PROTEUS: A Tool to generate pipelined Number Theoretic Transform Architectures for FHE and ZKP applications

FLORIAN HIRNER, Graz University of Technology, Austria
AHMET CAN MERT, Graz University of Technology, Austria
SUJOY SINHA ROY, Graz University of Technology, Austria

Emerging cryptographic algorithms such as fully homomorphic encryption (FHE) and zero-knowledge proof (ZKP) perform arithmetic involving very large polynomials. One fundamental and time-consuming polynomial operation is the Number theoretic transform (NTT) which is a generalization of the fast Fourier transform. Hardware platforms such as FPGAs could be used to accelerate the NTTs in FHE and ZKP protocols. One major problem is that the FHE and ZKP protocols require different parameter sets, e.g., polynomial degree and coefficient size, depending on their applications. Therefore, a basic research question is: How to design scalable hardware architectures for accelerating NTTs in the FHE and ZKP protocols?

In this paper, we present 'PROTEUS', an open-source and parametric tool that generates synthesizable bandwidth-efficient NTT architectures for user-specified parameter sets. The architectures can be tuned to utilize different memory bandwidths and parameters which is a very important design requirement in both FHE and ZKP protocols. The generated NTT architectures show a significant performance speedup compared to similar NTT architectures on FPGA. Further comparisons with state-of-the-art show a reduction of up to 23% and 35% in terms of DSP and BRAM utilization.

CCS Concepts: • **Computer systems organization** → *Reconfigurable computing*; • **Hardware** → **Hardware accelerators**.

Additional Key Words and Phrases: Parametric, Pipelined, NTT, FHE, ZKP

Authors' addresses: Florian Hirner, Graz University of Technology, Graz, Austria, florian.hirner@iaik.tugraz.at; Ahmet Can Mert, Graz University of Technology, Graz, Austria, ahmet.mert@iaik.tugraz.at; Sujoy Sinha Roy, Graz University of Technology, Graz, Austria, sujoy.sinharoy@iaik.tugraz.at.

1 INTRODUCTION

Advanced cryptographic protocols such as fully homomorphic encryption (FHE) and zero-knowledge proof (ZKP) have tremendous potentials in realizing new privacy-preserving applications that cannot otherwise be developed using classical cryptographic techniques. FHE [7] enables computations on the encrypted data without performing any decryption. Privacy-preserving outsourcing of computation to untrusted entities, such as a cloud, is a popular example use case of FHE. ZKP [9] systems enable a prover to prove to a verifier that a given statement is true without revealing any additional information.

Both FHE and ZKP constructions use polynomial evaluations such as the Number Theoretic Transform (NTT). The computational cost of NTT grows asymptotically as $O(n \log n)$ with the polynomial size n. Typically the polynomial size could be as large as 2^{16} in FHE, and 2^{16} to 2^{24} in ZKP. Processing such large polynomials take time and computational resources. Although there are efforts for algorithmic optimizations and improved software implementations to make FHE/ZKP more practical, they still require better performance improvements for compute-intensive operations like NTT. FPGA-based hardware acceleration has emerged as a strong candidate to improve the performance of NTT. To improve the speed of NTT in FHE and ZKP applications, several hardware accelerators have been proposed in the literature [5, 8, 12, 14–16, 18, 19, 29–31, 33]. However, most of these works are optimized for a specific set of parameters or target performance fixed during the design of the hardware architectures [19, 26]. Hence, they do not support more than one FHE or ZKP application and are not scalable.

Flexibility to support different parameter sets is an important requirement in both FHE and ZKP applications as their parameter sets change with the underlying computational constraints. Furthermore, FHE and ZKP constructions are still developing and there is little or less standardization in terms of parameters. Designing flexible hardware for NTT is a non-trivial task due to the complex memory access and control patterns of NTT. Some hardware accelerators [5, 16] provide run-time flexibility by supporting a limited set of specific parameters but increase the hardware cost enormously. Thus, design-time flexible NTT accelerators with a variety of parameter support have crucial importance for FHE and ZKP applications.

In this paper, we present 'PROTEUS' a tool that generates bandwidth-efficient and pipelined NTT architectures for developer specified parameter sets namely the polynomial size and modulus size. PROTEUS generated hardware architectures are design-time flexible and give optimal resource utilizations on FPGA platforms. The main impact of PROTEUS is that it saves precious design time and effort of a developer by generating optimal and synthesizable NTT architecture description in Verilog and SystemVerilog. The key contributions of this works are listed as follows:

- We analyze different types of NTTs and develop a framework, PROTEUS, that can generate bandwidth-efficient single-path delay feedback (SDF) and multi-path delay commutator (MDC) architectures for Radix-2 NTT.
- The proposed framework supports design-time flexibility in generating NTT hardware for different polynomial and coefficient sizes. This increases the adaptability for different FHE and ZKP use cases.
- We design PROTEUS in such a way that it can either be used as a standalone NTT unit for bandwidth-critical systems or as a basic building block inside a system targeting very large polynomial size. When a low-latency is desired for the NTT of a large polynomial, a divide-and-conquer approach can be used to decompose the computation into smaller NTTs, which could be instantiated as PROTEUS-generated SDF or MDC architectures.
- We design and implement efficient and parametric low-level arithmetic units of NTT such as integer multiplier, modular reduction unit and compact butterfly units. Specifically, we

adopted word-level Montgomery reduction for NTT-friendly primes approach [15] and proposed an algorithm to map it into FPGA using DSP units efficiently.
- We make our source code available at https://github.com/florianhirner/proteus. It should be noted that there are only few open-source NTT hardware in the literature. We believe our open-source parametric design will help researchers in this field.

To the best of our knowledge, PROTEUS is the first fully-parameterized and open-source SDF/MDC-based NTT architecture in the literature. The organization of the paper is as follows. In Sec. 2, we present preliminary information necessary for understanding of the paper. Sec. 3 introduces SDF-based and MDC-based NTT architectures. Sec. 4 presents an analysis of different NTT configurations. Sec. 5 presents the architecture of the proposed hardware generator. In Sec. 6 we present area and performance evaluation results and Sec. 7 concludes the paper.

2 PRELIMINARIES

In this section, NTT relevant notations and mathematical background are explained briefly.

2.1 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 irreducible 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 $\boldsymbol{a} \in R_q$ 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.

2.2 Number Theoretic Transformation (NTT) for FHE and ZKP

The NTT is used as a fundamental building block in FHE and ZKP protocols. NTT is a generalization of the classical Fast Fourier Transform (FFT) over $\mathbb{Z}_q$ where q is a prime satisfying $q \equiv 1 \pmod{n}$. 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 $0 \leq i < n$. The NTT uses the constant ω 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 $\boldsymbol{a}_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{\boldsymbol{a}}_i \cdot \omega^{ij} \pmod{q}$ for $0 \leq i < n$.

There are two approaches [24] 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 configuration and the DIF approach uses the Gentleman-Sande (GS) butterfly. For a given input coefficient pair (a, b) and the twiddle factor ω, the Cooley-Tukey and Gentleman-Sande butterflies output the coefficient pairs $\{a + b \cdot \omega, a - b \cdot \omega\}$ and $\{a + b, (a - b) \cdot \omega\}$ respectively.

In the NTT domain polynomial multiplication is a simple coefficient-wise operation [27]. An NTT-based polynomial multiplication 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 almost by a factor of two as only n-point NTT/INTT are required instead of $2n$-point NTT/INTT. NWC requires existence of a $2n$th root of the unity, say $\psi \in \mathbb{Z}_q$ which is possible only when

Algorithm 1 DIF NTT with GS Butterfly [11]

```
Input: a ∈ R_q (in normal order)
Input: ω (n/2 powers of ω in normal order)
Output: â ← NTT(a) ∈ R_q (in BR order)
1:  m ← 1
2:  for (y = n; y > 1; y = y/2) do
3:     s = y/2
4:     for (k = 0; k < m; k = k + 1) do
5:        j1 ← k · y, j2 ← j1 + s − 1, j3 ← 0
6:        for (j = j1; j ≤ j2; j = j + 1) do
7:           w ← ω[j3], u ← a[j], v ← a[j + s]
8:           a[j] ← (u + v) (mod q)
9:           a[j + s] ← (u − v) · w (mod q)
10:          j3 ← j3 + m
11:       end for
12:       m ← 2 · m, y ← s
13:    end for
14: end for
15: return a
```

Algorithm 2 DIT MNTT with CT Butterfly [11]

```
Input: a ∈ R_q (in normal order)
Input: ψ_rev (n powers of ψ in BR order)
Output: â ← MNTT(a) ∈ R_q (in BR order)
1:  t ← n
2:  for (m = 1; m < n; m = 2 · m) do
3:     t ← t/2
4:     for (i = 0; i < m; i = i + 1) do
5:        j1 ← 2 · i · t, j2 ← j1 + t − 1
6:        w ← ψ_rev[m + i]
7:        for (j = j1; j ≤ j2; j = j + 1) do
8:           u ← a[j]
9:           v ← a[j + t] · w (mod q)
10:          a[j] ← (u + v) (mod q)
11:          a[j + t] ← (u − v) (mod q)
12:       end for
13:    end for
14: end for
15: return a
```

$q \equiv 1 \pmod{2n}$. 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 $\boldsymbol{c}' = \mathrm{INTT}\big(\mathrm{NTT}(\boldsymbol{a}') \odot \mathrm{NTT}(\boldsymbol{b}')\big)$. 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 operations, respectively [20, 23]. This requires using the DIT approach for NTT with ψ and the DIF approach for INTT with ψ^{-1}. We refer to these new NTT and INTT as merged NTT (MNTT) and merged INTT (MINTT), respectively, for the rest of the paper. Algorithms for Radix-2 Iterative DIF-based NTT and Radix-2 Iterative DIT-based MNTT are given in Algorithm 1 and Algorithm 2, respectively. It should be noted that FHE constructions use MNTT/MINTT while ZKP schemes use NTT/INTT. PROTEUS provides support for both NTT/INTT and MNTT/MINTT.

Order of input and output coefficients: In-place NTT and 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 [3] for DIF and DIT approaches such as normal to bit-reversed order ($N \rightarrow R$) and bit-reversed to normal order ($R \rightarrow N$). 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 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, $\mathrm{NTT}^{DIT}_{N \rightarrow R}, \omega$ represents a DIT NTT that takes input polynomial in the normal order and generates the output polynomial in

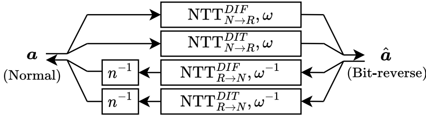


Fig. 1. Configurations for NTT/INTT

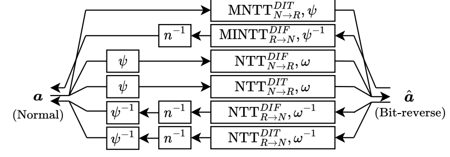


Fig. 2. Configurations for MNTT/MINTT

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. It is possible to use only N-to-N or R-to-R orders without mixing R with N. The disadvantage is that such an approach requires extra memory.

3 SDF AND MDC ARCHITECTURES FOR NTT

In hardware, there are various ways of designing architectures for DIT and DIF NTTs. Choosing an appropriate architecture depends on the platform and application constraints. The n-point iterative NTTs in Algorithm 1 and 2 have $\log_2(n)$ stages where each stage performs $\frac{n}{2}$ butterfly operations (steps 7-9 of Algorithm 1 and steps 8-11 of Algorithm 2). An iterative NTT implementation may use one or several butterfly units for computing the butterfly operations in all stages. Therefore, the iterative NTT is able to scale the latency. At the same time, an iterative NTT requires high bandwidth on-chip memory (proportional to the number of butterfly cores) and has high implementation complexity when the polynomials are of large degrees [14, 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 very large when the polynomial degree is large [6]. Single-path Delay Feedback (SDF) and Multi-path Delay Commutator (MDC) architectures, also referred to as pipelined architectures, use only one butterfly unit for each NTT stage. Hence, they instantiate only $\log_2(n)$ butterfly cores for computing n-pt NTTs as there are $\log_2(n)$ stages. Each butterfly core is exclusive to a specific NTT-stage and it performs $n/2$ butterfly operations of the corresponding stage. The pipelined architectures provide comparable throughput and performance with low bandwidth requirements [33]. In this work, we target SDF and MDC Radix-2 NTT implementations.

3.1 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, therefore $\log_2(n)$ in total. Each butterfly unit is coupled with one memory for temporarily storing several coefficients.

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 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 that requires coefficients separated by an offset of just 1. Fig. 3 shows a high-level view of an SDF-NTT architecture.

In order to provide inputs to the butterfly units with proper offset, the SDF architecture couples each butterfly unit with a memory. The memory works as a temporary storage. Let us consider

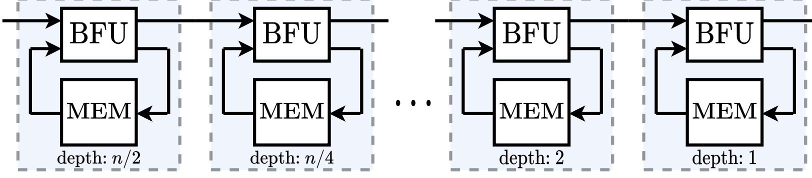


Fig. 3. High-level view of the Radix-2 SDF architecture. BFU and MEM stand for the butterfly and memory respectively.

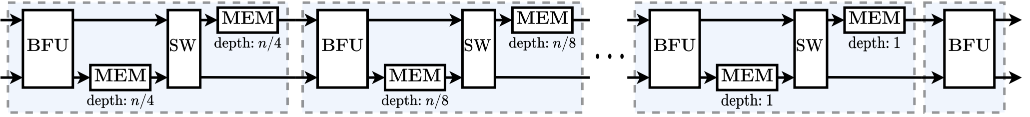


Fig. 4. High-level view of the Radix-2 MDC architecture. BUF, MEM and SW stand for the butterfly, memory, and switch respectively.

a 256-pt DIF NTT as a example. 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 memory one-by-one, and sends the coefficient pairs to the butterfly unit during the next 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.

A bottleneck of the aforementioned method is that the data is reversed during the computation. Meaning that the input data is in the normal order while the output data is in the bit-reversed order. Hence, a bit reversal at the end of the final pipeline stage is needed to change the output coefficients to to the normal order. However, the bit-reversal can be eliminated selecting the next INTT appropriately as described in Sec. 2.

3.2 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. Note that an MDC architecture performs the same butterfly operations as the SDF. However, MDC requires two smaller memory blocks per stage instead of one unlike SDF. The two memory blocks are used to prepare its output (i.e., reorder) for the next NTT stage.

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. Fig. 4 shows a high-level view of an MDC NTT architecture. Each stage is coupled with two memories, M_0 and M_1, of depth d, as shown in Fig. 4. The depth d of the memory units depends on the size of the input

polynomial n and the stage. It can be calculated as $d = n/(4 \cdot (s+1))$ 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 (MEM blocks in Fig. 4) and one switch. The switch can change the order of two coefficients depending on the required offset pattern in the next stage.

3.3 NTT for very large-degree polynomials using SDF/MDC architectures

Implementing NTT of a large-degree polynomial in hardware is not an easy task. It is even more challenging when high performance is targeted. As explained in Sec. 3, 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. SDF and MDC architectures for Radix-2 NTT achieve comparable performance with low bandwidth and area requirements. However, their performance is limited by the configuration (e.g., they use one butterfly unit for each NTT stage).

The hierarchical NTT approach divides a large-degree NTT operation into multiple smaller-degree NTTs [14, 33]. Specifically, it first reshapes a polynomial as a matrix. Then it performs NTT on columns of the matrix, transposes the matrix, and finally again performs NTT on columns of the matrix. The separation into smaller NTT operations makes it easier to implement and parallelize due to the smaller NTT size. An SDF and MDC-based architecture can be used to implement these smaller NTT operations which require low bandwidth. Furthermore, it is possible to instantiate multiple SDF/MDC architectures to perform multiple smaller-sized NTTs at the same time [33].

4 SELECTION OF PROPER NTT CONFIGURATION

Selection of the proper NTT configuration has a significant impact on the implementation complexity and performance. The configuration selection depends on the target application and the digital platform. As already explained in Sec. 2, DIT and DIF are two approaches to implement NTT and MNTT efficiently in hardware and software platforms, and there are various configurations for the order of input and output coefficients. In Table 1, we list 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 $N \rightarrow R$ 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 types of butterfly configurations namely, Cooley-Tukey for DIT and Gentleman-Sandy 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 that 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 types of 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 ($\text{NTT}^{DIT}_{N\rightarrow R}$, $\text{NTT}^{DIF}_{N\rightarrow R}$, $\text{NTT}^{DIT}_{R\rightarrow N}$, $\text{NTT}^{DIT}_{R\rightarrow N}$, $\text{MNTT}^{DIT}_{N\rightarrow R}$, $\text{MINTT}^{DIF}_{R\rightarrow N}$) for $n = 16$. The white boxes show offset between coefficients while the yellow boxes show root of unity 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

Option	Forward NTT	Bit Reverse?	Reordering?	Inverse NTT	Bit Reverse?
OP1	$\text{NTT}^{DIT}_{N\to R}$, w			$\text{NTT}^{DIF}_{R\to N}$, w^{-1}	
OP2	$\text{NTT}^{DIF}_{N\to R}$, w			$\text{NTT}^{DIT}_{R\to N}$, w^{-1}	
OP3	$\text{NTT}^{DIT}_{N\to R}$, w			$\text{NTT}^{DIT}_{R\to N}$, w^{-1}	
OP4	$\text{NTT}^{DIF}_{N\to R}$, w			$\text{NTT}^{DIF}_{R\to N}$, w^{-1}	
OP5	$\text{NTT}^{DIT}_{N\to R}$, w	✓		$\text{NTT}^{DIT}_{N\to R}$, w^{-1}	✓
OP6	$\text{NTT}^{DIF}_{N\to R}$, w	✓		$\text{NTT}^{DIF}_{N\to R}$, w^{-1}	✓
OP7	$\text{NTT}^{DIT}_{N\to R}$, w	✓	✓	$\to \text{NTT}^{DIT}_{N\to R}$, w	✓
OP8	$\text{NTT}^{DIF}_{N\to R}$, w	✓	✓	$\to \text{NTT}^{DIF}_{R\to N}$, w	✓

BR: Bit-reverse, RO: Reorder.

Table 1. Various options to perform NTT/INTT

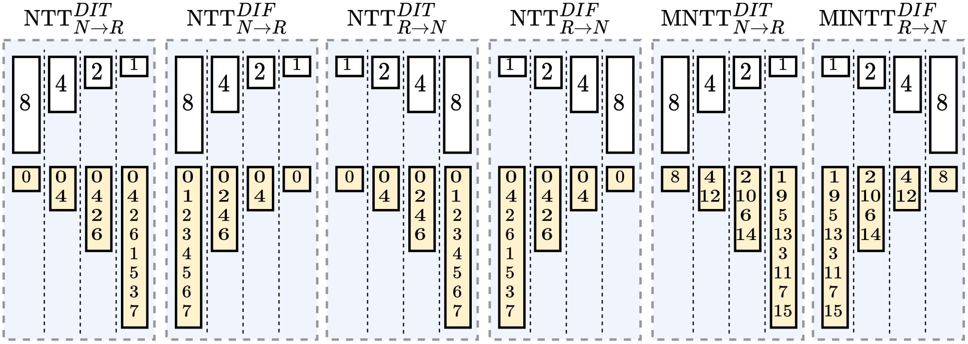


Fig. 5. Offset between coefficients and twiddle factor powers used in each stage of six different NTT/MNTT configurations for $n = 16$

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 1 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 [4], the authors show that it is possible to use the same twiddle factors for both forward and inverse NTTs. This special optimization requires reordering the input coefficients as shown in Eqn. 1. The OP7 and OP8 options in Table 1 show the NTT/INTT configurations that use the same twiddle factors for computing both NTT and INTT by employing bit-reverse 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. In this work, our design can generate hardware for all 8 options listed in Table 1.

$$(\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_{n-2}, \boldsymbol{a}_{n-1}) \rightarrow (\boldsymbol{a}_0, \boldsymbol{a}_{n-1}, \boldsymbol{a}_{n-2}, \ldots, \boldsymbol{a}_2, \boldsymbol{a}_1) \quad (1)$$

4.1 Reducing Memory Requirement for Twiddle Factors

An NTT operation requires $n/2$ different powers of twiddle factor ω, as mentioned in Sec. 2.2. 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. Therefore, an NTT/INTT implementation should store at least $n = n/2 + n/2$ twiddle factor powers. Even some works store more twiddle factors (e.g., they store the same twiddle factor several times in different memory locations) to simplify the twiddle factor access pattern during computations [13]. Works like [13, 31] store forward and inverse twiddle factors into two sets of ROMs and switch between them depending on the operation. However, this approach takes quite a toll on memory utilization since the number of twiddle factors doubles with each degree of the polynomial-size n.

We 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, as shown in Eqn. 2. Implying that the twiddle factor powers of NTT (ω^i) will be used to calculate the twiddle factor powers of INTT (ω^{-i}).

$$\omega^{-i} = \omega^n \cdot \omega^{-i} = \omega^{n/2} \cdot \omega^{n/2} \cdot \omega^{-i} = -\omega^{n/2-i} \quad (2)$$

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 is as -1 which leads to $\omega^{n/2} \cdot \omega^{n/2-i}$ becoming $-\omega^{n/2-i}$. Meaning 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 is also possible for MNTT/MINTT implementations since $\psi^{-i} = -\psi^{n-i}$.

4.2 Eliminating the multiplication with n^{-1} at the end of INTT

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 to implement and has a linear time cost, it still requires n extra modular multiplications. 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 [32], 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 and MINTT operations.

4.3 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 very limited run-time configurability. However, there are only a few works targeting SDF or MDC pipelined architectures with and without design-time configurability [19, 26, 29–31, 33]. In Table 2, we list

Work	NTT Architecture	Largest Reported Parameters	Parametric?	Open-source?
[12]	Iterative	up to $n = 2^{12}, \log_2(q) = 60$	✓	✓
[5]	Iterative	up to $n = 2^{11}, \log_2(q) = 31$	✓	✓
[18]	Iterative	up to $n = 2^{12}, \log_2(q) = 60$	✓	
[8]	Iterative	up to $n = 2^{13}, \log_2(q) = 52$	✓	
[31]	SDF	up to $n = 2^{12}, \log_2(q) = 60$	✓	
[33]	SDF	up to $n = 2^{10}, \log_2(q) = 768$		
[26]	MDC	$n = 2^{10}$		
[19]	MDC	$n = 2^8, q = 3329$		
[29]	MDC	up to $n = 2^{14}, \log_2(q) = 52$	✓	
[30]	MDC	$n = 2^{12}, \log_2(q) = 28$	✓	
Our*	SDF/MDC	up to $n = 2^{16}, \log_2(q) = 256$	✓	✓

* For larger polynomial degrees, a hierarchical NTT (Sec. 3.3) could use several SDF/MDC blocks.

Table 2. Related works in the literature

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 [12] and [5] make their source code available.

In [12, 13], 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 DIF NTT approach with N-to-R ordering. Thus, their architectures require costly bit-reversal operation 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 [18], which eliminates bit-reversal operation by employing both DIT and DIF approaches. They use CT and GS butterfly configurations for NTT and INTT operations, respectively. Also, they present formal proof for conflict-free memory access. The works in [5, 8] present iterative NTT architectures with both run-time and design-time configurability. In [31], the authors present SDF NTT architectures for multiple parameters. However, their implementation does not support MNTT/MINTT and their code is not open-source. PipeZK [33] proposes an SDF architecture as a part of a large hierarchical NTT architecture. Their work target the ASIC platform and is optimized for fixed parameters. Works in [19, 26] propose high throughput oriented Radix-2 MDC NTT architectures. Their design mainly targets post-quantum cryptographic schemes with small parameter sets and they use fixed parameters. In [29, 30], a parametric architecture for Radix-2 MDC NTT is presented. Their architecture can support different levels of parallelism and uses streaming permutation network for implementing the complex access pattern of NTT operation.

5 THE PROPOSED FRAMEWORK

In this section, we explain the proposed tool/framework in a top-down fashion, starting with the design of the parameter-flexible hardware generator tool. Then, we explain SDF and MDC Radix-2 NTT architectures. Finally, we explain low-level arithmetic units, parametric integer multiplier, parametric word-level Montgomery reduction unit and parametric butterfly circuits.

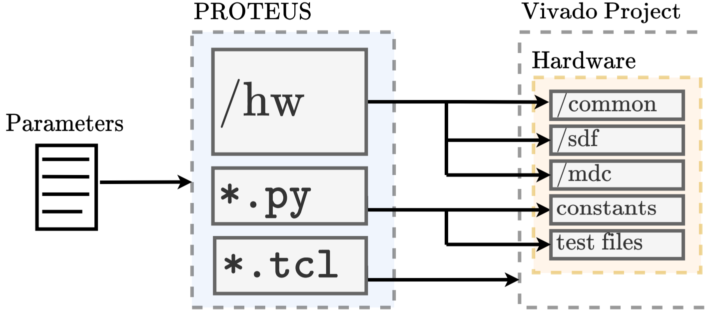


Fig. 6. The proposed framework PROTEUS

5.1 PROTEUS

PROTEUS is a parametric all-in-one solution to generate SDF and MDC Radix-2 NTT/MNTT architectures for FPGA. We provide the possibility to choose from a wide variety of configurations, as described in Sec. 4, to fit possible requirements. A high-level view of PROTEUS is shown in Fig. 6. The PROTEUS has three main components, (i) parametric hardware modules written in Verilog/SystemVerilog, (ii) a Python script, and (iii) a TCL script. First, the tool takes parameters, polynomial degree, coefficient modulus size (or coefficient modulus), the desired architecture type (SDF or MDC), and the desired operation type (NTT/INTT or MNTT/MINTT) as inputs. Then, it calls a Python script that generates the required constants for NTT/MNTT operation such as ROM files containing powers of $n/2n$-th root of unity. It also generates a corresponding parameter file that contains parameter definitions such as polynomial degree, number of butterfly stages, and coefficient modulus size. These parameters dictate the configuration of the hardware design of every unit within the architecture. Finally, a TCL script is called to trigger the creation of a corresponding design in the Xilinx Vivado tool. Further, we also provide the possibility to exchange our units with self-made ones by placing them inside the source folder.

The performance and implementation complexity of an NTT depends on the polynomial size n. As explained in Sec. 3, the Radix-2 DIF or DIT NTT algorithms perform the NTT of a polynomial in s stages where $s = log_2(n)$. In SDF and MDC architectures, s determines the number of instantiated butterfly units and affects the cost and structure of the overall design. We propose a design-time flexible architectural design that can configure itself through n.

For simplicity, consider the NTT of a polynomial with $n = 2^{10} = 1024$ coefficients. There will be $s = 10$ stages in the NTT and let us denote the stages 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 stages in a cascaded manner 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, a change of 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 will need to 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. Whereas, in the MDC Radix-2 NTT, each stage consists of one BFU, two FIFOs, and one commuter switch. In both types of 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.

5.2 High-level Architecture

5.2.1 Radix-2 NTT in the SDF configuration. Radix-2 NTT in the SDF configuration has one input and output port. This limited I/O bandwidth leads to a data collision if combined with a fully pipelined architecture. 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. 7 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. Yet 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. 7.

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 to 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. 9 and in the timing diagram in Fig 8. 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`). Note that the BFU unit does not perform any computations since the second input (`BFU_IN_0`) is empty. The BFU just delays the first half of the polynomial

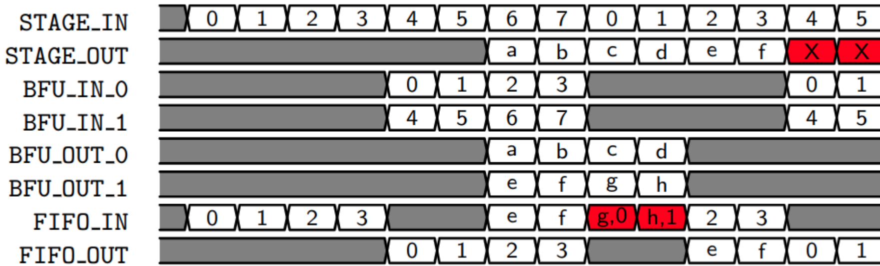


Fig. 7. A scenario within a Radix-2 SDF stage where data collision happens in `FIFO_IN`

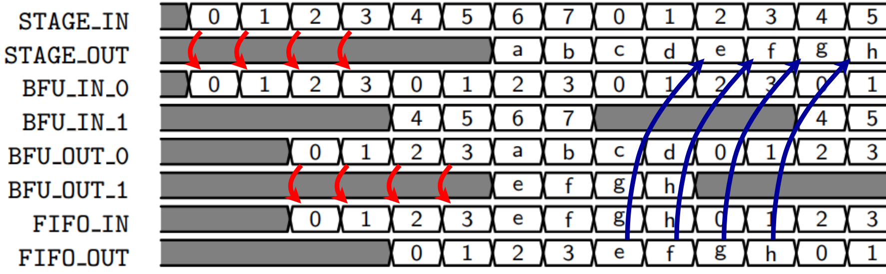


Fig. 8. A scenario within a Radix-2 SDF stage where data collision is avoided

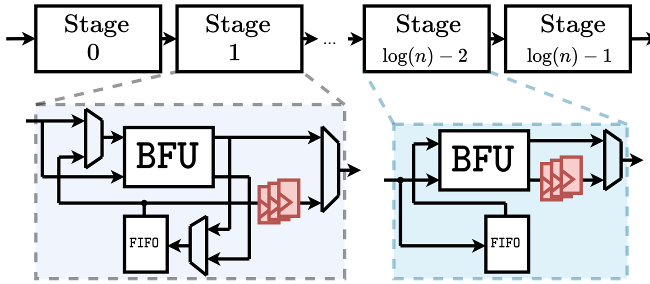


Fig. 9. Overview of our changed Radix-2 SDF configuration and its stages

by 2 cycles and passes it to the FIFO (shown with red arrows in Fig. 8). 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 (red-colored registers in Fig.9) after passing through the FIFO. Finally, the additional buffer passes FIFO outputs to the stage output (shown with blue arrows in Fig. 8). 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 that there is 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 ($l \gg \frac{n_s}{2}$), the dataflow inside the stage changes as displayed on the right side of Fig. 9. However, in this case, the dataflow is much simpler compared to 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. The generated hardware consumes a total number of $Memory_{SDF}$ bits for storage shown in Eqn. 3. The overall latency $Latency_{SDF}$ of one NTT with SDF architecture is shown in Eqn. 4.

$$Memory_{SDF} = \log_2(q) \cdot \sum_{s=0}^{log_2(n)-1} f_s \tag{3}$$

$$Latency_{SDF} \sim 2 \cdot n + \log_2(n) \cdot l \tag{4}$$

5.2.2 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 as well as 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. 5.1). 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 Radix2 MDC NTT architectures through a given polynomial size n. In the generated MDC architecture, the total number of bits consumed by the hardware ($Memory_{MDC}$) is shown in Eqn. 5 while the overall latency ($Latency_{MDC}$) of one NTT with MDC architecture is shown in Eqn. 6.

$$Memory_{MDC} = \log_2(q) \cdot \sum_{s=0}^{log_2(n)-1} f_s \tag{5}$$

$$Latency_{MDC} \sim n + \log_2(n) \cdot l \tag{6}$$

5.3 Low-level Arithmetic Units

As shown in Sec. 5, each NTT stage serves as a controller that coordinates multiple sub-modules. In contrast to the architectural design that mostly depends on the polynomial size n, the hardware of low-level arithmetic units depends on the coefficient modulus size, $\log_2(q)$. The parametric coefficient modulus size makes it challenging to design all the required arithmetic units such as a DSP-based integer multiplier, a Montgomery modular reduction unit, and a unified butterfly unit.

5.3.1 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. The multiplication unit should be able to cope with any given data size due to the parameterization in our design. Xilinx Vivado has an integrated IP generator that can generate integer multiplier circuits. However, it has support only up to 64-bit inputs and is not easy to combine with a parametric design like ours (e.g., it requires manual instantiating). Also, another issue of Xilinx-generated multiplier units is the high utilization of DSP units and high latency for large sizes.

Our parametric integer multiplier design uses a divide-and-conquer approach and it splits a multiplication operation, $c = a \cdot b$, into smaller multiplications to reduce the DSP count and latency. First, it splits both operands a and b into chunks via standard tiling [22]. These chunks can have different sizes up to w_0 or w_1 bits depending on the DSP architecture of the deployed FPGA platform.

Algorithm 3 Word-level Montgomery Reduction Algorithm for NTT-friendly Primes [15]

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

1: $T \leftarrow d$
2: **for** $(i = 0; i < L; i = i + 1)$ **do**
3: $T_H \leftarrow T \gg w$
4: $T_L \leftarrow T \pmod{2^w}$
5: $T2 \leftarrow -T_L \pmod{2^w}$
6: $cin \leftarrow T2[w-1] \vee T_L[w-1]$
7: $T \leftarrow (q_H \cdot T2) + T_H + cin$
8: **end for**
9: **if** $T \geq q$ **then**
10: $c = T - q$
11: **else**
12: $c = T$
13: **end if**
14: **return** c

In the case of Xilinx's newer Ultrascale+ architecture, DSP units can support up to 27-bit×18-bit signed integer multiplication. Hence, w_0 and w_1 can be up to 26 and 17 bits while older platforms can support up to 24 and 17 bits. Then, small chunks of a and b are multiplied by each other.

PROTEUS uses this approach and implements all small chunk multiplications in parallel by instantiating one DSP unit per multiplication. The total number of employed DSP units (DSP_{MUL}) can be calculated by using Eqn. 7. It should be noted that we let the synthesis tool implement multiplication using DSP or LUTs. This enables the tool to use LUT for small-sized multiplications to save DSP resources. This sometimes leads to a lower number of DSPs than Eqn. 7. Finally, DSP outputs are accumulated using a parametric carry-save-adder (CSA) tree. The multiplier unit is fully pipelined and its overall latency is the sum of the latency of one DSP unit and CSA tree. The proposed parametric integer multiplication unit takes the bit-size of modulus ($\log_2(q)$) and maximum chunk sizes (w_0 and w_1) as inputs and generates the corresponding multiplication circuit.

$$DSP_{MUL} \leq i \cdot j \text{ where } i = \lceil \frac{log_2(q)}{w_0} \rceil \text{ and } j = \lceil \frac{log_2(q)}{w_1} \rceil \tag{7}$$

5.3.2 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. There are several techniques to perform a modulo reduction. Barrett [2] and Montgomery [17] are two well-known techniques to perform modular reduction for a generic modulus q. Add-shift-based approaches [4, 10, 21] 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 [25] and techniques targeting modulus of certain form [15, 28]. 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 3.

Algorithm 4 Algorithm for Mapping Word-level Montgomery Reduction into FPGA

1: $s \leftarrow 24$ or 26 (based on the DSP architecture of target FPGA platform)
2: **if** ($\log_2(T_H) \leq 47$) **then**
3:   **if** ($\log_2(q_H) \leq s$) **then**
4:     Implement $T \leftarrow (q_H \cdot T2) + T_H + cin$ (line 7 of Algorithm 3) using one DSP unit.
5:   **else**
6:     Instantiate $v = \lceil \frac{\log_2(q_H)}{s} \rceil$ DSP units.
7:     The first DSP unit implements $(q_H[s-1:0] \cdot T2) + T_H + cin$.
8:     The remaining DSP units implement $(q_H[s \cdot i + s - 1 : s \cdot i] \cdot T2)$, $i \in [1, v-1]$.
9:     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.
10:   **end if**
11: **else**
12:   **if** ($\log_2(q_H) \leq s$) **then**
13:     Implement $r \leftarrow (q_H \cdot T2) + cin$ using one DSP unit.
14:     Implement $T \leftarrow r + T_H$ using one adder.
15:   **else**
16:     Instantiate $v = \lceil \frac{\log_2(q_H)}{s} \rceil$ DSP units.
17:     The first DSP unit implements $(q_H[s-1:0] \cdot T2) + cin$.
18:     The remaining DSP units implement $(q_H[s \cdot i + s - 1 : s \cdot i] \cdot T2)$, $i \in [1, v-1]$.
19:     DSP outputs and T_H are reduced to 2 using one CSA tree, then the final result is computed using one adder.
20:   **end if**
21: **end if**
22: **return** c

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 2-8 in Algorithm 3). Finally, a reduction operation is performed at the end (lines 9-13 in Algorithm 3). The word-level Montgomery algorithm is scalable and it enables efficient utilization of DSP units in FPGA. As shown in line 7 of Algorithm 3, the algorithm performs $T \leftarrow (q_H \cdot T2) + T_H + cin$ in each reduction step which 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 4, we present an algorithm to map this operation into FPGA using DSP units efficiently. When $\log_2(T_H)$ is less than 48, then the addition of T_H and cin can be implemented using DSP without using 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. 5.3.1, 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, then the addition of 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 either 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. 10 and Fig. 11 show the implementation of modular reduction circuit for parameters

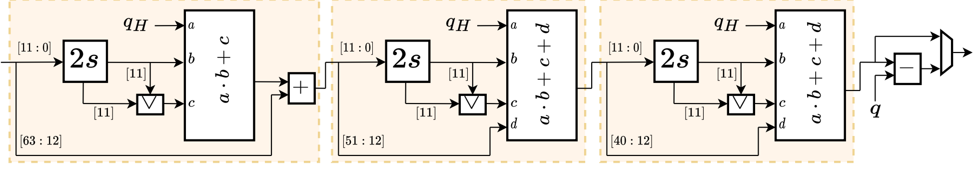


Fig. 10. Word-level Montgomery reduction circuit for $\log_2 q = 32$ and $w = 12$.

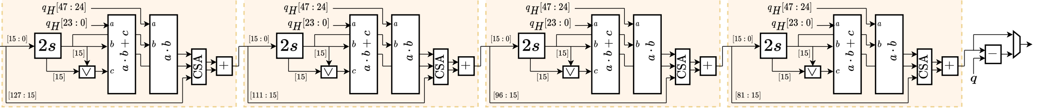


Fig. 11. Word-level Montgomery reduction circuit for $\log_2 q = 64$ and $w = 16$.

$\log_2(q) = 32$, $w = 12$ and $\log_2(q) = 64$, $w = 16$, respectively, where $2s$ and $\vee$ represent two's complement and logical OR operations.

It should be noted that the Algorithm 3 introduces an extra term at the output, R^{-1} where R is $2^{w \cdot L}$. To eliminate this extra constant, either output of the reduction operation or one of the input operands should be multiplied with R. Since one operand is always constant (either 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 does not add any extra latency.

5.3.3 Parametric CT, GS and Unified Butterfly Units. There are two approaches, DIT and DIF, to constructing efficient NTT algorithms that require so-called CT and GS butterfly configurations, as described in Sec. 2.2. Depending on the configuration of PROTEUS, we either require a CT, GS, or both butterflies combined. This reliance leads to the necessity of three different units.

Note that a butterfly unit consists of modular operations like addition, subtraction, and multiplication. Yet, the only difference between a CT and a GS butterfly is the placement of the modulo multiplication. The CT butterfly performs the required modular multiplication before the modulo subtraction while in GS it is performed afterward. In terms of parametrization, a butterfly unit is affected by the size of the modulus q, while n has no impact.

Each butterfly construct takes a, b, w as inputs and gives u, v as output. In case of CT, it outputs $u = a + (b \cdot w) \pmod q$ and $v = a - (b \cdot w) \pmod q$ while GS butterfly outputs $u = a + b \pmod q$ and $v = (a - b) \cdot w \pmod q$. Yet, a unified butterfly can give either an output in CT or GS, which depends on the configuration. The naive way to support both is to instantiate both GS and CT and select the desired outputs. This, however, would double the number of hardware resources, which is neither efficient nor desirable. Another approach [1] is adding a pair of addition and subtraction units to support both butterfly configurations. These additional units are either used or bypassed to act as a CT or GS butterfly unit. This approach saves one costly modular multiplication unit, yet, this is not ideal since it employs extra addition and subtraction units.

We present a different design for a unified butterfly that fits well into a pipelined architecture like PROTEUS which requires both butterfly operations during NTT computation. The unified butterfly takes a, b, w as inputs and gives u, v as output as in CT and GS. The dataflow inside our unified butterfly redirects via MUXes to either produce CT or GS butterfly outputs as shown in Fig. 12.

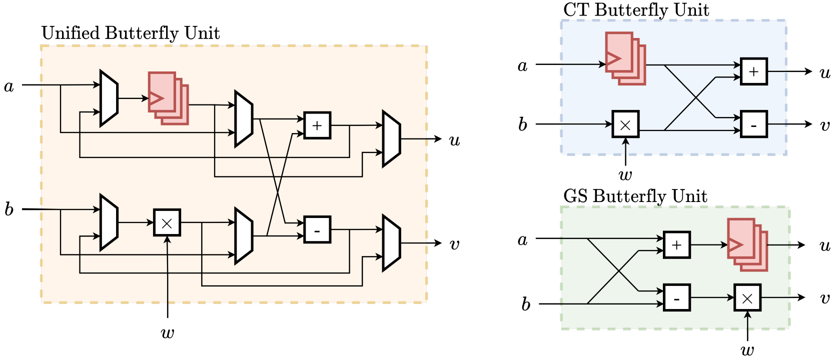


Fig. 12. Overview of CT, GS, and Unified butterfly units used in PROTEUS

6 EVALUATIONS

In this section, we present the area and performance results of hardware architectures that are 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. As described in Sec. 5.1, the proposed framework takes polynomial size (n), coefficient modulus (q), NTT/MNTT configuration (OP1 to OP8), and architecture type (SDF or MDC) as inputs, and generates the corresponding hardware. We obtained area and performance results using Xilinx Vivado 2019.1 for Xilinx Virtex-7 XCVX485T/Alveo U250 FPGA with default synthesis and place & route 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 on an actual Xilinx PYNQ FPGA board.

6.1 Evaluation of NTT configuration and parameter selection

In this section, we present 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 right NTT hardware. In Table 3, we present area utilization, latency (in terms of clock cycles), and an average latency of 100 operations for SDF and MDC NTT architectures with different configurations (OP1 to OP8), parameters, and modular reduction circuits. MDC architecture shows almost 2× better performance compared to SDF architecture at the expense of slightly larger LUT and DFF utilization. Further, it requires 2× 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, (i) a custom add-shift-based reduction unit for a constant modulus and (ii) word-level Montgomery reduction unit presented in Sec. 5.3.2. As shown in Table 3, Montgomery reduction uses 1.6× more DSP units with similar LUT and DFF utilization compared to 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. 4, NTT configuration has a significant impact on implementation complexity. Table 3 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

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

Table 3. Implementation and performance results for SDF and MDC NTT architectures for different parameters and design options (all results are collected for 150 MHz target clock frequency)

complexity due to different NTT and INTT orderings, as explained in Sec. 4. Hence, OP3 and OP4 configurations outperform only OP1 and OP2 configurations. The configurations OP5, OP6, OP7, OP8 eliminate different orderings of NTT and INTT, and show better area performance compared to OP1, OP2, OP3, 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 3, they use 50% more BRAMs compared to NTT/INTT configurations since they need to store pre-processing and post-processing constants (see Sec. 2.2).

We present the results of SDF and MDC architectures for a selection of various polynomial degrees ranging from 2^{10} to 2^{16} and moduli of size 28-bit and 64-bit in Table 4. We also visualize the change in the area utilization for different parameters and configurations in Fig. 13 and Fig. 14. Table 4, Fig. 13 and Fig. 14 show that PROTEUS can generate different NTT architectures that cover a wide range of parameters, area utilization and performance, easily by just changing a few parameters.

6.2 Comparison with the literature

In this section, we present the comparisons between PROTEUS-generated NTT architectures and related works in the literature [8, 12, 14, 16, 29–31, 33]. In Table 5, we present resource utilization,

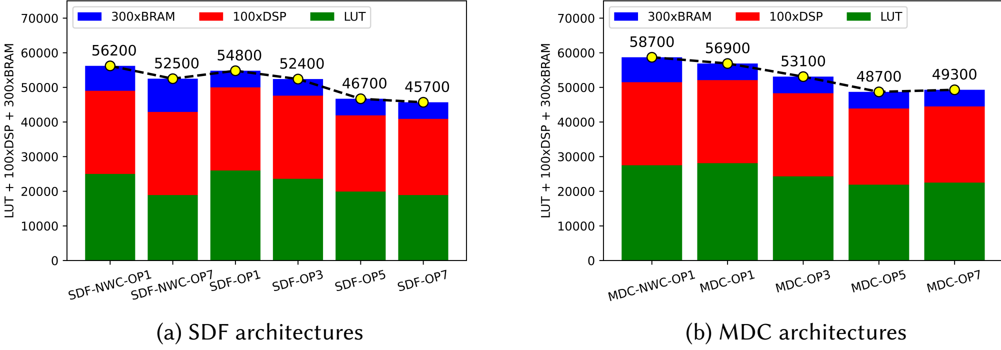
Design	$(n, \log_2(q))$	**LUT/FF/DSP/BRAM**	**Freq.**	**Lat./Avg.***
SDF-OP7/8	$(2^{10}, 28)$	5.2k/2.8k/36/2	150	2123/1035
SDF-OP1[a]		7.4k/3.6k/40/3	150	2128/1035
MDC-OP7/8		6.9k/3.2k/36/2	150	1100/518
MDC-OP1[a]		9.8k/4.7k/40/3	150	1124/518
SDF-OP7/8	$(2^{12}, 28)$	6.3k/3.4k/44/8	150	8382/4138
SDF-OP1[a]		8.9k/4.4k/48/12	150	8391/4138
MDC-OP7/8		8.4k/4.0k/44/8	150	4286/2070
MDC-OP1[a]		11.9k/5.7k/48/12	150	4214/2070
SDF-OP7/8	$(2^{14}, 28)$	7.8k/4.0k/52/29	150	32871/16549
SDF-OP1[a]		10.5k/5.2k/56/44	150	32890/16549
MDC-OP7/8		10.0k/4.8k/52/30	150	16487/8275
MDC-OP1[a]		12.9k/6.2k/56/44	150	16520/8275
SDF-OP7/8	$(2^{16}, 28)$	10.4k/4.8k/60/113	150	131201/66193
SDF-OP1[a]		13.6k/6.2k/64/170	150	131092/66193
MDC-OP7/8		12.2k/5.6k/60/114	150	65605/33097
MDC-OP1[a]		16.8k/7.9k/64/170	150	65674/33097
SDF-OP7/8	$(2^{10}, 64)$	15.7k/14.0k/180/4	150	2133/1035
SDF-OP1[a]		21.0k/16.4k/200/6	150	2138/1035
MDC-OP7/8		18.6k/14.9k/180/4	150	1110/518
MDC-OP1[a]		23.3k/18.8k/200/6	140	1134/518
SDF-OP7/8	$(2^{12}, 64)$	18.9k/16.8k/220/16	150	8293/4138
SDF-OP1[a]		25.0k/19.6k/240/24	150	8310/4138
MDC-OP7/8		22.6k/18.0k/220/16	150	4185/2069
MDC-OP1[a]		27.6k/22.5k/240/24	150	4214/2070
SDF-OP7/8	$(2^{14}, 64)$	22.6k/19.7k/260/64	150	32895/16549
SDF-OP1[a]		29.4k/22.8k/280/96	150	32914/16549
MDC-OP7/8		26.7k/21.2k/260/63	150	16473/8275
MDC-OP1[a]		32.8k/26.1k/280/95	135	16520/8275
SDF-OP7/8	$(2^{16}, 64)$	27.5k/23.1k/300/256	145	131225/66193
SDF-OP1[a]		35.2k/26.5k/320/384	135	131218/66193
MDC-OP7/8		31.3k/24.4k/300/255	150	65637/33097
MDC-OP1[a]		37.5k/29.9k/320/383	135	65706/33097

*: Latency/Average latency for 100 operation. [a]: Uses NWC technique.

Table 4. Implementation and performance results for SDF and MDC NTT architectures for $\log_2(q) = \{28, 64\}$ and $n = \{2^{10}, 2^{12}, 2^{14}, 2^{16}\}$ using Montgomery reduction.

performance, architecture and level of parallelism (e.g., number of butterfly units running in parallel) results of PROTEUS-generated and related works.

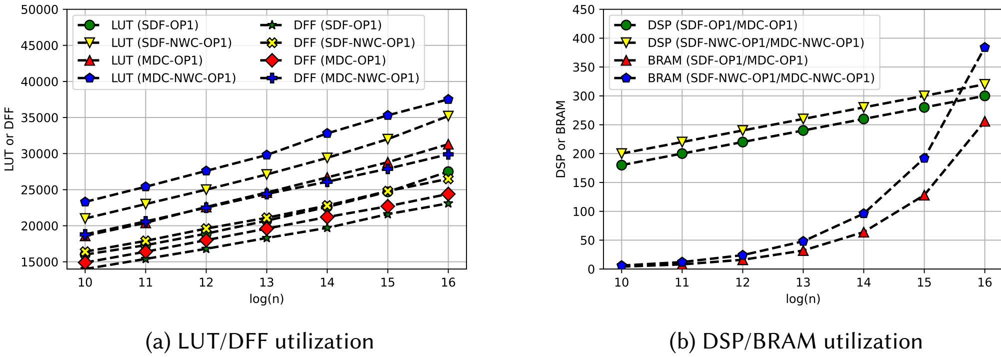
Parametric iterative NTT architectures can set the level of parallelism by changing the number of butterfly units. Compared to the low-cost iterative NTT architectures with one butterfly unit, our SDF and MDC NTT architectures show much better performance. For parameters $n = 2^{12}$ and $\log_2(q) = 64$, our SDF and MDC NTT architectures with OP7 configuration show 6× and 11.8× speedup (in terms of cycles), respectively, compared to the low-cost NTT architectures in [8, 12]. Compared to a balanced iterative NTT architecture with 8 butterfly cores [12], our SDF and MDC architectures show similar performance while using 11× less BRAM. High-performance iterative



(a) SDF architectures

(b) MDC architectures

Fig. 13. Comparison of resource utilization of SDF and MDC NTT architectures for different NTT/MNTT approaches using $n = 2^{12}$ and $\log_2(q) = 64$ with Montgomery reduction. The green, red and blue boxes represent the number of LUTs, 100×the number of DSPs and 300×the number of BRAMs [31] respectively.



(a) LUT/DFF utilization

(b) DSP/BRAM utilization

Fig. 14. Resource utilization of SDF and MDC NTT architectures for $\log_2(q) = 64$ and $n = 2^{10}$ to $n = 2^{16}$ with Montgomery reduction.

NTT architectures [8, 12, 14, 16] show better performance than our SDF and MDC architectures. However, they use much more resources and require high bandwidth. For example, the work in [14] shows 8.6× better performance compared to our MDC architecture for parameter $n = 2^{10}$ at the expense of 5.6× more LUT, 26× more DSP and 162× more BRAM. Although iterative NTT architectures in [5, 18] are parametric, they target very 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 [19, 26, 29, 30]. The works in [29, 30] can change their levels of parallelism and increase their performance at the expense of more resources and increased bandwidth. Compared to the MDC architecture with 4 parallel butterfly units [30], our MDC NTT architecture still shows 1.7× better performance while using fewer resources. In [19, 26], MDC NTT architectures for small parameters without any flexibility are presented. Thus, they are not compared with our work.

PipeNTT [31] is the current state-of-the-art work for parametric SDF Radix-2 NTT architecture. In Table 5, we compare our SDF and MDC NTT architectures with PipeNTT for parameters

Work	Platf.	$n, \log_2(q)$	LUT/FF/DSP/BRAM	Freq. (MHz)	Lat./Avg.* (in cycle)	Lat./Avg.* (in μs)	NTT Arch.	LoP†
[30]	Virtex 7	2^{10}, 28	95k/104k/640/80	215	-	0.9/-	MDC	32
			187k/205k/1280/128	212	-	0.75/-	MDC	64
[29]	Virtex 7	2^{10}, 28	206k/159k/640/80	210	-	1.1/-	MDC	32
[14]	Virtex 7	2^{10}, 32	77k/-/952/325.5	200	-	0.4/-	Iter.	32
[16]	Virtex 7	2^{10}, 32	39.6k/-/224/96	150	-	1.66/-	Iter.	32
SDF-OP7	Virtex 7	2^{10}, 28	5.2k/2.8k/36/2	150	2113/1035	14.09/6.90	SDF	1
MDC-OP7	Virtex 7	2^{10}, 28	6.9k/3.2k/36/2	150	1090/518	7.27/3.46	MDC	1
[8]	UV	2^{12}, 28	2.7k/2.4k/6/8	435	12302/-	28.2/-	Iter.	2
			11.8k/8.9k/24/16	379	3086/-	8.15/-	Iter.	8
[16]	Virtex 7	2^{12}, 32	39.6k/-/224/96	150	-	5.84/-	Iter.	32
[29]	XCU200	2^{12}, 30	54.1k/56.2k/288/84	250	-	24.7/-	MDC	4
SDF-OP7	Virtex 7	2^{12}, 28	6.3k/3.4k/44/8	150	8273/4138	55.31/27.91	SDF	1
MDC-OP7	Virtex 7	2^{12}, 28	8.4k/4.0k/44/8	150	4165/2069	27.91/13.8	MDC	1
[8]	UV	2^{12}, 60	2.6k/2.5k/26/21	144	24590/-	171.6/-	Iter.	1
			90.0k/77.0k/832/160	130	782/-	6.0/-	Iter.	32
[12]	Virtex 7	2^{12}, 60	2.7k/-/31/180	125	24708/-	197.6/-	Iter.	1
			23.2k/-/248/176	125	3276/-	26.2/-	Iter.	8
			99.3k/-/992/176	125	972/-	7.77/-	Iter.	32
[31]	Virtex 7	2^{12}, 60	17.0k/11.0k/286/24.5	150	8284/-	55.31/-	SDF	1
SDF-OP7	Virtex 7	2^{12}, 64	18.9k/16.8k/220/16	150	8293/4138	55.31/27.91	SDF	1
MDC-OP7	Virtex 7	2^{12}, 64	22.6k/18.0k/220/16	150	4185/2069	27.91/13.8	MDC	1

*: Latency/Average latency for 100 operation. †: Level of parallelism.

Table 5. Comparison table

$n = 2^{12}$ and $\log_2(q) = 64$. Compared to our SDF architecture, PipeNTT shows similar performance while using 1.3× more DSP and 1.5× more BRAM. Our MDC architecture outperforms PipeNTT while using less DSP and BRAM. PipeNTT suffers a high number of BRAM units because their implementation does not use twiddle factor optimization presented in 4.1. Thus, they have to employ extra BRAMs to store twiddle factors for INTT. For DSP utilization, our architecture shows better performance compared to PipeNTT due to our optimized word-level Montgomery modular reduction unit. When we use their area metric for comparison (LUT + 100×DSP + 300×BRAM in Table 1 of [31]), we show up to 35% better performance for various configurations. Their design provides limited configurability as they only support configuration OP1 for SDF while we support SDF and MDC architectures for several configurations.

In [33], authors propose an NTT architecture targeting very large-degree polynomials (e.g., $n = 2^{20}$) for ZKP applications. Their implementation uses a hierarchical approach and divides a large NTT into multiple smaller NTTs (e.g., $n = 2^{10}$) where they use SDF Radix-2 NTT architecture for implementing small NTTs. They also target very large coefficient modulus sizes ranging from 256-bit to 768-bit. They target the ASIC platform and do not provide any area and performance results for SDF NTT architecture. For proof-of-concept, we use PROTEUS to generate NTT architectures for a 256-bit coefficient modulus and several polynomial sizes, $n = \{2^{10}, 2^{12}, 2^{14}, 2^{16}\}$. We present performance and implementation results in Table 6. Note that we used a specially designed add-shift-based reduction circuit for 256-bit modular reduction operation.

Work	Platf.	$n, \log_2(q)$	LUT/FF/DSP/BRAM/URAM	Freq.	Lat./Avg.*
SDF-OP1[a]	AU250	2^{10}, 256	219k/48k/1650/20/4	125	4k/2k
		2^{12}, 256	261k/58k/1980/65/12	125	16k/8k
		2^{14}, 256	305k/67k/2310/307/16	125	32k/16k
		2^{16}, 256	356k/77k/2640/1417/16	125	131k/66k

*: Latency/Average latency for 100 operation. [a]: Uses NWC technique.

Table 6. Implementation results for $\log_2(q) = 256$ on Xilinx Alveo U250 FPGA

7 CONCLUSIONS

In this paper, we design a tool, called PROTEUS, to generate bandwidth-efficient SDF and MDC Radix-2 NTT architectures. Each architecture supports several configurations and parameters (polynomial degree and coefficient modulus sizes) that can be configured at design time. We also introduce algorithmic optimizations to reduce memory requirements for storing twiddle factors and eliminate the multiplication with scalar n^{-1} at the end of INTT. Further, we introduce several architectural optimizations for low-level parametric arithmetic units like the unified butterfly and word-level Montgomery reduction units. We performed experiments for a wide range of polynomial degrees and coefficient modulus sizes. A comparison with state-of-the-art works like PipeNTT [31] shows the advantages of our design approach. Both PipeNTT and PROTEUS implements SDF-based Radix2 NTT architecture, yet, PROTEUS uses up to 23% and 35% less resources in terms of DSPs and BRAMs, respectively, with a similar LUT utilization. Moreover, PROTEUS reduces the latency of NTT by 50% compared to PipeNTT when using its MDC-based architecture. These advantages combined with design-time flexibility make PROTEUS suitable as a basic building block for FHE and ZKP systems.

In the future, additional analysis and extension of PROTEUS for various Radix-2^n could make our design even more versatile. The usage of different Radix sizes would need an in-depth analysis of the corresponding data flow, yet, it would decrease the latency of NTT. Also, the elimination of twiddle factor memories and adapting on-the-fly twiddle factor generation could be explored for memory-constrained configurations.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 4 (8 2019), 17–61. https://doi.org/10.13154/tches.v2019.i4.17-61

[2] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings (Lecture Notes in Computer Science, Vol. 263)*. Springer, 311–323. https://doi.org/10.1007/3-540-47721-7_24

[3] Eleanor Chu and Alan George. 1999. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms.* CRC press.

[4] Wei Dai and Berk Sunar. 2016. cuHE: A homomorphic encryption accelerator library. In *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 169–186.

[5] Kemal Derya, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2022. CoHA-NTT: A Configurable Hardware Accelerator for NTT-based Polynomial Multiplication. *Microprocessors and Microsystems* 89 (2022), 104451.

[6] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. 2022. Basalisc: Flexible asynchronous hardware accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2205.14017* (2022).
[7] Craig Gentry. 2009. *A fully homomorphic encryption scheme.* Stanford university.
[8] Xiao Hu, Jing Tian, Minghao Li, and Zhongfeng Wang. 2022. AC-PM: An Area-Efficient and Configurable Polynomial Multiplier for Lattice Based Cryptography. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2022), 1–14. https://doi.org/10.1109/TCSI.2022.3218192
[9] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2009. Zero-Knowledge Proofs from Secure Multiparty Computation. *SIAM J. Comput.* 39, 3 (2009), 1121–1152. https://doi.org/10.1137/080725398
[10] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. 2015. Efficient Ring-LWE encryption on 8-bit AVR processors. In *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17.* Springer, 663–682.
[11] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. Cryptology ePrint Archive, Paper 2016/504. https://eprint.iacr.org/2016/504 https://eprint.iacr.org/2016/504.
[12] Ahmet Can Mert, Emre Karabulut, Erdinc Ozturk, Erkay Savas, and Aydin Aysu. 2020. An Extensive Study of Flexible Design Methods for the Number Theoretic Transform. *IEEE Trans. Comput.* (2020), 1–1. https://doi.org/10.1109/TC.2020.3017930
[13] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, Michela Becchi, and Aydin Aysu. 2020. A Flexible and Scalable NTT Hardware : Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography. In *2020 Design, Europe Conference (DATE).*
[14] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 2 (2019), 353–362.
[15] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD).* 253–260. https://doi.org/10.1109/DSD.2019.00045
[16] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2020. FPGA implementation of a run-time configurable NTT-based polynomial multiplication hardware. *Microprocessors and Microsystems* 78 (2020), 103219.
[17] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44 (1985), 519–521.
[18] Jianan Mu, Yi Ren, Wen Wang, Yizhong Hu, Shuai Chen, Chip-Hong Chang, Junfeng Fan, Jing Ye, Yuan Cao, Huawei Li, and Xiaowei Li. 2022. Scalable and Conflict-free NTT Hardware Accelerator Design: Methodology, Proof and Implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), 1–1. https://doi.org/10.1109/TCAD.2022.3205552
[19] Ziying Ni, Ayesha Khalid, Dur-e-Shahwar Kundi, Máire O'Neill, and Weiqiang Liu. 2022. Efficient Pipelining Exploration for A High-performance CRYSTALS-Kyber Accelerator. *IACR Cryptol. ePrint Arch.* (2022), 1093.
[20] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In *Progress in Cryptology – LATINCRYPT 2015*, Kristin Lauter and Francisco Rodríguez-Henríquez (Eds.). Springer International Publishing, Cham, 346–365.
[21] Claudia Patricia Renteria-Mejia and Jaime Velasco-Medina. 2017. High-throughput ring-LWE cryptoprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 8 (2017), 2332–2345.
[22] Debapriya Basu Roy, Debdeep Mukhopadhyay, Masami Izumi, and Junko Takahashi. 2014. Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves *(DAC '14).* Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2593069.2593234
[23] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact Ring-LWE Cryptoprocessor. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, Lejla Batina and Matthew Robshaw (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 371–391.
[24] Michael Scott. 2017. A Note on the Implementation of the Number Theoretic Transform. In *Cryptography and Coding - 16th IMA International Conference, IMACC 2017.* Springer. https://doi.org/10.1007/978-3-319-71045-7_13
[25] Silvan Streit and Fabrizio De Santis. 2017. Post-quantum key exchange on ARMv8-A: A new hope for NEON made simple. *IEEE Trans. Comput.* 67, 11 (2017), 1651–1662.
[26] Weihang Tan, Antian Wang, Yingjie Lao, Xinmiao Zhang, and Keshab K. Parhi. 2021. Pipelined High-Throughput NTT Architecture for Lattice-Based Cryptography. In *2021 Asian Hardware Oriented Security and Trust Symposium.* 1–4. https://doi.org/10.1109/AsianHOST53231.2021.9699608
[27] Franz Winkler. 1996. *Polynomial algorithms in computer algebra.* Springer Science & Business Media.
[28] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2021. A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme. In *2021 Design, Automation & Test in Europe Conference &*

Exhibition (DATE). IEEE, 1020–1025.

[29] Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2022. NTTGen: A Framework for Generating Low Latency NTT Implementations on FPGA. In *Proceedings of the 19th ACM International Conference on Computing Frontiers* (Turin, Italy) *(CF '22)*. 30–39.

[30] Tian Ye, Yang Yang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. 2021. FPGA Acceleration of Number Theoretic Transform. In *High Performance Computing*. Springer International Publishing, 98–117.

[31] Zewen Ye, Ray C.C. Cheung, and Kejie Huang. 2022. PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2022), 1–1. https://doi.org/10.1109/TCSII.2022.3184703

[32] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 2 (3 2020), 49–72.

[33] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture. In *48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*.