and MDC architectures use $1.75\times$ and $6.5\times$ less DSPs, and $3\times$ and $10.5\times$ less BRAMs, respectively, compared to state-of-the-art SDF and MDC-based NTT implementations.**

***Index Terms*—Parametric, Pipelined, NTT, FHE, ZKP.**

I. Introduction

Various advancements have emerged in modern cryptography, like fully homomorphic encryption (FHE), post-quantum cryptography (PQC), and zero-knowledge proofs (ZKP). All of these constructions heavily rely on polynomial arithmetic, particularly polynomial multiplication. However, this process can become computationally expensive when dealing with large polynomials. Number Theoretic Transform (NTT) reduces the polynomial multiplication complexity from $O(n^2)$ to $O(n \log_2 n)$. Thus, NTT plays a crucial role in constructing modern cryptographic schemes, enhancing the efficiency of polynomial arithmetic, especially for large polynomials.

The cryptographic schemes mentioned earlier rely on diverse parameter sets, including polynomial degree and coefficient size. However, this diversity poses a challenge when designing a configurable hardware architecture.

Most of the existing solutions in hardware, like [1]–[10], are designed and optimized to support specific parameter sets. There are also some efforts in literature targeting configurable NTT hardware [9], [10] that can support multiple parameters at run-time. Although run-time flexibility enables support for multiple parameters, it subsequently makes the hardware design inefficient due to the area overhead of extra reconfigurability. Therefore, certain works in the literature propose design-time (or compile-time) flexible architectures, where the design parameters are given before synthesis to generate hardware tailored to the given parameters. Design-time configurable architectures have shown better area efficiency compared to run-time flexible hardware. However, the limited number of works targeting design-time flexibility, either target high-bandwidth designs [8]–[12] or do not make the source code available [1]–[4], [6], [8]–[10]. This work aims to develop an open-source and bandwidth-efficient design-time flexible architecture for NTT.

Several approaches for designing and implementing NTT in hardware include hierarchical, iterative, and pipelined methods. The hierarchical approach involves dividing a large NTT operation into smaller parts, reducing the overall implementation cost. The iterative approach is scalable regarding area and performance but has high implementation complexity for large polynomials. On the other hand, the pipelined approach uses Single-path Delay Feedback (SDF) or Multi-path Delay Commutator (MDC) architectures to provide bandwidth-efficient and resource-aware implementations with comparable performance. However, in the existing literature, there are only a few works that specifically target the pipelined NTT approach using SDF and MDC architectures [1]–[4], [9], [10] (discussed in Sec. III-D).

In this paper, we present *Proteus*, open-source and parametric hardware that generates bandwidth-efficient, pipelined, and synthesizable Radix-2 NTT architectures using SDF and MDC approaches, for given polynomial size, coefficient modulus size, and NTT configuration at design-time. This versatile hardware supports several parameters of NTT configurations and features many low-level optimizations to achieve optimal architecture. Compared to state-of-the-art SDF/MDC NTT implementations, Proteus significantly reduces resource consumption while improving performance. One of Proteus's primary benefits is that it saves a hardware designer's precious design time and effort by generating efficient and synthesizable Radix-2 NTT architecture for a given parameter set, thus streamlining the process. The key contributions of our work are summarized as:

- We first present a comprehensive analysis of different configurations for NTT with several optimization techniques to reduce the hardware implementation cost.
- Our analysis shows that it is possible to reduce the implementation complexity of the pipelined NTT architectures without sacrificing performance by combining different configurations and optimization techniques. Although some low-level optimizations are already present in the literature [13], [14], we are the first work to combine these optimizations with different NTT configurations, present an analysis and show that these can be used to improve the area cost of the pipelined NTT architectures.
- We propose and implement Proteus, design-time flexible and parametric hardware that can generate synthesizable Radix-2 SDF/MDC-based pipelined NTT architectures for a given parameter set and configuration. Proteus supports various NTT configurations and incorporates

F. Hirner, A. C. Mert, S. S. Roy are with Institute of Applied Information Processing and Communications, Graz University of Technology, 8010 Graz, Austria (e-mail: {florian.hirner, ahmet.mert, sujoy.sinharoy}@iaik.tugraz.at).

This work was supported in part by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.

numerous optimizations that effectively minimize area consumption.
- We propose parametric implementations for low-level arithmetic units, integer multiplication, modular reduction, and compact butterfly units. Specifically, we adopt the word-level Montgomery reduction for NTT-friendly primes approach [15] and propose an algorithm that effectively maps it into FPGA using DSP units. Furthermore, we incorporated several low-level optimizations [13], [14] to achieve an efficient design.
- Compared to the state-of-the-art SDF-based NTT implementation [1], our implementation uses up to $1.75\times$ and $3\times$ less DSPs and BRAMs, respectively, while showing a similar performance. Furthermore, our implementation supports multiple NTT configurations while [1] supports a single configuration.
- We have made our source code available at https://github.com/florianhirner/proteus. It is worth noting that open-source NTT hardware solutions are relatively scarce within the existing literature. Furthermore, to the best of our knowledge, we present the first open-source parametric implementation of SDF/MDC-based NTT architecture.

The paper is organized as follows. In Sec. II, we present the necessary background, and Sec. III introduces SDF/MDC-based NTT architectures. Sec. IV provides a detailed analysis of different NTT configurations and optimizations. Sec. V illustrates our proposed hardware design, Proteus. In Sec. VI, we report evaluation results, and Sec. VII concludes the paper.

II. Preliminaries

A. Notations

The ring of integers modulo q is represented as $\mathbb{Z}_q$. Let $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ be the polynomial ring that consists of polynomials reduced by the polynomial $x^n + 1$ with coefficients in $\mathbb{Z}_q$. In this paper, q is a prime, and n is a power-of-2. The prime q is either congruent to 1 modulo n or $2n$. We use lowercase letters and lowercase bold font letters to represent integers (e.g., $a \in \mathbb{Z}_q$) and polynomials (e.g., $\boldsymbol{a} \in R_q$), respectively. The ith coefficient of a polynomial $\boldsymbol{a}$ is denoted as $\boldsymbol{a}_i$ or $\boldsymbol{a}[i]$. Therefore, the polynomial $\boldsymbol{a}$ is represented as $\boldsymbol{a} = \sum_{i=0}^{n-1} x^i \boldsymbol{a}_i$ or $\boldsymbol{a} = \sum_{i=0}^{n-1} x^i \boldsymbol{a}[i]$. The NTT of a polynomial $\boldsymbol{a} \in R_q$ is denoted as $\hat{\boldsymbol{a}}$. Let $\cdot$, $\times$, and $\odot$ represent the integer, polynomial and coefficient-wise multiplications, respectively.

B. Number Theoretic Transformation (NTT)

The NTT is used as a fundamental building block in modern cryptographic schemes. NTT is a generalization of the Fast Fourier Transform (FFT) over $\mathbb{Z}_q$ where $q \equiv 1 \pmod n$. An n-pt NTT takes an input polynomial $\boldsymbol{a} \in R_q$ and outputs the evaluation $\hat{\boldsymbol{a}}$ where $\hat{\boldsymbol{a}}_i = \sum_{j=0}^{n-1} \boldsymbol{a}_i \cdot \omega^{ij} \pmod q$ for $i \in [0, n)$. The NTT uses the constant ω called twiddle factor, which is an nth primitive root of the unity (i.e., $\omega^n \equiv 1 \pmod q$ and $\omega^i \neq 1 \pmod q \ \forall i < n$). The inverse NTT (INTT) transforms an NTT-output into a polynomial representation as $\boldsymbol{a}_i = n^{-1} \sum_{j=0}^{n-1} \hat{\boldsymbol{a}}_i \cdot \omega^{ij} \pmod q$ for $i \in [0, n)$.

Algorithm 1 DIF NTT with GS Butterfly [18]

Input: $\boldsymbol{a} \in R_q$ (in normal order), ω (powers of ω in normal order)
Output: $\hat{\boldsymbol{a}} \leftarrow \text{NTT}(\boldsymbol{a}) \in R_q$ (in bit-reversed order)

```
m ← 1
for (y = n; y > 1; y = y/2) do
    for (k = 0; k < m; k = k + 1) do
        j1 ← k · y, j2 ← j1 + (y/2) − 1, j3 ← 0
        for (j = j1; j ≤ j2; j = j + 1) do
            w ← ω[j3], u ← a[j], v ← a[j + y/2]
            a[j] ← (u + v), a[j + y/2] ← (u − v) · w
            j3 ← j3 + m
        end for
        m ← 2 · m, y ← y/2
    end for
end for
return a
```

There are two approaches [16] to compute an NTT, namely, decimation-in-time (DIT) and decimation-in-frequency (DIF). The DIT approach of NTT uses the Cooley-Tukey (CT) butterfly, and the DIF approach uses the Gentleman-Sande (GS) butterfly. For a given input coefficient pair (a, b) and a constant ω^i, the CT and GS butterflies output the coefficient pairs $\{a+b\cdot\omega^i, a-b\cdot\omega^i\}$ and $\{a+b, (a-b)\cdot\omega^i\}$ respectively. These approaches enable different NTT configurations (Sec. IV).

In the NTT domain, polynomial multiplication is a simple coefficient-wise operation [17]. An NTT-based polynomial multiplication of two n coefficient polynomials first zero pads each polynomial into $2n$-coefficients, then computes $2n$-pt NTTs of the two polynomials, then multiplies them coefficient-wise, and finally computes one $2n$-pt INTT to obtain the result of the polynomial multiplication. If the polynomial multiplication is performed in a polynomial ring, a modular reduction by the irreducible polynomial is performed.

When working in $R_q = \mathbb{Z}_q[x]/x^n + 1$ with n a power-of-2, a special optimization known as the negative wrapped convolution (NWC) could be used to reduce the computation overhead as only n-pt NTT/INTT are required instead of $2n$-pt NTT/INTT. NWC requires the existence of a $2n$th root of the unity, say $\psi \in \mathbb{Z}_q$, which is possible only when $q \equiv 1 \pmod{2n}$. For computing $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ in R_q using NWC, the input polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ are first scaled by the powers of ψ to obtain $\boldsymbol{a'} = (\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n-1}) \odot (\psi^0, \psi^1, \ldots, \psi^{n-1})$ and $\boldsymbol{b'} = (\boldsymbol{b}_0, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{n-1}) \odot (\psi^0, \psi^1, \ldots, \psi^{n-1})$ respectively, which we refer to as the pre-processing. Then, the standard n-pt NTT-based polynomial multiplication is used to obtain $\boldsymbol{c'} = \text{INTT}\big(\text{NTT}(\boldsymbol{a'}) \odot \text{NTT}(\boldsymbol{b'})\big)$. Finally, the coefficients of $\boldsymbol{c'}$ are multiplied by the powers of ψ^{-1} to obtain the original result $\boldsymbol{c} = (\boldsymbol{c'}_0, \ldots, \boldsymbol{c'}_{n-2}, \boldsymbol{c'}_{n-1}) \odot (\psi^0, \psi^{-1}, \ldots, \psi^{-(n-1)})$. The final scaling step is called post-processing in our paper.

It is possible to combine the pre- and post-processing with NTT and INTT, respectively [19], [20]. This requires using the DIT approach for NTT with ψ and the DIF approach for INTT with ψ^{-1}. In this paper, we refer to these NTT and INTT as merged NTT (MNTT) and merged INTT (MINTT), respectively. The algorithm for Radix-2 Iterative DIF-based NTT is given in Algorithm 1. Note that FHE/PQC generally uses MNTT/MINTT, while ZKP protocols use NTT/INTT. Proteus provides support for both NTT/INTT and MNTT/MINTT.

Order of input and output coefficients: In-place NTT and

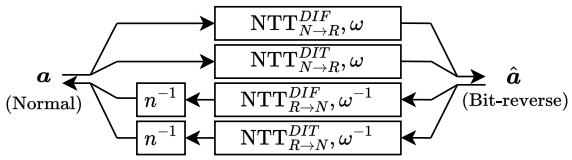

Fig. 1. Configurations for NTT/INTT.

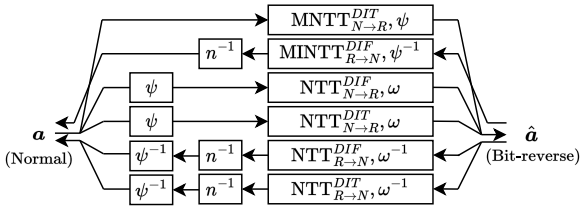

Fig. 2. Configurations for MNTT/MINTT.

MNTT operations change the order of the coefficient after the transformations. A DIF NTT takes a polynomial in the normal order (i.e., $\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n-1}$) and generates a polynomial in the bit-reversed order (i.e., $\boldsymbol{a}_{br(0)}, \boldsymbol{a}_{br(1)}, \ldots, \boldsymbol{a}_{br(n-1)}$) where $br(\cdot)$ represents bit-reverse of $\log_2(n)$-bit integer. It is possible to derive various configurations [21] for DIF and DIT approaches such as normal to bit-reversed order ($N \to R$ or N-to-R) and bit-reversed to normal order ($R \to N$ or R-to-N). We summarize all possible DIT and DIF configurations for NTT/INTT and MNTT/MINTT in Fig. 1 and Fig. 2, respectively.

We use superscript and subscript to represent the DIT/DIF type and input polynomial order change for an NTT/MNTT operation, respectively. We also use ω and ω^{-1} to represent NTT or INTT, respectively. For example, $\mathrm{NTT}^{DIT}_{N\to R},\omega$ represents a DIT NTT that takes the input polynomial in the normal order and generates the output polynomial in bit reversed order. In Fig. 1 and Fig. 2, the boxes with ψ, ψ^{-1} and n^{-1} represent the pre-processing, post-processing and scalar multiplication by n^{-1} operations, respectively.

III. SDF and MDC Architectures for NTT

Choosing an appropriate NTT architecture in hardware depends on the platform and application constraints. The n-point in-place Radix-2 DIF NTT in Algorithm 1 has $\log_2(n)$ stages where each stage performs $\frac{n}{2}$ butterfly operations (steps 7-8 of Algorithm 1). An iterative NTT implementation may use one or several butterfly units to compute the butterfly operations in all stages. Therefore, the iterative NTT can scale the latency and on-chip memory bandwidth. However, it requires high memory bandwidth and implementation complexity when the polynomials have large degrees [12], [15]. A fully or partially unrolled NTT architecture unrolls the NTT stages (step 2 of Algorithm 1) by instantiating many butterfly cores for each stage. Such an unrolled architecture can be pipelined to achieve high throughput. However, the area requirement of an unrolled NTT will be substantial when the polynomial degree is large [22].

SDF and MDC architectures, also called pipelined architectures, use only one butterfly unit for each NTT stage. Hence, they instantiate only $\log_2(n)$ butterfly cores for implementing an n-pt NTT unit. Each butterfly core is exclusive to a specific NTT stage, performing $n/2$ butterfly operations of the corresponding stage. The pipelined architectures provide comparable throughput and performance with low bandwidth requirements [2]. In this work, we target SDF and MDC Radix-2 NTT implementations.It is worth noting that both iterative and SDF/MDC NTT architectures implement the same algorithm (e.g., Algorithm 1) and have the same functionality. They only differ in implementation approach.

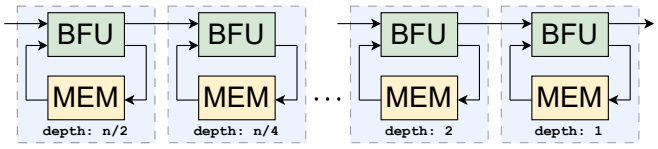

Fig. 3. Radix-2 SDF architecture. BFU and MEM are butterfly and memory.

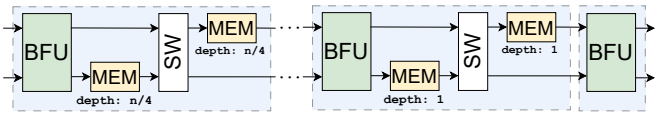

Fig. 4. Radix-2 MDC architecture. SW is a commuter switch.

A. Single-path Delay Feedback (SDF) Architecture

The Radix-2 SDF architecture takes one input coefficient per cycle and generates one output coefficient per cycle after filling the internal pipeline stages. It uses one butterfly unit for each stage, and each butterfly unit is coupled with one memory for temporarily storing several coefficients. Fig. 3 shows a high-level view of an SDF-NTT architecture.

A butterfly computation requires two coefficients where the coefficient indices are separated by an offset dependent on the stage. For example, a 256-pt DIF NTT (Algorithm 1) of a polynomial $\boldsymbol{a}$ takes coefficients whose indices are separated by an offset of 128 (e.g., ($\boldsymbol{a}_i$, $\boldsymbol{a}_{i+128}$) for $i = 0, \ldots, 127$) in the first stage. In the second stage, the butterfly unit takes coefficients separated by an offset of 64 (e.g., ($\boldsymbol{a}_i$, $\boldsymbol{a}_{i+64}$) for $i = 0, \ldots, 63$ and $i = 128, \ldots, 191$). This pattern continues until the last stage, which requires coefficients separated by an offset of just 1.

The SDF architecture couples each butterfly unit with a memory to provide inputs to the butterfly units with the proper offset. For 256-pt DIF NTT, the first stage is coupled with a memory of depth 128 as the coefficients in the butterfly are separated by an offset of 128. The SDF architecture takes and stores the first 128 input coefficients in the memory in the first 128 cycles. Then, it takes the following 128 inputs from the external source one-by-one while reading the first 128 coefficients from its own memory one-by-one, and sends the coefficient pairs to the butterfly unit during the subsequent 128 cycles one-by-one. In this way, the butterfly unit receives coefficient pairs in the correct order. The first output of each butterfly operation in the first stage (e.g., $\boldsymbol{a}_0$ to $\boldsymbol{a}_{127}$) is sent to the next stage while the second output (e.g., $\boldsymbol{a}_{128}$ to $\boldsymbol{a}_{255}$) is stored inside the memory to be sent to the next stage later. The next stage processes the polynomial in two parts, meaning that it separates the coefficients of each part by an offset of 64 (e.g., ($\boldsymbol{a}_i$, $\boldsymbol{a}_{i+64}$) for $i = 0, \ldots, 63$ and $i = 128, \ldots, 191$). It first processes the first part, then reads the second part from the memory of the first stage (e.g., $\boldsymbol{a}_{128}$ to $\boldsymbol{a}_{255}$) and

processes it. Each stage employs this technique to perform an NTT operation, yet with another offset.

B. Multi-path Delay Commutator (MDC) Architecture

An MDC architecture takes two coefficients per cycle as input and generates two outputs every cycle after filling the pipeline. It uses one butterfly unit, two memory units for reordering the data, and a switch commuter for each NTT stage, as shown in Fig. 4. Similar to the SDF architecture, a butterfly operation during NTT/INTT takes two coefficients where the two coefficient indices are separated by an offset dependent on the stages. Each stage is coupled with two memories of depth d for preparing its output (i.e., reorder) for the next NTT stage. The depth d of the memory units depends on the size of the input polynomial n and the stage. It can be calculated as $d = \frac{n}{2^{s+2}}$ where $s = 0, 1, \ldots, \log_2(n) - 1$ is the stage number. In the case of 256-pt DIF NTT, each memory in the first stage has a depth of 64. Inside each stage, the butterfly outputs are reordered for the next stage using two memories and one switch. The switch can change the order of two coefficients depending on the required offset pattern of the next stage.

C. Using SDF/MDC architectures with large polynomials

Implementing NTT of a large-degree polynomial in hardware is not an easy task and it is even more challenging when high performance is targeted. Iterative and unrolled NTT architectures can achieve high performance by employing several butterfly units running in parallel. However, their implementation complexity will be very high for a large degree polynomial. In contrast to iterative and unrolled NTT, the hierarchical NTT approach divides a large-degree NTT operation into multiple smaller-degree NTTs [2], [12]. The separation into smaller NTT operations makes it easier to implement and parallelize operations due to the smaller NTT size. An SDF and MDC-based architecture, like Proteus, can implement these smaller NTT operations requiring low bandwidth. Furthermore, it is possible to instantiate multiple SDF/MDC architectures to perform multiple smaller-sized NTTs at the same time [2].

D. Related Works

There are a plethora of works in the literature targeting efficient NTT hardware architectures. Most of these works target iterative NTT implementations with fixed parameters or minimal run-time configurability. However, only a few works are targeting SDF or MDC pipelined architectures with and without design-time configurability [1]–[4], [9], [10]. In Table I, we list all related NTT architectures in the literature targeting either a pipelined approach or supporting design-time configurability. We also report their NTT architecture method, the largest reported parameter, and the availability of their source code. It is worth mentioning that only [5] and [7] make their source code available. In [5], the authors present design-time configurable iterative NTT architectures. Their architecture takes the polynomial degree, coefficient modulus size, and the number of processing elements as inputs and generates a synthesizable iterative NTT architecture for these parameters. Their architectures use only the DIF NTT approach with N-to-R ordering. Thus, their architectures require costly bit-reversal operations between NTT and INTT. Similarly, since they only employ DIF NTT architecture, their implementation cannot be used for MNTT/MINTT operations. Mu *et al.* presented a parametric NTT architecture for iterative design approach [6], eliminating bit-reversal operation by employing DIT and DIF approaches. They use CT and GS butterfly configurations for NTT and INTT operations. Also, they present formal proof for conflict-free memory access. The works in [7], [8] present iterative NTT architectures with run-time and design-time configurability.

TABLE I
RELATED WORKS IN THE LITERATURE

Work	NTT Arc.	Largest Parameters	Param.?	Open?
[5]	Iterative	$n = 2^{12}, \log_2(q) = 60$	✓	✓
[7]	Iterative	$n = 2^{11}, \log_2(q) = 31$	✓	✓
[6]	Iterative	$n = 2^{12}, \log_2(q) = 60$	✓	
[8]	Iterative	$n = 2^{13}, \log_2(q) = 52$	✓	
[1]	SDF	$n = 2^{12}, \log_2(q) = 60$	✓	
[2]	SDF	$n = 2^{10}, \log_2(q) = 768$		
[3]	MDC	$n = 2^{10}$		
[4]	MDC	$n = 2^{8}, q = 3329$		
[10]	MDC	$n = 2^{14}, \log_2(q) = 52$	✓	
[9]	MDC	$n = 2^{12}, \log_2(q) = 28$	✓	
Our	SDF/MDC	$n = 2^{16}, \log_2(q) = 256$	✓	✓

In [1], the authors present SDF-based NTT architectures for a limited parameter set. Their implementation does not support MNTT/MINTT, which limits its usability. Furthermore, their code is not open-source. PipeZK [2] proposes an SDF architecture as a part of a large hierarchical NTT architecture. Their work targets the ASIC platform and is optimized for fixed parameters. Furthermore, their design does not provide support for MNTT/MINTT. Works in [3], [4] propose high throughput oriented Radix-2 MDC NTT architectures. Their design mainly targets post-quantum cryptographic schemes with small parameter sets (i.e., $n = 256$) and they use fixed parameters. In [9], [10], a parametric architecture for Radix-2 MDC NTT is presented. Their architecture can support different levels of parallelism and uses a streaming permutation network for implementing the complex access pattern of NTT operation. Our work, Proteus, is the only open-source parametric design supporting several parameters for SDF- and MDC-based NTT architectures.

IV. SELECTION OF PROPER NTT CONFIGURATION

The proper NTT configuration has a significant impact on implementation complexity and performance. The configuration selection depends on the target application and platform. As explained in Sec. II, DIT and DIF are two approaches to implement NTT and MNTT efficiently in hardware and software platforms, and various configurations for the order of input and output coefficients exist. Table II lists 8 different DIT and DIF options (denoted as OPi) with different coefficient orders. The table will help us to select the best NTT/INTT configuration for a targeted application or platform.

The first option (OP1) uses DIT NTT with $N \rightarrow R$ ordering and DIF INTT with $R \rightarrow N$ ordering. This approach does

not require any extra bit-reversal operation since the output order of the NTT is the same as the input order of the INTT. However, it requires two different butterfly configurations: CT for DIT and GS for DIF. This approach can also be used for MNTT and MINTT operations by simply changing ω and ω^{-1} to ψ and ψ^{-1} respectively as shown in Fig. 2. The exact alternative option is OP2, which uses DIF NTT and DIT INTT to avoid bit-reversal. When the target application requires only NTT/INTT and not MNTT/MINTT, then OP1 and OP2 are not optimal as they require two butterfly configurations. It is possible to eliminate using two types of butterfly configuration by employing DIT-only or DIF-only NTT/INTT with a different ordering as seen in OP3 and OP4 methods.

Having two different ordering (e.g., $N \rightarrow R$ and $R \rightarrow N$) configurations for NTT and INTT operations complicates implementation and increases resource usage. In Fig. 5, we visualize configurations of six NTT/MNTT approaches ($\text{NTT}^{DIT}_{N\rightarrow R}$, $\text{NTT}^{DIF}_{N\rightarrow R}$, $\text{NTT}^{DIT}_{R\rightarrow N}$, $\text{NTT}^{DIF}_{R\rightarrow N}$, $\text{MNTT}^{DIT}_{N\rightarrow R}$, $\text{MINTT}^{DIF}_{R\rightarrow N}$) for $n = 16$. The white boxes show offset between coefficients while the yellow boxes show ω/ψ powers used in a stage. $\text{NTT}^{DIT}_{N\rightarrow R}$ and $\text{NTT}^{DIT}_{R\rightarrow N}$ use the same butterfly type; however, they use different orderings (as shown in Fig. 5). Due to ordering difference, $\text{NTT}^{DIT}_{N\rightarrow R}$ and $\text{NTT}^{DIT}_{R\rightarrow N}$ have different coefficient offsets in each stage. In SDF and MDC architectures, this will create a huge burden on implementation as the memory depth of each stage depends on the offset between input coefficients. This can be solved either by (i) using memory units large enough for both configurations in each stage or by (ii) making the data flow order configurable using multiplexers. Both approaches will increase the implementation complexity significantly.

The NTT configurations OP5 and OP6 methods in Table II use the bit-reverse operation to eliminate having two different ordering for NTT and INTT operations. Bit-reverse is expensive in hardware, especially for iterative NTT architectures that generate multiple coefficients in each cycle. On the other hand, SDF and MDC architectures generate 1 and 2 coefficients per cycle, respectively. Thus, the bit-reverse operation can easily be implemented almost free of cost by writing the output coefficients into the memory in the correct order.

In [14], the authors show that using the same twiddle factors for both forward and inverse NTTs is possible. This unique optimization requires reordering the input coefficients from $(\boldsymbol{a}_0, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n-2}, \boldsymbol{a}_{n-1})$ to $(\boldsymbol{a}_0, \boldsymbol{a}_{n-1}, \ldots, \boldsymbol{a}_2, \boldsymbol{a}_1)$. The OP7 and OP8 options in Table II show the NTT/INTT configurations that use the same twiddle factors for computing both NTT and INTT by employing bit-reverse (BR) and reorder (RO) operations. Similar to OP5 and OP6, SDF and MDC architectures enable the implementation of BR and RO operations without any extra implementation cost. Proteus can generate hardware for all options listed in Table II.

Reducing memory for twiddle factors: An NTT operation requires $n/2$ different powers of twiddle factor ω, as mentioned in Sec. II-B. Similarly, INTT requires $n/2$ different powers of ω^{-1}. Each NTT/INTT stage uses a part of twiddle factor powers during computations. For example, the first stage of $\text{NTT}^{DIF}_{N\rightarrow R}$ uses all $n/2$ powers of ω, the second stage uses only half of the powers ($n/4$ powers), and so on.

TABLE II
VARIOUS OPTIONS TO PERFORM NTT/INTT

Option	NTT	BR?	RO?	INTT	BR?
OP1	$\text{NTT}^{DIT}_{N\rightarrow R}, w$			$\text{NTT}^{DIF}_{R\rightarrow N}, w^{-1}$	
OP2	$\text{NTT}^{DIF}_{N\rightarrow R}, w$			$\text{NTT}^{DIT}_{R\rightarrow N}, w^{-1}$	
OP3	$\text{NTT}^{DIT}_{N\rightarrow R}, w$			$\text{NTT}^{DIT}_{R\rightarrow N}, w^{-1}$	
OP4	$\text{NTT}^{DIF}_{N\rightarrow R}, w$			$\text{NTT}^{DIF}_{R\rightarrow N}, w^{-1}$	
OP5	$\text{NTT}^{DIT}_{N\rightarrow R}, w$	✓		$\text{NTT}^{DIT}_{N\rightarrow R}, w^{-1}$	✓
OP6	$\text{NTT}^{DIF}_{N\rightarrow R}, w$	✓		$\text{NTT}^{DIF}_{N\rightarrow R}, w^{-1}$	✓
OP7	$\text{NTT}^{DIT}_{N\rightarrow R}, w$	✓	✓	$\text{NTT}^{DIT}_{N\rightarrow R}, w$	✓
OP8	$\text{NTT}^{DIF}_{N\rightarrow R}, w$	✓	✓	$\text{NTT}^{DIF}_{N\rightarrow R}, w$	✓

BR: Bit-reverse, RO: Reorder.

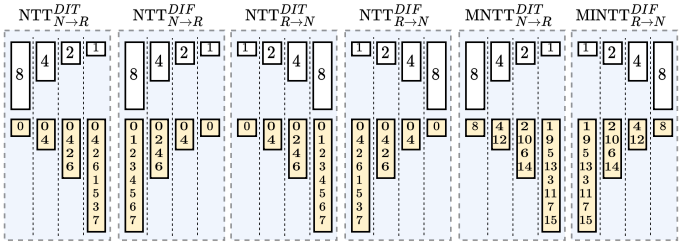

Fig. 5. Offset between coefficients and twiddle factor powers used in each stage of six different NTT/MNTT configurations for $n = 16$.

Therefore, an NTT/INTT implementation should store at least $n = n/2 + n/2$ twiddle factor powers. We used this approach and exploited the mathematical properties of the twiddle factor that $\omega^n \equiv 1 \pmod q$ and $\omega^{n/2} \equiv -1 \pmod q$ to halve the total number of twiddles factors that need to be stored in the form $\omega^{-i} = \omega^n \cdot \omega^{-i} = \omega^{n/2} \cdot \omega^{n/2} \cdot \omega^{-i} = -\omega^{n/2-i}$. This implies that the twiddle factor powers of NTT (ω^i) will be used to calculate the twiddle factor powers of INTT (ω^{-i}). A specific ω^{-i} can be written as $\omega^n \cdot \omega^{-i}$. If we split the term ω^n into $\omega^{n/2} \cdot \omega^{n/2}$, then we can write the formula $\omega^n \cdot \omega^{-i}$ as $\omega^{n/2} \cdot \omega^{n/2-i}$. The property of $\omega^{n/2}$ allows us to write it as -1 which leads to $\omega^{n/2} \cdot \omega^{n/2-i}$ becoming $-\omega^{n/2-i}$. This means that ω^{-i} can be computed by using $-\omega^{n/2-i}$ and this is quite cheap in hardware since it just requires an additional subtraction circuit to get the modular inverse of a twiddle factor power during INTT from a twiddle factor power. In our work, we used this optimization and reduced the required twiddle factor storage by 50% in exchange for $log_2(n)$ additional subtraction units. Note that this also applies to MNTT/MINTT since $\psi^{-i} = -\psi^{n-i}$.

Eliminating the multiplication with n^{-1}: INTT and MINTT require the coefficients of the resultant polynomial to be scaled by n^{-1} in $\mathbb{Z}_q$. Although this scaling operation is straightforward and has a linear time cost, it still requires n extra operations. This extra latency for the scaling operation can skipped by merging the scaling with the post-processing operation in NWC, as shown in Fig. 2. In [13], the authors proposed a technique to replace the multiplication by n^{-1} at the end of INTT with the multiplication by 2^{-1} at the end of each INTT butterfly operation. This is implementation friendly because for an odd prime q, $a/2$ in $\mathbb{Z}_q$ can easily be computed as $(a \gg 1) + a[0] \cdot (\frac{q+1}{2})$. Thus, the extra n multiplication operation can be eliminated using two extra adders in the butterfly unit. In our work, we used this technique to reduce the latency of INTT/MINTT operations.

Bit-reverse and reorder operations: As shown in Table II,

options OP5, OP6, OP7, and OP8 require bit-reverse or reorder operations, which could have high implementation costs in hardware, for iterative NTT architectures in particular. Since SDF-based NTT architectures read one coefficient from input memory and write one coefficient to the output memory per cycle, bit-reverse and reorder operations can be performed costlessly by reading and writing coefficients with proper addressing. The same approach can be applied to the MDC-based NTT architecture as well. Our work used this method to implement bit-reverse and reorder operations without any performance and area costs.

V. The Proposed Architecture

In this section, we explain Proteus's proposed architecture top-down. We explain the required parameters and their effects on the architectural design. Then, we explain SDF and MDC Radix-2 NTT architectures. Finally, we explain low-level arithmetic units, parametric integer multipliers, parametric word-level Montgomery reduction units, and parametric butterfly.

A. Overall design of Proteus

Proteus is an all-in-one solution that generates SDF and MDC Radix-2 NTT/MNTT architectures for FPGA, offering various configurable options (Sec. IV). Proteus takes its parameters before synthesizing the target architecture. Specifically, for SDF and MDC architectures, it takes i) polynomial parameters (polynomial size n and coefficient modulus size $\log_2 q$), ii) type of butterfly unit (CT, GS or unified) that enables NTT configurations OP1 to OP8, iii) type of modular reduction method (parametric Montgomery unit or user-defined add-shift based unit for constant prime). For a given parameter set, it generates a synthesizable NTT architecture.

The performance and implementation complexity of SDF/MDC-based NTT depends on the polynomial-size n and the modulus size $\log_2 q$. The Radix-2 DIF or DIT NTT algorithms perform the NTT of a polynomial in s stages where $s = log_2(n)$ as explained in detail in Sec. III. In SDF and MDC, s determines the number of instantiated butterfly units, affecting the overall design's cost and structure. We propose a design-time flexible architectural design that can configure itself through n. Consider the NTT of a polynomial with $n = 2^{10} = 1024$ coefficients for simplicity. There will be $s = 10$ stages denote as $s_0, s_1, ..., s_9$ to represent the data flow of the Radix-2 NTT. Coefficients of the polynomial will move through the cascaded stages, starting with stage s_0. To develop the automatic architecture generator, Proteus, it is crucial to understand how a change in the parameter n affects the data flow. For example, changing the parameter n from 2^{10} to 2^{11} leads to an increase in the number of NTT stages from 10 to 11. Hence, Proteus must add a new stage in the pipelined NTT architecture when n increases from 2^{10} to 2^{11}. Besides adding the new stage, there will be a change in the data flow, meaning that the output of s_9 now gets forwarded to the new stage s_{10} and the output of s_{10} becomes the new final result.

Besides the number of total stages, the parameter n also changes the total resource utilization. Each stage of the SDF Radix-2 NTT consists of only one butterfly unit (BFU) and one FIFO. Meanwhile, in the MDC Radix-2 NTT, each stage consists of one BFU, two FIFOs, and one commuter switch. In both architectures, each stage processes the whole polynomial of size n in n_s chunks where $n_s = n/2^s$. Therefore, the FIFO depth f_s in each stage depends on the size of each input chunk n_s. In SDF architecture, FIFO depth in stage s will be $n_s/2$. In MDC architecture, the depth of FIFOs is $n_s/4$. The size of coefficient modulus, $\log_2(q)$, determines the size of arithmetic units, data width of FIFO units, and configurations of integer multiplier and reduction units.

B. High-level Architecture

1) Radix-2 NTT in the SDF configuration: Radix-2 NTT in the SDF configuration has one input and output port. If combined with a fully pipelined architecture, this limited I/O bandwidth leads to a data collision. The data collision happens when the computation process of a BFU takes more than one cycle (which is the case in a pipelined BFU). The timing diagram in Fig. 6 explains how a data collision occurs when the BFU has a latency of 2 cycles, and the input polynomial is of size 8 coefficients. At the initial stage of the computation, the first half of the input polynomial (four coefficients 0, 1, 2, 3 in `STAGE_IN`) is sent into the FIFO (`FIFO_IN`). Then, the FIFO delays the first half of the input polynomial by 4 cycles to align it with the second half of the polynomial (four coefficients 4, 5, 6, 7 in `STAGE_IN`). When the second half of the polynomial arrives, it is sent directly to the BFU unit. The BFU unit now receives the first half of the polynomial (0, 1, 2, 3 in `BFU_IN_0`) as the first input and the second half of the polynomial (4, 5, 6, 7 in `BFU_IN_1`) as the second input. After a computation latency of 2 cycles, the first output of the BFU unit (a, b, c, d in `BFU_OUT_0`) proceeds directly to the stage output. The second output of the BFU (e, f, g, h in `BFU_OUT_1`) is sent back into the FIFO to be delayed until the output port of the stage is available again. However, this causes a data collision in `FIFO_IN` because the new input of the stage (two new coefficients 0, 1 in `STAGE_IN`) and the second output of the BFU unit (two coefficients g, h in `BFU_OUT_1`) need to be stored in FIFO. The data collision on the input of the FIFO is shown in red ((g, 0), (h, 1) in `FIFO_IN`) in Fig. 6.

A method to solve the data collision issue is to adjust the data flow of each stage. An in-depth analysis reveals a dependency between the data flow and the size n_s of the input polynomial in combination with the latency l of the BFU unit. Our solution is to denote two different dataflows depending on the input size and the BFU latency. The latency l of the BFU can be either smaller-equal or greater than $n_s/2$. In case where the latency is smaller-equal than the polynomial input size ($l \leq \frac{n_s}{2}$), the dataflow inside the stage needs to be implemented as displayed on the left side of Fig. 8 and in the timing diagram in Fig 7. Again, we will use an example where the BFU has a latency of 2 cycles, and the input polynomial is of size 8 coefficients. At the initial stage of the computation, the first half of the input polynomial (four coefficients 0, 1, 2, 3 in `STAGE_IN`) is sent to the first input of the BFU (`BFU_IN_0`). The BFU unit performs no computations since

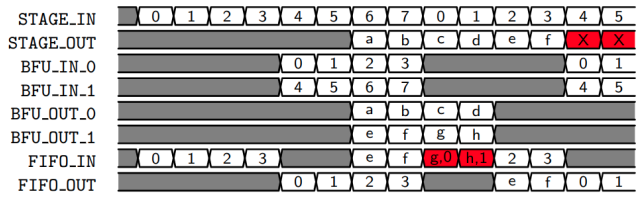

Fig. 6. A scenario within a Radix-2 SDF stage where data collision happens

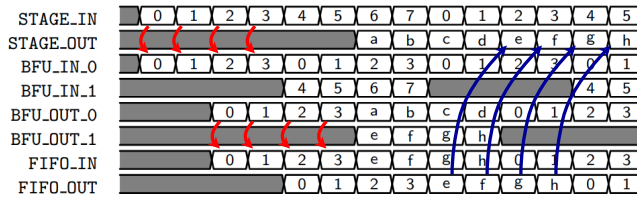

Fig. 7. A scenario within a Radix-2 SDF stage where data collision is avoided.

the second input (BFU_IN_0) is empty. The BFU delays the first half of the polynomial by 2 cycles and passes it to the FIFO (shown with red arrows in Fig. 7). This time, the FIFO delays the first half of the input polynomial by 2 cycles to align it with the second half of the polynomial (four coefficients 4, 5, 6, 7 in STAGE_IN). Now that both halves of the polynomial are aligned, the BFU unit performs the butterfly operation. After a computation latency of 2 cycles, the first output of the BFU unit (a, b, c, d in BFU_OUT_0) proceeds directly to the stage output. The second output of the BFU (e, f, g, h in BFU_OUT_1) is sent as input to the FIFO. Yet, this time, the second output (e, f, g, h) is sent to an additional buffer (registers at the base of blue arrows in Fig. 7) after passing through the FIFO. Finally, the additional buffer passes FIFO outputs to the stage output (shown with blue arrows in Fig. 7). This procedure guarantees a total delay of 4 cycles to buffer (e, f, g, h) until the output port of the stage is available again. This also guarantees no data collision since the first half of the new polynomial inputs (0, 1, 2, 3) is not passed directly to the FIFO. Instead, it first passes through the BFU unit, which delays it by 2 cycles. This modification in the dataflow solves the data collision at the input of the FIFO.

In the case where the latency is greater than the polynomial input size ($l > \frac{n_s}{2}$), the dataflow inside the stage changes as displayed on the right side of Fig. 8. However, in this case, the dataflow is much simpler than the other case where l is smaller-equal $\frac{n_s}{2}$. The first half of the input polynomial (four coefficients 0, 1, 2, 3 in STAGE_IN) is sent into the FIFO (FIFO_IN). Then, the FIFO delays the first half of the input polynomial by 4 cycles to align it with the second half of the polynomial (four coefficients 4, 5, 6, 7 in STAGE_IN). After a computation latency of 2 cycles, the first output of the BFU unit (a, b, c, d in BFU_OUT_0) proceeds directly to the stage output while the second output of the BFU (e, f, g, h in BFU_OUT_1) is sent to a small buffer. This buffer delays (e, f, g, h) until the output port of the stage is available again.

Proteus generates parametric hardware for Radix-2 SDF NTT architectures through a given polynomial size n and a BFU computation latency l. The generated hardware consumes a total number of $\log_2(q) \cdot \sum_{s=0}^{log_2(n)-1} f_s$ bits in terms of total storage. The overall latency of one NTT with SDF architecture is approximately $2n + \log_2(n) \cdot l$.

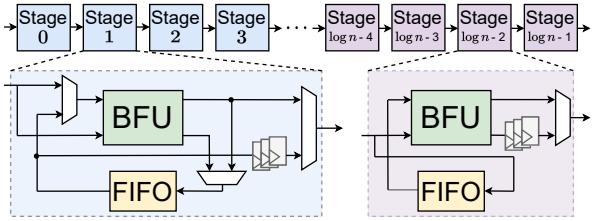

Fig. 8. Overview of our changed Radix-2 SDF configuration and its stages.

2) Radix-2 NTT in the MDC configuration: Radix-2 NTT in the MDC configuration requires two inputs per cycle, which doubles the required bandwidth compared to the Radix-2 SDF. In contrast to a stage of the SDF architecture, a stage s in the MDC architecture consists of one butterfly unit, a switch unit, and two $n_s/4$ deep FIFOs.

In Radix-2 MDC, the inputs and outputs can be fed directly into and out of the stage. The stage input is passed to the butterfly unit first. After computation, the output is either sent to a FIFO or directly into a switch unit. The switch unit swaps two input data depending on a selection signal. This signal is either high or low for $n_s/2$ cycles where n_s is the length of the stage chunk input (see Sec. V-A). In combination with FIFOs before and after the switch, it is possible to mimic the required transformation pattern. The logic of Radix-2 MDC is much simpler when compared to Radix-2 SDF since it does not reuse its FIFOs for input and output storing.

Proteus generates parametric hardware for Radix-2 MDC NTT architectures through a given polynomial size n and a BFU computation latency l. In the generated MDC architecture, the total number of bits consumed by the hardware is around $\log_2(q) \cdot \sum_{s=0}^{log_2(n)-1} f_s$. In contrast, the overall latency of one NTT with MDC architecture is approximately $n + \log_2(n) \cdot l$.

C. Low-level Arithmetic Units

In contrast to the high-level architectural design that depends on the polynomial-size n, the hardware of low-level arithmetic units depends on the coefficient modulus size, which is $\log_2(q)$.

1) Parametric Integer Multiplier: The structure of the integer multiplier is crucial for the entire design since NTT requires multiplication for each butterfly operation per NTT/INTT stage. Xilinx Vivado has an integrated IP generator that can generate integer multiplier circuits, which support inputs of up to 64 bits. Vivado-generated multipliers have high utilization of DSP units, and the recommended number of pipeline stages is high for large input sizes. We designed and implemented a parametric integer multiplier unit that uses a divide-and-conquer approach splitting a multiplication into smaller ones via standard tiling [23] to reduce the requirement of DSP slice. Note that the smaller computations are calculated in parallel to reduce the overall latency. The outputs of the DSPs are accumulated using a parametric carry-save-adder (CSA) tree. The multiplier unit is fully pipelined, and its latency is the sum of the latency of one DSP and CSA tree.

2) Parametric Word-level Montgomery Reduction Unit: Each integer multiplication unit requires an additional modulo reduction to keep the result within the modulo ring of q.

Algorithm 2 Word-level Montgomery Algorithm [15]

Input: d, $q = q_H \cdot 2^w + 1$, w (word size), $L = \lceil \frac{\log_2 q}{w} \rceil$ (iterations)
Output: $c = d \cdot R^{-1} \pmod q$ where $R = 2^{w \cdot L}$

1: $T \leftarrow d$
2: **for** $(i = 0; i < L; i = i + 1)$ **do**
3: $\quad T_H \leftarrow T \gg w$, $T_L \leftarrow T \pmod{2^w}$
4: $\quad T2 \leftarrow -T_L \pmod{2^w}$
5: $\quad cin \leftarrow T2[w-1] \vee T_L[w-1]$
6: $\quad T \leftarrow (q_H \cdot T2) + T_H + cin$
7: **end for**
8: **return** $c = (T \geq q)\ ?\ T - q\ :\ T$

Algorithm 3 Mapping Word-level Montgomery Red. to FPGA

1: $s \leftarrow 24$ or 26 (based on the DSP architecture of target FPGA)
2: **if** $(\log_2(T_H) \leq 47)$ **then**
3: $\quad$ **if** $(\log_2(q_H) \leq s)$ **then**
4: $\quad\quad$ Implement $T \leftarrow (q_H \cdot T2) + T_H + cin$ using one DSP unit.
5: $\quad$ **else**
6: $\quad\quad$ Instantiate $v = \lceil \frac{\log_2(q_H)}{s} \rceil$ DSP units. The first DSP unit implements $(q_H[s-1:0] \cdot T2) + T_H + cin$. The other DSP units implement $(q_H[si+s-1:si] \cdot T2)$, $i \in [1, v-1]$. If 3 or more DSPs are instantiated, DSP outputs are reduced to 2 using one CSA tree; then, the final result is computed using one adder. Otherwise, the final result is computed using one adder.
7: $\quad$ **end if**
8: **else**
9: $\quad$ **if** $(\log_2(q_H) \leq s)$ **then**
10: $\quad\quad$ Implement $r \leftarrow (q_H \cdot T2) + cin$ using one DSP unit.
11: $\quad\quad$ Implement $T \leftarrow r + T_H$ using one adder.
12: $\quad$ **else**
13: $\quad\quad$ Instantiate $v = \lceil \frac{\log_2(q_H)}{s} \rceil$ DSP units. The first DSP unit implements $(q_H[s-1:0] \cdot T2) + cin$. The other DSP units implement $(q_H[si+s-1:si] \cdot T2)$, $i \in [1, v-1]$. DSP outputs and T_H are reduced to 2 using one CSA tree, then the final result is computed using one adder.
14: $\quad$ **end if**
15: **end if**

There are several techniques to perform a modulo reduction. Barrett [24] and Montgomery [25] are two well-known techniques for modular reduction for a generic modulus q. Add-shift-based approaches [14], [26], [27] can also be utilized when a sparse prime such as a Solinas or Mersenne is used as the modulus. There are also lazy reduction methods [28] and techniques targeting modulus of certain form [15], [29]. In this work, we adopted a word-level Montgomery reduction algorithm tailored for NTT-friendly primes, proposed in [15], and mapped it into FPGA efficiently in a parametric design setting. The algorithm divides a modular reduction operation into smaller chunks, and it uses the form of NTT-friendly primes, $q = q_H \cdot 2^w + 1$ where $w \leq \log_2(n)$, to simplify the reduction operation. The word-level Montgomery reduction algorithm for NTT-friendly primes is shown in Algorithm 2.

The algorithm takes $d = a \cdot b$ and prime modulus $q = q_H \cdot 2^w + 1$ as inputs and performs reduction operation in $L = \lceil \log_2(q)/w \rceil$ steps where w-bit (word size) reduction is performed in each step (lines 2-7 in Algorithm 2). Finally, a reduction operation is performed at the end (line 8 in Algorithm 2). The word-level Montgomery algorithm is scalable, enabling efficient utilization of DSP units in FPGA. As shown in line 7 of Algorithm 2, the algorithm performs $T \leftarrow (q_H \cdot T2) + T_H + cin$ in each reduction step which

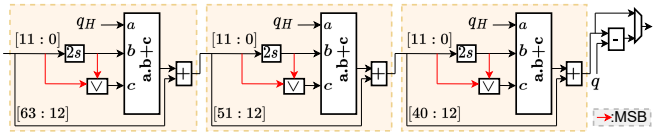

Fig. 9. WL Montgomery reduction circuit for $\log_2 q = 32$ and $w = 12$.

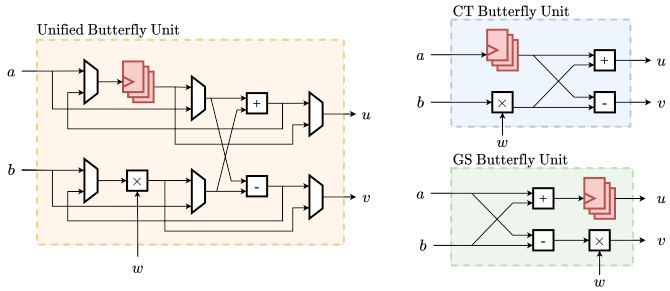

Fig. 10. Overview of CT, GS, and Unified butterfly units used in Proteus

involves one $(\log_2(q) - w)$-bit $\times$ w-bit multiplication, and additions with $(2 \cdot \log_2(q) - i \cdot w)$-bit and 1-bit integers. Based on the parameter selection (e.g., $\log_2(q)$ and w), this operation can be implemented using a single Xilinx DSP unit, which can perform $A \cdot B + C + carry$ for 25/27-bit A, 18-bit B, 48-bit C and 1-bit $carry$. In Algorithm 3, we present a method to map this operation into FPGA using DSP units efficiently. When $\log_2(T_H)$ is less than 48, the addition of T_H and cin can be implemented using DSP without any extra fabric LUTs. If $\log_2(q_H)$ is larger than the DSP input operand size, then it is divided into smaller parts using the divide-and-conquer approach, as explained in Sec. V-C1, and $\log_2(q_H) \cdot T2$ multiplication is implemented using multiple DSPs. Finally, DSP results are accumulated using one CSA tree. When $\log_2(T_H)$ is equal to or greater than 48, adding T_H is also implemented using adders. The proposed mapping algorithm uses $L \cdot \lceil \log_2(q_H)/s \rceil$ DSPs where s is 24 or 26, depending on the FPGA platform.

The proposed parametric modular reduction unit takes the bit-size of modulus $(\log_2(q))$, word size (w), and the number of reduction steps (L) as input and generates the corresponding reduction circuit. Fig. 9 shows the implementation of modular reduction circuit for parameters $\log_2(q) = 32$, $w = 12$ and $L = 3$, where $2s$ and $\vee$ represent two's complement and logical OR operations.

It should be noted that the Algorithm 2 introduces an extra term at the output, R^{-1} where R is $2^{w \cdot L}$. To eliminate this extra constant, either the output of the reduction operation or one of the input operands should be multiplied with R. Since one operand is always constant (ω^i or ψ^i) during NTT/MNTT operation, we multiply all precomputed constants by R before loading them to the hardware. Since this operation is performed offline and once for each parameter, it adds no extra latency.

3) Parametric CT, GS, and Unified Butterfly Units: Two approaches, DIT and DIF, to constructing efficient NTT algorithms require so-called CT and GS butterfly configurations, as described in Sec. II-B. Depending on Proteus's configuration, we require a CT, GS, or both butterflies combined. This reliance leads to the necessity of three different units. We present a different design for a unified butterfly that fits nicely into a pipelined architecture like Proteus, which requires both

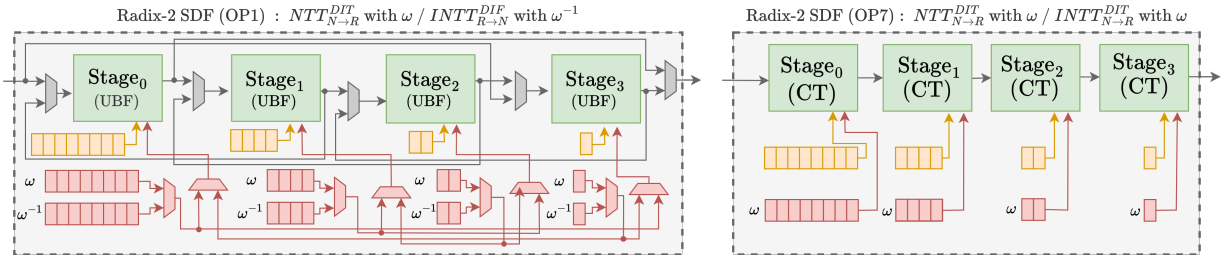

Fig. 11. Radix-2 SDF NTT architecture with OP1 (left) and OP7 (right) for $n = 16$. Yellow and red boxes represent data and twiddle factor memories.

butterfly operations during NTT computation. The dataflow inside our unified butterfly redirects via MUXes to either produce CT or GS outputs as shown on the left side of Fig. 10.

D. Comparison of different NTT Architectures

Proteus can generate two different NTT architectures, Radix-2 SDF and Radix-2 MDC. Each architecture supports 18 configurations described in Sec. IV. We illustrate the flexibility and efficiency of Proteus by comparing two Proteus-generated NTT architectures. The first one is Radix-2 SDF with OP1, which is more oriented towards flexibility, while the other one is Radix-2 SDF with OP7, which is more focused on minimizing resource utilization.

Fig. 11 illustrates both architectures with a polynomial of size $n = 2^4$. The figure shows that the design of OP1 is much more complex than OP7. This complexity is due to the bidirectional data flow that is required to support all configurations in one architecture, such as $\text{NTT}^{DIT}_{N\to R}$, $\text{NTT}^{DIT}_{R\to N}$, $\text{NTT}^{DIF}_{N\to R}$, and $\text{NTT}^{DIF}_{R\to N}$. This requires extra logic to route data and twiddle factors to different stages as needed. It also requires a unified butterfly core that can perform CT and GS. Compared to OP1, OP7 does not support all transformation types since it is more streamlined towards a specific scenario, namely $\text{NTT}^{DIT}_{N\to R}$. This pivoting towards a single transformation type reduces resource utilization significantly regarding LUTs. In addition to this, it also allows for more optimization, such as using a more straightforward butterfly design and NTT twiddle factor re-usage for INTT. This not only halves the required memory but also simplifies routing due to less logic involved.

The required NTT configuration depends on the target application. In the case of ZKPs systems, mostly NTT/INTT is required [2] while FHE and PQC use MNTT/MINTT. If an implementation targets support for several cryptosystems, it would be best for a designer to adopt OP1. If a single scenario is targeted (e.g., server-side FHE computations), then flexibility can be sacrificed for lower implementation complexity by adopting configurations such as OP5, OP6, OP7, or OP8. To that end, Proteus provides designer freedom when selecting an NTT configuration for a particular use case.

VI. Evaluations

This section presents the area and performance results of hardware architectures generated for several parameters and configurations using Proteus. Then, we compare our results with related works in the literature. We coded the architectural units of Proteus using Verilog/SystemVerilog and verified their functionality using RTL simulations. As described in Sec. V-A, the proposed hardware takes polynomial size (n), coefficient modulus size ($\log_2 q$), NTT/MNTT configuration (OP1 to OP8), and modular reduction type for SDF or MDC architectures as inputs, and generates the corresponding hardware. We obtained area and performance results using Xilinx Vivado 2019.1 for Xilinx Virtex-7 XCVX485T FPGA with 150 MHz target frequency and default synthesis/implementation settings. As a proof of concept, we also generated, implemented, and verified all possible NTT configurations for the polynomial sizes 2^4 to 2^{10} and modulus sizes 32-bit and 64-bit using the Montgomery reduction algorithm on an actual Xilinx PYNQ FPGA board. We use Python models to generate test vectors for verification. These models are also included within our GitHub repository.

A. Evaluation of NTT configuration and parameter selection

This section presents the area and performance results of Proteus-generated NTT architectures for various configurations. Proteus provides several options for different configurations and lets the user choose the proper NTT hardware. In Table III, we present area utilization, latency (in clock cycles), and an average latency of 100 operations for SDF and MDC NTT architectures with different configurations (OP1 to OP8) and modular reduction circuits. MDC architecture shows almost $2\times$ better performance than SDF architecture at the expense of slightly larger LUT and DFF utilization. Further, it requires $2\times$ bandwidth compared to SDF. Proteus generates a parametric word-level Montgomery reduction unit for a given parameter set. It also lets the user replace it with a custom modular reduction unit. We evaluated all configurations for two different modular reduction units, (a) a custom add-shift-based reduction unit for a constant modulus and (b) word-level Montgomery reduction unit presented in Sec. V-C2. As shown in Table III, Montgomery reduction uses $1.6\times$ more DSP units with similar LUT and DFF utilization than add-shift-based reduction for a large NTT parameter. However, the add-shift-based reduction unit supports only one modulus, while the Montgomery reduction unit supports a wide range of moduli for a given parameter set.

As explained in Sec. IV, NTT configuration significantly impacts implementation complexity. Table III shows that OP1 and OP2 configurations have the highest LUT and DFF utilization compared to other configurations because they use a unified butterfly configuration. The configurations OP3 and OP4 do not use unified butterfly units; however, they still have high implementation complexity due to different NTT and INTT orderings, as explained in Sec. IV. Hence, OP3 and OP4 configurations outperform only OP1 and OP2 configurations. The OP5, OP6, OP7, and OP8 configurations eliminate

TABLE III
IMPLEMENTATION AND PERFORMANCE RESULTS FOR SDF AND MDC NTT ARCHITECTURES FOR DIFFERENT CONFIGURATIONS

Design	$(n, \log_2(q)) = (2^{12}, 64)$		$(n, \log_2(q)) = (2^{10}, 28)$	
	LUT/FF/DSP/BR	Lat./Avg.*	LUT/FF/DSP/BR	Lat./Avg.*
SDF				
OP1/2[a]	23.6k/11.8k/144/16	8298/4138	8.7k/3.9k/20/2	2118/1035
OP3/4[a]	21.1k/11.8k/144/16	8298/4138	7.8k/4.0k/20/2	2118/1035
OP5/6[a]	17.6k/11.3k/132/16	8293/4138	6.4k/3.7k/18/2	2113/1035
OP7/8[a]	16.6k/10.6k/132/16	8293/4138	6.0k/3.5k/18/2	2113/1035
OP1[a,c]	22.5k/12.7k/144/24	8310/4138	8.7k/4.5k/20/3	2118/1035
OP1/2[b]	26.0k/18.5k/240/16	8370/4138	7.5k/3.2k/40/2	2128/1035
OP3/4[b]	23.6k/18.5k/240/16	8359/4138	7.3k/3.3k/40/2	2128/1035
OP5/6[b]	19.9k/17.5k/220/16	8359/4138	5.7k/3.0k/36/2	2123/1035
OP7/8[b]	18.9k/16.8k/220/16	8382/4138	5.2k/2.8k/36/2	2123/1035
OP1[b,c]	25.0k/19.5k/220/24	8391/4138	7.4k/3.6k/40/3	2128/1035
MDC				
OP1/2[a]	25.7k/15.7k/144/16	4214/2070	9.7k/5.4k/20/2	1114/518
OP3/4[a]	21.8k/12.5k/144/16	4190/2069	8.0k/4.2k/20/2	1114/518
OP5/6[a]	19.7k/11.8k/132/16	4185/2069	7.2k/3.9k/18/2	1090/518
OP7/8[a]	20.2k/11.8k/132/16	4185/2069	7.4k/3.9k/18/2	1090/518
OP1[a,c]	24.9k/15.7k/144/24	4214/2070	9.5k/5.5k/20/3	1114/518
OP1/2[b]	28.1k/22.3k/240/16	4262/2070	10.1k/4.6k/40/2	1124/518
OP3/4[b]	24.3k/19.2k/240/16	4251/2070	7.3k/3.4k/40/2	1124/518
OP5/6[b]	21.9k/18.0k/220/16	4251/2070	6.7k/3.2k/36/2	1100/518
OP7/8[b]	22.5k/18.0k/220/16	4286/2070	6.9k/3.2k/36/2	1100/518
OP1[b,c]	27.5k/22.5k/240/24	4214/2070	9.8k/4.7k/40/3	1124/518

*: Latency/Average latency for 100 operation. [a]: Using add-shift based reduction for constant prime modulus. [b]: Using word-level Montgomery based reduction for variable prime modulus. [c]: Uses NWC technique.

different orderings of NTT and INTT and show better area performance than the OP1, OP2, OP3, and OP4 configurations. Compared to OP1/OP2, OP7/OP8 configurations use up to 31% less LUT. The configurations with the NWC technique also show high resource usage as they use a unified butterfly unit and have different orderings for NTT and INTT. As shown in Table III, they use 50% more BRAMs compared to NTT/INTT configurations since they need to store pre-processing and post-processing constants (see Sec. II-B).

We also visualize the change in the area utilization for different parameters and configurations in Fig. 12 and Fig. 13, that show Proteus can generate different NTT architectures that cover a wide range of parameters, area utilization, and performance, efficiently by just changing a few parameters.

B. Comparison with the literature

In this section, we present the comparisons between Proteus-generated NTT architectures and related works in the literature [1], [2], [5], [8]–[12]. In Table IV, we present resource utilization, performance, architecture and level of parallelism (e.g., number of butterfly units running in parallel) results of Proteus-generated and related works.

Parametric iterative NTT architectures can set the level of parallelism by changing the number of butterfly units. Compared to the low-cost iterative NTT architectures with one butterfly unit, our SDF and MDC NTT architectures perform much better. For parameters $n = 2^{12}$ and $\log_2(q) = 64$, our SDF and MDC NTT architectures with OP7 configuration show $6.1/9.1\times$ and $7.1/10.4\times$ speedup (in terms of cycles), respectively, compared to the low-cost NTT architectures in [5], [8]. Compared to a balanced iterative NTT architecture with eight butterfly cores [5], our SDF and MDC architectures show similar performance while using $11\times$ less BRAM. High-performance iterative NTT architectures [5], [8], [11], [12]

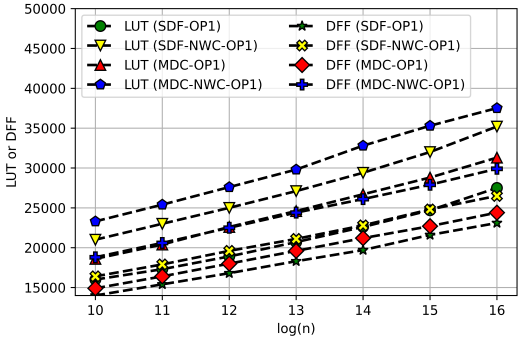

Fig. 12. LUT/DFF utilization of SDF and MDC NTT architectures for $\log_2(q) = 64$ and $n = 2^{10}$ to $n = 2^{16}$ with Montgomery reduction.

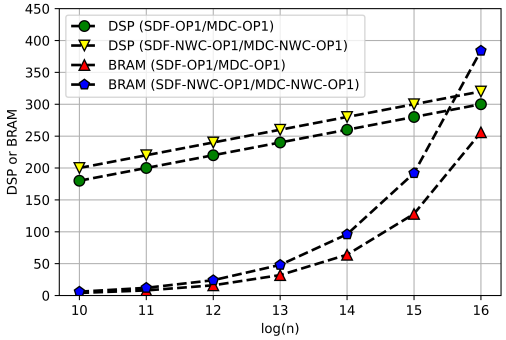

Fig. 13. DSP/BRAM utilization of SDF and MDC NTT architectures for $\log_2(q) = 64$ and $n = 2^{10}$ to $n = 2^{16}$ with Montgomery reduction.

show better performance than our SDF and MDC architectures. However, they use significantly more resources and require high bandwidth. For example, the work in [12] shows $8.5\times$ better performance compared to our MDC architecture for parameter $n = 2^{10}$ at the expense of $14.8/11.2\times$ more LUT, $26\times$ more DSP and $162\times$ more BRAM. Although iterative NTT architectures in [6], [7] are parametric, they target small parameter sets. Hence, they are not included in the comparison table.

There are only a few works in the literature for MDC NTT architectures [3], [4], [9], [10]. In [3], [4], MDC NTT architectures for small parameters without any flexibility are presented. Thus, they are not compared with our work. The works in [9], [10] can change their levels of parallelism and increase their performance at the expense of more resources and bandwidth. Compared to the MDC architecture with four parallel butterfly units [10], our MDC NTT architecture still shows $1.8\times$ better performance while using fewer resources.

PipeNTT [1] is the current state-of-the-art work for parametric SDF Radix-2 NTT architecture. In Table IV, we compare our SDF and MDC NTT architectures with PipeNTT for parameters $n = 2^{12}$, $\log_2(q) = 64$ and $n = 2^{10}$, $\log_2(q) = 28$. Compared to our SDF architecture, PipeNTT performs similarly, using up to $1.75\times$ more DSP and $3\times$ more BRAM. Our MDC architecture outperforms PipeNTT by up to $2.0\times$ while using less DSP and BRAM. PipeNTT requires many BRAM units because their implementation does not use twiddle factor optimization presented in Sec. IV. Thus, they must employ extra BRAMs to store twiddle factors for INTT. Our architecture performs better for DSP utilization than PipeNTT due to our optimized word-level Montgomery modular reduction

TABLE IV
COMPARISON TABLE

Work	Plt.	LUT/FF/DSP/BR	Frq. ‡	Latency* (in μs)	Latency* (in cc)	NTT Arc.	LP †
$n = 2^{10}$, $\log_2(q) = 28 - 32$ **bits**							
[9][a]	V7	95k/104k/640/80	215	0.9/-	-/-	MDC	32
		187k/205k/1280/128	212	0.75/-	-/-	MDC	64
[10][a]	V7	206k/159k/640/80	210	1.1/-	-/-	MDC	32
[12][b]	V7	77k/-/952/325.5	200	0.4/-	80/80	Iter.	32
[11][b]	V7	39.6k/-/224/96	150	1.66/-	250/250	Iter.	32
[1]	V7	3.4k/3.1k/63/6	175	12.1/6	2.1k/-	SDF	1
OP7[1,b]	V7	5.2k/2.8k/36/2	150	14.1/6.9	2.1k/1.0k	SDF	1
OP7[2,b]	V7	6.9k/3.2k/36/2	150	7.3/3.5	1.1k/0.5k	MDC	1
$n = 2^{12}$, $\log_2(q) = 28 - 32$ **bits**							
[8]	A7	2.7k/2.4k/6/8	435	28.2/-	12.3k/-	Iter.	2
		11.8k/8.9k/24/16	379	8.15/-	3.0k/-	Iter.	8
[11][b]	V7	39.6k/-/224/96	150	5.84/-	876/-	Iter.	32
[10][a]	AU	54.1k/56.2k/288/84	250	24.7/-	-/-	MDC	4
OP7[1,b]	V7	6.3k/3.4k/44/8	150	55.8/27.5	8.3k/4.1k	SDF	1
OP7[2,b]	V7	8.4k/4.0k/44/8	150	28.5/13.8	4.2k/2.0k	MDC	1
$n = 2^{12}$, $\log_2(q) = 60 - 64$ **bits**							
[8]	A7	2.6k/2.5k/26/21	144	172/-	24.5k/-	Iter.	1
		90k/77k/832/160	130	6.0/-	782/-	Iter.	32
[5][b]	V7	2.7k/-/31/180	125	198/-	24.7k/-	Iter.	1
		23.2k/-/248/176	125	26/-	3.2k/-	Iter.	8
		99.3k/-/992/176	125	7.77/-	972/-	Iter.	32
[1]	V7	17.0k/11.0k/286/24.5	150	55.2/27.5	8.2k/-	SDF	1
OP7[1,b]	V7	18.9k/16.8k/220/16	150	55.8/27.5	8.3k/4.1k	SDF	1
OP7[2,b]	V7	22.6k/18.0k/220/16	150	28.5/13.8	4.2k/2.0k	MDC	1

*: Latency/Avg. latency for 100 operations. †: Level of parallelism. ‡: Freq. in MHz.
[1]: Radix-2 SDF. [2]: Radix-2 MDC. V7: Virtex-7. A7: Artix-7. AU: Alveo U200.
[a]: Using add-shift based reduction for constant prime modulus.
[b]: Using word-level Montgomery based reduction for variable prime modulus.

unit. When we use their area metric for comparison (LUT + 100×DSP + 300×BRAM in Table 1 of [1]), we show up to 35% better performance for various configurations. Their design provides limited reconfigurability. They only support configuration OP1 for SDF, while we support SDF and MDC architectures for several configurations.

VII. CONCLUSIONS

This paper introduces Proteus, a parametric hardware that can generate bandwidth-efficient SDF and MDC Radix-2 NTT architectures. Proteus supports configurable parameters and incorporates algorithmic and architectural optimizations to reduce memory requirements and improve performance. Experimental results demonstrate that Proteus outperforms the state-of-the-art regarding resource utilization while reducing the overall NTT latency. Proteus's design-time flexibility makes it suitable as a fundamental building block for FHE, PQC, and ZKP systems. Future work includes extending Proteus for different Radix configurations and implementing on-the-fly twiddle factor generation for memory-constrained platforms.

REFERENCES

[1] Z. Ye, R. C. Cheung, and K. Huang, "Pipentt: A pipelined number theoretic transform architecture," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–1, 2022.

[2] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2021.

[3] W. Tan, A. Wang, Y. Lao, X. Zhang, and K. K. Parhi, "Pipelined high-throughput ntt architecture for lattice-based cryptography," in *2021 Asian Hardware Oriented Security and Trust Symposium*, 2021, pp. 1–4.

[4] Z. Ni, A. Khalid, D. Kundi, M. O'Neill, and W. Liu, "Efficient pipelining exploration for A high-performance crystals-kyber accelerator," *IACR Cryptol. ePrint Arch.*, p. 1093, 2022.

[5] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[6] J. Mu, Y. Ren, W. Wang, Y. Hu, S. Chen, C.-H. Chang, J. Fan, J. Ye, Y. Cao, H. Li, and X. Li, "Scalable and conflict-free ntt hardware accelerator design: Methodology, proof and implementation," *IEEE Trans. on Computer-Aided Design of Int. Cir. and Sys.*, pp. 1–1, 2022.

[7] K. Derya, A. C. Mert, E. Öztürk, and E. Savaş, "Coha-ntt: A configurable hardware accelerator for ntt-based polynomial multiplication," *Microprocessors and Microsystems*, vol. 89, p. 104451, 2022.

[8] X. Hu, J. Tian, M. Li, and Z. Wang, "Ac-pm: An area-efficient and configurable polynomial multiplier for lattice based cryptography," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14, 2022.

[9] T. Ye, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Fpga acceleration of number theoretic transform," in *High Performance Computing*. Springer International Publishing, 2021, pp. 98–117.

[10] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: A framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. CF '22, 2022, p. 30–39.

[11] A. C. Mert, E. Öztürk, and E. Savaş, "Fpga implementation of a run-time configurable ntt-based polynomial multiplication hardware," *Microprocessors and Microsystems*, vol. 78, p. 103219, 2020.

[12] ——, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, pp. 353–362, 2019.

[13] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt," *IACR Transactions on CHES*, vol. 2020, no. 2, p. 49–72, 3 2020.

[14] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library." Springer, 2016, pp. 169–186.

[15] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *2019 22nd Euromicro Conf. on Digital System Design*, 2019, pp. 253–260.

[16] M. Scott, "A note on the implementation of the number theoretic transform," in *Cryptography and Coding - 16th IMA International Conference, IMACC 2017*. Springer, 2017.

[17] F. Winkler, *Polynomial algorithms in computer algebra*. Springer Science & Business Media, 1996.

[18] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," Cryptology ePrint Archive, Paper 2016/504, 2016.

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

[20] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers," in *Progress in Cryptology – LATINCRYPT 2015*, 2015, pp. 346–365.

[21] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[22] R. Geelen, M. Van Beirendonck, H. V. Pereira, B. Huffman, T. McAuley, B. Selfridge, D. Wagner, G. Dimou, I. Verbauwhede, F. Vercauteren *et al.*, "Basalisc: Flexible asynchronous hardware accelerator for fully homomorphic encryption," *arXiv preprint arXiv:2205.14017*, 2022.

[23] D. B. Roy, D. Mukhopadhyay, M. Izumi, and J. Takahashi, "Tile before multiplication: An efficient strategy to optimize dsp multiplier for accelerating prime field ecc for nist curves."

[24] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO '86*, vol. 263, 1986, pp. 311–323.

[25] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[26] Z. Liu, H. Seo, S. Sinha Roy, J. Großschädl, H. Kim, and I. Verbauwhede, "Efficient ring-lwe encryption on 8-bit avr processors." Springer, 2015, pp. 663–682.

[27] C. P. Renteria-Mejia and J. Velasco-Medina, "High-throughput ring-lwe cryptoprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2332–2345, 2017.

[28] S. Streit and F. De Santis, "Post-quantum key exchange on armv8-a: A new hope for neon made simple," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1651–1662, 2017.

[29] F. Yaman, A. C. Mert, E. Öztürk, and E. Savaş, "A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme," in *2021 DATE*. IEEE, 2021, pp. 1020–1025.