electronics

MDPI

Article

Lazy Modular Reduction for NTT

Geumtae Kim [1], Eunyoung Seo [2,*], Yongwoo Lee [3,*], Young-Sik Kim [2] and Jong-Seon No [1]

[1] The Department of Electrical and Computer Engineering, Institute of New Media and Communications (INMC), Seoul National University, Seoul 08826, Republic of Korea; kkt1513@snu.ac.kr (G.K.); jsno@snu.ac.kr (J.-S.N.)
[2] The Department of Electrical Engineering and Computer Science, Daegu Gyeongbuk Institute of Science and Technology, Daegu 42988, Republic of Korea; ysk@dgist.ac.kr
[3] The Department of Electrical and Computer Engineering, Inha University, Incheon 22212, Republic of Korea
* Correspondence: eyseo_dgist@dgist.ac.kr (E.S.); yongwoo@inha.ac.kr (Y.L.)

Abstract: The number theoretic transform (NTT) is a fundamental operation in cryptography, especially for lattice-based cryptographic schemes. This paper introduces *LazyNTT*, a novel method that reduces the number of Montgomery multiplications required in the NTT computation by replacing some of them with standard multiplication without modular reduction. This approach enhances the performance of the NTT computation and modular polynomial multiplication in lattice-based cryptographic schemes. The proposed *LazyNTT* can be generalized by increasing the number of standard multiplications. The experimental results show that the proposed *LazyNTT* improves the cycle counts of the NTT by up to 28% and 9%, respectively, by allowing two and one standard multiplications.

Keywords: number theoretic transform (NTT); Montgomery multiplication; modular reduction; post-quantum cryptography (PQC); lattice-based cryptography

check for updates

Citation: Kim, G.; Seo, E.; Lee, Y.; Kim, Y.-S.; No, J.-S. Lazy Modular Reduction for NTT. *Electronics* **2024**, *13*, 4887. https://doi.org/10.3390/electronics13244887

Academic Editors: Changhoon Lee, YoHan Park, Tao Huang, Tsz Hon Yuen, Guanglin Zhang, Shihao Yan and Jusak Jusak

Received: 25 October 2024
Revised: 3 December 2024
Accepted: 9 December 2024
Published: 11 December 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The advent of quantum computers breaks classical public key cryptosystems such as RSA [1] using Shor's algorithm [2]. As a result, the development of post-quantum cryptography (PQC) has become crucial. The National Institute of Standards and Technology (NIST) has initiated a standardization process for PQC [3], with many candidates based on lattice cryptography.

Lattice-based cryptography heavily relies on modular arithmetic, particularly modular polynomial multiplication, which constitutes a significant computational bottleneck. The number theoretic transform (NTT) reduces the complexity of modular polynomial multiplication from $O(n^2)$ to $O(n \log n)$ by leveraging component-wise multiplication. Despite this, modular reduction within the NTT remains computationally expensive.

This paper proposes *LazyNTT*, a method designed to reduce the number of Montgomery multiplications [4] in the NTT computation, which is an efficient workaround for naive modular reduction but is still more expensive than the standard integer multiplication. By replacing some of them with standard multiplications at intermediate stages of the NTT computation, we achieve significant computational complexity improvements.

1.1. Contributions

In this paper, we propose a new variant of the NTT with a faster runtime by reducing the number of Montgomery multiplications. The detailed contributions are given as follows:

1. We propose two versions of *LazyNTT* that reduce the number of Montgomery multiplications in the NTT computation by replacing some of them with standard multiplications during the intermediate stages of the NTT.
2. We demonstrate the effectiveness of the proposed algorithm in practical parameters used in Falcon [5] and Kyber [6], showing that this method can significantly reduce the total cycle counts of the NTT computation.

3. The proposed method is applicable to various lattice-based cryptographic schemes, including PQC [7–11], and it can naturally be generalized for the further replacement of Montgomery multiplications.

1.2. Related Works

Extensive research has been conducted on the NTT due to its critical role in cryptography. In the hardware domain, efficient implementations of the NTT on FPGA have been presented in [12,13]. They optimized memory access patterns and minimized the latency of the NTT computation. Additionally, the performance improvement of polynomial multiplication using the NTT by employing a unified butterfly unit has been presented in [14]. In the software domain, performance improvements have been achieved using the Harvey butterfly structure in the radix-4 NTT [15] and optimizing cyclic convolution operations to speed up the NTT [16]. In the context of PQC, efficient NTT implementations using a hardware accelerator and instruction set for Kyber [7,8], parallelism with FPGA for Dilithium [9,10], and twiddle factor generation for Falcon [11,17] have been presented. In the field of homomorphic encryption (HE), some have focused on efficiently implementing the NTT using GPUs [18–20]. Specialized hardware designs to optimize the NTT for HE also have been proposed in [21,22]. Previous works are summarized in Table 1 for better clarity.

Table 1. Related works on number theoretic transform.

Platform	Papers	Cryptosystems	Technique
FPGA	[13]	HE	Parallelization, automatic NTT design
	[21,22]		On-the-fly twiddle factor generation
	[12]	Kyber	Double-bandwidth ping-pong memory access
	[14]	PQC	Unified butterfly unit
GPU	[18]	HE	Optimized memory access patterns
	[19]		Multi-stream asynchronous computation
	[20]		On-the-fly twiddle factor generation
CPU	[15]	HE	Radix-4 Harvey butterfly
	[16]	PQC	Modified cyclic convolution

Our work builds on these advancements by specifically addressing the computational load of modular reduction within the NTT. To the best of our knowledge, a direction focusing on replacing only the type of multiplications used in the NTT has not been explored before. Additionally, previous works on the NTT have been developed across various platforms and cryptosystems, making it difficult to compare them under the same criterion. Therefore, we use the official implementation code of Falcon as a reference, which is selected as the PQC standard by NIST. We believe that comparing computational complexity with this reference code is appropriate for demonstrating the meaningful results of *LazyNTT*. By replacing some Montgomery multiplications with standard multiplications without modular reduction, we achieve a notable computational complexity gain, which complements existing optimization in both hardware and software implementations.

1.3. Organization

The structure of this paper is as follows: Section 2 introduces the basic notations and preliminaries needed for understanding Montgomery multiplication and the NTT. It also includes a detailed explanation of the radix-2 NTT, which is the primary focus of the proposed method. Section 3 describes the proposed Standard–Montgomery version (*SM-LazyNTT*) and Standard–Standard–Montgomery version (*SSM-LazyNTT*) algorithms. Their designs, operational principles, and generalization are also included in this section.

Section 4 provides the implementation results and an analysis of the improved computational complexity achieved by *SM-LazyNTT* and *SSM-LazyNTT* compared to the original NTT. Section 5 concludes the paper with a summary of the proposed method and suggestions for future works.

2. Preliminaries

2.1. Notations

A ciphertext modulus is denoted as q, and n represents both the length of the polynomial and the NTT. Throughout this paper, we assume that n is a power of two. For a polynomial $f(x)$, let f_i denote the i-th coefficient of the polynomial. Vectors are indicated as bold letters, for example $\mathbf{f}$, and it represents a vector containing all the coefficients of $f(x)$, i.e., $\mathbf{f} = (f_0, f_1, \cdots, f_{n-1})$. We denote the j-th component of the vector $\mathbf{f}$ as $\mathbf{f}[j]$. We use $\overline{a}$ to indicate a value in Montgomery space and similarly use $\overline{\mathbf{a}}$ to denote a vector containing values in Montgomery space. The NTT of the polynomial $f(x)$ is denoted as $\hat{\mathbf{f}}$. R is a scaling factor used in Montgomery multiplication. The integer ring is denoted as $\mathbb{Z}_q$ and the polynomial ring with the modulus is $\mathbb{Z}_q[x]$. For a polynomial $\phi(x)$, we define the quotient ring $\mathbb{Z}_q[x]/\phi(x)$. Standard multiplication is denoted by $\cdot$ and Montgomery multiplication is denoted by $*$. Component-wise multiplication between two vectors, $\hat{\mathbf{f}}$ and $\hat{\mathbf{g}}$, is denoted as $\hat{\mathbf{f}} \circ \hat{\mathbf{g}}$, which means that each element in the same index is multiplied as $\hat{f}_i \cdot \hat{g}_i$. When the context is clear, we may omit the modular reduction notation, mod q. Let ω be a primitive n-th root of unity, and let α be a primitive $2n$-th root of unity in $\mathbb{Z}_q$, such that

$$\omega^n \equiv 1 \mod q \quad \text{and} \quad \alpha^{2n} \equiv 1 \mod q,$$

with $\omega^i \not\equiv 1 \mod q$ and $\alpha^j \not\equiv 1 \mod q$ for any integers $0 < i < n$ and $0 < j < 2n$, respectively.

2.2. Montgomery Multiplication

In integer modular arithmetic, while there are simple constant-time methods for addition and subtraction, multiplication is more complex due to the division by the modulus q required in the modular reduction step. Montgomery multiplication [4] is a widely used technique in cryptography for efficient modular multiplication. It allows the modulo multiplication of integers without directly performing division by the modulus, which is computationally expensive. Instead, it first transforms the numbers into a special domain called Montgomery space by multiplying them with a constant, R, and taking the result modulo q as

$$\overline{a} = a \cdot R \mod q.$$

This transformation enables more efficient modular arithmetic operations by replacing the division by q with cheaper operations such as multiplication, addition, and bit-shifting.

An element, a, in $\mathbb{Z}_q$ is mapped to Montgomery space as $\overline{a} = a \cdot R \mod q$. Addition and subtraction in Montgomery space are defined as in $\mathbb{Z}_q$, such that $\overline{a} \pm \overline{b} = a \cdot R \pm b \cdot R = (a \pm b) \cdot R \mod q$. However, multiplication doubles the factor R, and thus Montgomery multiplication $(*)$ is defined as follows:

$$a * b = (a \cdot b \cdot R^2) \cdot R^{-1} \mod q = (a \cdot b) \cdot R \mod q.$$

To implement the steps of multiplying R^{-1} and reducing modulo q efficiently, we use the extended Euclidean algorithm. Algorithm 1 describes the whole procedures. Unlike Montgomery multiplication, which multiplies two values in Montgomery space, we use the standard multiplication in this paper to refer to the multiplication between one value in Montgomery space and the other value in standard space.

Algorithm 1 Montgomery Multiplication

Require: $\overline{a}, \overline{b}, R, q$
Ensure: $\overline{a} \cdot \overline{b} \cdot R^{-1} \mod q$
Pre-computation:
$(R', q') \leftarrow$ Extended Euclidean(R, q)
$R' \leftarrow R' \mod q$ ▷ $RR' \equiv 1 \mod q$
$q' \leftarrow q' \mod R$ ▷ $qq' \equiv 1 \mod R$
Montgomery Multiplication:
$x \leftarrow \overline{a} \cdot \overline{b}$
$m \leftarrow (x \cdot q') \mod R$
$u \leftarrow (x - m \cdot q)/R$
if $u \leq 0$ **then**
    $u \leftarrow u + q$
end if
return u

2.3. Number Theoretic Transform

The NTT is a specialized form of the discrete Fourier transform (DFT) that operates in an integer ring. Unlike the DFT, which handles complex numbers, the NTT utilizes integer modular arithmetic, making it suitable for error-free and efficient computations in cryptography and polynomial multiplication. For efficient NTT computation, a structure analogous to the fast Fourier transform (FFT) [23] can be employed.

Let $\phi(x)$ be the n-th cyclotomic polynomial, whose roots are the primitive n-th roots of unity, and $\phi_1(x) = x - 1$. The NTT can be defined as a ring homomorphism that maps a polynomial in $\mathbb{Z}_q[x]/\phi(x)$ to $(\mathbb{Z}_q[x]/\phi_1(x))^n$. Here, the product of the quotient ring $(\mathbb{Z}_q[x]/\phi_1(x))^n$ can be considered an integer vector $(\mathbb{Z}_q)^n$, which is the vector of evaluations at the powers of a primitive root of unity.

Considering a polynomial multiplication over the quotient ring $\mathbb{Z}_q[x]/(x^n + 1)$ for two polynomials,

$$f(x) = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1}$$

and

$$g(x) = g_0 + g_1 x + \cdots + g_{n-1} x^{n-1},$$

the result $h(x) = (f(x) \times g(x))/(x^n + 1) \in \mathbb{Z}_q[x]/(x^n + 1)$ has coefficients such that

$$h_k = \sum_{i=0}^{k} f_i g_{k-i} - \sum_{i=k+1}^{n-1} f_i g_{n+k-i} \mod q.$$

This requires $O(n^2)$ multiplication complexity using a direct convolution approach. However, modular polynomial multiplication using the NTT achieves $O(n \log n)$ complexity with the following steps:

(1) Calculating the NTT of two polynomials.

$$\hat{\mathbf{f}} = NTT(f(x))$$

$$\hat{\mathbf{g}} = NTT(g(x))$$

(2) The component-wise multiplication of two NTTs.

$$\hat{\mathbf{h}} = \hat{\mathbf{f}} \circ \hat{\mathbf{g}}$$

(3) Calculating the inverse NTT of the result.

$$h(x) = iNTT(\hat{\mathbf{h}})$$

For the NTT, using $\phi(x) = x^n \pm 1$ enhances efficiency due to the presence of the roots of unity $\omega^n \equiv 1 \mod q$ for $x^n - 1$ and $\alpha^{2n} \equiv 1 \mod q$ for $x^n + 1$. These roots of unity enable a structured and cyclic approach to polynomial multiplication, allowing the use of FFT structure that breaks down the problem into smaller subproblems, leading to $O(n \log n)$ complexity. Moreover, the cyclic nature imposed by $x^n \pm 1$ allows for efficient modular arithmetic operations, ensuring that the polynomial degree is maintained within n, thereby simplifying modular reduction steps. For the quotient rings $\mathbb{Z}_q[x]/(x^n - 1)$ and $\mathbb{Z}_q[x]/(x^n + 1)$, the former is known as positive wrapped convolution (PWC) [24] and the latter as negative wrapped convolution (NWC) [25].

In the context of PQC, the NTT usually refers to the NWC version. For example, in Falcon and Dilithium, $\phi(x) = x^n + 1$ is used for the NTT. To perform polynomial multiplication using the NTT, the modulus is chosen to ensure that the primitive $2n$-th root of unity exists in $\mathbb{Z}_q$ such that $q \equiv 1 \mod 2n$. Two examples are as follows:

- Falcon uses $q = 12{,}289 = 6 \cdot 2048 + 1$ for $n = 1024$ or 512 [5].
- Dilithium uses $q = 8{,}380{,}417 = 16{,}368 \cdot 512 + 1$ for $n = 256$ [26].

In Kyber, the parameters $q = 3329 = 13 \cdot 2^8 + 1$ and $n = 256$ are used, but a primitive 512-th root of unity does not exist in $\mathbb{Z}_{3329}$. Therefore, Kyber does not use the NWC-based NTT. Instead, during the NTT, the polynomial $f(x)$ is converted into 128 polynomials with a degree of 1 rather than 256 integers. Although this paper focuses on NWC, the proposed method can be naturally extended to the PWC environment and works similarly well.

The NTT using NWC transforms a polynomial $f(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ to its evaluations at points $x = \alpha^{2i+1}$ for $i = 0, \ldots, n-1$. Since α^{2i} is also a n-th root of unity, we only consider α^{2i+1}, which is a primitive $2n$-th root of unity.

2.4. Radix-2 NTT Using FFT Structure

In the context of the NTT, radix-2 refers to a processing step, where two terms are handled together in a structure commonly known as a butterfly operation. By leveraging the divide-and-conquer strategy inherent in the FFT, the radix-2 NTT recursively splits the polynomial into smaller sub-NTTs, each half the size of the previous one, and applies the butterfly operation to combine the results. We will refer to this single iteration as a "stage". Within a stage, we refer to each spot according to the index as a "position". The butterfly operation involves the pairwise addition and subtraction of polynomial coefficients where one of them is multiplied by a power of the root of unity. This recursive structure ensures a significant reduction in computational complexity, making the transform efficient in both time and space, as in Algorithm 2.

In Algorithm 2, $\overline{\mathbf{G}}$ refers to the vector that stores all odd powers of the root of unity α^{2i+1} for $0 \leq i \leq n-1$, which are converted into Montgomery space. The first "for" loop (line 3) shows that there are a total of $\log n$ stages, with each iteration representing the computation of one stage. The second "for" loop (line 5) handles the computation of the m butterfly structure sets, with each structure using a different element of $\overline{\mathbf{G}}$ in that stage. The third "for" loop (line 7) processes h butterfly structures using the same element of $\overline{\mathbf{G}}$.

Generally, the radix-2 NTT is calculated using following steps.

1. Divide all coefficients into two sets.
2. Perform the NTT of each half.
3. Recursively divide the coefficients in each NTT until a 2-point NTT is reached.
4. Combine the results to compute the final n-point NTT.

This process is conducted recursively, until the 2-point NTT is obtained.

Algorithm 2 Radix-2 NTT

Require: $\overline{\mathbf{f}}$, $\overline{\mathbf{G}}$, $\log n$, q, and R
1: $n \leftarrow 2^{\log n}$
2: $t \leftarrow n$
3: **for** $m \leftarrow 1$ to n **do**
4: $h \leftarrow t/2$
5: **for** $u \leftarrow 0$ to $m-1$ **do**
6: $v \leftarrow u \cdot t$
7: **for** $v \leftarrow 0$ to $h-1$ **do**
8: $\overline{r}_1 \leftarrow \overline{\mathbf{f}}[v]$
9: $\overline{r}_2 \leftarrow \overline{\mathbf{f}}[v+h]$
10: $\overline{x} \leftarrow \overline{r}_1$
11: $\overline{y} \leftarrow \overline{r}_2 * \overline{\mathbf{G}}[m+u]$
12: $\overline{r}_1 \leftarrow \overline{x} + \overline{y} \mod q$
13: $\overline{r}_2 \leftarrow \overline{x} - \overline{y} \mod q$
14: $v \leftarrow v+1$
15: **end for**
16: **end for**
17: $t \leftarrow h$
18: $m \leftarrow 2 \cdot m$
19: **end for**

3. Lazy Modular Reduction for NTT

All multiplications during the NTT computation are carried out using Montgomery multiplication. Although Montgomery multiplication performs integer modular reduction efficiently, it still involves several multiplications, additions, and bit-shift operations. Notably, we observe that the NTT computation can proceed to subsequent stages without completing modular reduction in every intermediate stage. We propose a method, *LazyNTT*, where some of the Montgomery multiplications in intermediate stages of the NTT are replaced with standard multiplications without modular reduction. Although the results in intermediate stages are not reduced modulo, q, our goal is to ensure that the final NTT result is reduced modulo, q, not necessarily in every stage. Therefore, by increasing the value of the parameter R used in Montgomery multiplication, standard multiplications can replace some of the Montgomery multiplications required in the NTT computation. Since the algorithm occasionally skips modular reduction during the NTT computation, we refer to the proposed method as *LazyNTT*.

In this paper, we propose two versions of *LazyNTT*. The first version replaces one out of two Montgomery multiplications with a standard multiplication, using both in an alternating manner during the NTT computation. Based on the order of the standard (S) and Montgomery (M) multiplications used, we refer to this version as *SM-LazyNTT*. The second version replaces the first two of three Montgomery multiplications with standard multiplications. We similarly refer to this version as *SSM-LazyNTT*. Although this paper focuses only on the SM and SSM versions of *LazyNTT*, the approach can be easily generalized to replace more Montgomery multiplications. The generalization technique is described in Section 3.5.

3.1. SM-LazyNTT

A polynomial, $f(x)$, of the length $n = 2^k$ can be expressed in radix-2 form as below.

$$f(x) = \sum_{i_{k-1}=0}^{1} x^{i_{k-1}} \Big(\sum_{i_{k-2}=0}^{1} x^{2 \cdot i_{k-2}} \big(\cdots \big(\sum_{i_0=0}^{1} x^{2^{k-1} \cdot i_0} \cdot f_{2^{k-1} \cdot i_0 + \cdots + 2 \cdot i_{k-2} + i_{k-1}} \big) \big) \Big) \tag{1}$$

For the above polynomial $f(x)$, we can count the number of required multiplications as follows.

Remark 1. *A total of $k \cdot 2^{k-1}$ multiplications are required for the NTT computation of $f(x)$.*

1. From the radix-2 butterfly structure, it is clear that the number of multiplications required in each stage is exactly half of the length, 2^{k-1}.
2. For any $i \in [1, k]$, there are 2^{i-1} sub-NTTs in the ith stage, each of size 2^{k-i+1}. For each sub-NTT, a butterfly operation is applied to pairs of elements, which involves one multiplication by a power of the root of unity.
3. Since there are 2^{k-i+1} elements in each sub-NTT, the number of multiplications required per sub-NTT is 2^{k-i}.
4. Thus, $2^{i-1} \cdot 2^{k-i} = 2^{k-1}$ multiplications are required in the ith stage. In total, $k \cdot 2^{k-1}$ multiplications are required for the entire NTT computation with k stages.

In (1), the ith coefficient f_i is multiplied by $x^{2^{k-1} \cdot i_0}$ in the innermost bracket first, corresponding to the first stage. In the second stage, it is multiplied by $x^{2^{k-2} \cdot i_1}$, and this continues until it is finally multiplied by $x^{i_{k-1}}$. As we unfold the brackets through all k stages, the exponent of x accumulates to the value $2^{k-1} \cdot i_0 + 2^{k-2} \cdot i_1 + \cdots + i_{k-1}$, which is equal to the coefficient index i. This corresponds to the binary representation of i, indicating that each bit is from i_0 to i_{k-1}. Therefore, throughout the NTT computation, f_i encounters Montgomery multiplications with modular reduction when $i_l = 1$ for $0 \leq l \leq k-1$. On the other hand, when $i_l = 0$, there is no multiplication and no modular reduction.

These multiplications can be computed using the proposed *SM-LazyNTT* algorithm based on the following principles:

1. The last multiplication encountered in each index is always performed using a Montgomery multiplication.
2. The second-to-last multiplication, i.e., the one immediately preceding the Montgomery multiplication, is replaced with a standard multiplication.
3. The multiplication preceding the standard multiplication is again performed using a Montgomery multiplication, ensuring that standard and Montgomery multiplications alternate, with the multiplication sequence always ending in a Montgomery multiplication.

We can make the following claim, which is the condition for the replacement of a Montgomery multiplication with a standard multiplication in *SM-LazyNTT*.

Claim 1. *To replace a Montgomery multiplication with a standard multiplication, the condition $R > q^2$ should be satisfied, and there should be at least one Montgomery multiplication in the subsequent stages of the NTT computation.*

Proof. In the original Montgomery multiplication, R is set to be a power of two greater than the modulus q, performing modular reduction within every multiplication.

1. In Montgomery multiplication, the result of the multiplication of two values in $\mathbb{Z}_q$ should be less $R \cdot q$. As described in Section 2.2, this ensures the correct result by subsequent conditional addition. For *SM-LazyNTT*, one standard multiplication will make the value be within the range of $[0, q^2)$, since it is performed without modular reduction. Then, if this value is multiplied by another element in $\mathbb{Z}_q$ using Montgomery multiplication, the result should be less than $R \cdot q$. This leads to the condition that $q^3 < R \cdot q$, which implies that $R > q^2$.
2. After a Montgomery multiplication is replaced by a standard multiplication without modular reduction, there should be at least one Montgomery multiplication remaining in the subsequent stages to ensure that the result is properly reduced modulo, q. Clearly, the last multiplication of each coefficient throughout the NTT should be conducted using Montgomery multiplication to perform modular reduction in the end. At the point of the multiplication just before the last Montgomery multiplication, modular reduction can be skipped because following Montgomery multiplication involves modular reduction.

□

Also, the number of replaceable positions in *SM-LazyNTT* can be determined by the following claim.

Claim 2. *Out of a total of* $k \cdot 2^{k-1}$ *Montgomery multiplications,* $(k-1) \cdot 2^{k-2}$ *can be replaced with standard multiplications.*

Proof. From the Remark 1, there are 2^{k-1} multiplications for each stage.

1. The replaceable positions among all these Montgomery multiplications depend on whether there are any Montgomery multiplications in the subsequent stages.
2. We count the number of replaceable multiplications starting from the final stage. In the final kth stage, all 2^{k-1} multiplications are performed using Montgomery multiplication to ensure that the multiplication results are correctly reduced modulo, q. In the preceding $(k-1)$th stage, half of the Montgomery multiplications (that is, 2^{k-2}) can be replaceable, since they are followed by Montgomery multiplications in the next kth stage. Thus, in the $(k-1)$th stage, the indices with Montgomery multiplication are partitioned exactly in half, depending on whether there is subsequent Montgomery multiplication in the kth stage or not. Only for the indices with subsequent Montgomery multiplication, modular reduction can be skipped and standard multiplication can be used instead of Montgomery multiplication.
3. Similarly, in the $(k-2)$th stage, 2^{k-2} Montgomery multiplications out of the total 2^{k-1} can be replaceable, since there are remaining Montgomery multiplications in the following $(k-1)$th and kth stages.
4. Thus, for every stage except the final one, half of the required multiplications can be replaced with standard multiplications. As a result, a total of $(k-1) \cdot 2^{k-2}$ Montgomery multiplications out of $k \cdot 2^{k-1}$ can be replaced, achieving asymptotically half-reduced multiplicative complexity.

□

The proposed *SM-LazyNTT* algorithm is given in Algorithm 3. The input $\bar{\mathbf{f}}$ is the coefficient vector of $f(x)$. The input $\overline{\mathbf{G}}$ stores all odd powers of the primitive $2n$-th root of unity, α^{2i+1}, in Montgomery space, as in Algorithm 2. The input $\mathbf{G}$ stores all odd powers of the primitive $2n$-th root of unity, but they are in standard space. In lines 11, 37, and 53 (blue colored) of Algorithm 3, Montgomery multiplications are used. However, in lines 21 and 43 (red colored), standard multiplications are used, providing modular reduction-free multiplication. Unlike the original NTT, where computations proceed sequentially with indices increasing by 1 (as in the "for" loop in line 7 of Algorithm 2), the proposed *SM-LazyNTT* requires specifying the positions of standard and Montgomery multiplications. To determine these positions, the index jumping vector $\mathbf{J}$ is used. Details about $\mathbf{J}$ are described in Section 3.3. The two for-loops starting in lines 33 and 49 correspond to the second-to-last stage and the final stage, respectively. In the second-to-last stage, each sub-NTT contains one standard multiplication and one Montgomery multiplication, and thus the jumping index is not used. Additionally, in the final stage, only Montgomery multiplications are performed, which is why a separate loop is created for it.

In *SM-LazyNTT*, larger intermediate values can arise. These larger values are subsequently reduced to the range of $[0, q)$ through Montgomery multiplication. After one standard multiplication, the resulting values range from 0 to q^2. By using the subtraction part in a butterfly operation, the result ranges from $-q^2$ to q^2. The conditional addition (line 8 of Algorithm 1) in Montgomery multiplication is performed to the value but uses q^2 instead of q, ensuring that the final result remains in the range from 0 to q^2.

Algorithm 3 *SM-LazyNTT*

Require: $\overline{\mathbf{f}}$, $\overline{\mathbf{G}}$, $\mathbf{G}$, $\mathbf{J}$, $\log n$, q

$n' \leftarrow 2^{(\log n - 2)}$
for $m \leftarrow 1$ to n' **do**
  $t \leftarrow n/2$
  $h \leftarrow t/2$
  for $u \leftarrow 0$ to $m-1$ **do**
    $j_1 \leftarrow 0, j_2 \leftarrow 0, v \leftarrow 0$
    for $d \leftarrow 0$ to $h-1$ **do**
      $\overline{r}_1 \leftarrow \overline{\mathbf{f}}[v]$
      $\overline{r}_2 \leftarrow \overline{\mathbf{f}}[v+t]$
      $\overline{x} \leftarrow \overline{r}_1$
      $\overline{y} \leftarrow \overline{r}_2 * \overline{\mathbf{G}}[m+u]$ ▷ Montgomery multiplication
      $\overline{\mathbf{f}}[v] \leftarrow \overline{x} + \overline{y} \mod q$
      $\overline{\mathbf{f}}[v+t] \leftarrow \overline{x} - \overline{y} \mod q$
      $v \leftarrow v + (4 - \mathbf{J}[j_1])$
      $j_1 \leftarrow j_1 + 1$
    end for
    for $d \leftarrow 0$ to $h-1$ **do**
      $\overline{r}_1 \leftarrow \overline{\mathbf{f}}[v+1]$
      $\overline{r}_2 \leftarrow \overline{\mathbf{f}}[v+1+t]$
      $\overline{x} \leftarrow \overline{r}_1$
      $\overline{y} \leftarrow \overline{r}_2 \cdot \mathbf{G}[m+u]$ ▷ Standard multiplication
      $\overline{\mathbf{f}}[v+1] \leftarrow \overline{x} + \overline{y}$
      $\overline{\mathbf{f}}[v+1+t] \leftarrow \overline{x} - \overline{y}$
      $v \leftarrow v + \mathbf{J}[j_2]$
      $j_2 \leftarrow j_2 + 1$
    end for
    $v \leftarrow v + n$
  end for
  $n \leftarrow t$
  $m \leftarrow 2 \cdot m$
end for
$v \leftarrow 0$
for $u \leftarrow 0$ to $m-1$ **do**
  $\overline{r}_1 \leftarrow \overline{\mathbf{f}}[v]$
  $\overline{r}_2 \leftarrow \overline{\mathbf{f}}[v+2]$
  $\overline{x} \leftarrow \overline{r}_1$
  $\overline{y} \leftarrow \overline{r}_2 * \overline{\mathbf{G}}[m+u]$ ▷ Montgomery multiplication
  $\overline{\mathbf{f}}[v] \leftarrow \overline{x} + \overline{y} \mod q$
  $\overline{\mathbf{f}}[v+2] \leftarrow \overline{x} - \overline{y} \mod q$
  $\overline{r}_1 \leftarrow \overline{\mathbf{f}}[v+1]$
  $\overline{r}_2 \leftarrow \overline{\mathbf{f}}[v+3]$
  $\overline{x} \leftarrow \overline{r}_1$
  $\overline{y} \leftarrow \overline{r}_2 \cdot \mathbf{G}[m+u]$ ▷ Standard multiplication
  $\overline{\mathbf{f}}[v+1] \leftarrow \overline{x} + \overline{y}$
  $\overline{\mathbf{f}}[v+3] \leftarrow \overline{x} - \overline{y}$
  $v \leftarrow v + 4$
end for
$v \leftarrow 0, m \leftarrow 2 \cdot m$
for $u \leftarrow 0$ to $m-1$ **do**
  $\overline{r}_1 \leftarrow \overline{\mathbf{f}}[v]$
  $\overline{r}_2 \leftarrow \overline{\mathbf{f}}[v+1]$
  $\overline{x} \leftarrow \overline{r}_1$
  $\overline{y} \leftarrow \overline{r}_2 * \overline{\mathbf{G}}_1[m+u]$ ▷ Montgomery multiplication
  $\overline{\mathbf{f}}[v] \leftarrow \overline{x} + \overline{y} \mod q$
  $\overline{\mathbf{f}}[v+1] \leftarrow \overline{x} - \overline{y} \mod q$
  $v \leftarrow v + 2$
end for

3.2. SSM-LazyNTT

Building on the previous section about *SM-LazyNTT*, we can further consider a method that replaces one more Montgomery multiplication. That is, up to two consecutive preceding multiplications before Montgomery multiplication can be replaced with standard multiplications. In this case, we can state the condition for the replacement as follows.

Claim 3. *To replace up to two consecutive Montgomery multiplications with standard multiplications, the condition $R > q^3$ should be satisfied, and there should be at least one Montgomery multiplication in the subsequent stages of the NTT computation.*

Proof. From the Claim 1, we know that the result of a Montgomery multiplication should be less than $R \cdot q$.

1. After two consecutive standard multiplications without modular reduction, the result lies in the range $[0, q^3)$. To perform the next Montgomery multiplication with an element in $\mathbb{Z}_q$, the condition $q^4 < R \cdot q$ should be satisfied, which implies that $R > q^3$.
2. Similarly, there should be at least one Montgomery multiplication in the subsequent NTT stages to ensure that the computed result is properly modular reduced.

□

For a polynomial, $f(x)$, in (1), the number of standard multiplications and Montgomery multiplications in the *SSM-LazyNTT* algorithm are counted by the following claim, where $HW(i)$ represents the Hamming weight of i.

Claim 4. *In the proposed SSM-LazyNTT algorithm, the number of Montgomery multiplications is*

$$\sum_{i=0}^{2^k-1} \left\lceil \frac{HW(i)}{3} \right\rceil$$

and the number of standard multiplications is

$$\sum_{i=0}^{2^k-1} \left(HW(i) - \left\lceil \frac{HW(i)}{3} \right\rceil \right).$$

Proof. We know that multiplication is performed in the positions where the bit is 1 in the binary representation of an index, i.

1. Since the final multiplication of each index should be performed by Montgomery multiplication, the possible patterns of the multiplications are combinations of M, S-M, and S-S-M.
2. Since Montgomery multiplication is only performed in the last of every three multiplications, the number of Montgomery multiplications required in each index i is given by $\left\lceil \frac{HW(i)}{3} \right\rceil$. The remaining multiplications are performed as standard multiplications, and, thus, the number of replaceable multiplications in each index is $\left(HW(i) - \left\lceil \frac{HW(i)}{3} \right\rceil \right)$.

□

The proposed *SSM-LazyNTT* algorithm is presented in Algorithm 4. It is similar to Algorithm 3, but with some key differences. First, unlike in *SM-LazyNTT*, where standard and Montgomery multiplications are equally partitioned, here, there are more standard multiplications. Therefore, we define the vectors **S** and **M**, specifying the number of multiplications required in each stage. Second, unlike in *SM-LazyNTT*, where a single jumping vector **J** is sufficient to indicate both positions of the standard multiplication and Montgomery multiplication, here, the separate index jumping vectors $\mathbf{J}_S$ and $\mathbf{J}_M$ for standard and Montgomery multiplication are required, respectively.

Algorithm 4 *SSM-LazyNTT*

Require: $\bar{\mathbf{f}}, \overline{\mathbf{G}}, \mathbf{G}, \mathbf{J}_S, \mathbf{J}_M, \log n, q$

$n' \leftarrow 2^{(\log n - 3)}$
$\mathbf{M} = \{170, 85, 43, 22, 11, 5, 2\}$
$\mathbf{S} = \{342, 171, 85, 42, 21, 11, 6\}$
for $m \leftarrow 1$ to n' **do**
  $t \leftarrow n/2$
  $i_M \leftarrow (10 - \log n), i_S \leftarrow (10 - \log n)$
  for $u \leftarrow 0$ to $m - 1$ **do**
    $j_1 \leftarrow 0, j_2 \leftarrow 0, v \leftarrow 0$
    for $d \leftarrow 0$ to $M[i_M]$ **do**
      $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v]$
      $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + t]$
      $\bar{x} \leftarrow \bar{r}_1$
      $\bar{y} \leftarrow \bar{r}_2 * \overline{\mathbf{G}}[m + u]$ ▷ Montgomery multiplication
      $\bar{\mathbf{f}}[v] \leftarrow \bar{x} + \bar{y} \mod q$
      $\bar{\mathbf{f}}[v + t] \leftarrow \bar{x} - \bar{y} \mod q$
      $v \leftarrow v + \mathbf{J}_M[j_1]$
      $j_1 \leftarrow j_1 + 1$
    end for
    for $d \leftarrow 0$ to $S[i_S]$ **do**
      $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v + 1]$
      $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + 1 + t]$
      $\bar{x} \leftarrow \bar{r}_1$
      $\bar{y} \leftarrow \bar{r}_2 \cdot \mathbf{G}[m + u]$ ▷ Standard multiplication
      $\bar{\mathbf{f}}[v + 1] \leftarrow \bar{x} + \bar{y}$
      $\bar{\mathbf{f}}[v + 1 + t] \leftarrow \bar{x} - \bar{y}$
      $v \leftarrow v + \mathbf{J}[j_2]$
      $j_2 \leftarrow j_2 + 1$
    end for
    $v \leftarrow v + n$
  end for
  $n \leftarrow t, m \leftarrow 2 \cdot m$
end for
$v \leftarrow 0$
for $u \leftarrow 0$ to $m - 1$ **do**
  $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v]$
  $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + 4]$
  $\bar{x} \leftarrow \bar{r}_1$
  $\bar{y} \leftarrow \bar{r}_2 * \overline{\mathbf{G}}[m + u]$ ▷ Montgomery multiplication
  $\bar{\mathbf{f}}[v] \leftarrow \bar{x} + \bar{y} \mod q$
  $\bar{\mathbf{f}}[v + 4] \leftarrow \bar{x} - \bar{y} \mod q$
  for $d \leftarrow 1$ to 3 **do**
    $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v + d]$
    $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + d + 4]$
    $\bar{x} \leftarrow \bar{r}_1$
    $\bar{y} \leftarrow \bar{r}_2 \cdot \mathbf{G}[m + u]$ ▷ Standard multiplication
    $\bar{\mathbf{f}}[v + d] \leftarrow \bar{x} + \bar{y}$
    $\bar{\mathbf{f}}[v + d + 4] \leftarrow \bar{x} - \bar{y}$
  end for
  $v \leftarrow v + 8$
end for
$v \leftarrow 0, m \leftarrow 2 \cdot m$
for $u \leftarrow 0$ to $m - 1$ **do**
  $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v]$
  $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + 2]$
  $\bar{x} \leftarrow \bar{r}_1$
  $\bar{y} \leftarrow \bar{r}_2 * \overline{\mathbf{G}}_1[m + u]$ ▷ Montgomery multiplication
  $\bar{\mathbf{f}}[v] \leftarrow \bar{x} + \bar{y} \mod q$
  $\bar{\mathbf{f}}[v + 2] \leftarrow \bar{x} - \bar{y} \mod q$
  $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v + 1]$
  $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + 3]$
  $\bar{x} \leftarrow \bar{r}_1$
  $\bar{y} \leftarrow \bar{r}_2 \cdot \mathbf{G}[m + u]$ ▷ Standard multiplication
  $\bar{\mathbf{f}}[v + 1] \leftarrow \bar{x} + \bar{y}$
  $\bar{\mathbf{f}}[v + 3] \leftarrow \bar{x} - \bar{y}$
end for
$v \leftarrow 0, m \leftarrow 2 \cdot m$
for $u \leftarrow 0$ to $m - 1$ **do**
  $\bar{r}_1 \leftarrow \bar{\mathbf{f}}[v]$
  $\bar{r}_2 \leftarrow \bar{\mathbf{f}}[v + 1]$
  $\bar{x} \leftarrow \bar{r}_1$
  $\bar{y} \leftarrow \bar{r}_2 * \overline{\mathbf{G}}_1[m + u]$ ▷ Montgomery multiplication
  $\bar{\mathbf{f}}[v] \leftarrow \bar{x} + \bar{y} \mod q$
  $\bar{\mathbf{f}}[v + 1] \leftarrow \bar{x} - \bar{y} \mod q$
  $v \leftarrow v + 2$
end for

However, since these jumping vectors are determined deterministically once the size of the NTT, n, is fixed, they can be precomputed and stored before the algorithm begins, and, thus, this does not pose any issue.

Similarly, the third-to-last, second-to-last, and final stages are also separated into their own for-loops. In the for-loop starting in line 34, i.e., the third-to-last stage, there is only one Montgomery multiplication per sub-NTT, and, thus, the jumping vector is not used here.

In *SSM-LazyNTT*, larger intermediate values can arise, as in *SM-LazyNTT*. A notable point is that the ranges of values differ after the first and second standard multiplications; the first standard multiplication results in values ranging from 0 to q^2, while the second results in values ranging from 0 to q^3. To ensure that these values remain positive and within correct ranges, different conditional additions (adding q^2 and q^3, respectively) are applied for each case. This ensures that the results are always positive and modularly reduced during subsequent Montgomery multiplications.

3.3. Jumping Vector

Unlike the original NTT, which uses only Montgomery multiplication, the proposed algorithms mix standard multiplication and Montgomery multiplication. Therefore, it is necessary to determine in each stage which indices use standard multiplication and which use Montgomery multiplication. In Algorithms 3 and 4, the jumping vectors $\mathbf{J}$, $\mathbf{J}_M$, and $\mathbf{J}_S$ indicate in which indices standard or Montgomery multiplication should be applied. These vectors are deterministic and can be predetermined once the NTT size, n, is defined.

As shown in Figures 1 and 2, by considering Montgomery and standard multiplications from the final stage, we can determine all the multiplications to be performed as either Montgomery multiplications or standard multiplications. Red lines represent standard multiplications, black lines represent Montgomery multiplications, and dashed lines represent no multiplication. Clearly, we do not generate or compute these vectors in each execution. Instead, they can be precomputed and stored for use.

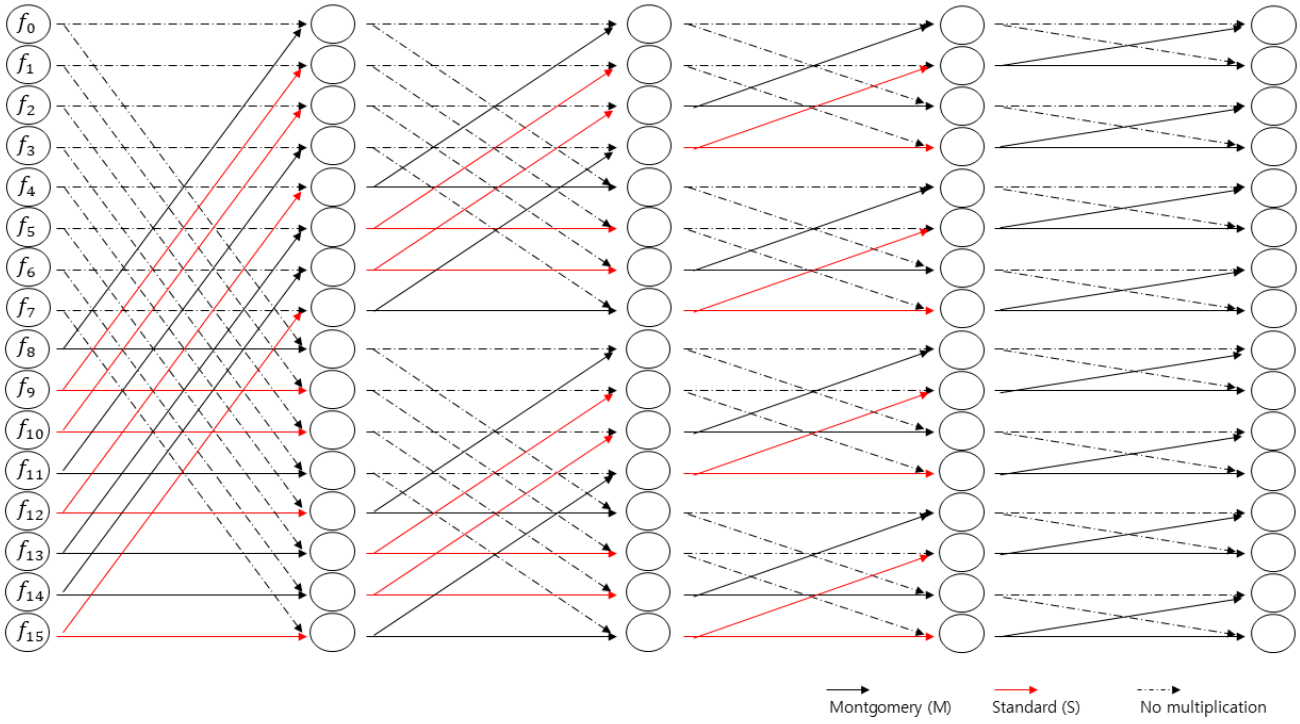


Figure 1. The positions taking standard and Montgomery multiplications in the *SM-LazyNTT* computation of the length $n = 16$.

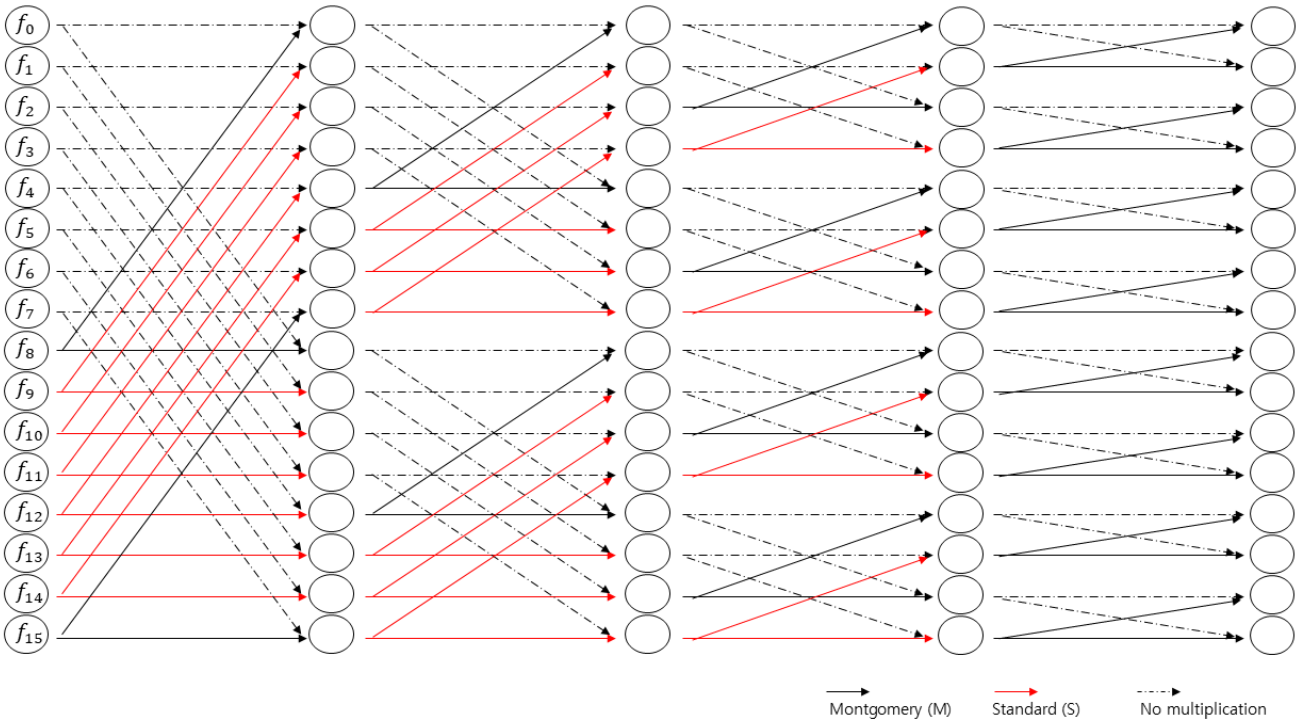


Figure 2. The positions taking standard and Montgomery multiplications in the *SSM-LazyNTT* computation of the length $n = 16$.

3.4. Memory Usage Analysis

LazyNTT may produce larger intermediate values compared to the original NTT and requires slightly more memory, as it utilizes not only $\overline{\mathbf{G}}$, the powers of the root of unity in Montgomery space, but also $\mathbf{G}$, those in standard space. However, the additional memory consumption is not expensive. For cases where $q = 12{,}289$ or $q = 3329$, a 16-bit variable is used, requiring a total of $n \times 16$ bits. Furthermore, since the jumping index values range from 1 to 3 for *SM-LazyNTT* and from 1 to 7 for *SSM-LazyNTT* with a length of $(n/2 - 1)$, eight-bit variables are sufficient to store the indices. This requires $(n/2 - 1) \times 8$ bits in software implementation, but two or three bits are actually enough, respectively. In Algorithm 4, the arrays M and S are needed to store the number of standard and Montgomery multiplications for running multiplications for loops of varying lengths. In summary, the additional memory required compared to the original NTT is not substantial, and this trade-off of using a small amount of additional memory to achieve speed improvement is quite meaningful.

3.5. Generalization

The proposed *LazyNTT* algorithm is not limited to only one or two replacements of Montgomery multiplication but can be extended to more replacements. When multiplying a total of $(t + 1)$ elements in $\mathbb{Z}_q$, we can proceed with $(t - 1)$ standard multiplications instead of Montgomery multiplications. We apply Montgomery multiplication only in the final tth multiplication to ensure that the result is in the range of $\mathbb{Z}_q$. The condition for this to hold is $R > q^t$, since the result x should satisfy $0 \le x < q^{t+1} \le q \cdot R$.

In general, for an algorithm that uses $t - 1$ standard multiplications out of a total of t required multiplications, referred to as $S^{t-1}M$-*LazyNTT*, the following claim holds.

Claim 5. *In the $S^{t-1}M$-LazyNTT algorithm, the number of Montgomery multiplications is*

$$\sum_{i=0}^{2^k-1} \left\lceil \frac{HW(i)}{t} \right\rceil,$$

and the number of standard multiplications is

$$\sum_{i=0}^{2^k-1}\left(HW(i)-\left\lceil\frac{HW(i)}{t}\right\rceil\right).$$

Proof. Similarly to Claim 4, we know that Montgomery multiplication is only performed in the last of all consecutive multiplications in each index. Thus, the number of required Montgomery multiplications in each index, i, is given by $\left\lceil\frac{HW(i)}{t}\right\rceil$. The remaining multiplications can be performed by standard multiplications, and, thus, the number of replaceable multiplications in each index is $\left(HW(i)-\left\lceil\frac{HW(i)}{t}\right\rceil\right)$. □

In Falcon [5], the prime modulus and Montgomery scaling factor are $q = 12{,}289$ and $R = 2^{16}$. In the original Falcon implementation [27], since each multiplication is performed using Montgomery multiplication, R only needs to be a power of two larger than q. However, to apply the proposed *SM-LazyNTT* and *SSM-LazyNTT* algorithms, given that the size of q is 14 bits, R should be at least 2^{28} and 2^{42}, respectively. For the Kyber parameter $q = 3329$, given that the size of q is 12 bits, R should be at least 2^{24} and 2^{36}, respectively.

In the original NTT, modular reduction is performed after each multiplication, ensuring that no value exceeds the modulus q. However, in the proposed algorithm, modular reductions are skipped in some of the intermediate stages, requiring the storage of larger values, which sometimes necessitates a 64-bit variable. For *SM-LazyNTT* with $q = 12{,}289$, after one standard multiplication, the result can reach up to q^2, which corresponds to a maximum of 28 bits. To perform the subsequent Montgomery multiplication of this value, a variable capable of holding up to 42 bits is required. In the case of *SSM-LazyNTT*, after two standard multiplications, the value can reach up to 42 bits, and to perform the subsequent Montgomery multiplication, a variable capable of holding up to 56 bits is necessary.

$S^{t-1}M$*-LazyNTT* can be the generalized version of the proposed *LazyNTT*; however, there are some practical obstacles for its implementation. First, variables larger than 64 bits are required for going beyond *SSM-LazyNTT*. For example, in *SSSM-LazyNTT*, R should be 2^{56} for $q = 12{,}289$ to perform three consecutive standard multiplications, and following Montgomery multiplication would demand a variable capable of $70 = 56 + 14$ bits for its computation. Thus, while it is theoretically possible to extend the algorithm, the need for higher bit-width processing may introduce overhead, potentially diminishing the performance benefits. Additionally, memory usage would increase accordingly to store these larger values. These are more seriously considered in both hardware and software implementations of the generalized version of *LazyNTT*.

4. Implementation Results

We implemented two versions of the proposed algorithm, *SM-LazyNTT* and *SSM-LazyNTT*, using the NTT function in Falcon implementation [27]. The experiments were conducted on a Linux server with an AMD Ryzen 9 5950X 16-Core Processor CPU with 64 GB of memory. We used a "GCC" compiler, and no compiler optimization options were applied. The number of iterations was one million. We compared the proposed algorithms with the original NTT algorithm in terms of the number of cycles for different parameters, q and n. The implementation source code is publicly available at https://github.com/Yongwoo-Lee-ccl/fast-ntt (accessed on 24 October 2024).

Figures 3 and 4 show the performance comparison results of the proposed algorithms, *SM-LazyNTT* and *SSM-LazyNTT*, with the original NTT in Falcon [27]. Clearly, the required cycle counts were reduced in the proposed algorithms. As shown in Tables 2 and 3, we achieved a speedup gain of around 28% with $q = 12{,}289$ and 24% with $q = 3329$ for *SSM-LazyNTT* and around 9% with both parameters for *SM-LazyNTT*.

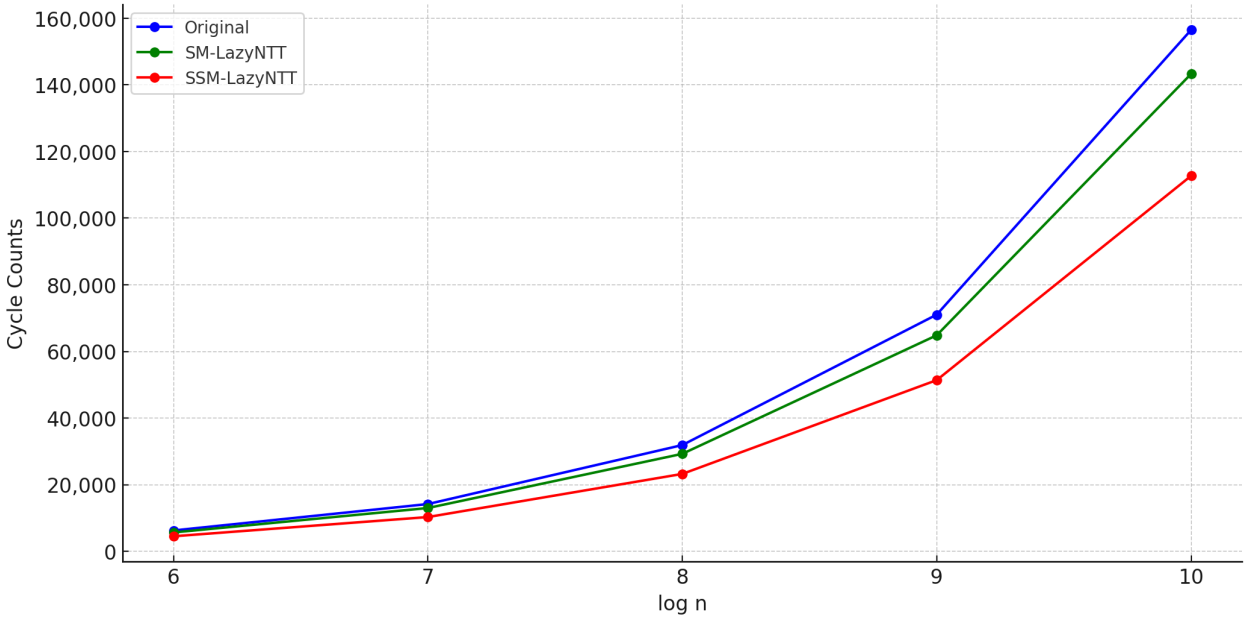


Figure 3. A performance comparison of the original and the proposed *SM-LazyNTT* and *SSM-LazyNTT* algorithms with $q = 12{,}289$.

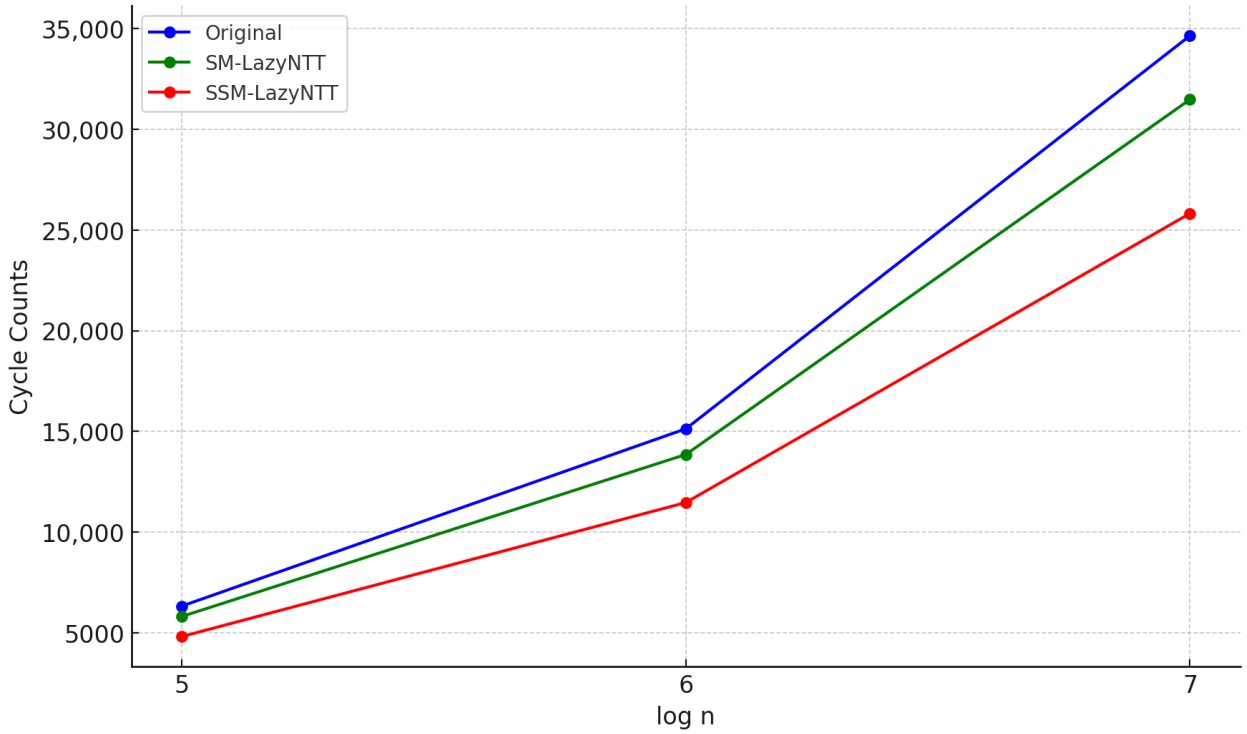


Figure 4. A performance comparison of the original and the proposed *SM-LazyNTT* and *SSM-LazyNTT* algorithms with $q = 3329$.

Table 2. A performance comparison of the original and the proposed *SM-LazyNTT* and *SSM-LazyNTT* algorithms based on the number of cycles with $q = 12{,}289$.

	$q = 12{,}289$				
$\log n$	6	7	8	9	10
SM-LazyNTT	5660	13,000	29,258	64,843	143,297
SSM-LazyNTT	4514	10,287	23,230	51,384	112,701
Original	6255	14,187	31,889	71,041	156,509
Speedup (SM/Original)	0.905	0.916	0.918	0.913	0.916
Speedup (SSM/Original)	0.722	0.725	0.728	0.723	0.720

Table 3. A performance comparison of the original and the proposed *SM-LazyNTT* and *SSM-LazyNTT* algorithms based on the number of cycles with $q = 3329$.

	$q = 3329$		
$\log n$	5	6	7
SM-LazyNTT	5823	13,860	31,469
SSM-LazyNTT	4826	11,476	25,816
Original	6339	15,135	34,636
Speedup (SM/Original)	0.919	0.916	0.909
Speedup (SSM/Original)	0.761	0.758	0.745

5. Discussion and Conclusions

As the NTT plays an important role in lattice-based cryptosystems, reducing its computational load can be an important contribution. We proposed a new and faster NTT algorithm, *LazyNTT*, and its two versions of implementation. The implementation results showed that *SSM-LazyNTT* and *SM-LazyNTT* were practically efficient, improving the runtime of the NTT by up to 28% and 9%, respectively. Moreover, the proposed technique could be naturally generalized to replace more than two Montgomery multiplications with standard integer multiplications. Even with the replacement of just one or two multiplications, we observed meaningful computational complexity improvements compared to the original NTT and confirmed the proper functionality of the proposed methods. Furthermore, the proposed *LazyNTT* only replaces some Montgomery multiplications with standard multiplications, without changing any other structures of the NTT; the runtime improvement can be achieved without compromising security.

In this paper, *LazyNTT* was implemented for PQC parameters, especially Falcon and Kyber. However, it can also be extended to other cryptographic schemes, such as HE. For example, HE typically employs larger parameters and longer NTT lengths compared to PQC. Consequently, *LazyNTT* could replace a larger number of multiplications, potentially leading to a higher reduction in the number of cycles. Furthermore, when implementing software with large parameters requiring 128-bit variables or more, although there may be additional costs, improved speed with *LazyNTT* compensates the costs and introduces a suitable implementation for homomorphic encryption.

Some future works can be considered in the following directions. First, implementing the generalized version of the proposed *LazyNTT* can be a promising research topic. Second, this approach can be extended not only to the radix-2 NTT but also by mixing NTTs that use various sizes of radix. Third, this technique can be applied to the inverse NTT, thereby improving the overall polynomial multiplication process. Fourth, it can be applied to various lattice-based cryptographic schemes, such as digital signatures, homomorphic encryption, and zero knowledge proofs. Finally, implementing *LazyNTT* on various platforms, such as GPUs or FPGAs, can be a research direction for broader applications.

Author Contributions: G.K. developed and improved the overall idea of this paper, wrote the C code implementation, and derived the experimental results. E.S. initially proposed the idea of this paper. E.S. and Y.L. contributed to the programming and discussed the idea together. Y.-S.K. and J.-S.N. guided the paper's overall direction and structure. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) grant funded by the Korean government(MSIT) (RS-2024-00399401, Development of Quantum-Safe Infrastructure Migration and Quantum Security Verification Technologies).

Data Availability Statement: The data presented in this study are openly available in [fast-ntt] at https://github.com/Yongwoo-Lee-ccl/fast-ntt.git (accessed on 24 October 2024).

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of this study; in the collection, analysis, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Rivest, R.L.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
2. Shor, P.W. Algorithms for quantum computation: Discrete logarithms and factoring. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 20–22 November 1994.
3. Alagic, G.; Apon, D.; Cooper, D.; Dang, Q.; Dang, T.; Kelsey, J.; Lichtinger, J.; Miller, C.; Moody, D.; Peralta, R.; et al. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*; US Department of Commerce: Washington, DC, USA, **2022**.
4. Montgomery, P.L. Modular multiplication without trial division. *Math. Comput.* **1985**, *44*, 519–521. [CrossRef]
5. Fouque, P.A.; Hoffstein, J.; Kirchner, P.; Lyubashevsky, V.; Pornin, T.; Prest, T.; Ricosset, T.; Seiler, G.; Whyte, W.; Zhang, Z.; et al. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. *Post-Quantum Cryptogr. Stand.* **2018**, *36*, 1–75.
6. Bos, J.; Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schanck, J.M.; Schwabe, P.; Seiler, G.; Stehlé, D. Crystals-Kyber: A CCA-secure module-lattice-based KEM. In Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, UK, 24–26 April 2018; pp. 353–367.
7. Yaman, F.; Mert, A.C.; Öztürk, E.; Savaş, E. A hardware accelerator for polynomial multiplication operation of Crystals-Kyber PQC scheme. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 1020–1025.
8. Nannipieri, P.; Di Matteo, S.; Zulberti, L.; Albicocchi, F.; Saponara, S.; Fanucci, L. A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up Crystals algorithms. *IEEE Access* **2021**, *9*, 150798–150808. [CrossRef]
9. Nguyen, T.H.; Kieu-Do-Nguyen, B.; Pham, C.K.; Hoang, T.T. High-speed NTT Accelerator for Crystals-Kyber and Crystals-Dilithium. *IEEE Access* **2024**, *12*, 34918–34930. [CrossRef]
10. Li, B.; Yan, Y.; Wei, Y.; Han, H. Scalable and parallel optimization of the number theoretic transform based on FPGA. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2023**, *32*, 291–304. [CrossRef]
11. Alsuhli, G.; Saleh, H.; Al-Qutayri, M.; Mohammad, B.; Stouraitis, T. Efficient twiddle factor generation for post quantum cryptography Falcon-based number theoretic transform. *Authorea Prepr.* **2024**. [CrossRef]
12. Zhang, C.; Liu, D.; Liu, X.; Zou, X.; Niu, G.; Liu, B.; Jiang, Q. Towards efficient hardware implementation of NTT for Kyber on FPGAs. In Proceedings of the 2021 IEEE International Symposium on Circuits and Systems (ISCAS), Daegu, Republic of Korea, 22–28 May 2021; pp. 1–5.
13. Yang, Y.; Kuppannagari, S.R.; Kannan, R.; Prasanna, V.K. NTTGen: A framework for generating low latency NTT implementations on FPGA. In Proceedings of the 19th ACM International Conference on Computing Frontiers, Turin, Italy, 17–22 May 2022; pp. 30–39.
14. Derya, K.; Mert, A.C.; Öztürk, E.; Savaş, E. CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication. *Microprocess. Microsystems* **2022**, *89*, 104451. [CrossRef]
15. Bradbury, J.; Drucker, N.; Hillenbrand, M. NTT software optimization using an extended Harvey butterfly. *Cryptol. Arch.* **2021**, 1396. Available online: https://eprint.iacr.org/2021/1396 (accessed on 24 October 2024).
16. Longa, P.; Naehrig, M. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Proceedings of the Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, 14–16 November 2016; Proceedings 15; Springer: Berlin/Heidelberg, Germany, 2016; pp. 124–139.
17. Lyubashevsky, V.; Seiler, G. NTTRU: Truly fast NTRU using NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2019**, 180–201. [CrossRef]
18. Özerk, Ö.; Elgezen, C.; Mert, A.C.; Öztürk, E.; Savaş, E. Efficient number theoretic transform implementation on GPU for homomorphic encryption. *J. Supercomput.* **2022**, *78*, 2840–2872. [CrossRef]

19. Goey, J.Z.; Lee, W.K.; Goi, B.M.; Yap, W.S. Accelerating number theoretic transform in GPU platform for fully homomorphic encryption. *J. Supercomput.* **2021**, *77*, 1455–1474. [CrossRef]
20. Kim, S.; Jung, W.; Park, J.; Ahn, J.H. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs. In Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC), Beijing, China, 27–30 October 2020; pp. 264–275.
21. Kim, S.; Lee, K.; Cho, W.; Nam, Y.; Cheon, J.H.; Rutenbar, R.A. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 3–6 May 2020; pp. 56–64.
22. Duong-Ngoc, P.; Kwon, S.; Yoo, D.; Lee, H. Area-efficient number theoretic transform architecture for homomorphic encryption. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, *70*, 1270–1283. [CrossRef]
23. Cooley, J.W.; Tukey, J.W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **1965**, *19*, 297–301. [CrossRef]
24. Agarwal, R.C.; Burrus, C.S. Number theoretic transforms to implement fast digital convolution. *Proc. IEEE* **1975**, *63*, 550–560. [CrossRef]
25. Chu, E.; George, A. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*; CRC Press: Boca Raton, FL, USA, 1999.
26. Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schwabe, P.; Seiler, G.; Stehlé, D. Crystals-Dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2018*, 238–268. [CrossRef]
27. NTT Code from the Falcon Project. Available online: https://falcon-sign.info/impl/falcon.h.html (accessed on 24 October 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.