Improved Plantard Arithmetic for Lattice-based Cryptography

Junhao Huang[1], Jipeng Zhang[2], Haosong Zhao[1], Zhe Liu[2,3], Ray C. C. Cheung[4], Çetin Kaya Koç[5,2], Donglong Chen[1*]

[1] Guangdong Provincial Key Laboratory of Interdisciplinary Research and Application for Data Science, BNU-HKBU United International College, Zhuhai, China
huangjunhao@uic.edu.cn, zhaohaosonguic@gmail.com, donglongchen@uic.edu.cn
[2] Nanjing University of Aeronautics and Astronautics, Nanjing, China jp-zhang@outlook.com
[3] Zhejiang Lab, Hangzhou, China zhe.liu@nuaa.edu.cn
[4] City University of Hong Kong, Hong Kong, China r.cheung@cityu.edu.hk
[5] University of California Santa Barbara, Santa Barbara, USA cetinkoc@ucsb.edu
[*] Corresponding Author.

Abstract. This paper presents an improved Plantard's modular arithmetic (Plantard arithmetic) tailored for Lattice-Based Cryptography (LBC). Based on the improved Plantard arithmetic, we present faster implementations of two LBC schemes, Kyber and NTTRU, running on Cortex-M4. The intrinsic advantage of Plantard arithmetic is that one multiplication can be saved from the modular multiplication of a constant. However, the original Plantard arithmetic is not very practical in LBC schemes because of the limitation on the unsigned input range. In this paper, we improve the Plantard arithmetic and customize it for the existing LBC schemes with theoretical proof. The improved Plantard arithmetic not only inherits its aforementioned advantage but also accepts signed inputs, produces signed output, and enlarges its input range compared with the original design. Moreover, compared with the state-of-the-art Montgomery arithmetic, the improved Plantard arithmetic has a larger input range and smaller output range, which allows better lazy reduction strategies during the NTT/INTT implementation in current LBC schemes. All these merits make it possible to replace the Montgomery arithmetic with the improved Plantard arithmetic in LBC schemes on some platforms. After applying this novel method to Kyber and NTTRU schemes using 16-bit NTT on Cortex-M4 devices, we show that the proposed design outperforms the known fastest implementation that uses Montgomery and Barrett arithmetic. Specifically, compared with the state-of-the-art Kyber implementation, applying the improved Plantard arithmetic in Kyber results in a speedup of 25.02% and 18.56% for NTT and INTT, respectively. Compared with the reference implementation of NTTRU, our NTT and INTT achieve speedup by 83.21% and 78.64%, respectively. As for the LBC KEM schemes, we set new speed records for Kyber and NTTRU running on Cortex-M4.

Keywords: Kyber, NTTRU, NTT, Cortex-M4, modular arithmetic, lattice-based cryptography

1 Introduction

Quantum computers are being developed rapidly. Google [AAB+19] reported their 53-qubit processor in 2019. Two years later, a Chinese research team [WBC+21] announced a 66-qubit processor named *Zuchongzhi*. As is known, Shor's algorithm can break the conventional public-key cryptosystems such as RSA and ECC on a sufficiently large quantum computer. Theoretically, a quantum computer with a few thousand qubits should

break the 2048-bit RSA. However, there are still noise issues not being well understood. A recent research [GE21] shows that factoring 2048-bit integer in RSA within 8 hours needs a quantum computer with about 20 million noisy qubits. This implies the current progress of quantum computers is still far from breaking the currently deployed 2048-bit RSA or 256-bit ECC. On the other hand, IBM's roadmap of quantum computers [Hac20] claims that the number of qubits could be more than doubled per year, making a one million-qubit machine possible in 2030. The progress of experimental quantum computers and the surprising developments in the future motivated the cryptographic community to expedite the study of post-quantum cryptographic algorithms to find suitable alternatives to the traditional public-key cryptosystems within a decade or less.

In order to promote the development of post-quantum cryptography (PQC), NIST initiated a standardization project in 2016 to solicit, evaluate, and standardize the post-quantum cryptographic algorithms. Until now, in the third round of the standardization process, seven finalists, including four key encapsulation mechanisms (KEM) and three digital signature algorithms, have been selected. Among these algorithms, three out of the four KEM finalists (i.e., Kyber [ABD+20], Saber [BMD+20], and NTRU [CDH+20]) are from lattice-based cryptography (LBC), which indicates that LBC is competitive in terms of security and implementation efficiency. Evaluating the implementation efficiency of these candidates is an essential task [AASA+20] not only in the third round of the NIST standardization project but also in the future commercial deployment.

Among all the operations in LBC, polynomial multiplication is one of the core operations. Therefore, a relatively low complexity method called Number Theoretic Transform (NTT) algorithm is widely used to reduce the computational complexity of polynomial multiplication. During the algorithm design of Kyber and NTTRU, the underlying ring was properly chosen so that NTT could be used. Modular multiplication is the dominant operation inside NTT, and the most commonly used modular multiplication algorithms are Montgomery multiplication [Mon85] and Barrett multiplication [Bar86], which were originally designed for large-moduli cryptosystems like RSA and ECC. However, the moduli of the NIST LBC candidates are relatively small, for example, 3329 for Kyber [ABD+20], 12289 for Newhope [ADPS16], and 7681 for NTTRU [LS19]. Therefore, the product of two polynomial coefficients will extend to at most a word size (32 or 64 bits). Based on the word size characteristics of these moduli, Thomas Plantard proposed an efficient specialized word size modular multiplication (Plantard multiplication) [Pla21]. The proposed Plantard multiplication by a constant introduces an $l \times 2l$-bit multiplication operation ($l = 16/32$ bit). Plantard suggested that if the $l \times 2l$-bit multiplication can be implemented in one multiplication instruction in the target platform, the Plantard multiplication by a constant would consume one multiplication fewer than its Montgomery as well as Barrett counterparts and could be well deployed in LBC schemes.

Motivation. Plantard [Pla21] purely focused on the theoretical design yet leaves much room for further exploration. The original Plantard arithmetic cannot be efficiently applied to the existing LBC schemes for the following reasons. First, the original Plantard multiplication ([Pla21, Algorithm 8]) only presents a solution for multiplying unsigned integers mod an odd modulus q. Previous literature [Sei18, BKS19, ABCG20, GKS21, AHKS22] has shown that using unsigned integers in the NTT implementation of LBC schemes is inefficient because it would introduce an extra addition into each butterfly unit. Second, the input range of the original Plantard multiplication is mere $[0, q]$, which requires expensive modular reductions on the polynomial coefficients after each NTT layer.

Previous reports have offered solutions to these problems by adapting the Montgomery arithmetic. Alkim et al. [ADPS16] showed that the large input range of Montgomery reduction enables the so-called lazy reduction technique to reduce the use of expensive modular reduction. Seiler et al. [Sei18] proposed a signed version of Montgomery re-

duction on the Intel platform, which further speeds up the butterfly units. The follow-up work [BKS19, ABCG20, GKS21] further extended the signed Montgomery reduction to the Cortex-M4 platform. The fastest Montgomery reduction for 16-/32-bit moduli [ABCG20, GKS21, AHKS22] only consumes 2 cycles on the Cortex-M4 platform. Even though Montgomery arithmetic could provide an efficient solution to most of the LBC schemes, this paper shows that Plantard arithmetic could be a potential alternative to Montgomery arithmetic in the LBC schemes with l-bit odd modulus, given that the target platform could implement the $l \times 2l$-bit multiplication in one multiplication.

Our Contributions. This paper aims to improve, customize, and implement the Plantard arithmetic for the existing LBC schemes. Our contributions are threefold:

- First, we present an improved Plantard arithmetic for signed integers with its correctness proof. The proposed method is inspired by the observation that the moduli in the existing LBC schemes are normally several times smaller than the modulus constraint in the original Plantard arithmetic.
- Second, we detail the advantages of the improved Plantard arithmetic over its original version, Montgomery and Barrett arithmetic. With some tweaks over the original algorithm, the improved Plantard arithmetic not only supports signed integers but also extends its input range and narrows down its output range, thus enabling better lazy reduction strategies in NTT/INTT. Besides, it inherits the advantage of the Plantard multiplication, i.e., saving one multiplication when one of the two operands is a constant. All these merits make it possible to replace the Montgomery and Barrett arithmetic with the improved Plantard arithmetic in LBC schemes on some platforms.
- Third, we present an efficient implementation of the improved Plantard arithmetic for 16-bit odd moduli using the **smulw{b,t}** instruction to perform the 16×32-bit multiplication on Cortex-M4. By replacing the state-of-the-art Montgomery and Barrett arithmetic with the improved Plantard arithmetic, we demonstrate that the proposed design enables more efficient polynomial arithmetic, namely NTT, INTT, and modular reduction, resulting in faster Kyber and NTTRU implementations on Cortex-M4. To the best of our knowledge, this is the first assembly implementation of NTTRU on Cortex-M4, and we set new speed records for Kyber and NTTRU.

It should be noted that our optimizations for Kyber and NTTRU are not limited to Cortex-M4 and can be extended to Cortex-M7 as well as some 32-bit microcontrollers without SIMD extensions, e.g., the SiFive Freedom E310 with a 32-bit E31 RISC-V core [SiF] (see Subsection 4.3 for more details about the extensibility of our optimizations).

Code. The implementations of Kyber and NTTRU are open source and available at https://github.com/UIC-ESLAS/ImprovedPlantardArithmetic.

2 Preliminaries

In this section, we first briefly introduce the variants of the Learning With Errors (LWE) problem. Then, the algorithms of Kyber and NTTRU are described. Finally, the details of the NTT and related modular arithmetic are given.

The Learning With Errors (LWE) problem is systematically defined by Regev in [Reg05] for constructing public-key cryptosystems under the quantum random oracle model. Lyubashevsky et al. [LPR10] introduced a Ring algebraic structure for LWE, in effect solving the inefficient problem of LWE-based schemes; this variant is called Ring-LWE. More recently, Langlois and Stehlé [LS15] proposed the Module-LWE problem,

which bridges LWE and Ring-LWE. One of the NIST PQC KEM finalists, Kyber, is constructed based on Module-LWE. The original NTRU, which was proposed in 1996 and first published in 1998 [HPS98] by Hoffstein, Pipher, and Silverman, has gone through a long research history. As one of the KEM finalists, NTRU has been significantly improved in terms of security and efficiency compared with the original design.

2.1 Kyber

The security of Kyber is based on the Module-LWE problem, whose formal definition is given as follows:

$$(\mathbf{A}, \mathbf{b} = \mathbf{A}^T\mathbf{s} + \mathbf{e}). \tag{1}$$

Here, the secret vector $\mathbf{s}$ is sampled from a centered binomial distribution $B_\eta(R_q^{k\times 1})$, where k is the dimension of the underlying Module-LWE problem. The public matrix $\mathbf{A}$ is sampled from a uniform random distribution $\mathcal{U}(R_q^{k\times k})$. The decisional Module-LWE problem indicates that $(\mathbf{A}, \mathbf{b})$ is computationally indistinguishable from a uniform random sampling.

The polynomial ring used within Kyber is $R_q = \mathbb{Z}_q[X]/(X^n+1)$ with $q = 3329$ and $n = 256$ across all parameter sets. Each polynomial coefficient in R_q can be accommodated in a 16-bit integer, which enables a 16-bit NTT polynomial multiplication. Kyber constructs a CCA-secure KEM from a CPA-secure public key encryption (PKE). Kyber PKE consists of key generation (Algorithm 1), encryption (Algorithm 2), and decryption (Algorithm 3). We refer readers to Kyber's specification [ABD+20] for more details about the CCA-secure KEM. The core operation of Kyber in key generation and encryption is the matrix-vector multiplication, i.e., $\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s})$ and $\hat{\mathbf{A}}^T \circ \hat{\mathbf{t}}$, where each element of the matrix and vector is a polynomial.

Algorithm 1 Kyber.PKE Key Generation [ABD+20]

Output: $pk = (\hat{\mathbf{b}}, \rho), sk = \hat{\mathbf{s}}$
1: $seed \leftarrow \{0, \cdots, 255\}^{32}$
2: $\rho, \sigma \leftarrow \mathrm{SHAKE256}(64, seed)$
3: $\hat{\mathbf{A}} \leftarrow \mathrm{GenMatrixA}(\rho)$
4: $\mathbf{s} \leftarrow \mathrm{SampleVec}(\sigma, 0)$
5: $\mathbf{e} \leftarrow \mathrm{SampleVec}(\sigma, 1)$
6: $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$
7: **return** $pk = (\hat{\mathbf{b}}, \rho), sk = \hat{\mathbf{s}}$

Algorithm 2 Kyber.PKE Encryption [ABD+20]

Input: $pk = (\hat{\mathbf{b}}, \rho)$, message $\mu \in R_q$, seed $coin \in \{0, \cdots, 255\}^{32}$
Output: Ciphertext $(\mathbf{u}', h)$
1: $\hat{\mathbf{A}} \leftarrow \mathrm{GenMatrixA}(\rho)$
2: $\mathbf{s}' \leftarrow \mathrm{SampleVec}(coin, 0)$
3: $\mathbf{e}' \leftarrow \mathrm{SampleVec}(coin, 1)$
4: $\mathbf{e}'' \leftarrow \mathrm{SampleVec}(coin, 2)$
5: $\hat{\mathbf{t}} \leftarrow \mathrm{NTT}(\mathbf{s}')$
6: $\mathbf{u} \leftarrow \mathrm{NTT}^{-1}\left(\hat{\mathbf{A}}^T \circ \hat{\mathbf{t}}\right) + \mathbf{e}'$
7: $\mathbf{v}' \leftarrow \mathrm{NTT}^{-1}\left(\hat{\mathbf{b}}^T \circ \hat{\mathbf{t}}\right) + \mathbf{e}'' + \mu$
8: **return** $(\mathbf{u}' = \mathrm{Compress}(\mathbf{u}), h = \mathrm{Compress}(\mathbf{v}'))$

Algorithm 3 Kyber.PKE Decryption [ABD+20]

Input: Ciphertext $c = (\mathbf{u}', h)$, secret key $sk = \hat{\mathbf{s}}$
Output: Message $\mu \in R_q$
1: $\mathbf{u} \leftarrow \text{Decompress}(\mathbf{u}')$
2: $\mathbf{v}' \leftarrow \text{Decompress}(h)$
3: **return** $\mu = \mathbf{v}' - \text{NTT}^{-1}\left(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})\right)$

2.2 NTRU and NTTRU

NTRU, as a finalist in the third round of the NIST PQC project, is the combination of two first-round candidates, NTRUEncrypt [ZCHW] and NTRU-HRSS-KEM [SHRS]. The KEM specified in the NTRU's specification is constructed using a generic transformation of a correct deterministic public-key encryption scheme (correct DPKE)[CDH+20]. The polynomial arithmetic of NTRU operates in three polynomial rings $\mathbb{Z}_3[x]/\Phi_{\mathbf{n}}, \mathbb{Z}_q[x]/\Phi_{\mathbf{n}}$, and $\mathbb{Z}_q[x]/\left(\mathbf{\Phi_1} \cdot \mathbf{\Phi_n}\right)$ with $\Phi_1 = (x-1)$ and $\Phi_{\mathbf{n}} = \left(x^{n-1} + x^{n-2} + \cdots + 1\right)$.

NTTRU[LS19], an NTT-friendly variant of NTRU, is constructed using a partially correct probabilistic public-key encryption scheme (partially correct PPKE). The speed of NTTRU is faster than other NTRU variants, including the third-round finalist NTRU and alternate candidate NTRU Prime. However, the authors only provided security proof under the classical random oracle model (ROM) instead of the quantum random oracle model (QROM).

The public key encryption scheme of NTTRU consists of key generation (Algorithm 4), encryption (Algorithm 5), and decryption (Algorithm 6). The polynomial ring of NTTRU is $R_q = \mathbb{Z}_q[X]/(X^{768} - X^{384} + 1)$, where $q = 7681$. The β_2^{768} is a binomial distribution, which generates $a_1, a_2, a_3, a_4 \leftarrow \{0,1\}^{768}$, and outputs $a_1 + a_2 - a_3 - a_4 \bmod^{\pm} 3 \in R_q$. The modular reduction ($\bmod^{\pm} q$) maps integers to the range $(-\frac{q}{2}, \frac{q}{2})$. The NTT's suitability for NTTRU comes from its polynomial ring $\mathbb{Z}_{7681}[X]/(X^{768} - X^{384} + 1)$, and we will give more details in Subsection 2.3.

Algorithm 4 NTTRU.PKE Key Generation [LS19]

Output: $sk = f, pk = h$
1: $f' \leftarrow \beta_2^{768}$
2: $f := 3f' + 1$
3: if f is not invertible in R_q, restart
4: $g \leftarrow \beta_2^{768}$
5: $h := 3g/f$
6: **return** $(sk = f, pk = h)$ ▷ both sk and pk are in NTT representation

Algorithm 5 NTTRU.PKE Encryption [LS19]

Input: Message m, randomness r, $pk = h$
Output: Ciphertext c
1: $c := hr + m$ (computed in NTT representation)
2: **return** c

2.3 NTT

In this paper, NTT is classified into 16-bit NTT [LS19, ABD+20] and 32-bit NTT [CHK+21, ZHLR22] according to the modulus size. For example, the modulus size of Kyber and NTTRU are both smaller than 2^{16}, so the NTT of these moduli is classified into the 16-bit

Algorithm 6 NTTRU.PKE Decryption [LS19]

Input: Ciphertext c, $sk = f$ (both in NTT representation)
Output: Message m
1: $m := \left(cf \bmod^{\pm} q\right) \bmod^{\pm} 3$
2: **return** m

NTT. The modulus of Dilithium is $q = 8380417$, which is larger than 16-bit and smaller than 23-bit; thus, the NTT of modulus $q = 8380417$ is classified into the 32-bit NTT. We mainly focus on the 16-bit NTT in this paper.

NTT multiplication over the polynomial ring $\mathbb{Z}_q[X]/f(X)$ is based on the fact that the polynomial $f(X)$ can be factored as

$$f(X) = \prod_{i=0}^{n-1} f_i(X) (\bmod q),$$

where $f_i(X)$ are small-degree (degree-0 for a complete-NTT, higher degree for an incomplete-NTT [CHK+21]) polynomials. The polynomial multiplication of $a, b \in \mathbb{Z}_q[X]/f(X)$ is first calculated using NTT as

$$a_i = a \bmod f_i(X) \text{ and } b_i = b \bmod f_i(X) \tag{2}$$

for $i = 0, \cdots, n-1$. Then we compute the base multiplication as $a_0 b_0, \cdots, a_{n-1} b_{n-1}$. Finally, using the inverse NTT (INTT), we find the result polynomial c such that $c = ab \bmod f(X)$.

For Kyber, the polynomial $f(X) = X^{256} + 1$ can be factored, with the help of a primitive 256-th root of unity ζ, as

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \zeta^{2i+1}\right).$$

This factorization is incomplete since $f(X)$ is factored as degree-1 polynomials instead of degree-0. Therefore, the NTT of Kyber is a 7-layer incomplete-NTT.

For NTTRU, the polynomial $f(X) = X^{768} - X^{384} + 1$ is initially split into $(X^{384} + 684)(X^{384} - 685)$, then all the way down to irreducible polynomials $X^3 \pm r$. The NTT of NTTRU consists of 8 layers of butterflies, and we refer readers to [LS19] for more details.

The commonly used computing structures of NTT are Cooley-Tukey (CT) butterfly [CT65] and Gentleman-Sande (GS) butterfly [GS66]. The CT butterfly accepts inputs in normal order and generates outputs in bit-reverse order. In contrast, the GS butterfly takes inputs in bit-reverse order and produces outputs in normal order. Therefore, a hybrid structure which uses CT butterfly for NTT and GS butterfly for INTT will accept inputs and generate outputs both in normal order without any bit-reversal operations.

2.4 Modular Arithmetic

In this paper, we divide the modular reduction into two types: mod and $\bmod^{\pm}$, where $c \bmod q$ obtains a result in $[0, q)$ while $c \bmod^{\pm} q$ gets a signed result in $(-\frac{q}{2}, \frac{q}{2})$. The "modular reduction" in this paper refers to both types of modular reduction. The modular multiplication $c = a \cdot b \bmod q$ is normally divided into two parts: the multiplication $c = a \cdot b$ and the modular reduction $c \bmod q$. The modular reduction $c \bmod q$ can be done by $c - \lfloor c/q \rfloor q$, where the division operation is non-constant-time in modern computers. The non-constant-time characteristic of division prevents it from being used in the implementation of cryptographic algorithms due to security concerns. The commonly used constant time

modular reduction algorithms are Montgomery reduction [Mon85] and Barrett reduction [Bar86]. Their basic idea is to replace the non-constant-time division with constant-time multiplications and shifts.

In current LBC schemes, the known fastest modular arithmetic is the signed version of Montgomery multiplication [Sei18], which is given in Algorithm 7. The constant β is often set to word size 2^l so that the modulus q fits in one word. For instance, $\beta = 2^{16}$ when $q < 2^{16}$ and $\beta = 2^{32}$ when $2^{16} \leq q < 2^{32}$. The algorithm replaces the division by multiplications and shifts operations. It requires 3 multiplications.

Algorithm 7 Signed Montgomery multiplication [Sei18]

Input: Operand a, b such that $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$, where $\beta = 2^l$ with the machine word size l, the odd modulus $q \in (0, \frac{\beta}{2})$

Output: $r \equiv ab\beta^{-1} \bmod q, r \in (-q, q)$

1: $c = c_1\beta + c_0 = a \cdot b$
2: $m = c_0 \cdot q^{-1} \bmod^{\pm} \beta$ ▷ $\bmod^{\pm}$ obtains a signed product, q^{-1} is a precomputed constant
3: $t_1 = \lfloor m \cdot q/\beta \rfloor$ ▷ shift right operation
4: $r = c_1 - t_1$
5: **return** r

The Barrett reduction [Bar86] approximately computes $\lfloor c/q \rfloor$ using $\lfloor (c \cdot \lambda)/2^{2l'+\gamma} \rfloor$, where λ is a precomputed constant ($\lambda = \lfloor 2^{2l'+\gamma}/q \rfloor$) and l' satisfies $2^{l'-1} < q < 2^{l'}$. Note that γ is an arbitrary integer used to control the accuracy of the approximation (the larger the value, the higher the accuracy). The algorithm of Barrett multiplication is given in Algorithm 8, and it also requires 3 multiplications. Barrett algorithm is different from Montgomery algorithm. Its implementation generally consists of a shift operation for a non-word-size offset ($2l' + \gamma$), which would require an explicit shift instruction during the implementation on Cortex-M4. We can get the signed version with minor modifications[1].

Algorithm 8 Barrett multiplication [Bar86]

Input: Operand a, b such that $0 \leq a \cdot b < 2^{2l'+\gamma}$, the modulus q satisfying $2^{l'-1} < q < 2^{l'}$, and the precomputed constant $\lambda = \left\lfloor 2^{2l'+\gamma}/q \right\rfloor$

Output: $r \equiv a \cdot b \bmod q, r \in [0, q]$

1: $c = a \cdot b$
2: $t = \lfloor (c \cdot \lambda)/2^{2l'+\gamma} \rfloor$
3: $r = c - t \cdot q$
4: **return** r

Plantard [Pla21] proposed a new modular multiplication (Plantard multiplication) tailored for the word size moduli and claimed that the proposed algorithm outperforms other existing solutions, including Montgomery and Barrett multiplication, in some application scenarios. The original Plantard multiplication is given in Algorithm 9. Here, we denote $X \bmod 2^{l'}$ as $[X]_{l'}$, $X >> l'$ as $[X]^{l'}$, where l' is a positive integer. The Plantard multiplication also needs 3 multiplications, which is the same as Montgomery and Barrett multiplication. But the key advantage of Plantard multiplication is that one could precompute the product of b and $q^{-1} \bmod 2^{2l}$ when operand b is a constant. This introduces an $l \times 2l$-bit multiplication $a \cdot (bq')$. If the target platform can compute the $l \times 2l$-bit multiplication in one multiplication instruction, then one multiplication could be saved for the overall modular multiplication. This advantage mainly comes from the fact that

[1] Signed Barrett reduction in Kyber's implementation https://github.com/pq-crystals/kyber/blob/master/ref/reduce.c

the product of two operands $c = a \cdot b$ is only used once in Plantard multiplication, while its Montgomery and Barrett counterparts use it twice (see step 2,4 in Algorithm 7 and step 2,3 in Algorithm 8).

Algorithm 9 Original Plantard Multiplication [Pla21]

Input: Unsigned integers $a, b \in [0, q]$, $q < \frac{2^l}{\phi}, \phi = \frac{1+\sqrt{5}}{2}, q' \equiv q^{-1} \bmod 2^{2l}$, where l is the machine word size

Output: $r \equiv ab(-2^{-2l}) \bmod q$ where $r \in [0, q]$

1: $r = \left[\left([[abq']_{2l}]^l + 1\right) q\right]^l$ ▷ bq' could be precomputed when b is constant

2: **return** r

Despite this advantage, it should be noted that the original algorithm is designed for unsigned integer inputs within the range $[0, q]$. It has been well-illustrated that using a signed version of modular arithmetic in NTT/INTT will further speed up the computation. Therefore, the original Plantard multiplication could be extended to support signed arithmetic and purchase faster NTT/INTT implementations.

2.5 Target Platform: Cortex-M4

Our target platform is STM32F407G-DISC1, with a 32-bit ARM Cortex-M4 processor implementing the ARMv7E-M instruction sets. It provides 16 32-bit general-purpose registers, but only 14 of them are available for programming. Because Cortex-M4 supports floating-point (FP) computation, 32 32-bit FP registers are available for programming and can be used to "cache" frequently-used values.

The Cortex-M4 also supports Single Instruction Multiple Data (SIMD) instructions that can operate two halfwords or four bytes in parallel. The essential instructions in our implementation include **smul{b,t}{b,t}**, **smulw{b,t}**, and **ldrd**. We use Rd to represent the destination register; Rm and Rn represent two source registers. The **smulbb** instruction multiplies the bottom halfword of Rm by the bottom halfword of Rn and writes the result to Rd. The **smulwb** instruction multiplies Rm by the bottom halfword of Rn; extracts the most significant 32 bits of the result, and writes it to Rd. The **ldrd** R8, R9, [R3, #0x20] instruction loads R8 from the address R3+32 and loads R9 from the address R3+36.

3 Improved Plantard Arithmetic

This section presents the improved Plantard arithmetic, which was initially proposed in [Pla21]. Our improvement is built by first imposing a slight restriction on the modulus based on the observation of current LBC schemes (e.g., Kyber and NTTRU); then, several tweaks are proposed to improve the Plantard arithmetic. Detailed correctness proof and advantages of the improved algorithm will be presented.

3.1 Improved Plantard Multiplication

The improved Plantard multiplication is presented in Algorithm 10. Due to the adaptation of signed numbers, we denote $(X \bmod^{\pm} 2^{l'})$ as $[X]_{l'}$, $(X >> l')$ as $[X]^{l'}$ for a positive integer l'. Here, $(X >> l')$ is equivalent to $\lfloor X/2^{l'} \rfloor$ or $\lceil X/2^{l'} \rceil$ for a positive or negative number, respectively. For simplicity, we use $\lfloor X/2^{l'} \rfloor$ to represent $(X >> l')$ for sign-unknown number X unless a negative number is explicitly specified. We denote q as an odd modulus, l as the minimum word size (e.g., 16, 32 or 64 bits) such that $q < 2^{l-\alpha-1}$, where α is a small integer.

Algorithm 10 Improved Plantard multiplication

Input: Two signed integers $a, b \in [-q2^{\alpha}, q2^{\alpha}], q < 2^{l-\alpha-1}, q' = q^{-1} \bmod^{\pm} 2^{2l}$
Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$ where $r \in (-\frac{q}{2}, \frac{q}{2})$

1: $r = \left[\left([[abq']_{2l}]^{l} + 2^{\alpha}\right) q\right]^{l}$
2: **return** r

The improved Plantard multiplication is based on the observation that the moduli adopted by most of the existing LBC schemes are several times smaller than the restriction $q < \frac{2^l}{\phi}$ in the original algorithm [Pla21], e.g., 12-bit modulus 3329 in Kyber, 13-bit modulus 7681 in NTTRU, 23-bit modulus 8380417 in Dilithium. Thus, there are still margins between the original constraint and the moduli adopted in LBC schemes. Besides, all of the current state-of-the-art modular arithmetic is implemented with signed integers, which can eliminate extra additions of the multiple of q to ensure a positive output of the butterfly unit. Considering that the Plantard multiplication proposed by Thomas Plantard has the advantage of multiplying a constant (i.e., one multiplication is saved) but only supports unsigned integers, we believe that adapting signed integers for this algorithm will lead to more efficient modular arithmetic in LBC schemes.

We start by first introducing a small integer $\alpha \geq 0$, which narrows down the range of the modulus q from $q < \frac{2^l}{\phi}$ to $q < 2^{l-\alpha-1} < \frac{2^{l-\alpha}}{\phi}$. Based on this tweak, we can then replace the original main step $r = \left[\left([[abq']_{2l}]^{l} + 1\right) q\right]^{l}$ by $r = \left[\left([[abq']_{2l}]^{l} + 2^{\alpha}\right) q\right]^{l}$. In order to adapt signed inputs for this algorithm, we modify the inequality that maintains the correctness of this algorithm from $0 < q2^l - p_0 q + ab < 2^{2l}$ to $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$ under certain circumstances (see Subsection 3.2 for more details). We will show the correctness proof and advantages of the improved algorithm in the following two parts.

3.2 Proof of the Improved Plantard Multiplication

Theorem 1 states the correctness of Algorithm 10, and we will give the specific proof below.

Theorem 1 (Correctness)**.** *Let q be an odd modulus, l be the minimum word size (power of 2 number, e.g., 16, 32, and 64) such that $q < 2^{l-\alpha-1}$, where $\alpha \geq 0$, then Algorithm 10 is correct for $-q2^{\alpha} \leq a, b \leq q2^{\alpha}$.*

Proof of Theorem 1. The main step of Algorithm 10 is to compute $r = \left[\left([[abq']_{2l}]^{l} + 2^{\alpha}\right) q\right]^{l}$, namely:

$$r = \left\lfloor \frac{\left(\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^{\alpha}\right) q}{2^l} \right\rfloor.$$

1. We first check that $r \in (-\frac{q}{2}, \frac{q}{2})$. Since $\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor \in [-2^{l-1}, 2^{l-1} - 1]$, we have

$$\left\lceil \frac{(-2^{l-1} + 2^{\alpha})q}{2^l} \right\rceil \leq r \leq \left\lfloor \frac{(2^{l-1} - 1 + 2^{\alpha})q}{2^l} \right\rfloor$$
$$\left\lceil -\frac{q}{2} + \frac{q}{2^{l-\alpha}} \right\rceil \leq r \leq \left\lfloor \frac{q}{2} + \frac{(2^{\alpha} - 1)q}{2^l} \right\rfloor.$$

Since $\frac{q}{2^{l-\alpha}} < \frac{1}{2}$, we can get $r > -\frac{q}{2}$ from the left-hand side of the inequation. Let's

consider $\frac{q}{2} + \frac{(2^\alpha-1)q}{2^l}$ on the right-hand side; since $q < 2^{l-\alpha-1}$, we have

$$\frac{(2^\alpha-1)q}{2^l} < \frac{q2^\alpha}{2^l} < \frac{2^\alpha 2^{l-\alpha-1}}{2^l} = \frac{1}{2}.$$

Because q is an odd number, then

$$\left\lfloor \frac{q}{2} + \frac{(2^\alpha-1)q}{2^l} \right\rfloor = \left\lfloor \frac{q}{2} \right\rfloor < \left\lfloor \frac{q+1}{2} \right\rfloor.$$

Therefore, the result r lies in $(-\frac{q}{2}, \frac{q}{2})$.

2. Then, we check that $r = ab(-2^{-2l}) \bmod^{\pm} q$. The proof is very similar to the one in [Pla21], except that we introduce a small integer $\alpha \geq 0$ and impose a stricter constraint $q < 2^{l-\alpha-1}$ on q. Since q is an odd number, there exists a $p = abq^{-1} \bmod^{\pm} 2^{2l}$ so that

$$pq - ab \equiv \left(abq^{-1}\right) q - ab \bmod 2^{2l} \equiv ab - ab \bmod 2^{2l} \equiv 0 \bmod 2^{2l}.$$

Therefore, $pq - ab$ is divisible by 2^{2l}, so

$$ab\left(-2^{-2l}\right) \bmod q \equiv \frac{pq-ab}{2^{2l}}.$$

Let $p_1 = \left\lfloor \frac{p}{2^l} \right\rfloor, p_0 = p - p_1 2^l = p \bmod^{\pm} 2^l$ and $p_0 \in [-2^{l-1}, 2^{l-1})$, then instead of analyzing $q2^l - p_0 q + ab$ in the original work, we slightly modify the equation to $q2^{l+\alpha} - p_0 q + ab$. The tweak here is the key to the improved Plantard arithmetic, which allows us to accept both positive and negative inputs and extend the range of the inputs a, b to $[-q2^\alpha, q2^\alpha]$. We will further prove that this modification will not change its correctness under the constraints in Theorem 1.

The correctness of the Plantard multiplication proposed by Thomas Plantard is based on the inequality: $0 < q2^l - p_0 q + ab < 2^{2l}$. We now check that our modified equation $q2^{l+\alpha} - p_0 q + ab$ also satisfies this inequality:

$$0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l} \tag{3}$$

under the restrictions $q < 2^{l-\alpha-1}, \alpha \geq 0,$ and $-q2^\alpha \leq a, b \leq q2^\alpha$.

(1) When $ab > 0$ and $p_0 < 0$, we have

$$\begin{aligned} q2^{l+\alpha} - p_0 q + ab &\leq q2^{l+\alpha} + q2^{l-1} + q^2 2^{2\alpha} = q\left(2^{l+\alpha} + 2^{l-1} + q2^{2\alpha}\right) \\ &< 2^{l-\alpha-1}\left(2^{l+\alpha} + 2^{l-1} + 2^{l-\alpha-1}2^{2\alpha}\right) \\ &= 2^{l-\alpha-1}\left(3 \cdot 2^{l+\alpha-1} + 2^{l-1}\right) = 3 \cdot 2^{2l-2} + 2^{2l-\alpha-2} \\ &= \left(3 + 2^{-\alpha}\right) \cdot 2^{2l-2} \leq 4 \cdot 2^{2l-2} = 2^{2l} \text{ when } \alpha \geq 0. \end{aligned}$$

Therefore, for a small integer $\alpha \geq 0$, we have $q2^{l+\alpha} - p_0 q + ab < 2^{2l}$. The proof of the right-hand side of Equation 3 ends.

(2) When $ab < 0$ and $p_0 > 0$, we have

$$\begin{aligned} q2^{l+\alpha} - p_0 q + ab &> q2^{l+\alpha} - q2^{l-1} - q^2 2^{2\alpha} = q\left(2^{l+\alpha} - 2^{l-1} - q2^{2\alpha}\right) \\ &> q\left(2^{l+\alpha} - 2^{l-1} - 2^{l-\alpha-1}2^{2\alpha}\right) = q\left(2^{l+\alpha-1} - 2^{l-1}\right). \end{aligned}$$

It is obvious that $2^{l+\alpha-1} - 2^{l-1} \geq 0$ for any small integer $\alpha \geq 0$. Therefore, we have $q2^{l+\alpha} - p_0 q + ab > 0$. The proof of the left-hand side of Equation 3 ends. Overall, we obtain that

$$0 < \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} < 1.$$

Combining with the fact that $pq - ab$ is divisible by 2^{2l}, we have the following equation:

$$ab\left(-2^{-2l}\right) \bmod q \equiv \frac{pq-ab}{2^{2l}} \equiv \left\lfloor \frac{pq-ab}{2^{2l}} + \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} \right\rfloor \equiv \left\lfloor \frac{qp_1 2^l + q2^{l+\alpha}}{2^{2l}} \right\rfloor$$
$$\equiv \left\lfloor \frac{q(p_1 + 2^\alpha)}{2^l} \right\rfloor \equiv \left\lfloor \frac{q\left(\left\lfloor \frac{abq^{-1} \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^\alpha\right)}{2^l} \right\rfloor.$$

For signed inputs, it is equivalent to $ab(-2^{-2l}) \bmod^{\pm} q = \left[\left(\left[[abq']_{2l}\right]^l + 2^\alpha\right) q\right]^l = r.$ □

In summary, the improved Plantard multiplication is achieved by first imposing a stricter constraint on the modulus q. Then, the inequality that guarantees the correctness is modified into $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$. Note that the improved Plantard multiplication can be easily modified into a word size modular reduction (Plantard reduction).

3.3 Comparisons

Original Plantard multiplication. Although the main steps between the improved and original Plantard multiplications have only a minor difference, our tweaks make it more efficient and practical in LBC schemes thanks to the following merits.

- The improved algorithm supports signed inputs and produces a result in $(-\frac{q}{2}, \frac{q}{2})$, while the original version only accepts unsigned integers in $[0, q]$. The advantage of using signed integers in LBC schemes has been fully discussed in previous work, i.e., eliminating an extra addition during the butterfly computation.
- Besides, our tweaks extend the input range of Plantard multiplication from $[0, q]$ up to $[-q2^\alpha, q2^\alpha]$ and change the output range from $[0, q]$ to $(-\frac{q}{2}, \frac{q}{2})$, thus eliminating the final correction in the original version [Pla21, Algorithm 8]. A larger input range is especially useful in NTT/INTT because supporting more redundancy can reduce unnecessary modular reduction on the coefficients (refer to the lazy reduction strategy in Subsubsection 4.2.3).

Montgomery and Barrett arithmetic. The improved modular arithmetic also has several merits over the state-of-the-art Montgomery and Barrett arithmetic in LBC schemes.

- First of all, as introduced in [Pla21], the intrinsic advantage of the Plantard multiplication is that it costs one multiplication fewer than Montgomery and Barrett multiplication when one of the two inputs is fixed. This is achieved by precomputing the product of q' and the constant input modulo 2^{2l}. Moreover, the Barrett arithmetic may require an explicit shift operation for a non-word-size offset, which will further decrease its efficiency. We refer readers to Subsubsection 4.2.4 for a detailed comparison with the Barrett arithmetic.
- Secondly, the improved Plantard multiplication extends the input range to $[-q2^\alpha, q2^\alpha]$. We can easily modify this algorithm into the Plantard reduction that accepts input in $[-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$ (see Algorithm 13 for details). Compared with the Montgomery reduction that accepts $[-2^{l-1}q, 2^{l-1}q]$, the improved Plantard reduction supports more redundancy if α is big enough. Therefore, we recommend choosing the largest α that satisfies the prerequisites so that the improved algorithm has the largest input range. The input range of the proposed modular reduction is hard to compare

with the Barrett reduction directly because the input range of the Barrett arithmetic depends on the parameter setting, i.e., q and γ in Algorithm 8. The Barrett reduction used inside NTT/INTT in Kyber and NTTRU only needs to reduce 16-bit signed integers. Our modular reduction (see Algorithm 15) can also cover all 16-bit signed integers and can be a perfect replacement for the Barrett reduction in the NTT/INTT of these LBC schemes.

- The output range of the improved algorithm is narrowed down to $(-\frac{q}{2}, \frac{q}{2})$, which is the same as the state-of-the-art Barrett reduction presented in [AHKS22, Algorithm 3.2]. Compared with the output range $[-q, q]$ of the Montgomery multiplication, the improved Plantard multiplication halves or slows down the growing rate of the coefficient size in the NTT with CT butterflies or the INTT with GS butterflies, respectively. Together with the larger input range, the improved Plantard multiplication enables better lazy reduction strategies in NTT/INTT (see Subsubsection 4.2.3).

With all these merits over the Montgomery and Barrett arithmetic, the improved Plantard arithmetic has two weak spots. If we precompute the product of q' and the twiddle factors modulo 2^{2l} in NTT, the size of the precomputed intermediate result will be doubled. First, it needs to deal with the $l \times 2l$-bit multiplication, which may not be applicable to architectures like AVX2 and NEON. However, we show that this method is perfectly suitable for Cortex-M4, Cortex-M7 and some 32-bit microcontrollers (see Subsection 4.3). Second, the double-size twiddle factors are normally placed into the Read-Only Memory (ROM) instead of the Random Access Memory (RAM). Consequently, the side effect is that we need more cycles to load them into registers (see Subsubsection 4.2.2 for detailed discussion); this will not affect the stack usage. Nevertheless, arithmetic and experimental analyses show that the overall cycle reduction obtained from the aforementioned merits could offset the cycle increment introduced by twiddle factor loading.

4 Optimized Implementation on Cortex-M4

In this section, we present an efficient implementation of the improved Plantard arithmetic for 16-bit word-size moduli on Cortex-M4 and apply it to Kyber and NTTRU. Then, the extensibility of the improved Plantard arithmetic on other platforms and schemes is discussed. Note that the Plantard arithmetic mentioned below refers to the improved Plantard arithmetic presented in Section 3.

4.1 Efficient Plantard Arithmetic for 16-bit Modulus

Before moving forward into the implementation details, it is assumed that there exists a small integer $\alpha \geq 0$ such that the prerequisite ($q < 2^{l-\alpha-1}$) mentioned in Theorem 1 is established (which is the case for most of the LBC schemes, including Kyber and NTTRU). The word size in the following parts is set to $l = 16$ to support the 16-bit arithmetic in Kyber and NTTRU. As recommended in Subsection 3.3, the maximum α that satisfies Theorem 1 is $\alpha = 3/\alpha = 2$ for Kyber/NTTRU.

Based on the Plantard multiplication described in Algorithm 10, we propose a 2-cycle modular multiplication by a constant for 16-bit moduli. The detailed instruction sequence is shown in Algorithm 11. This is achieved by observing that a part of the main step of the Plantard multiplication is $abq' \bmod^{\pm} 2^{2l}$. If the multiplicand b is a constant, one can directly precompute $bq' \bmod^{\pm} 2^{2l}$ where $q' = q^{-1} \bmod^{\pm} 2^{2l}$. Recall that the input of Algorithm 10, a and b, satisfies $a, b \in [-q2^{\alpha}, q2^{\alpha}]$ and $ab \in [-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$. We can always reduce the constant b down to $[0, q)$ during the precomputation of bq'. Therefore, the input range of a can be extended to $[-q2^{2\alpha}, q2^{2\alpha}]$. Besides, for the parameter α, we have $2^{l-\alpha-2} < q < 2^{l-\alpha-1}$ when the maximum α is chosen. Therefore, by combining

Algorithm 11 The 2-cycle improved Plantard multiplication by a constant on Cortex-M4

Input: An l-bit signed integer $a \in [-2^{l-1}, 2^{l-1})$, a precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod^{\pm} 2^{2l}$
Output: $r_{top} = ab(-2^{-2l}) \bmod^{\pm} q, r_{top} \in (-\frac{q}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **smulwb** r, bq', a ▷ $r \leftarrow [[abq']_{2l}]^l$
3: **smlabb** $r, r, q, q2^{\alpha}$ ▷ $r_{\text{top}} \leftarrow [q[r]_l + q2^{\alpha}]^l$
4: **return** r_{top}

Algorithm 12 The 3-cycle signed Montgomery multiplication on Cortex-M4 [ABCG20]

Input: Two l-bit signed integers a, b such that $ab \in [-q2^{l-1}, q2^{l-1})$
Output: $r_{top} = ab2^{-l} \bmod^{\pm} q, r_{top} \in (-q, q)$

1: **mul** c, a, b
2: **smulbb** $r, c, -q^{-1}$ ▷ $r \leftarrow [c]_l \cdot \left(-q^{-1}\right)$
3: **smlabb** r, r, q, c ▷ $r_{\text{top}} \leftarrow [[r]_l \cdot q]^l + [c]^l$
4: **return** r_{top}

$2^{l-\alpha-2} < q$ and $a \in [-2^{2\alpha}q, 2^{2\alpha}q]$, we conclude that a lies in $[-2^{l+\alpha-2}, 2^{l+\alpha-2}]$. For any integer $\alpha > 0$, the input a covers every 16-bit signed integer, i.e., $a \in [-2^{l-1}, 2^{l-1})$.

The main step of Algorithm 10 is $[([[abq']_{2l}]^l + 2^{\alpha})q]^l$. The **smulwb** instruction in Algorithm 11 computes $[[abq']_{2l}]^l$ in 1 cycle, and the intermediate result is saved in the bottom half of r. Here, adding 2^{α} to the bottom half of r before the multiplication with q might produce a carry to the top half of the 32-bit register r. This would cause an error when the top half of r is not zero. This problem can be addressed by computing the main step separately as $[q \cdot [[abq']_{2l}]^l + q2^{\alpha}]^l$ with the **smlabb** instruction. Here, the term $q2^{\alpha}$ is fixed and can be precomputed without extra runtime overhead. The final result locates in the top half of r. Compared with Algorithm 12, the state-of-the-art 16-bit Montgomery multiplication implementation on Cortex-M4 [ABCG20], we reduce one multiplication by precomputing the product of q^{-1} and the constant operand b. As for its comparison with the Barrett arithmetic, we refer to Subsubsection 4.2.4 for more details.

Note that [LS19] proposed a similar trick for Montgomery multiplication with the AVX2 instructions, which is improved over the work in [Sei18]. It is achieved by precomputing $q^{-1}/-q^{-1}$ with the bottom half of c (see step 2 of Algorithm 7/Algorithm 12). Unfortunately, it is not suitable for Cortex-M4 because the trick shown in [LS19] highly relies on the two-instruction-fashion multiplication, whereas the 16/32-bit multiplication on Cortex-M4 is carried out in a single instruction. Applying their optimization on Cortex-M4 introduces extra multiplication instructions as stated in [BKS19, Subsection 3.2].

As for the multiplication of two variables, we present a 2-cycle Plantard reduction over a 32-bit signed product $c \in [-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$, which is shown in Algorithm 13. Instead of using **smulwb** as we did in Algorithm 11, the **mul** instruction is used to compute $[[cq']_{2l}]^l$, and the intermediate result locates in the top half of r. The result is then obtained by the **smlatb** instruction. It is worth noting that the input range of Algorithm 13 is $q2^{2\alpha-l+1}$ times larger than the Montgomery reduction (the reduction version of Algorithm 12). Since α is chosen to ensure $q < 2^{l-\alpha-1}$, we have $q2^{\alpha+1} < 2^{l-\alpha-1} \cdot 2^{\alpha+1} = 2^l$. As for the maximum α, we have $q2^{\alpha+1} \approx 2^l$, then $q2^{2\alpha+1-l} = q2^{\alpha+1-l} \cdot 2^{\alpha} \approx 2^{\alpha}$. Therefore, the input range of the Plantard reduction is approximately 2^{α} times larger than the Montgomery reduction. The excellent input range and output range of the improved Plantard arithmetic enable better lazy reduction strategies in NTT/INTT (see Subsubsection 4.2.3).

Algorithm 13 The 2-cycle improved Plantard reduction on Cortex-M4

Input: A $2l$-bit signed integer $c \in [-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$
Output: $r_{top} = c(-2^{-2l}) \bmod^{\pm} q, r_{top} \in (-\frac{q}{2}, \frac{q}{2})$
1: $q' \leftarrow q^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **mul** r, c, q'
3: **smlatb** $r, r, q, q2^{\alpha}$
4: **return** r_{top}

4.2 Efficient 16-bit NTT/INTT Implementation

This subsection details the improvements the improved Plantard arithmetic brings to the butterfly unit, layer merging, and lazy reduction in NTT/INTT.

4.2.1 Butterfly Unit

The core operations in NTT/INTT are performed in the butterfly unit. Commonly, there are two types of butterfly structures, namely the CT butterfly and GS butterfly. In Kyber, using CT butterfly in NTT and GS butterfly in INTT is a common strategy to avoid coefficients flipping. However, recent reports [CHK+21, AHKS22] state that using CT butterfly for both NTT and INTT in LBC schemes would also result in faster code. For both strategies in 16-bit NTT/INTT, one needs to compute CT/GS butterfly over two 32-bit packed integers and return two 32-bit packed results [BKS19, ABCG20]. The improved Plantard multiplication by a constant (Algorithm 11) can be well adapted into the double CT/GS butterflies with the help of the **smulw{b,t}** instruction on Cortex-M4.

Algorithm 14 Double CT butterfly on Cortex-M4

Input: Two 32-bit packed signed integers a, b (each containing a pair of 16-bit signed coefficients), the 32-bit twiddle factor ζ
Output: $a = (a_{top} + b_{top}\zeta)||(a_{bottom} + b_{bottom}\zeta), b = (a_{top} - b_{top}\zeta)||(a_{bottom} - b_{bottom}\zeta)$
1: **smulwb** t, ζ, b
2: **smulwt** b, ζ, b
3: **smlabb** $t, t, q, q2^{\alpha}$
4: **smlabb** $b, b, q, q2^{\alpha}$
5: **pkhtb** $t, b, t,$ asr#16
6: **usub16** b, a, t
7: **uadd16** a, a, t
8: **return** a, b

The key operation in the butterfly unit is the modular multiplication by a proper twiddle factor ζ, which is a precomputed constant. It is common to store the twiddle factors in the Montgomery domain when using Montgomery arithmetic. Similar to Montgomery arithmetic, Plantard arithmetic also produces a result in a special domain, namely $c(-2^{-2l}) \bmod^{\pm} q$. Therefore, to integrate Plantard arithmetic into the butterfly unit, we also store the twiddle factors in the "Plantard" domain by multiplying each twiddle factor with a constant $-2^{2l} \bmod q$. Besides, to utilize the improved Plantard multiplication by a constant (Algorithm 11) in the butterfly unit, one also needs to multiply $q^{-1} \bmod^{\pm} 2^{2l}$ by the twiddle factor ζ in the "Plantard" domain modulo 2^{2l}, namely $\zeta = (\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1} \bmod^{\pm} 2^{2l}$. Note that this precomputation would result in a $2l$-bit twiddle factor. Although it would introduce extra cycles for data loading, this overhead can be offset by the improvements brought by the improved Plantard arithmetic.

Algorithm 14 illustrates the detailed instruction sequence of the double CT butterfly, which computes $a = (a_{top} + b_{top}\zeta)||(a_{bottom} + b_{bottom}\zeta), b = (a_{top} - b_{top}\zeta)||(a_{bottom} -$

$b_{bottom}\zeta$) over packed arguments a, b. Thanks to the SIMD instruction **smulw{b,t}**, one can pack two 16-bit coefficients into one 32-bit register b. Then, one can perform the Plantard multiplication (Algorithm 11) by a proper twiddle factor ζ over the top and bottom half of b using the **smulwt** and **smulwb** instructions, respectively. The follow-up instruction sequence is the same as the previous work [ABCG20]. In summary, we obtain a 7-instruction double CT butterfly for packed arguments, which reduces 2 instructions compared with the one that uses Montgomery multiplication. Similarly, we can also obtain a 7-instruction double GS butterfly for INTT.

4.2.2 Layer Merging

As stated in Subsubsection 4.2.1, using the improved Plantard arithmetic will double the size of the twiddle factors. It would bring extra **load** instructions to load the 32-bit twiddle factors compared with the original 16-bit version. However, the improvements brought by the improved Plantard arithmetic can easily bury this overhead thanks to the layer merging techniques. There are two efficient layer merging strategies for Kyber on Cortex-M4, which are presented in [ABCG20] and [AHKS22], respectively. The layer merging strategy in [ABCG20] is the 3-layer merging strategy (3-3-1), while [AHKS22] adopts the 4-layer merging strategy (4-3). Since the first 4-layer NTT re-uses the same 15 twiddle factors multiple times, [AHKS22] proposes to cache the 15 16-bit twiddle factors into 8 FP registers and replace the memory access instruction with the cheaper **vmov** instruction. Apart from using the FP registers and the **vmov** instruction, the 4-layer merging strategy is actually built upon the 3-layer merging strategy in [ABCG20].

The 3-layer merging strategy in [ABCG20] allows us to compute 8 butterflies at the cost of loading 1, 2, or 4 32-bit twiddle factor(s) in each layer. Note that loading 2 consecutive 32-bit twiddle factors can be achieved by a 3-cycle **ldrd** instruction on Cortex-M4, which is merely 1-cycle slower than the original **ldr** instruction to load 2 consecutive 16-bit twiddle factors. We rearrange the twiddle factors in the same order as they are used so that one can use **ldrd** to load 2 consecutive 32-bit twiddle factors. Overall, by integrating the Plantard arithmetic into the 3-layer merging strategy, one could reduce 8 cycles (reduce 1 cycle for each butterfly) with 0, 1, or 2 extra cycle(s) to load the 32-bit twiddle factors in each layer. For the 8 butterflies that only require 1 twiddle factor, one could reduce 8 cycles at no extra cost since **ldr** has the same cost as **ldrh**.

As for the 4-layer merging strategy, due to the double size of the twiddle factors, one needs to load 15 twiddle factors into 15 FP registers with the **vldm** instruction. Compared with 8 FP registers in the original design, our design consumes 7 extra cycles. Besides, during each iteration of the first 4 layers, one needs 7 extra **vmov** instructions to retrieve the twiddle factors from the FP registers to general registers. This extra cycle consumption can be totally covered by the cycle reduction brought by the improved Plantard arithmetic.

In order to reveal the actual effect brought by the improved modular arithmetic in our implementation, both 3-layer and 4-layer merging strategies are used in Kyber, while the better 4-layer merging strategy is adopted in NTTRU. Besides, for the implementation of NTTRU and Kyber with the 3-layer merging strategy, we use CT butterflies in NTT and GS butterflies in INTT. As for the Kyber implementation with the 4-layer merging strategy, the CT butterflies are adopted in both NTT and INTT, similar to [AHKS22].

4.2.3 Lazy Reduction

To better illustrate the improvements the improved Plantard arithmetic brings to both CT and GS butterflies, we only discuss the lazy reduction strategies for the NTT with CT butterflies and INTT with GS butterflies here. When Montgomery arithmetic is applied to NTT and INTT, all of the coefficients will grow by q after each layer of NTT,

while half of the coefficients will double after each layer of INTT. The well-known lazy reduction technique can minimize the number of modular reductions in NTT/INTT, which is mainly determined by the input range of the modular multiplication. Compared with the Montgomery arithmetic, the output range of the improved Plantard arithmetic is halved, while the input range of the improved Plantard reduction (Algorithm 13) is about 2^α times larger than Montgomery reduction (i.e., the reduction version Algorithm 12). Therefore, applying the improved Plantard arithmetic in NTT/INTT can halve/decrease the growing rate of coefficients compared with the one that uses Montgomery arithmetic, thus enabling better lazy reduction strategies.

CT Butterfly. Specifically, for the modulus $q = 3329$ ($9q < 2^{l-1} < 10q$) in Kyber, the forward NTT has 7 layers of butterflies. Therefore, the 7 layers of butterflies that use Montgomery multiplication will expand all of the 256 coefficients by $7q$. Therefore, at least one modular reduction is required to reduce the 256 coefficients to perform the follow-up base multiplication. On the contrary, when applying Plantard multiplication, the coefficients only grow by $3.5q$ in 7 layers. Since the inputs of NTT are always smaller than q, $4.5q$ lies in the modular multiplication input range $[-q2^\alpha, q2^\alpha]$ of Kyber (α=3 for Kyber). Therefore, the output coefficients of NTT with Plantard multiplication can be directly used in the base multiplication without modular reduction.

It should be noted that the coefficient expansion would have a bigger effect on schemes with bigger modulus like NewHope [ADPS16] and NTTRU [LS19]. As for the modulus $q = 7681$ ($4q < 2^{l-1} < 5q$) in NTTRU, all of the 768 coefficients need one modular reduction after every three layers of butterflies when Montgomery multiplication is used. Since the forward NTT in NTTRU has 8 layers of butterflies, we need two modular reductions after the third and sixth layers of butterflies. The last two layers of butterflies will produce coefficients smaller than $3q$. These coefficients cannot be directly used in Montgomery multiplication because $(3q)^2 = 9q^2$, which is out of the input range of $q2^{l-1}$. Therefore, the forward NTT implemented with Montgomery arithmetic in NTTRU needs three modular reductions for 768 coefficients. However, when using the improved Plantard multiplication in NTTRU, we only need one modular reduction for 768 coefficients after the seventh-layer butterflies. The final layer of butterflies only expand the coefficients up to $1q$, which lies in the input range of $[-q2^\alpha, q2^\alpha]$ ($\alpha = 2$ for NTTRU), and these coefficients can be directly used in the base multiplication. In sum, two modular reductions for 768 coefficients are saved compared with the implementation that uses Montgomery arithmetic, which fully illustrates the benefits of a larger input range and smaller output range of our method.

GS Butterfly. As for the INTT with GS butterflies in Kyber, the advantages of applying the Plantard arithmetic are twofold. First, the maximum input value of INTT is halved after the matrix-vector multiplication (i.e., Step 6 of Algorithm 2). If one uses Montgomery arithmetic in the matrix-vector multiplication, the coefficients produced in this process would be smaller than $2q, 3q,$ and $4q$ for Kyber512, Kyber768, and Kyber1024, respectively. Therefore, one modular reduction of coefficients is required after the first and second layers of butterflies in INTT for Kyber768/Kyber1024 and Kyber512, respectively. In our case, the modular reduction will have a one-layer delay since the matrix-vector multiplication generates coefficients smaller than $1q, 1.5q,$ and $2q$. Second, since the maximum value of the reduced coefficient is $0.5q$ after each modular reduction, 4 layers of butterflies can be conducted over the reduced coefficients instead of 3 when using Montgomery's method. After the second-layer butterflies of Kyber768/Kyber1024, the modular reduction is applied for all of the 128 coefficients; then all coefficients have a maximum value of $0.5q$. The third-layer butterflies have a step size of 8, and the first 8 coefficients ($a_0 \sim a_7$) will first grow to $8q$ after the sixth-layer butterflies. The final-layer butterflies have a step size of 128. Two sets of 8 coefficients ($a_0 \sim a_7, a_{128} \sim a_{135}$) have a maximum value of $8q$, while the rest of the coefficients are smaller than $4q$. Therefore, we only

Algorithm 15 Double Plantard reduction for packed coefficients

Input: A 32-bit packed integers $a = a_{top}||a_{bottom}$ where a_{top}, a_{bottom} are two 16-bit signed coefficients

Output: $r = (a_{top} \bmod^{\pm} q)||(a_{bottom} \bmod^{\pm} q)$, $-q/2 < r_{top}, r_{bottom} < q/2$

1: const $\leftarrow (-2^{2l} \bmod q) \cdot (q^{-1} \bmod^{\pm} 2^{2l}) \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **smulwb** t, const, a
3: **smulwt** a, const, a
4: **smlabt** $t, t, q, q2^{\alpha}$
5: **smlabt** $a, a, q, q2^{\alpha}$
6: **pkhtb** r, a, t, asr#16
7: **return** r

need to perform modular reduction on these 16 coefficients instead of all of the 128 coefficients in previous reports, and other coefficients will be naturally reduced by the modular multiplication with the twiddle factors and 128^{-1} in the final-layer butterflies.

The GS butterflies are also adopted for the INTT in NTTRU. The optimization brought by our method is similar to the INTT in Kyber, except that the modular reduction for half of the coefficients is required after every 3 layers of butterflies with Plantard arithmetic instead of 2 when using Montgomery arithmetic. Thus, only 2 modular reductions for 384 coefficients are required after the third and sixth layers of butterflies.

4.2.4 Double Plantard Reduction for Packed Coefficients

The modular reduction of coefficients is necessary if the next operation in NTT/INTT would cause an overflow. Previous work [ABCG20] uses double Montgomery reduction or double Barrett reduction (i.e., Algorithm 10 and Algorithm 8 in [ABCG20]) to reduce two packed coefficients on Cortex-M4, which consume 7 cycles and 8 cycles, respectively. The double Barrett reduction in [ABCG20, Algorithm 8] requires two explicit shift operations for packed coefficients. Recent work further reduces its cycle counts down to 6 cycles by eliminating two explicit shift operations using the **smlawb** and **smlawt** instructions (see [AHKS22, Algorithm 3.2]). In this work, an even better 5-cycle double Plantard reduction is proposed and shown in Algorithm 15. This algorithm is based on the fact that the Plantard reduction over a 16-bit signed integer can be viewed as a Plantard multiplication by a constant $-2^{2l} \bmod q$. And one can multiply q^{-1} by the constant $-2^{2l} \bmod q$ to obtain a precomputed constant, thus saving one multiplication by q^{-1}. Note that the input range of Algorithm 15 is the same as Algorithm 11. Since Algorithm 15 generates signed output in $(-\frac{q}{2}, \frac{q}{2})$, this algorithm is only applicable inside NTT/INTT. The Barrett reduction is still required outside NTT/INTT to obtain a positive result.

4.3 Extensibility on Other Platforms and Schemes

The efficiency of the Plantard arithmetic for 16-bit moduli relies on the **smulw{b,t}** instruction to perform the 16×32-bit multiplication on Cortex-M4. The 16×32-bit multiplication would make it difficult to apply the Plantard arithmetic to architectures like AVX2 and NEON. However, the proposed optimizations for 16-bit NTT are not limited to Cortex-M4 but can also be extended to Cortex-M7 as both of the devices share the same SIMD extensions [Lor16]. Moreover, the Plantard arithmetic for 16-bit moduli may still outperform the Montgomery and Barrett arithmetic on some 32-bit microcontrollers without such powerful SIMD extensions as Cortex-M4/M7. Take the SiFive Freedom E310 microcontroller [SiF], equipped with a 32-bit E31 RISC-V core, as an example. The instruction sequence of Algorithm 11 on RISC-V is shown in Algorithm 16. Here, the 16×32-bit multiplication is directly implemented with the 32×32-bit multiplication

Algorithm 16 The improved Plantard multiplication by a constant on RISC-V

Input: A 32-bit signed integer $a \in [-q2^{2\alpha}, q2^{2\alpha}]$, a precomputed $2l$-bit integer bq' where b is a constant, $q' = q^{-1} \bmod^{\pm} 2^{2l}$
Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in (-\frac{q}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **mul** r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
3: **srai** $r, r, \#16$ ▷ $r \leftarrow [[abq']_{2l}]^l$
4: **mul** r, r, q
5: **add** $r, r, q2^{\alpha}$ ▷ $r \leftarrow q[[abq']_{2l}]^l + q2^{\alpha}$
6: **srai** $r, r, \#16$ ▷ $r \leftarrow [q[[abq']_{2l}]^l + q2^{\alpha}]^l$
7: **return** r

instruction, and the result locates in the top half of r. One extra shift operation is required to shift it down to the bottom half for the follow-up operations. Compared with its Montgomery and Barrett counterparts [Gre20, Listing 5.2.10] on RISC-V, the improved Plantard multiplication by a constant reduces one multiplication instruction (the intrinsic advantage of the Plantard arithmetic) and introduces an extra shift instruction. On the SiFive Freedom E310 microcontroller, one multiplication instruction costs 5 cycles while one shift instruction costs 1 cycle [SiF]. Therefore, the improved Plantard arithmetic is better than its Montgomery and Barrett counterparts on this platform. Similarly, for other 32-bit microcontrollers without SIMD instruction, our method for 16-bit moduli would still outperform its Montgomery and Barrett counterparts if multiplication instruction is slower than the shift instruction.

Similar analysis results can be obtained on the Plantard arithmetic for 32-bit moduli. The improved Plantard arithmetic would still outperform its Montgomery and Barrett counterparts if the target platform provides native 32×64-bit or 64×64-bit multiplication instructions, and the multiplication instruction is slower than the shift instruction. In this work, we only focus on accelerating Kyber and NTTRU with 16-bit NTT on Cortex-M4. We leave the exploration of Plantard arithmetic's practical applications on various schemes and platforms as future work.

5 Results and Comparisons

5.1 Benchmarking Setup

The benchmarking results are obtained on STM32F407G-DISC1 with the STM32F407VGT6 MCU. The board is equipped with 1MiB of flash memory and 192 KiB of RAM. The benchmarking setup can refer to pqm4 [KRSS], and the microcontroller is clocked at 24 MHz to avoid wait states during memory operations as recommended in pqm4. The compile tool is arm-non-eabi-gcc version 10.2.1, and we compile our code with -flto and -O3 options, which are the same as [ABCG20] and [AHKS22]. The Keccak implementation is token from pqm4, and the hardware random number generator (RNG) of the microcontroller is used in our implementation.

The proposed Kyber implementation is initially built upon the pqm4 code, which consists of the optimizations presented in [ABCG20, BKS19]. Later work [AHKS22] presents a faster Kyber implementation, including a high-speed and stack-friendly version. Although their high-speed Kyber implementation has better efficiency, their stack usage is almost doubled. We integrate the improved Plantard arithmetic into their stack-friendly implementation to benchmark the speed performance of Kyber. The proposed NTTRU implementation is based on the reference code [LS19] that uses the symmetric primitives from OpenSSL. To immigrate its reference code to Cortex-M4, we use the SHA2 and AES

Table 1: Cycle counts for the core polynomial arithmetic in Kyber and NTTRU on Cortex-M4, i.e., NTT, INTT, base multiplication, and base inversion.

Scheme	Implementation	NTT	INTT	Base Mult	Base Inv
Kyber	[ABCG20]	6 822	6 951	2 291	-
	This work[a]	5 441	5 775	2 421	-
	Speed-up	20.24%	16.92%	-5.67%	-
	Stack[AHKS22]	5 967	5 917	2 293	-
	Speed[AHKS22]	5 967	5 471	1 202	-
	This work[b]	4 474	4 684/4 819/4 854	2 422	-
	Speed-up (stack)	25.02%	20.84%/18.56%/17.97%	-5.58%	-
	Speed-up (speed)	25.02%	14.38%/11.92%/11.28%	-101.41%	-
NTTRU	[LS19]	102 881	97 986	44 703	100 249
	This work	17 274	20 931	10 550	40 763
	Speed-up	83.21%	78.64%	76.40%	59.34%

[a] Implementation based on [ABCG20], [b] Implementation based on the stack-friendly code of [AHKS22].

implementations in pqm4, which are based on the work in [AP21].

5.2 Performance of the Polynomial Arithmetic

The cycle counts of the core polynomial arithmetic in Kyber and NTTRU are presented in Table 1. Using the improved Plantard arithmetic, the proposed Kyber implementation based on [ABCG20] achieves 20.24% and 16.92% speed-ups for NTT and INTT, respectively. The speed-optimized implementation of [AHKS22] trades the speed with extra stack usage while implementing the matrix-vector multiplication and base multiplication, which explains their two times faster base multiplication. Besides, the coefficients of their matrix-vector multiplication lie in $[-q, q]$, which reduces one modular reduction of 128 coefficients at the beginning of their INTT implementation. On the contrary, the matrix-vector multiplication of their stack-friendly implementation produces coefficients smaller than kq, which requires extra modular reductions in INTT and results in a slower INTT implementation.

After applying the improved Plantard arithmetic in their stack-friendly implementation, we halve the coefficient size of the matrix-vector multiplication down to $\frac{1}{2}kq$; then, three different lazy reduction strategies are adopted for the INTT in Kyber512, Kyber768, and Kyber1024, respectively. Overall, the proposed implementation based on the stack-friendly code of [AHKS22] achieves 25.02% speed-up for NTT and 20.84%/18.56%/17.97% speed-ups for the INTT in Kyber512/Kyber768/Kyber1024, respectively. Our INTT implementation is also 14.38%/11.92%/11.28% faster than their speed-optimized INTT implementation in Kyber512/Kyber768/Kyber1024, respectively. All these speed-ups mainly come from the efficient Plantard multiplication by a constant and better lazy reduction strategies. The results clearly reveal the improved Plantard arithmetic's benefits to the NTT/INTT on Cortex-M4. The base multiplication of Kyber consists of modular multiplication by a constant and modular multiplication of two variables, which increases the register usage pressure and requires extra cycles to handle this problem.

The polynomial arithmetic implementation of NTTRU is similar to Kyber. By fully utilizing the SIMD instructions, the double butterflies, double modular reduction over packed coefficients, double base multiplication, and double base inversion are first presented by applying the improved Plantard arithmetic. The assembly implementation of NTTRU obtains excellent speed-ups compared with its reference implementation in [LS19]. Table 1 shows that the speed-ups for NTT, INTT, base multiplication, and base inversion are 83.21%, 78.64%, 76.40%, and 59.34%, respectively. The speed-ups for NTT and INTT mainly come from the efficient Plantard multiplication by a constant, better

Table 2: Cycle counts (cc) and stack usage (Bytes) for KeyGen, Encaps, and Decaps on Cortex-M4. k is the dimension of the underlying Module-LWE problem for Kyber. The first row of each entry indicates the cycle count and the second row refers to stack usage.

Scheme	Implementation	KeyGen			Encaps			Decaps		
		$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$
Kyber	[ABCG20]	454k	741k	1 177k	548k	893k	1 367k	506k	832k	1 287k
		2 464	2 696	3 584	2 168	2 640	3 208	2 184	2 656	3 224
	This work[a]	446k	729k	1 162k	542k	885k	1 357k	497k	818k	1 270k
		2 464	2 696	3 584	2 168	2 640	3 208	2 184	2 656	3 224
	Stack[AHKS22]	439k	717k	1 139k	534k	871k	1 329k	484k	797k	1 233k
		2 608	3 056	3 576	2 160	2 660	3 236	2 176	2 676	3 252
	Speed[AHKS22]	438k	711k	1 129k	531k	864k	1 316k	479k	787k	1 217k
		4 268	6 732	7 748	5 252	6 284	7 292	5 260	6 308	7 300
	This work[b]	430k	702k	1 119k	526k	861k	1 314k	472k	780k	1 211k
		2 608	3 056	3 576	2 160	2 660	3 236	2 176	2 676	3 252
NTTRU	[LS19]	526k			431k			559k		
		9 384			8 748			10 324		
	This work	267k			237k			254k		
		9 372			7 452			8 816		

[a] Implementation based on [ABCG20], [b] Implementation based on the stack-friendly code of [AHKS22].

lazy reduction techniques, and parallel implementation utilizing the SIMD instructions on Cortex-M4. The speed-up for base inversion is relatively smaller than others because most of the modular multiplications in this operation are implemented as the modular multiplications of two variables. Therefore, the efficient Plantard multiplication by a constant (Algorithm 11) can not be applied to base inversion.

5.3 Performance of Schemes

Table 2 shows the cycle counts and stack usage of the KEM protocols, namely key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps). The Kyber implementation based on either [ABCG20] or the stack-friendly code of [AHKS22] achieves better speed performance than its Montgomery-based counterpart with the same stack usage. For the Kyber implementation based on [ABCG20], the speed-ups for KeyGen, Encaps, and Decaps are 1.27%-1.76%, 0.73%-1.09%, and 1.32%-1.78%, respectively. As for the Kyber implementation based on the stack-friendly code in [AHKS22], we achieve speed-ups of 1.76%-2.09%, 1.13%-1.50%, and 1.78%-2.48% for KeyGen, Encaps, and Decaps, respectively. It is worth noting that our implementation based on the stack-friendly version of [AHKS22] outperforms their high-speed version with approximately half of the stack. If the stack usage is not one of the primary improved goals, one can also integrate the improved Plantard arithmetic into their speed-optimized implementation for better efficiency. As stated in previous work [KRSS19], the dominance of the hashing functions of Kyber will reduce the overall effect of the optimized polynomial arithmetic, which explains the relatively small speed-ups for the KEM protocols of Kyber.

As the first assembly NTTRU implementation on Cortex-M4, we obtain 49.24%, 45.01%, and 54.56% speed-ups for KeyGen, Encaps, and Decaps compared with its reference implementation. The excellent speed-ups mainly come from the highly optimized assembly implementation of NTT, INTT, base multiplication, and base inversion implemented with the improved Plantard arithmetic. It is obvious that NTTRU outperforms every variant of Kyber by a large margin in terms of speed on Cortex-M4. Compared with the fastest Kyber512, NTTRU provides 37.91%, 55.03%, 46.19% faster KeyGen, Encaps, and Decaps with approximately 3.45∼4.05 times larger stack usage, respectively. In sum, although it is not a candidate for the NIST PQC competition, its efficiency might make it suitable for some specific application scenarios or platforms.

Acknowledgments

The authors would like to thank the anonymous reviewers, especially the shepherd: Bo-Yin Yang, and our friend Hao Cheng for their constructive suggestions and comments on our paper. This work is partially supported by the National Key R&D Program of China (No.2020AAA0107703 and No.2021YFB3100700), the National Natural Science Foundation of China (No. 62002023 and No.62132008), the Guangdong Provincial Key Laboratory of Interdisciplinary Research and Application for Data Science, BNU-HKBU United International College (2022B1212010006), Guangdong Higher Education Upgrading Plan (2021-2025) (UIC R0400001-22), Guangdong Higher Education Key Platform and Research Project (No. 2020KQNCX100), Hong Kong Innovation and Technology Commission (InnoHK Project CIMDA), and Hong Kong Research Grants Council (Project 11204821).

References

[AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020.

[ABD+20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation. 2020.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Daan Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, 2022.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers new bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019.

[BMD+20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 3 Submission), 2020. NIST Post-Quantum Cryptography Standardization Project.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M Schanck, Tsunekazu Saito, Peter Schwabe, William Whyte, Keita Xagawa, et al. NTRU: Algorithm specifications and supporting documentation. 2020.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings new speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[GE21] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, 2021.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, 2021.

[Gre20] Denisa Greconici. Kyber on RISC-V. Master's thesis, 2020.

[GS66] W. Morven Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA*, volume 29 of *AFIPS Conference Proceedings*, pages 563–578. AFIPS / ACM / Spartan Books, Washington D.C., 1966.

[Hac20] Robert Hackett. IBM plans a huge leap in superfast quantum computing by 2023, 2020.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. *IACR Cryptol. ePrint Arch.*, page 844, 2019.

[Lor16] Thomas Lorenser. The DSP capabilities of ARM Cortex-M4 and Cortex-M7 processors. *ARM White Paper*, 29, 2016.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):180–201, 2019.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Trans. Emerg. Top. Comput.*, 9(3):1506–1518, 2021.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *Cryptology ePrint Archive*, 2018.

[SHRS] John M. Schanck, Andreas Hulsing, Joost Rijneveld, and Peter Schwabe. NTRU-HRSS-KEM: NIST Round 1 Submission.

[SiF] SiFive. Sifive FE310-G002 Manual. https://sifive.cdn.prismic.io/sifive/b56b304f-cd2d-421b-9c14-6b35c33f172e_fe310-g002-manual-v1p4.pdf.

[WBC+21] Yulin Wu, Wan-Su Bao, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, and Yajie et al. Du. Strong quantum computational advantage using a superconducting quantum processor. *Phys. Rev. Lett.*, 127:180501, Oct 2021.

[ZCHW] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, and William Whyte. NTRUEncrypt: NIST Round 1 Submission.

[ZHLR22] Jipeng Zhang, Junhao Huang, Zhe Liu, and Sujoy Sinha Roy. Time-memory trade-offs for Saber+ on memory-constrained RISC-V platform. *IEEE Transactions on Computers*, 2022.