Algorithmic Views of Vectorized Polynomial Multipliers – NTRU

Han-Ting Chen[1], Yi-Hua Chung[2], Vincent Hwang[2,3(✉)], and Bo-Yin Yang[2(✉)]

[1] National Taiwan University, Taipei, Taiwan
r10922073@csie.ntu.edu.tw
[2] Academia Sinica, Taipei, Taiwan
yhchiara@gmail.com, vincentvbh7@gmail.com, by@crypto.tw
[3] Max Planck Institute for Security and Privacy, Bochum, Germany

Abstract. The lattice-based post-quantum cryptosystem NTRU is used by Google for protecting Google's internal communication. In NTRU, polynomial multiplication is one of bottleneck. In this paper, we explore the interactions between polynomial multiplications, Toeplitz matrix-vector products, and vectorization with architectural insights. For a unital commutative ring R, a positive integer n, and an element $\zeta \in R$, we reveal the benefit of vector-by-scalar multiplication instructions while multiplying in $R[x]/\langle x^n - \zeta \rangle$.

We aim at designing an algorithm exploiting no algebraic and number–theoretic properties of n and ζ. An obvious way is to multiply in $R[x]$ and reduce modulo $x^n - \zeta$. Since the product in $R[x]$ is a polynomial of degree at most $2n-2$, one usually chooses a polynomial modulus $\boldsymbol{g}$ such that (i) $\deg(\boldsymbol{g}) \geq 2n-1$, and (ii) there exists a well-studied fast polynomial multiplication algorithm f for multiplying in $R[x]/\langle \boldsymbol{g} \rangle$.

We deviate from common approaches and point out a novel insight with dual modules and vector-by-scalar multiplications. Conceptually, we relate the module-theoretic duals of $R[x]/\langle x^n - \zeta \rangle$ and $R[x]/\langle \boldsymbol{g} \rangle$ with Toeplitz matrix-vector products, and demonstrate the benefit of Toeplitz matrix-vector products with vector-by-scalar multiplication instructions. It greatly reduces the register pressure, and allows us to multiply with essentially no permutation instructions that are commonly used in vectorized implementation.

We implement the ideas for the NTRU parameter sets `ntruhps2048677` and `ntruhrss701` on a Cortex-A72 implementing the Armv8.0-A architecture with the single-instruction-multiple-data (SIMD) technology Neon. For polynomial multiplications, our implementation is 2.18× and 2.23× for `ntruhps2048677` and `ntruhrsss701` than the state-of-the-art optimized implementation. We also vectorize the polynomial inversions and sorting network by employing existing techniques and translating AVX2-optimized implementations into Neon. Compared to the state-of-the-art optimized implementation, our key generation, encapsulation, and decapsulation for `ntruhps2048677` are 7.67×, 2.48×, and 1.77× faster, respectively. For `ntruhrss701`, our key generation, encapsulation, and decapsulation are 7.99×, 1.47×, and 1.56× faster, respectively.

Keywords: Toeplitz matrix · NTRU · Vectorization · Dual Module

A. Chattopadhyay et al. (Eds.): INDOCRYPT 2023, LNCS 14460, pp. 177–196, 2024.
https://doi.org/10.1007/978-3-031-56235-8_9

1 Introduction

At PQCrypto 2016, the National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography (PQC) Standardization Process for replacing existing standards for public-key cryptography with quantum-resistant cryptosystems [15]. For lattice-based cryptosystems, polynomial multiplications had been the most time-consuming operations. In this paper, we investigate the interations between the underlying mathematical structure of polynomial rings and the architectural insights of vector-by-scalar multiplication instructions in instruction set architectures (ISAs).

In the NTRU submission [6] to the NIST PQC Standardization, polynomial rings of the form $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ and $\mathbb{Z}_q[x]\Big/\left\langle \sum_{i=0}^{n-1} x^i \right\rangle$ are used where $\mathbb{Z}_q$ is an integer ring, and n is a prime. Since $x^n-1=(x-1)\sum_{i=0}^{n-1} x^i$, multiplications in $\mathbb{Z}_q[x]\Big/\left\langle \sum_{i=0}^{n-1} x^i \right\rangle$ is often implemented as $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ followed by reduction modulo $\sum_{i=0}^{n-1} x^i$. In this paper, we focus on the polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n-1\rangle$.

Common approaches for multiplying two size-n polynomials in $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ usually multiply in $\mathbb{Z}_q[x]$ and reduce modulo x^n-1. Let $\boldsymbol{a}, \boldsymbol{b}$ be polynomials in $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ and f be an algebra monomorphism computing $\boldsymbol{ab}=f^{-1}\left(f(\boldsymbol{a})f(\boldsymbol{b})\right)$ in $\mathbb{Z}_q[x]$. Recent work [13] showed that the module-theoretic dual $f(\boldsymbol{a})^*$ can be used for multiplying a Toeplitz matrix and a vector. Since polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ can be regarded as a Toeplitz matrix-vector multiplication, we don't need the reduction modulo x^n-1 anymore.

In this paper, we point out the architectural benefit of Toeplitz matrix-vector products for ISAs implementing vector-by-scalar multiplication instructions. We show that the outer-product approach multiplying two matrices in cubic time implies efficient Toeplitz matrix-vector products with vector-by-scalar multiplication instructions.

1.1 Contributions

We summarize our contributions as follows.

- We point out the architectural benefit of Toeplitz matrix-vector products for vectorization and implement the ideas on a Cortex-A72 implementing Armv8.0-A where vector-by-scalar multiplication instructions are supported.
- We explain that Toeplitz matrix-vector product is actually a generic approach – it is only tied to the shape of polynomial rings and not the underlying monomorphism. Prior work [13] doesn't seem to observe this and they compared the Toeplitz matrix-vector product with Toom–Cook and the plain polynomial multiplication with number-theoretic transform[1] followed by reduction modulo x^n-1.

[1] Number-theoretic transform refers to a broad family of algebra monomorphisms that doesn't contain Toom–Cook.

– For the performance of polynomial multiplications, we outperform the state-of-the-art optimized implementation by 2.18× and 2.23× for the NTRU parameter sets `ntruhps2048677` and `ntruhrss701`, respectively.
– For the overall performance of the scheme, our `ntruhps2048677` key generation, encapsulation, and decapsulation is 7.67×, 2.48×, and 1.77× faster than the state-of-the-art optimized implementation; our `ntruhrss701` key generation, encapsulation, and decapsulation is 7.99×, 1.47×, and 1.56× faster than the state-of-the-art optimized implementation.

1.2 Code

Our source code is publicly available at
https://github.com/vector-polymul-ntru-ntrup/NTRU.

1.3 Structure of This Paper

This paper is structured as follows: Sect. 2 describes our target operations and platforms. Section 3 surveys polynomial transformations used for multiplications. Section 4 goes through the benefit of Toeplizt matrix–vector products. Section 5 describes our implementations. We show the performance numbers in Sect. 6.

2 Preliminaries

Sections 2.1 describe the polynomials rings in NTRU, and Sect. 2.2 describes our target platform Cortex-A72.

2.1 Polynomials in NTRU

The NTRU submission comprises two families NTRU-HPS and NTRU-HRSS. Both operate on polynomial rings $\mathbb{Z}_3[x]/\langle\Phi_n\rangle$, $\mathbb{Z}_q[x]/\langle\Phi_n\rangle$, and $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ where q is a power of 2, n is a prime, and Φ_n is the nth cyclotomic polynomial, which for prime n is $\frac{x^n-1}{x-1} = \sum_{i<n} x^i$. We target the parameter sets `ntruhps2048677` ($(q,n) = (2048, 677)$) and `ntruhrss701` ($(q,n) = 8192, 701$). For more parameter sets and details, we refer to the specification [6]. While NTRU also requires inversions in $\mathbb{Z}_3[x]/\langle\Phi_n\rangle$ and $\mathbb{Z}_q[x]/\langle x^n-1\rangle$, we focus on multiplying polynomials in $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ and $\mathbb{Z}_{8192}[x]/\langle x^{701}-1\rangle$.

2.2 Cortex-A72

Our target platform is the ARM Cortex-A72. Cortex-A72 implements the 64-bit Armv8.0-A instruction set architecture. It is a superscalar Central Processing Unit (CPU) with an in-order frontend and an out-of-order backend. We summarize some architectural features relevant to this paper, and refer to [1] for more details about the pipelines.

SIMD Registers. In Armv8.0-A, there are 32 architectural 128-bit SIMD registers each viewable as packed 8-, 16-, 32-, or 64-bit elements. The width of the element is specified the suffices `.16B` `.8H`, `.4S`, and `.2D` respectively on the register name. For referencing a certain lane, we use the annotation `.H[5]` for the 5th (zero-based) halfword of the register and similarly for other lanes and data widths [2, Figure A1-1].

Armv8-A Vector Instructions. A plain `mul` multiplies corresponding vector elements and returns same-sized results. Additionally, `mul` also refers to another instruction encoding — vector-by-scalar multiplication — if the last operand is a lane of a register. In this case `mul` multiplies the vector by a scalar (the lane value). This simple feature plays significant roles on maximizing register utilization and minimizing permutations. There are many variants of multiplications: `mla/mls` computes the same product vector and accumulates to or subtracts from the destination. Next, the shifts: `shl` shifts left; `sshr` arithmetically shifts right. For basic arithmetic, the usual `add/sub` adds/subtracts the corresponding elements. Then we have permutations — `uzp{1,2}` extracts the even and odd positions respectively from a pair of vectors and concatenates the results into a vector. `zip{1,2}` takes the bottom and top halves of a pair of vectors and riffle-shuffles them into the destination.

3 Polynomial Multiplications

This section surveys the Chinese remainder theorem for polynomial rings and Toom–Cook, and is structured as follows. We assume all the rings are commutative and unital in this paper. Sect. 3.1 reviews the Chinese remainder theorem for polynomial rings. This forms the basis of various fast polynomial ring transformations. Section 3.2 reviews Toom–Cook. Section 3.3 reviews the bit losses of Toom–Cook.

3.1 The Chinese Remainder Theorem for Polynomial Rings

Let $n = \prod_l n_l$ and $\boldsymbol{g}_{i_0,\ldots,i_{h-1}} \in R[x]$ be coprime polynomials for $i_l \in [0, n_l)$. The CRT gives us the following the isomorphism

$$\prod_{i_0,\ldots,i_{l-1}} \frac{R[x]}{\left\langle \prod_{i_l,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \prod_{i_0,\ldots,i_l} \frac{R[x]}{\left\langle \prod_{i_{l+1},\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle}$$

for all $l = 1, \ldots, h-1$[2]. We call each of the isomorphism "a layer of computation" and "a layer" for short. Usually, multiplications in $\prod_{i_0,\ldots,i_{h-1}} R[x] \Big/ \left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle$

[2] For possibly non-commutative unital rings, we only have $R[x]/\left(\langle \boldsymbol{g}_i \rangle \cap \langle \boldsymbol{g}_j \rangle\right) \cong R[x]/\langle \boldsymbol{g}_i \rangle \times R[x]/\langle \boldsymbol{g}_j \rangle$ for coprime polynomials $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$. If R is commutative, $R[x]$ is also commutative and we have $\langle \boldsymbol{g}_i \rangle \cap \langle \boldsymbol{g}_j \rangle = \langle \boldsymbol{g}_i \rangle \langle \boldsymbol{g}_j \rangle = \langle \boldsymbol{g}_i \boldsymbol{g}_j \rangle$. This leads to $R[x]/\langle \boldsymbol{g}_i \boldsymbol{g}_j \rangle \cong R[x]/\langle \boldsymbol{g}_i \rangle \times R[x]/\langle \boldsymbol{g}_j \rangle$ in our context.

are cheap. If all the layers are cheap, we have an algorithmic improvement for multiplying polynomials in $R[x]\Big/\left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle$. If the n_l is a small constant, then it is usually cheap to decompose a polynomial ring into a product of n_l polynomial rings.

3.2 Toom–Cook (TC) and Karatsuba

For a positive integer n, we define $R[x]_{<n}$ as $\{\boldsymbol{a}(x) \in R[x] | \deg(\boldsymbol{a}(x)) < n\}$, the set of polynomials with degree less than n. Toom–Cook [7,17] and Karatsuba [11] are divide-and-conquer approaches for multiplying polynomials in $R[x]$. We can also use them for multiplying polynomials in $R[x]_{<n}$. We introduce $y \sim x^{\frac{n}{k}}$ (zero-pad so that $k|n$) [4], and map $R[x]_{<n} \hookrightarrow R[x]\big/\left\langle x^{\frac{n}{k}} - y \right\rangle [y]_{<k} \hookrightarrow R'[y]_{<k}$ for $R' = R[x]/\langle \boldsymbol{g} \rangle$ with $\deg \boldsymbol{g} \geq \frac{2n}{k} - 1$.

For $\boldsymbol{a}, \boldsymbol{b} \in R'[y]_{<k}$, a k-way Toom–Cook computes $\boldsymbol{ab} \in R'[y]_{<2k-1}$ via *evaluating* $\boldsymbol{a}, \boldsymbol{b}$ at suitably chosen s_i's in R'. In other words, we apply the map $R'[y]_{<k} \hookrightarrow R'[y]\Big/\left\langle \prod_{i=0}^{2k-2} (y - s_i) \right\rangle \cong \prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$.

If one of the *evaluation points* is $s_i = \infty$, the corresponding map into $R'[y]/\langle y - s_i \rangle$ takes the highest degree coefficient (deg-$(k-1)$ for $\boldsymbol{a}, \boldsymbol{b}$, deg-$(2k-2)$ for $\boldsymbol{ab}$). [11] chose $k = 2$ at $\{s_i\}_i = \{0, 1, \infty\}$; [17] chose $\{s_i\}_i = \{0, \pm 1, \ldots, \pm(k-1)\}$; and [18, Page 31] replaced $-k+1$ with ∞. We write $\mathbf{TC}_{(2k-1)\times k}$ for the matrix mapping the coefficients of a deg $< k$ polynomial into $\prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$ and $\mathbf{TC}^{-1}_{(2k-1)\times(2k-1)}$ for the matrix mapping $\prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$ into $R[y]\Big/\left\langle \prod_{i=0}^{2k-2} (y - s_i) \right\rangle$.

A key observation is that while working over $\mathbb{Z}_{2^k}$ for $k = 5$ and $\{s_i\} = \left\{0, \pm 1, \pm 2, \pm\frac{1}{2}, 3, \infty\right\}$, $\mathbf{TC}^{-1}_{9\times 9}$ only requires "division by 8". This implies 3-bit losses. The matrix $\mathbf{TC}^{-1}_{9\times 9}$ will be stated explicitly in the full version.

3.3 Enlarging Coefficient Rings

We briefly explain how to divide a power of 2 when 2 is not invertible, for example while working over $\mathbb{Z}_{2^k}$. Suppose we want $r \in \mathbb{Z}_{2^k}$. We instead compute $2^\epsilon r \in \mathbb{Z}_{2^{k+\epsilon}}$, and right-shift $2^\epsilon r$ by ϵ bits [4, Section 7, Paragraph "What to do when 2 is not invertible"]. For our Toom–Cook defined over $\mathbb{Z}_{2^k}$, we would compute in $\mathbb{Z}_{2^{16}}$ so $r = \frac{2^{16-k} r}{2^{16-k}} \in \mathbb{Z}_{2^k}$ can be derived by right-shifting $2^{16-k} r \in \mathbb{Z}_{2^{16}}$ by $16 - k$ bits.

4 Toeplitz Matrix–Vector Product

In this section, we go through the benefit of Toeplitz matrix–vector products. The fundamental of using Toeplitz matrix–vector product is best described via R-modules, dual R-modules, and associative R-algebra. When the context is clear, we call an R-module a module and an associative R-algebra an algebra.

Section 4.1 reviews some basics about modules and algebras. Section 4.2 distinguish the inner-product-based and outer-product-based approaches for matrix–vector product. Section 4.3 introduces Toeplitz matrix–vector product. Section 4.4 explains the benefit of vector-by-scalar multiplications. Section 4.5 presents the generic Toeplitz matrix–vector product conversion from ring monomorphisms computing the double-size products.

4.1 Module and Associative Algebra

This section goes through some basics about modules, dual modules, and associative algebras. Readers familiar with these basic algebraic structures can skip this section.

Module and Dual Module. Let $(M, +)$ be an abelian group and R a ring. We turn M into an R-module by introducing a scalar multiplication $\cdot_M : R \times M \to M$ (we write $r \cdot_M \boldsymbol{a}$ for $(\cdot_M)(r, \boldsymbol{a})$) satisfying the following:

- $\forall \boldsymbol{a}, \boldsymbol{b} \in M, \forall r, s \in R, (r + s) \cdot_M (\boldsymbol{a} + \boldsymbol{b}) = r \cdot_M \boldsymbol{a} + r \cdot_M \boldsymbol{b} + s \cdot_M \boldsymbol{a} + s \cdot_M \boldsymbol{b}$.
- $\forall \boldsymbol{a} \in M, 1 \cdot_M \boldsymbol{a} = \boldsymbol{a}$.
- $\forall \boldsymbol{a} \in M, \forall r, s \in R, (rs) \cdot_M \boldsymbol{a} = r \cdot_M (s \cdot_M \boldsymbol{a})$.

We call $(M, +, \cdot_M)$ a left R-module. One can define a right R-module in a similar way by identifying a scalar multiplication from $M \times R$ to M. Since we assume R is commutative, we do not distinguish between left and right R-modules and simply call them R-modules. For elements $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1} \in M$, if they are linearly independent and every element in M can be expressed as a linear combination of $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}$, we call $\{\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}\}$ a basis of M and n the rank. A free module of rank n is a module with a basis of n elements and is very close to an n-dimensional vector space in our context. We denote by R^n for the free module of rank n. Notice that a ring R and a polynomial ring $R[x]/\langle \boldsymbol{g} \rangle$ are free modules, and the matrix ring $M_{n \times n}(R)$ is an R-module.

An R-module homomorphism is a map $\eta : M \to N$ satisfying:

$$\forall r \in R, \forall \boldsymbol{a}, \boldsymbol{b} \in M, \eta(r \cdot_M \boldsymbol{a} + \boldsymbol{b}) = r \cdot_N \eta(\boldsymbol{a}) + \eta(\boldsymbol{b}).$$

One can verify that the set of R-module homomorphisms $\mathrm{Hom}_R(M, R)$ from M to R is an R-module. We call $\mathrm{Hom}_R(M, R)$ the dual of M, and denote it as M^*. If M is a free R-module of finite rank, it is isomorphic to M^*. For an R-module homomorphism $\eta : M \to N$, we define the transpose of η as the R-module homomorphism $\eta^* : N^* \to M^*$ sending $\boldsymbol{a}^*$ to $\boldsymbol{a}^* \circ \eta$.

Associative Algebra. For rings R and $\mathcal{A}$, we turn $\mathcal{A}$ into an associative R-algebra by introducing a module structure. One identifies the module addition with the ring addition, and provide a scalar multiplication $\cdot_{\mathcal{A}} : R \times \mathcal{A} \to \mathcal{A}$ for the module structure satisfying

$$\forall r \in R, \forall \boldsymbol{a}, \boldsymbol{b} \in \mathcal{A}, r \cdot_{\mathcal{A}} (\boldsymbol{a}\boldsymbol{b}) = (r \cdot_{\mathcal{A}} \boldsymbol{a})\boldsymbol{b} = \boldsymbol{a}(r \cdot_{\mathcal{A}} \boldsymbol{b}).$$

An R-algebra homomorphism is a map that is a ring homomorphism and a module homomorphism at the same time.

Obviously, a polynomial ring is an R-algebra and all the ring monomorphisms in Sect. 3 are also module monomorphisms; therefore, they are algebra monomorphisms.

4.2 Matrix–Vector Products

There are two basic ways to multiply a matrix by a vector. For a matrix M, we denote $M[i_0][i_1]$ for the (i_0, i_1)-th entry, $M[i_0][-]$ for the i_0-th row, and $M[-][i_1]$ for the i_1-th column of M. Let A be an $n_0 \times n_1$ matrix a B be a column vector of n_1 elements. We wish to compute the matrix-vector product $C = AB$. Algorithm 1 computes the result with several inner products of the rows of the matrix and the vector. Algorithm 2 accumulates several products of the columns of the matrix and the corresponding elements of the vector.

Algorithm 1. Inner-product-based matrix–vector multiplication.

1: **for** $i_0 = 0, \ldots, n_0 - 1$ **do**
2: **for** $i_1 = 0, \ldots, n_1 - 1$ **do**
3: $C[i_0] = C[i_0] + A[i_0][i_1]B[i_1]$
4: **end for**
5: ▷ Inner product of the vectors $A[i_0][-]$ and $B[-]$.
6: **end for**

Algorithm 2. Outer-product-based matrix–vector multiplication.

1: **for** $i_1 = 0, \ldots, n_1 - 1$ **do**
2: **for** $i_0 = 0, \ldots, n_0 - 1$ **do**
3: $C[i_0] = C[i_0] + A[i_0][i_1]B[i_1]$
4: **end for**
5: ▷ Outer product of the vectors $A[-][i_1]$ and $B[i_1]$.
6: **end for**

In the context of a vector instruction set, the former translates into vector-by-vector multiplications with interleaved operands, requiring transposition of the inputs and outputs, and a larger number of registers. The latter can be easily implemented with vector-by-scalar multiplications, requiring much fewer permutation instructions and less rigid instruction scheduling. It is easily seen that in the context of matrix multiplications, Algorithm 1 is a special case of the inner product approach (cf. Algorithm 3), and Algorithm 2 is a special case of the outer product approach (cf. Algorithm 4). We also call them accordingly.

Algorithm 3. Inner-product-based matrix–matrix multiplication.

```
for i_0 = 0, ..., n_0 − 1 do
    for i_1 = 0, ..., n_1 − 1 do
        for i_2 = 0, ..., n_2 − 1 do
            A[i_0][i_1] = C[i_0][i_1] + A[i_0][i_2]B[i_2][i_1]
        end for
                          ▷ Inner product of the vectors A[i_0][-] and B[-][i_1].
    end for
end for
```

Algorithm 4. Outer-product-based matrix–matrix multiplication.

```
for i_2 = 0, ..., n_2 − 1 do
    for i_0 = 0, ..., n_0 − 1 do
        for i_1 = 0, ..., n_1 − 1 do
            C[i_0][i_1] = C[i_0][i_1] + A[i_0][i_2]B[i_2][i_1]
        end for
    end for
                          ▷ Outer product of the vectors A[-][i_2] and B[i_2][-].
end for
```

4.3 Toeplitz Matrices

Let M be an $m \times n$ matrix over the ring R. We call it a Toeplitz matrix if it takes the form

$$M = \begin{pmatrix} a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \\ a_n & a_{n-1} & \cdots & a_2 & a_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m+n-3} & a_{m+n-4} & \cdots & a_{m-1} & a_{m-2} \\ a_{m+n-2} & a_{m+n-3} & \cdots & a_m & a_{m-1} \end{pmatrix}, \text{for all possible } i, j, M_{i,j} = M_{i+1,j+1}.$$

We denote M as $\mathbf{Toeplitz}_{m\times n}(a_{m+n-2}, \ldots, a_0)$.

Toeplitz Matrices for Weighted Convolutions. For a weighted convolution $\boldsymbol{c} = \boldsymbol{ab} = \left(\sum_i a_i x^i\right)\left(\sum_i b_i x^i\right) \in R[x]/\langle x^n - \zeta\rangle$, we choose an $n' \geq n$, zero-pad $\boldsymbol{a}$ and $\boldsymbol{c}$ to size-n' polynomials $\boldsymbol{a}'$ and $\boldsymbol{c}'$, respectively, and define $\texttt{Expand}_{n\to n',\zeta} = \left(\sum_{i<n} b_i x^i, \zeta\right) \mapsto \left(\underbrace{0, \ldots, 0}_{n'-n}, b_{n-1}, \ldots, b_0, \zeta b_{n-1}, \ldots, \zeta b_1, \underbrace{0, \ldots, 0}_{n'-n}\right)$. We have

$$\boldsymbol{c}' = \mathbf{Toeplitz}_{n'\times n'}\left(\texttt{Expand}_{n\to n',\zeta}\left(\boldsymbol{b}\right)\right)\boldsymbol{a}'.$$

$\mathbf{Toeplitz}_{n\times n}\left(\texttt{Expand}_{n\to n,\zeta}(-)\right)(-)$ is exactly the `asymmetric_mul` by [3, Section 4.2]. See [8, Paragraph "A Toeplitz matrix view of asymmetric multiplication", Sect. 8.3.2] for explanations.

4.4 Small-Dimensional Cases

Toeplitz matrix–vector multiplications are extensively used in our implementations. For a fast polynomial ring transformation resulting weighted convolutions, we apply the outer-product-based Toeplitz matrix–vector multiplication. Existing works [3,14,16] applied the inner product approach with pre-and post-transposes. The Toeplitz structure admits fast construction of the full matrix. For a weighted convolution over $x^4 - \zeta$, we apply $\texttt{Expand}_{4\to4,\zeta}$ with `ext` instructions, and accumulate vector-by-scalar products. Algorithm 5 is an illustration.

Algorithm 5. Outer product approach for $R[x]/\langle x^4 - \zeta \rangle$.

Inputs: $\boldsymbol{a} = a_0 + a_1x + a_2x^2 + a_3x^3$, $\boldsymbol{b} = b_0 + b_1x + b_2x^2 + b_3x^3$.
Outputs: $\boldsymbol{c} = \boldsymbol{ab} \bmod (x^4 - \zeta)$.

1: $\texttt{b} = b_3||b_2||b_1||b_0$
2: $\texttt{t0} = a_3||a_2||a_1||a_0$
3: Compute $\texttt{t} = \zeta a_3||\zeta a_2||\zeta a_1||\zeta a_0$ with Barrett multiplication.
4: ▷ [3] proposed an interleaved version of this; others [14,16] reduced the interleaved partial results instead.
5: ▷ The remaining steps are different from [3].
6: `ext t1, t, t0, #`$3 \cdot 4$ ▷ $\texttt{t1} = a_2||a_1||a_0||\zeta a_3$
7: `ext t2, t, t0, #`$2 \cdot 4$ ▷ $\texttt{t2} = a_1||a_0||\zeta a_3||\zeta a_2$
8: `ext t3, t, t0, #`$1 \cdot 4$ ▷ $\texttt{t3} = a_0||\zeta a_3||\zeta a_2||\zeta a_1$
9: $(\texttt{lo},\texttt{hi}) = (\texttt{smull},\texttt{smull2})(\texttt{t0}, b_0)$
10: $(\texttt{lo},\texttt{hi}) = (\texttt{lo},\texttt{hi})(\texttt{smlal},\texttt{smlal2})(\texttt{t1}, b_1)$
11: $(\texttt{lo},\texttt{hi}) = (\texttt{lo},\texttt{hi})(\texttt{smlal},\texttt{smlal2})(\texttt{t2}, b_2)$
12: $(\texttt{lo},\texttt{hi}) = (\texttt{lo},\texttt{hi})(\texttt{smlal},\texttt{smlal2})(\texttt{t3}, b_3)$
13: $\boldsymbol{c} =$ `Montgomery_long`$(\texttt{lo},\texttt{hi})$

Generally speaking, once the Toeplitz matrix is constructed via `ext`s or memory loads (recall that we can instead store an $n \times n$ Toeplitz matrix as an array of $2n-1$ elements), vector-by-scalar multiplications significantly reduce the register pressure and remove the follow up permutation instructions. We illustrate the differences between inner-product-based and outer-product-based Toeplitz matrix–vector multiplication for

$$\begin{pmatrix} a_0 & a_1' & a_2' & a_3' \\ a_1 & a_0 & a_1' & a_2' \\ a_2 & a_1 & a_0 & a_1' \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

where $a_1' = \zeta a_3, a_2' = \zeta a_2$, and $a_3' = \zeta a_1$ for the weighted convolutions defined in $R[x]/\langle x^4 - \zeta \rangle$. Figure 2 illustrates the register view of inner-product-based Toeplitz matrix–vector multiplication and Fig. 1 for the outer-product-based one. For Fig. 2, we apply $\log_2 4 \cdot \frac{4}{2} = 4$ pairs of $(\texttt{trn1},\texttt{trn2})$ to each operand to reach the register view. While applying vector-by-vector multiplications, the interleaved operands occupy 11 registers and the interleaved partial results occupy

4 or 8 registers (this depends on the coefficient ring). Finally, we also need to transpose the interleaved results with 4 pairs of (`trn1`, `trn2`). On the other hand, Fig. 1 requires no additional permutations and avoids the interleaved operands and results. This implies nearly no permutation instructions and very low register pressure.

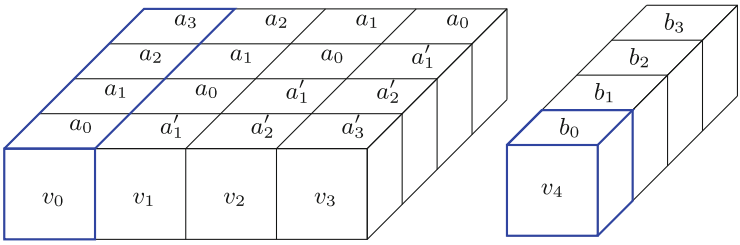

Fig. 1. Outer-product-based Toeplitz matrix–vector multiplication via vector-by-scalar multiplication. No permutations are required once we have data in registers $v_0, \ldots, v_3$. We only need 5 registers $v_0, \ldots, v_4$ for holding the operands and 1 or 2 registers for the partial results. (Color figure online)

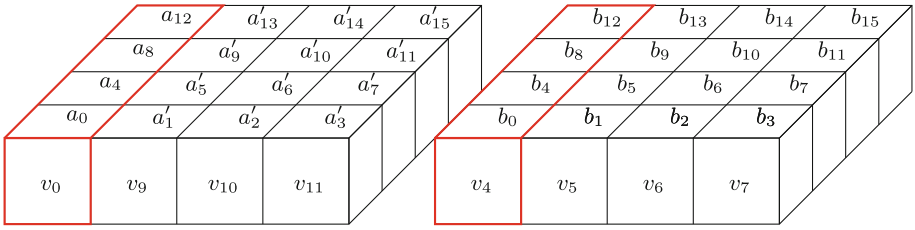

Fig. 2. Inner-product-based Toeplitz matrix–vector multiplication via vector-by-vector multiplication. One load $(a_0, \ldots, a_3), \ldots, (a_{12}, \ldots, a_{15})$ into registers $(v_0, \ldots, v_3)$, and transpose the registers as a 4×4 matrix. Same for $(v_4, \ldots, v_7)$ holding $(b_0, \ldots, b_3), \ldots, (b_{12}, \ldots, b_{15})$ and $(v_8, \ldots, v_{11})$ holding $(c_0, \ldots, c_3), \ldots, (c_{12}, \ldots, c_{15})$. Notice that we need to hold the registers $v_0, v_4, \ldots, v_{11}$ for computing $a_0b_0 + c_1b_1 + c_2b_2 + c_3b_3$. Therefore, we need 11 registers (we don't need (c_0, c_4, c_8, c_{12})) for the operands. Since we also need registers for holding the partial results (4 registers for $\mathbb{Z}_{2^{16}}$ and 8 registers otherwise), the register pressure is high and forbids us to generalize to size-16 computations. (Color figure online)

4.5 Large-Dimensional Toeplitz Transformation

There are several benefits when working on Toeplitz matrices. Firstly, we only need to store $m + n - 1$ coefficients $M_{m-1,0}, \ldots, M_{0,0}, \ldots, M_{0,n-1}$ of the matrix. Secondly, additions/subtractions of two Toeplitz matrices require only

$m+n-1$ additions/subtractions in R. Finally, submatrices from adjacent rows and columns are also Toeplitz matrices. These properties enable efficient divide-and-conquer computations when the dimension is large.

For the sake of generality, multiplying two polynomials $\boldsymbol{a}, \boldsymbol{b} \in R[x]_{<k}$ will be considered as $\boldsymbol{ab} \in R[x]_{<n}$ with $n \geq 2k-1$. Given an $\boldsymbol{a} \in R[x]_{<k}$, we write $(\boldsymbol{a}, -) : R^k \to R^n$ for the module homomorphism $\boldsymbol{b} \mapsto \boldsymbol{ab}$ and $(\boldsymbol{a}, -)^*$ its transpose. Suppose we have an R-algebra S where multiplications are much faster than in $R[x]_{<n}$, the Toeplitz matrix-vector product (TMVP) can be defined for an R-algebra homomorphism $f : R[x]_{<n} \to S$ with $f|_{R[x]_{<k}}$ a monomorphism.

Definition 1. Let S be an R-algebra and $f : R[x]_{<n} \to S$ be an R-algebra homomorphism, with $f_k \coloneqq f|_{R[x]_{<k}} : R[x]_{<k} \to S$ a monomorphism. Furthermore, let $\mathrm{rev}_{k\to k} : R^k \to R^k$ be the index reversal map and $\mathrm{id}_{m\to n} : R^m \to R^n$ be the inclusion (pad 0's) map for $m \leq n$. The TMVP associated with f refers to the following module homomorphisms:

$$\left(\mathbf{Toeplitz}_{k\times k}(-)\right)(\boldsymbol{a}) = \mathrm{rev}_{k\times k} \circ f_k^* \circ (f_k(\boldsymbol{a}), -)^* \circ (f^{-1})^* \circ \mathrm{id}_{(2k-1)\to n}.$$

We call $(f^{-1})^* \circ \mathrm{id}_{(2k-1)\to n}$ split–matrix, $f_k(\boldsymbol{a})$ split–vector, $(f_k(\boldsymbol{a}), -)^*$ base multiplication, and f_k^* interpolation. If $n = 2k-1$, $f = \mathbf{TC}_{(2k-1)\times(2k-1)}$, then this is the k-way Toeplitz-TC matrix–vector product [12,13]. Generally, any R-algebra monomorphism suffices. See Appendices A for a formal proof and B for examples. We go through a higher-level overview of the idea.

Since f is a ring monomorphism, we implement the module homomorphism $(\boldsymbol{a}, -)$ as $\mathrm{id}_{n\to(2k-1)} \circ f^{-1} \circ (f_k(\boldsymbol{a}), -) \circ f_k$, take the transpose of $(\boldsymbol{a}, -)$, and relate $(\boldsymbol{a}, -)^*$ to the Toeplitzation $\mathbf{Toeplitz}_{k\times k}(-)$ and the right-vector-multiplication $(-)(\boldsymbol{a})$. This allows us to convert any fast computation for $(\boldsymbol{a}, -)$ into something for $\left(\mathbf{Toeplitz}_{k\times k}(-)\right)(\boldsymbol{a})$. Since $\left(\mathbf{Toeplitz}(-)\right)(\boldsymbol{a}) = \mathrm{rev}_{k\times k} \circ (\boldsymbol{a}, -)^*$, and $(-, \boldsymbol{a})_{R[x]/\langle x^k-\zeta\rangle} = \mathrm{id}_{k\to k} \circ \left(\mathbf{Toeplitz}(-)\right)(\boldsymbol{a}) \circ \mathtt{Expand}_{k\to k,\zeta}$ as shown in Fig. 3, we eventually have a fast computation for $(-, \boldsymbol{a})_{R[x]/\langle x^k-\zeta\rangle}$.

5 Implementations

We propose two implementations for `ntruhps2048677` with 16-bit arithmetic modulo 65536: (i) `Toom-Cook` implements Toom–Cook with the splitting sequence $5 \to 3 \to 3 \to 2$, and (ii) `Toeplitz-TC` computes the Toeplitz matrix–vector product derived from `Toom-Cook`. Our `Toom-Cook` applies a more aggressive divide-and-conquer than prior works [9,14] by carefully choosing the point set for evaluations. Our `Toeplitz-TC` reveals the benefit of vector-by-scalar multiplications, which is more significant than the findings of [13].

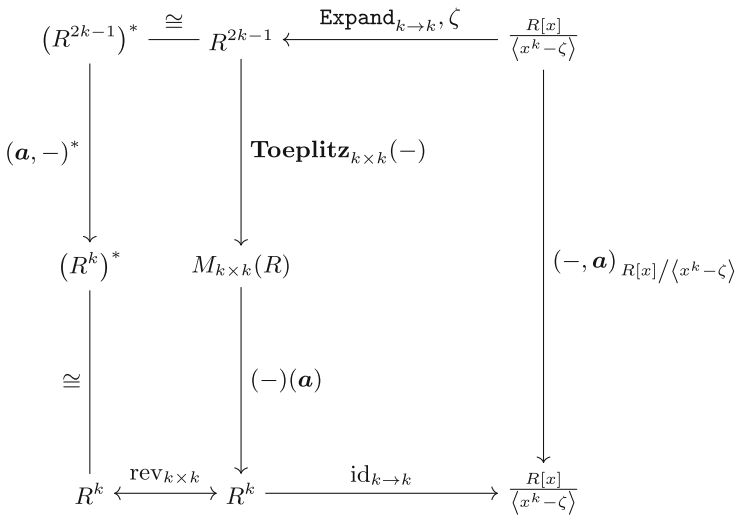

Fig. 3. Relations between $(-, \boldsymbol{a})_{R[x]/\langle x^k-1\rangle}$, $(\mathbf{Toeplitz}_{k\times k}(-))(\boldsymbol{a})$, and $(\boldsymbol{a}, -)^*$.

For `ntruhrss701`, we implement the `Toeplitz-TC` approach with the same splitting sequence $5 \to 3 \to 3 \to 2$ with the 3's referring to 3-way Karatsuba instead of Toom-3. We skip the Toom–Cook approach since it is obviously not worth implementing given the experience from `ntruhps2048677`. Since the implementation `Toeplitz-TC` of `ntruhrss701` is very close to the one for `ntruhps2048677`, we skip the description for `ntruhrss701`.

Section 5.1 describes the `Toom-Cook` approach, and Sect. 5.2 describes the `Toeplitz-TC` approach. Additionally, we summarize existing strategies multipliying polynomials in `ntruhps2048677` in Table 1.

5.1 `Toom-Cook`

We first describe our chosen Toom–Cook splitting sequence and implementation considerations. We then detail our memory optimization for the interpolation of $\mathbf{TC}^{-1}$.

Chosen Splitting Sequence. We choose the splitting sequence Toom–5 $\to$ two Toom–3's $\to$ Karatsuba. We first zero-pad the size-677 polynomials to size-720 for ease of vectorization and compute in $\mathbb{Z}_{2^{16}}$. Since the coefficient ring of `ntruhps2048677` is $\mathbb{Z}_{2048}$ and $\frac{2^{16}}{2048} = 2^5$, divisions by 2^e for $e = 0, \ldots, 5$ translate into shifting e bits. We choose the splitting sequence $5 \to 3 \to 3 \to 2$. Our `Toom-Cook` consists of one layer of $\mathbf{TC}_{9\times 5}$, two layers of $\mathbf{TC}_{5\times 3}$'s, one layer of

Table 1. Overview of divide-and-conquer strategies multiplying polynomials in $R[x]_{<720}$ for `ntruhps2048677`. We first start with $R[x]_{<720}$ and alternatingly list all the number of subproblems of divide-and-conquer and the resulting ring. For example, the sequence $R[x]_{<720}, 4 \to 7, R[x]_{<180}$ means that size-720 polynomials are first sectioned into four size-180 polynomials, and mapped to seven size-180 polynomials, and the resulting polynomial multiplications defined in $R[x]_{180}$.

	[13]	[14]	This work
Ring	$R[x]_{<720}$	$R[x]_{<720}$	$R[x]_{<720}$
Divide-and-conquer	$4 \to 7$	$3 \to 5$	$5 \to 9$
Ring	$R[x]_{<180}$	$R[x]_{<240}$	$R[x]_{<144}$
Divide-and-conquer	$3 \to 5$	$4 \to 7$	$3 \to 5$
Ring	$R[x]_{<60}$	$R[x]_{<60}$	$R[x]_{<48}$
Divide-and-conquer	$3 \to 5$	$2 \to 3$	$3 \to 5$
Ring	$R[x]_{<20}$	$R[x]_{<30}$	$R[x]_{<16}$
Divide-and-conquer	$2 \to 3$	$2 \to 3$	$2 \to 3$
Ring	$R[x]_{<10}$	$R[x]_{<15}$	$R[x]_{<8}$

$\mathbf{TC}_{3\times2}$, 675 size-8 schoolbooks, one layer of $\mathbf{TC}_{3\times3}^{-1}$, two layers of $\mathbf{TC}_{5\times5}^{-1}$'s, and one layer of $\mathbf{TC}_{9\times9}^{-1}$. We choose the point sets $\left\{0, \pm1, \pm2, \pm\frac{1}{2}, 3, \infty\right\}$ (cf. Sect. 3.2) for $\mathbf{TC}_{9\times5}$ and $\{0, \pm1, 2, \infty\}$ for $\mathbf{TC}_{5\times3}$. The interpolation matrices $\mathbf{TC}_{9\times9}^{-1}$, $\mathbf{TC}_{5\times5}^{-1}$, and $\mathbf{TC}_{3\times3}^{-1}$ incur 3-, 1-, and 0-bit losses of precision, respectively. These add up to 5 bits, allowing us to invert correctly.

Comparisons to Prior Splitting Sequence [14]. [14] treated each polynomial as a size-720 polynomial, and applied Toom–Cook with the splitting sequence $3 \to 4 \to 2 \to 2$. The polynomial size goes down to 240 after the Toom-3, 60 after the Toom-4, and 15 after two Karatsuba's. Since 60 is not a multiple of 8, [14] basically padded to size-64 polynomials before Karatsuba. In this paper, we instead split via the sequence $5 \to 3 \to 3 \to 2$ down to size-8 schoolbooks. Our evaluation points for Toom-5 has the same precision loss as Toom-4. This is 1 fewer bit than the standard $\{0, \pm1, \pm2, \pm3, 4, \infty\}$. We also avoid zero-padding in vectorization. We merge the two Toom-3 layers (for both $\mathbf{TC}_{5\times3}$ and $\mathbf{TC}_{5\times5}^{-1}$) to reduce memory operations.

Memory Optimizations for Interpolations. Let $k|n$, $\boldsymbol{g}'$ be a polynomial of degree at least $\frac{2n}{k} - 1$, and $R' = R[x]/\langle \boldsymbol{g}' \rangle$. Recall that $\mathbf{TC}_{(2k-1)\times k}$ computes $R[x]/\langle x^{\frac{n}{k}} - y \rangle\ [y] \hookrightarrow R'[y]\Big/\left\langle \prod_{i=0}^{2k-2} (y - s_i) \right\rangle \cong \prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$ and results in computations in $R[x]/\langle \boldsymbol{g}' \rangle$. After examining the source code, we find that prior works [9,14] inverted the steps $\cong$ and $\hookrightarrow$ separately. Algorithm 6 is an illustration. Inverting $\cong$ means applying the interpolation matrix and inverting $\hookrightarrow$ means accumulating the overlapped coefficients while substituting y with

$x^{\frac{n}{k}}$ in each of the polynomials in $R[x]/\langle \boldsymbol{g}' \rangle$. We instead alternate between the inversions of $\cong$ and $\hookrightarrow$ to reduce memory operations, *in essence merging two layers of computations*.

Algorithm 6. $\mathbf{TC}_{5\times5}^{-1}$ by [9,14].

Input: Size-3 polynomials $p_0, \ldots, p_4$.
Output: `c[0-10]` $= \mathbf{TC}_{5\times5}^{-1}(p_0, p_1, p_2, p_3, p_4)$.

1: Declare array `mem[5]`.
2: **for** $i = \{0, 1, 2\}$ **do**
3: `mem[0-4]` $= \mathbf{TC}_{5\times5}^{-1}(p_0\texttt{[}i\texttt{]}, \ldots, p_4\texttt{[}i\texttt{]})$.
4: ▷ Memory read and write.
5: **for** $j = \{0, \ldots, 4\}$ **do**
6: `c[`$2j+i$`]` = `c[`$2j+i$`]` + `mem[`j`]`
7: **end for**
8: ▷ Memory read and write.
9: **end for**

Algorithm 7. Our $\mathbf{TC}_{5\times5}^{-1}$.

Input: Size-3 polynomials $p_0, \ldots, p_4$.
Output: `c[0-10]` $= \mathbf{TC}_{5\times5}^{-1}(p_0, p_1, p_2, p_3, p_4)$.

1: Registers `r[11]`.
2: `r[0-4]` $= \mathbf{TC}_{5\times5}^{-1}(p_0\texttt{[0]}, \ldots, p_4\texttt{[0]})$
3: `r[6-10]` $= \mathbf{TC}_{5\times5}^{-1}(p_0\texttt{[2]}, \ldots, p_4\texttt{[2]})$
4: ▷ Memory read.
5: `r[5]` $= 0$
6: **for** $j = \{0, \ldots, 4\}$ **do**
7: `r[`$i+1$`]` $=$ `r[`$i+1$`]` $+$ `r[`$i+6$`]`
8: **end for**
9: **for** $j = \{0, \ldots, 5\}$ **do**
10: `c[`$2j$`]` $=$ `r[`j`]`
11: **end for**
12: ▷ Memory write.
13: `r[0-4]` $= \mathbf{TC}_{5\times5}^{-1}(p_0\texttt{[1]}, \ldots, p_4\texttt{[1]})$
14: ▷ Memory read.
15: **for** $j \leftarrow 0$ to 4 **do**
16: `c[`$2j+1$`]` $=$ `r[`j`]`
17: **end for**
18: ▷ Memory write.

5.2 `Toeplitz-TC`

We apply the Toeplitz matrix–vector product with **TC**'s as the underlying monomorphisms and choose the same splitting sequence $5 \to 3 \to 3 \to 2$. We call it `Toeplitz-TC`.

Our `Toeplitz-TC` with the Splitting Sequence $5 \to 3 \to 3 \to 2$. Algorithm 8 describes our `Toeplitz-TC` implementation. Essentially, we regard size-677 polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ as size-720 polynomials. In practice, we zero-pad $\boldsymbol{a}$ and $\boldsymbol{b}$ to length 680 and omit the computations involving the indices $680, \ldots, 719$. Then, we apply one layer of `Toeplitz-TC-5`, two layers of `Toeplitz-TC-3`'s, and one layer of `Toeplitz-TC-2`. Algorithm 9 describes our `Toeplitz-TC--3--3--2` following `Toeplitz-TC--5`.

Each steps of Algorithms 8 and 9 is implemented as a subroutine. We merge computations while using all available registers without register spills. Initializations to zeros and the corresponding computations are also omitted for efficiency. While applying **TC**, $\mathbf{TC}'^{-1*}$, and $\mathbf{TC}'^{*}$, we prefer shifts over multiplications and reuse intermediate values.

Algorithm 8. `Toeplitz-TC` for `ntruhps2048677`.

Input: size-720 polynomials `a`, `b`.
Output: the size-677 polynomial $\mathtt{c} = \mathtt{ab} \bmod (x^{677} - 1)$.

1: Declare `uint16_t buff_a[9][288], buff_b[9][144], buff_c[9][144]`.
2: `buff_a[0-8][0-287]` $= \mathbf{TC}^{-1*}_{9\times 9}$ (`a`)
3: ▷ See Section 4.5 for definition.
4: `buff_b[0-8][0-143]` $= \mathbf{TC}_{9\times 5}$ (`b`)
5: **for** $i = \{0, \ldots, 8\}$ **do**
6: `buff_c[`i`][0-143]` $=$ `Toeplitz-TC-3-3-2` (`buff_a[`i`][0-287], buff_b[`i`][0-143]`)
7: **end for**
8: `c[0-676]` $= \mathbf{TC}^{*}_{9\times 5}$ (`buff_c[0-8][0-143]`)

Algorithm 9. `Toeplitz-TC-3-3-2`.

Input: a 144×144 Toeplitz matrix `M`, and a size-144 vector `v`.
Output: the size-144 vector $\mathtt{c} = \mathtt{M} \cdot \mathtt{v}$.

1: Declare `uint16_t M1[5][96], M2[5][5][3][16]`.
2: Declare `uint16_t v1[5][5][16], c1[5][5][16]`.
3: `M1[0-4][0-95]` $= \mathbf{TC}^{-1*}_{5\times 5}$ (`M[0-143][0-143]`)
4: **for** $i = \{0, \ldots, 4\}$ **do**
5: `M2[`i`][0-4][0-2][0-15]` $= \left(\mathbf{TC}^{-1*}_{3\times 3} \circ \mathbf{TC}^{-1*}_{5\times 5}\right)$ (`M1[`i`][0-95]`)
6: **end for**
7: `v1[0-4][0-4][0-15]` $= (\mathbf{TC}_{5\times 3} \circ \mathbf{TC}_{5\times 3})$ (`v`)
8: `c1[`i`][`j`][0-15]` $= \mathbf{TC}^{*}_{3\times 2}$ (`M2[`i`][`j`][0-2][0-15]` $\cdot\, \mathbf{TC}_{3\times 2}$ (`b1[`i`][`j`][0-15]`))
9: `c[0-143]` $= (\mathbf{TC}^{*}_{5\times 3} \circ \mathbf{TC}^{*}_{5\times 3})$ (`c1[0-4][0-4][0-15]`)

Comparisons to [13]. [13] implemented the Toeplitz matrix–vector product with $\mathbf{TC}_{(2k-1)\times k}$ as the underlying monomorphisms on Cortex-M4, but they chose the splitting sequence $4 \to 3 \to 2 \to 2$. We improve the efficiency by applying a more aggressive splitting sequence. For the first layer, we use Toeplitz–TC–5 instead of Toeplitz–TC–4. Both strategies yield 3-bit losses. Although our $\mathbf{TC}^{-1*}_{9\times 9}$, $\mathbf{TC}_{9\times 5}$, and $\mathbf{TC}^{*}_{9\times 5}$ require more multiplications, we have a smaller number of schoolbooks, which is the bottleneck of the computation. Compared to [13], our Cortex-A72 implementation reaches the best performance with size-8 schoolbooks instead of size-16 ones. Also, [13] used $\mathbf{TC}^{-1*}_{(2k-1)\times(2k-1)}$, $\mathbf{TC}_{(2k-1)\times k}$ and $\mathbf{TC}^{*}_{(2k-1)\times k}$ to compute while we multiply some constants to the precomputed matrices for easier computation. The modified $\mathbf{TC}^{-1*}_{(2k-1)\times(2k-1)}$, $\mathbf{TC}_{(2k-1)\times k}$ and $\mathbf{TC}^{*}_{(2k-1)\times k}$ will be shown in the full version.

6 Results

We present the performance numbers in this section. We focus on polynomial multiplications, leaving the fast constant-time GCD [5] as future work.

6.1 Benchmark Environment

We use the Raspberry Pi 4 Model B featuring the quad-core Broadcom BCM2711 chipset. It comes with a 32 kB L1 data cache, a 48 kB L1 instruction cache, and a 1 MB L2 cache and runs at 1.5 GHz. For hashing, we use the `aes`, `sha2`, and `fips202` from PQClean [10] without any optimizations due to the lack of corresponding cryptographic units. For the randombytes, [3] used the `randombytes` from SUPERCOP which in turn used `chacha20`. We extract the conversion from `chacha20` into `randombytes` from SUPERCOP and replace `chacha20` with our optimized implementations using the pipelines `I0/I1`, `F0/F1`. We use the cycle counter of the PMU for benchmarking. Our programs are compilable with GCC 10.3.0, GCC 11.2.0, Clang 13.1.6, and Clang 14.0.0. We report numbers for the binaries compiled with GCC 11.2.0.

6.2 Performance of Vectorized Polynomial Multiplications

Table 2 summarizes the performance of vectorized polynomial multiplications. All of our implementations outperform the state-of-the-art Toom–Cook [14]. For `ntruhps2048677`, our `Toeplitz-TC` and `Toom-Cook` are 2.18× and 1.56× faster than [14]. Comparing `Toeplitz-TC` and `Toom-Cook` based on the same splitting sequence, the result is consistent to [13]. But the most significant reason is the use of vector-by-scalar multiplications. This finding is new. For `ntruhrss701`, we outperform [14]'s implementation by 2.23×.

Table 2. Overview of `polymuls`.

	`ntruhps2048677`	`ntruhrss701`
Implementation	Cycles	
[14]	58 286	70 061
`Toeplitz-TC`	26 784	31 478
`Toom-Cook`	37 318	–

6.3 Performance of Schemes

Before comparing the overall performance, we first illustrate the performance numbers of some other critical subroutines. Most of our optimized implementations of these subroutines are not seriously optimized except for parts involving polynomial multiplications. We simply translate existing techniques and AVX2-optimized implementations into Neon. Notice that inversions over $\mathbb{Z}_2$ and $\mathbb{Z}_3$, and sorting networks are implemented in a generic sense. With fairly little effort, they can be used for other parameter sets.

Inversions. For `ntruhps2048677`, we need one inversion in $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ and one inversion in $\mathbb{Z}_3[x]\Big/\left\langle \frac{x^{677}-1}{x-1}\right\rangle$. The inversion in $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ consists of one inversion in $\mathbb{Z}_2[x]/\langle x^{677}-1\rangle$ and lifting to $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ with eight polynomial multiplications since the coefficient ring is $\mathbb{Z}_{2048}$. We use the 1-bit form of $\mathbb{Z}_2$ for the inversion over $\mathbb{Z}_2$ without any algorithmic improvements and obtain a 20.41× speedup, leading to 10.27× overall speedup for the inversion over $\mathbb{Z}_{2048}$. The rest of the improvement for inversion over $\mathbb{Z}_{2048}$ comes from our improved polynomial multiplications (we use `Toeplitz-TC` here). For the inversion in $\mathbb{Z}_3[x]\Big/\left\langle \frac{x^{677}-1}{x-1}\right\rangle$, we use bitsliced implementation and obtain a 8.6× speedup. For `ntruhrss701`, we outperform obtain 22.63×, 10.04×, 9.46× performance improvement for inversions over $\mathbb{Z}_2$, $\mathbb{Z}_{8192}$, and $\mathbb{Z}_3$, respectively. Table 3 summarizes the performance of inversions.

Table 3. Performance of inversions in NTRU.

Operation	Ref	Ours	Ref	Ours
	`ntruhps2048677`		`ntruhrss701`	
`poly_Rq_inv`	3 506 621	341 482	3 938 579	392 478
`poly_R2_inv`	2 791 906	136 776	3 175 330	140 290
`poly_S3_inv`	4 153 823	482 005	4 765 259	503 590
`crypto_sort_int32`	104 691	17 819	–	–

Sorting Network. We translate AVX2-optimized sorting network into Neon.

Performance of `ntruhps2048677` and `ntruhrss701`. Table 4 summarizes our `ntruhps2048677` and `ntruhrss701`. We compare our `Toeplitz-TC` to the existing NTRU implementations on Cortex-A72 [14]. For `ntruhps2048677`, our key generation is 7.67× faster. The main contribution is our optimized inversions, multiplications lifting the inverse in $\mathbb{Z}_2[x]/\langle x^{677}-1\rangle$, followed by polynomial multiplications in $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ (for lifting) and sorting network. Our `ntruhps2048677` encapsulation is 2.48× faster. The main contribution is the sorting network followed by polynomial multiplications. Our `ntruhps2048677` decapsulation is 1.77× faster. The improvement entirely comes from the improved polynomial multiplications. For `ntruhrss701`, our key generation, encapsulation, and decapsulation are 7.99×, 1.47×, and 1.56× faster than [14], respectively.

Finally, Table 5 details the numbers of `ntruhps2048677` and `ntruhrss701` with `Toeplitz-TC`. Notice that only performance-critical subroutines are shown.

Table 4. Overall cycles of `ntruhps2048677` and `ntruhrss701`. **K** stands for key generation, **E** stands for encapsulation, and **D** stands for decapsulation.

Operation	`ntruhps2048677`			`ntruhrss701`		
	K	E	D	K	E	D
Ref	8 245 039	227 980	331 274	9 397 305	134 737	365 558
[14]	7 686 272	196 526	212 265	8 599 610	87 380	221 986
`Toeplitz-TC`	1 002 187	79 213	120 208	1 076 810	59 625	142 174
`Toom-Cook`	1 127 089	88 037	146 422	–	–	–

Table 5. Detailed performance numbers of `ntruhps2048677` and `ntruhrss701` with `Toeplitz-TC`. Only performance-critical subroutines are shown.

`ntruhps2048677`		`ntruhrss701`	
Operation	Cycles	Operation	Cycles
`crypto_kem_keypair`	1 002 187	`crypto_kem_keypair`	1 076 810
`owcpa_keypair`	990 579	`owcpa_keypair`	1 069 128
`poly_S3_inv`	482 005	`poly_S3_inv`	503 590
`poly_Rq_mul`(×5)	5× 26 784	`poly_Rq_mul`(×5)	5× 31 478
`poly_Rq_inv`	341 482	`poly_Rq_inv`	392 478
`poly_R2_inv`	136 776	`poly_R2_inv`	140 290
`poly_Rq_mul`(×8)	8× 26 784	`poly_Rq_mul`(×8)	8× 31 478
`sort`	17 819		
`randombytes`	12 054	`randombytes`	6 294
`crypto_kem_enc`	79 213	`crypto_kem_enc`	59 625
`owcpa_enc`	32 501	`owcpa_enc`	41 559
`poly_Rq_mul`	26 784	`poly_Rq_mul`	31 478
`randombytes`	13 023	`randombytes`	6 202
`sort`	18 040		
`sha3`	5 148	`sha3`	5 296
`crypto_kem_dec`	120 208	`crypto_kem_dec`	142 174
`owcpa_dec`	100 842	`owcpa_dec`	120 485
`poly_Rq_mul`(×2)	2× 26 784	`poly_Rq_mul`(×2)	2× 31 478
`poly_S3_mul`	28 341	`poly_S3_mul`	33 319
`sha3`	18 867	`sha3`	21 342

A Proof for the Toeplitz Transformation

For an algebra homomorphism $f : R[x]_{<n} \to S$ with $f_k \coloneqq f|_{R[x]_{<k}}$ a monomorphism, and module homomorphism $(\boldsymbol{a}, -) = \begin{cases} R^k \to R^n \\ \boldsymbol{b} \mapsto \boldsymbol{ab} \end{cases}$ where $n \geq 2k-1$, we have

$$\left(\mathbf{Toeplitz}_{k\times k}(-)\right)(\boldsymbol{a}) = \mathrm{rev}_{k\times k} \circ f_k^* \circ (f_k(\boldsymbol{a}), -)^* \circ (f^{-1})^* \circ \mathrm{id}_{(2k-1)\to n}.$$

Proof. Observe $(\boldsymbol{a}, -)^* = f_k^* \circ (f_k(\boldsymbol{a}), -)^* \circ \left(f^{-1}\right)^* \circ \mathrm{id}_{(2k-1)\to n}$, it remains to show $\left(\mathbf{Toeplitz}_{k\times k}(-)\right)(\boldsymbol{a}) = \mathrm{rev}_{k\times k} \circ (\boldsymbol{a}, -)^*$. Let $\boldsymbol{z} = (z_0, \ldots, z_{2k-2})$, $[k] = \{0, \ldots, k-1\}$, and $\mathbf{0}_{m_0,m_1}$ the $m_0 \times m_1$ matrix of zeros. We have:

$$
\begin{aligned}
& \left(\mathrm{rev}_{k\times k} \circ \mathbf{Toeplitz}_{k\times k}(\boldsymbol{z})\right)(\boldsymbol{a}) \\
= \;& (z_{i+j})_{(i,j)\in[k]^2} \, (a_j)_{(j,0)\in[k]\times[1]} \\
= \;& \left(\sum_{j\in[k]} z_{i+j} a_j\right)_{(i,0)\in[k]\times[1]} \\
= \;& \sum_{j\in[k]} (z_{i+j} a_j)_{(i,0)\in[k]\times[1]} \\
= \;& \sum_{j\in[k]} (\mathbf{0}_{k,j} \quad a_j I_k \quad \mathbf{0}_{k,k-j-1}) \, (z_h)_{(h,0)\in[2k-1]\times[1]} \\
= \;& \mathbf{Toeplitz}_{k\times(2k-1)} (\mathbf{0}_{1,k-1}, a_0, \ldots, a_{k-1}, \mathbf{0}_{1,k-1}) \, (z_h)_{(h,0)\in[2k-1]\times[1]} \\
= \;& (\boldsymbol{a}, -)^*(\boldsymbol{z}).
\end{aligned}
$$

Applying $\mathrm{rev}_{k\times k}$ from the left finishes the proof (cf. [18, Theorem 6]).

B Examples of Toeplitz Transformations

We give some examples of f's implementing $\begin{pmatrix} z_1 & z_2 \\ z_0 & z_1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_0 \end{pmatrix}$:

$$
\begin{aligned}
& \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} z_1 & z_2 \\ z_0 & z_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \\
= \;& \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_0 & 0 & 0 \\ 0 & a_0 + a_1 & 0 \\ 0 & 0 & a_1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} \\
= \;& \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \end{pmatrix} \begin{pmatrix} a_0 + a_1 & 0 & 0 \\ 0 & a_0 + \omega_3 a_1 & 0 \\ 0 & 0 & a_0 + \omega_3^2 a_1 \end{pmatrix} \mathbf{F}_3^{-1} \begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} \\
= \;& \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \end{pmatrix} \begin{pmatrix} a_0 + a_1 & 0 & 0 & 0 \\ 0 & a_0 + \omega_4 a_1 & 0 & 0 \\ 0 & 0 & a_0 + \omega_4^2 a_1 & 0 \\ 0 & 0 & 0 & a_0 + \omega_4^3 a_1 \end{pmatrix} \mathbf{F}_4^{-1} \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ 0 \end{pmatrix}
\end{aligned}
$$

where $\mathbf{F}_k^{-1} = \left(\mathbf{F}_k^{-1}\right)^T$ is the inverse of the cyclic size-k FFT.

References

1. ARM: Cortex-A72 Software Optimization Guide (2015). https://developer.arm.com/documentation/uan0016/a/

2. ARM: Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile (2021). https://developer.arm.com/documentation/ddi0487/gb/?lang=en
3. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B.Y., Yang, S.Y.: Neon NTT: faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 221–244 (2022). https://tches.iacr.org/index.php/TCHES/article/view/9295
4. Bernstein, D.J.: Multidigit multiplication for mathematicians (2001). https://cr.yp.to/papers.html#m3
5. Bernstein, D.J., Yang, B.Y.: Fast constant-time GCD computation and modular inversion. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(3), 340–398 (2019). https://tches.iacr.org/index.php/TCHES/article/view/8298
6. Chen, C., et al.: NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project [15] (2020). https://ntru.org/
7. Cook, S.A., Aanderaa, S.O.: On the minimum computation time of functions. Trans. Am. Math. Soc. **142**, 291–314 (1969)
8. Hwang, V.B.: Case Studies on Implementing Number-Theoretic Transforms with Armv7-M, Armv7E-M, and Armv8-A. Master's thesis (2022). https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A
9. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In: Deng, R., Gauthier-Umana, V., Ochoa, M., Yung, M. (eds.) Applied Cryptography and Network Security. ACNS 2019. LNCS, vol. 11464, pp. 281–301. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21568-2_14
10. Kannwischer, M.J., Schwabe, P., Stebila, D., Wiggers, T.: PQClean. https://github.com/PQClean
11. Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk, vol. 145, no. 2, pp. 293–294 (1962)
12. Írem Keskinkurt Paksoy, Cenk, M.: TMVP-based Multiplication for Polynomial Quotient Rings and Application to Saber on ARM Cortex-M4. Cryptology ePrint Archive (2020). https://eprint.iacr.org/2020/1302
13. Írem Keskinkurt Paksoy, Cenk, M.: Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication (2022). https://eprint.iacr.org/2022/300
14. Nguyen, D.T., Gaj, K.: Optimized Software Implementations of CRYSTALS-Kyber, NTRU, and Saber Using NEON-Based Special Instructions of ARMv8 (2021). third PQC Standardization Conference
15. NIST, the US National Institute of Standards and Technology: Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography
16. Sanal, P., Karagoz, E., Seo, H., Azarderakhsh, R., Kermani, M.M.: Kyber on ARM64: compact implementations of Kyber on 64-bit ARM Cortex-A processors. Cryptology ePrint Archive, Report 2021/561 (2021). https://eprint.iacr.org/2021/561
17. Toom, A.L.: The complexity of a scheme of functional elements realizing the multiplication of integers. In: Soviet Mathematics Doklady, vol. 3, no. 4, pp. 714–716 (1963)
18. Winograd, S.: Arithmetic Complexity of Computations, vol. 33. Siam, New Delhi (1980)