Algorithmic Views of Vectorized Polynomial Multipliers for NTRU and NTRU Prime (Long Paper)

Han-Ting Chen¹, Yi-Hua Chung², Vincent Hwang²,³, Chi-Ting Liu¹,² and Bo-Yin Yang²

¹ National Taiwan University, Taipei, Taiwan
² Academia Sinica, Taipei, Taiwan
³ Max Planck Institute for Security and Privacy, Bochum, Germany

Abstract.
This paper explores the design space of vector-optimized polynomial multiplications in the lattice-based key-encapsulation mechanisms NTRU and NTRU Prime. Since NTRU and NTRU Prime do not support straightforward applications of number-theoretic transforms, the state-of-the-art vector code either resorted to Toom–Cook, or introduced various techniques for coefficient ring extensions. All these techniques lead to a large number of small-degree polynomial multiplications, which is the bottleneck in our experiments.

For NTRU Prime, we show how to reduce the number of small-degree polynomial multiplications to nearly 1/4 times compared to the previous vectorized code with the same functionality. Our transformations are based on careful choices of FFTs, including Good–Thomas, Rader’s, Schönhage’s, and Bruun’s FFTs. For NTRU, we show how to deploy Toom-5 with 3-bit losses.

Furthermore, we show that the Toeplitz matrix–vector product naturally translates into efficient implementations with vector-by-scalar multiplication instructions which do not appear in all prior vector-optimized implementations.

We choose the ARM Cortex-A72 CPU which implements the Armv8-A architecture for experiments, because of its wide uses in smartphones, and also the Neon vector instruction set implementing vector-by-scalar multiplications that do not appear in most other vector instruction sets like Intel’s AVX2.

Even for platforms without vector-by-scalar multiplications, we expect significant improvements compared to the state of the art, since our transformations reduce the number of multiplication instructions by a large margin.

Compared to the state-of-the-art optimized implementations, we achieve 2.18× and 6.7× faster polynomial multiplications for NTRU and NTRU Prime, respectively. For full schemes, we additionally vectorize the polynomial inversions, sorting network, and encoding/decoding subroutines in NTRU and NTRU Prime. For ntruhps2048677, we achieve 7.67×, 2.48×, and 1.77× faster key generation, encapsulation, and decapsulation, respectively. For ntrulp761, we achieve 3×, 2.87×, and 3.25× faster key generation, encapsulation, and decapsulation, respectively. For sntrup761, there are no previously optimized implementations and we significantly outperform the reference implementation.

Keywords: NTRU · NTRU Prime · Neon · FFT · Toeplitz matrix · Toom–Cook
# Algorithmic Views of Vectorized Polynomial Multipliers for NTRU and NTRU Prime

## Contents

1. **Introduction**
   - 1.1 Polynomials in NTRU and NTRU Prime  
   - 1.2 Review of Transformations  
   - 1.3 Prior Works, Motivations, and Contributions  
   - 1.4 Code  
   - 1.5 Structure of this Paper

2. **Preliminaries**
   - 2.1 ARM Cortex-A72  
   - 2.2 Modular Reductions and Multiplications in Armv8-A

3. **Polynomial Multiplications**
   - 3.1 The Chinese Remainder Theorem for Polynomial Rings  
   - 3.2 Cooley–Tukey FFTs  
   - 3.3 Bruun-Like FFTs  
   - 3.4 Good–Thomas FFTs  
   - 3.5 Rader’s FFT for Odd Prime $p$  
   - 3.6 Toom–Cook (TC) and Karatsuba  
   - 3.7 Schonhage’s and Nussbaumer’s FFTs  
   - 3.8 Enlarging Coefficient Rings

4. **Toeplitz Matrix–Vector Product**
   - 4.1 Module and Associative Algebra  
   - 4.2 Matrix–Vector Products  
   - 4.3 Toeplitz Matrices  
   - 4.4 Small Dimensional Cases  
   - 4.5 Large Dimensional Toeplitz Transformation

5. **Implementations**
   - 5.1 Good–Thomas for “Big by Small” Polynomial Multiplications  
   - 5.2 NTRU Implementations over $\mathbb{Z}_{65536}$  
   - 5.3 NTRU Prime Implementations over $\mathbb{Z}_{4591}$

6. **Results**
   - 6.1 Benchmark Environment  
   - 6.2 Performance of Vectorized Polynomial Multiplications  
   - 6.3 Performance of Schemes

7. **Discussions**

A. **Proof for the Toeplitz Transformation**

B. **Examples of Toeplitz Transformations**

C. **Matrix multiplications**

D. **Detailed Numbers of Polynomial Multiplications**

E. **Performance of Inversions, Encoding, and Decoding**

F. **Detailed Numbers of NTRU Prime**

G. **Detailed Numbers of NTRU**
1 Introduction

At PQCrypto 2016, the National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization Process for replacing existing standards for public-key cryptography with quantum-resistant cryptosystems. For lattice-based cryptosystems, polynomial multiplications had been the most time-consuming operations. Recently standardized \[\text{Dilithium, Kyber, and Falcon}\] wrote number-theoretic transforms (NTTs) into their specifications in response.

NTRU is still being used by Google \[\text{KMS22}\], and OpenSSH 9.0 still defaults to NTRU Prime. However, the polynomial rings of NTRU and NTRU Prime (and Saber) do not allow NTT-based multiplications directly. The state-of-the-art vectorized implementations resorted to Toom–Cook \[\text{Too63, CA69}\], introduced various techniques extending coefficient rings, or computed the results over \(\mathbb{Z}\) by bounding the maximum value. All these approaches lead to a large number of small-degree polynomial multiplications, which is the bottleneck in our experiments. We study the compatibility of vectorization and various algorithmic techniques in the literature. We choose the ARM Cortex-A72 implementing the Armv8-A architecture (which naturally comes with the SIMD technology Neon) for this work. Armv8-A is currently the most prevalent architecture for mobile devices and Apple hardware.

We are interested in the following questions regarding vectorized polynomial multiplications for NTRU and NTRU Prime on Armv8-A, exemplified by the Cortex-A72:

- **NTRU Prime**: Can we avoid Schönhage \[\text{Sch77}\] and Nussbaumer \[\text{Nus80}\] while maintaining the vectorization friendliness? \[\text{BBCT22}\] showed that vectorized “big by big” polynomial multiplication takes \(\sim 1.5 \times\) cycles of the “big by small” one with AVX2. \[\text{BBCT22}\] applied Schönhage and Nussbaumer, each doubling the sizes of the coefficient rings for the ease of vectorizations. This leads to a large number of small-degree polynomial multiplications. We explain how to avoid the doubling with a series of careful choices of fast Fourier transforms (FFTs) in Section 1.3.2.

- **NTRU**: What is the fastest implementation when moving from Cortex-M4 to Cortex-A72? On Cortex-M4, the fastest polynomial multiplications alternated between NTT-based and non-NTT-based approaches as implementations gradually improved. We explore the optimizations that can be directly adapted to vectorization in Section 1.3.4.

- **Multiplication instructions**: How would the algorithmic choices change with the presence of vector-by-scalar multiplications? Neon multiplication instructions usually come with vector-by-vector and vector-by-scalar encodings, as opposed to Intel AVX2 with only the former. We answer this question in Section 1.3.5. The use of vector-by-scalar multiplications eventually leads to the fastest polynomial multiplications in NTRU and NTRU Prime on Cortex-A72.

1.1 Polynomials in NTRU and NTRU Prime

1.1.1 NTRU Prime

The NTRU Prime submission comprises two families Streamlined NTRU Prime and NTRU LPRime. Both operate on the polynomial ring \(\mathbb{Z}_q[x]/\langle x^q - x - 1 \rangle\) where \(q\) and \(p\) are primes such that the ring is a finite field. We target the polynomial multiplications for parameter sets \texttt{sntrup761} and \texttt{ntrulpr761} where \(q = 4591\) and \(p = 761\). One should note that \texttt{sntrup761}, which is used by OpenSSH, uses a (Quotient) NTRU structure, and requires inversions in \(\mathbb{Z}_q[x]/\langle x^{761} - x - 1 \rangle\) and \(\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1 \rangle\). We refer the readers to the specification \[\text{BBC+20}\] for more details. With no other assumptions on the inputs, we call a polynomial multiplication “big by big”. If one of the inputs is guaranteed to be ternary, we call it a “big by small” polynomial multiplication. Note that big-by-big
multiplications in NTRU Prime is required only if we apply the fast constant-time GCD for the key generation [BY19]. Although we left this optimization as a future work, we still optimize both big-by-big and big-by-small polynomial multiplications.

### 1.1.2 NTRU

The NTRU submission comprises two families NTRU-HPS and NTRU-HRSS. Both operate on polynomial rings $\mathbb{Z}_q[x]/\langle \Phi_n \rangle$, $\mathbb{Z}_p[x]/\langle \Phi_n \rangle$, and $\mathbb{Z}_p[x]/\langle x^n - 1 \rangle$ where $q$ is a power of 2, $n$ is a prime, and $\Phi_n$ is the $n$th cyclotomic polynomial, which for prime $n$ is $\frac{x^n - 1}{x - 1} = \sum_{i<n} x^i$. We target the parameter set $ntruhps2048677$ where $q = 2048$ and $n = 677$. For more parameter sets and details, we refer to the specification [CDH+20]. While NTRU also requires inversions in $\mathbb{Z}_q[x]/\langle \Phi_n \rangle$ and $\mathbb{Z}_p[x]/\langle x^n - 1 \rangle$, we focus on the polynomial multiplications in $\mathbb{Z}_{2048}[x]/\langle x^{677} - 1 \rangle$.

### 1.2 Review of Transformations

We assume all rings have identity 1 and are commutative. The polynomial ring $R[x]$ can be seen as infinite sequences in $R$ with finitely many non-zero elements; the degree of such a sequence (a polynomial) is the index (0-based) of the last non-zero element. We define the set of polynomials with a degree less than $n$ as $R[x]_{<n}$. A polynomial $g \in R[x]$ defines an ideal $\langle g \rangle := gR[x]$ and a quotient ring $R[x]/\langle g \rangle$. We often multiply in $R[x]/\langle g \rangle$ for a monic $g$ of degree $n$ with the convention that $\deg(g) = -1 \implies g = 0$. We regard $R[x]/\langle g \rangle$ as an (associative) $R$-algebra [Jac12, Chapter 7] over $R$. This allows us to characterize $R[x]/\langle g \rangle$ as a ring and a module. We use $R[x]/\langle g \rangle$ while focusing on the ring or $R$-algebra view, and $R^n$ for the module view. For arbitrary $R$-algebra monomorphism with $R[x]/\langle g \rangle$ as the domain, one can derive optimizations via a series of ring or module monomorphisms.

If $g = x^n - \zeta$, $R[x]/\langle g \rangle$ is a weighted convolution [CF94]; if further $\zeta = -1$, it is a negacyclic convolution; or if $\zeta = 1$, it is a cyclic convolution. For $a = \sum_i a_i x^i, b = \sum_i b_i x^i \in R[x]/\langle x^n - \zeta \rangle$, the weighted convolution $c = ab \in R[x]/\langle x^n - \zeta \rangle$ is the size-$n$ polynomial $\sum_i c_i x^i$ with $c_i = \sum_{j=0}^{i} a_j b_{i-j} + \zeta^{n-i} \sum_{j=1}^{n-i} a_j b_{n-i-j}$.

For a coprime factorization $g = \prod_{i_0} \cdots i_{k-1} g_{i_0 \cdots i_{k-1}}$, the Chinese remainder theorem (CRT) allows us to multiply polynomials in $R[x]/\langle g \rangle$ with the series of isomorphisms

$$R[x]/\langle \prod_{i_0 \cdots i_{k-1}} g_{i_0 \cdots i_{k-1}} \rangle \cong \prod_{i_0} R[x]/\langle \prod_{i_1 \cdots i_{k-1}} g_{i_0 \cdots i_{k-1}} \rangle \cong \cdots \cong \prod_{i_0 \cdots i_{k-1}} R[x]/\langle g_{i_0 \cdots i_{k-1}} \rangle.$$ 

Cooley–Tukey FFT [CT65] chooses $g_{i_0 \cdots i_{k-1}}$'s as binomials and Bruun’s FFT [Bru78, Mur96] chooses $g_{i_0 \cdots i_{k-1}}$'s as trinomials of the form $x^{2m} + \alpha x^m + 1$. The finite field version of Bruun’s FFT enables the choices $g_{i_0 \cdots i_{k-1}} = x^{2m} + \alpha x^m + \beta$ [BC87]. For $g = x^p - 1$ with prime $p$, Rader’s FFT [Rad68] computes the map $R[x]/\langle x^p - 1 \rangle \cong \prod_p R[x]/\langle x - \omega_p \rangle$ via a polynomial multiplication modulo $x^{p-1} - 1$ where $\omega_p$ is a principal $p$-th root of unity. A $k$-way Toom–Cook computes via the monomorphism $R[x]/\langle \prod_{i=0}^{2k-2} (x - s_i) \rangle \cong \prod_{i=0}^{2k-2} R[x]/\langle x - s_i \rangle$ for some suitably chosen $s_i$’s. The case $k = 2$ with $(s_0, s_1, s_2) = (0, 1, \infty)$ is called Karatsuba [KO62]. We usually use Toom-$k$ when the $s_i$’s are clear. Cooley–Tukey, Rader, and Bruun require $R$ to endow some special structures.

There are ideas that do not ask for special structures or require simple structures for $R$. If $g = x^n - 1$ for an $n$ with the coprime factorization $n = \prod q_i a_i$. Good–Thomas FFT [Goo58] converts $R[x]/\langle x^n - 1 \rangle$ into $\bigotimes_{d} R[x_d]/\langle x_d^{a_d} - 1 \rangle$. We then choose our favorite transformations for each of $R[x_d]/\langle x_d^{a_d} - 1 \rangle$ and tensor them. For a factorization $n = n_0 a_1$, Schönhage’s and Nussbaumer’s FFTs first start with writing $g_0 |_{y=xa_1} = g(x)$. Schönhage requires $g_0(y^n - 1)$ and an injection $R[x]/\langle x^{a_1} - y \rangle \mapsto R[x]/\langle y \rangle$.
with \( h|\Phi_n(x) \). Nussbaumer requires \( g_0|\Phi_n(x) \) and an injection \( R[x,y]/(x^{n_1}-y,g_0) \rightarrow R[x,y]/(h,g_0) \) with \( h(x^{2n_1}-1) \). The injections usually double the sizes of polynomial rings. For finite index sets \( J \subset I \), truncation [CF94, vdH04] computes the product by restricting the domain and image of a monomorphism. For an \( \eta : R[x]/(\prod_{i \in I} g_i) \cong \prod_{i \in J} R[x]/(g_i) \), we define \( \eta_J \) as \( R[x]/(\prod_{j \in J} g_j) \cong \prod_{i \in J} R[x]/(g_i) \). \( \eta_J \) now computes a product with size less than or equal to \( \text{deg}(\prod_{j \in J} g_j) \) while requiring \( R \) the special structure defining \( \eta_I \).

Finally, to invert a monomorphism, it suffices to identify the homomorphic image. We then perform additions, subtractions, multiplications, and divisions of elements in \( R \). If an element is not invertible, we can still perform a “division” by extracting the quotient. By the correctness of an algorithm, if we omit the division by \( a \in R \), then the result is the \( a \)-multiple of the desired one. If replace the coefficient ring \( R \) by \( aR \) for the image, the desired result can be obtained by the actual division by \( a \).

### 1.3 Prior Works, Motivations, and Contributions

We survey prior works and explain our motivations.

#### 1.3.1 NTRU Prime: Prior Works

Table 1: Prior works of ntrulpr761/sntrup761 on Cortex-M4. GT = Good–Thomas, MR = mixed–radix, I = implicit, VF = vectorization–friendly, and VUNF = vectorization–unfriendly.

<table>
<thead>
<tr>
<th>pqm4</th>
<th><a href="MR*">ACC+21</a></th>
<th><a href="GT">ACC+21</a></th>
<th>[AHY22]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toom-4</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GT (I, VUNF)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>GT (I, VF)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Rader</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Big by small</th>
<th>Big by big</th>
<th>Coefficient ring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>( \mathbb{Z}_{4591} )</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>-</td>
<td>( \mathbb{Z}_{4591} )</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>-</td>
<td>( \mathbb{Z} )</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>-</td>
<td>( \mathbb{Z} )</td>
</tr>
</tbody>
</table>

| Performance | 223 871 | 152 177 | 159 176 | 151 374 |

* Fastest mixed-radix approach in [ACC+21].

[BBB+20] was already using Good–Thomas FFT for computing big-by-small polynomial multiplications in NTRU Prime on Intel Haswell processor with AVX2. [BBB+20] computed the product as if one is working over the coefficient ring \( \mathbb{Z} \) by choosing sufficiently many moduli bounding the maximum value of the results in \( \mathbb{Z} \).

[ACC+21] implemented the same idea on Cortex-M4 and introduced alternative strategies without switching from \( \mathbb{Z}_q \) (\( q = 4591 \) for ntrulpr761/sntrup761) to \( \mathbb{Z} \). They introduced two such strategies: (i) FFT for \( \mathbb{Z}_q[x]/(x^{1620}-1) \) with one radix-2, three radix-3, and one radix-5 butterflies; and (ii) FFT for \( \mathbb{Z}_q[x]/(x^{1530}-1) \) with one radix-17 and two radix-3 butterflies. The former leads to size-6 polynomial multiplications and the later leads to size-10 polynomial multiplications. [ACC+21] explained how to use
Rader’s FFT for converting the radix-17 NTT into a size-16 cyclic convolution. [Haa21] implemented [ACC+21]’s vectorization-unfriendly Good–Thomas on Cortex-A72.

[BBCT22] extended [BBC+20]’s AVX2 work to big-by-big polynomial multiplications and introduced how to craft roots of unity with (truncated) Schön­hage and Nussbaumer. They were also aware of [ACC+21]’s strategies over \( \mathbb{Z}_{4591} \), but crafted radix-2 roots of unity for the ease of vectorization with AVX2 instead [BBCT22, Section 3.3]. For every applications of Schön­hage and Nussbaumer, the number of coefficients is doubled. This implies a large number of small-degree polynomial multiplications after FFTs.

[AHY22] proposed several improvements on Cortex-M4. They extended the existence of subtraction from radix-2 to arbitrary radices [AHY22, Section 3.1], and explained how vectorization-friendly transformations result in flexible code-size optimizations [AHY22, Section 3.3]. They proposed vectorization-friendly Good–Thomas for permuting on-the-fly with compact code size.

The most extensively studied parameter sets are ntrulpr761/sntrup761. Table 1 summarizes prior works on Cortex-M4, Table 2 summarizes prior vector-optimized works.

Table 2: Prior works of ntrulpr761/sntrup761 with vector instructions. SN = Schön­hage–Nussbaumer, and E = explicit.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Haswell</th>
<th>Cortex-A72</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>[BBC+20]</td>
<td><a href="GT">BBCT22</a></td>
</tr>
<tr>
<td>Idea</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GT (E, VUNF)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Schön­hage</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Nuss­­baumer</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Truncation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Applicability</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Big by small</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Big by big</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Coefficient ring</td>
<td>( \mathbb{Z} )</td>
<td>( \mathbb{Z} )</td>
</tr>
<tr>
<td>Performance</td>
<td>18080</td>
<td>16992</td>
</tr>
</tbody>
</table>

1.3.2 NTRU Prime: Motivations and Contributions

We propose four implementation strategies in this paper: (i) Good–Rader–outer, (ii) Good–Rader–Bruun, (iii) Good–Thomas, and (iv) Good–Schön­hage–Bruun. Each of our strategies aims at addressing some particular implementation challenges. We go through the design rationale of Good–Rader–Bruun and Good–Schön­hage–Bruun, and will return to Good–Rader–outer and Good–Thomas shortly.

Problem 1: truncated Schön­hage vs vectorization-friendly Good–Thomas and Schön­hage. Our first problem is to analyze the compatibility of vectorization-friendly Good–Thomas and Schön­hage. [BBCT22] applied Schön­hage to \( \mathbb{Z}_q[x]/\langle x^{512} - 1 \rangle \langle x^{1024} + 1 \rangle \). We instead compute over \( \mathbb{Z}_q[x]/\langle x^{1536} - 1 \rangle \) by first pulling out the factor 3 via vectorization-friendly Good–Thomas. For the power-of-two cyclic NTT, we apply Schön­hage. Compared to truncated Schön­hage, our approach results in twice as many subproblems of half size. This answers questions 6 and 7 in [Hwa22, Section 10.1].
Problem 2: Nussbaumer vs Bruun. Our second problem is to reduce the number of polynomial multiplications. After applying Schönhage, we now have to multiply polynomials in the ring \( \mathbb{Z}[x]/(x^k + 1) \) where \( k = 6 \) in [BBCT22] and \( k = 5 \) in this work. [BBCT22] applied Nussbaumer and doubled the number of coefficients. We avoid this doubling with Bruun’s FFT. Our approach results in only half of size-8 polynomial multiplications with slightly expensive computations.

Problem 3: Schönhage vs Rader-17. Our third problem aims at removing Schönhage. We first observe that the Schönhage in [BBCT22] reduced a size-1536 problem to several size-64 problems. We are looking for a multiple of 17 close to \( 1536/64 = 48 \). We choose 51 since one can define a size-51 cyclic NTT nicely over \( \mathbb{Z}_q \). We optimize further by extending the size-51 cyclic NTT to size-102.

Problem 4: generalize Bruun over \( x^{2k} + c \) for \( c \neq \pm 1 \). The last problem is to apply Bruun’s FFT after the size-102 cyclic NTT. After the cyclic NTT, the goal is to multiply polynomials over \( \mathbb{Z}_q[x]/(x^{16} \pm \omega_1^{51}) \). The composed multiplication over a finite field shows that the factorization follows the same pattern of factorizing \( \mathbb{Z}_q[x]/(x^{16} \pm 1) \) since \( 51 \perp 16 \).

We propose Good–Schönhage–Bruun based on Problems 1 and 2, and adapt the resulting computation into Good–Rader–Bruun based on Problems 3 and 4.

Table 3: NTRU Prime strategies of this work (Cortex-A72). GRo = Good–Rader–outer, GRB = Good–Rader–Bruun, GT = Good–Thomas, GSB = Good–Schönhage–Bruun.

<table>
<thead>
<tr>
<th>NTRU Prime strategies</th>
<th>GRo</th>
<th>GRB</th>
<th>GT</th>
<th>GSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idea</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Good–Thomas (explicit, VF, Section 3.4)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Schönhage (Section 3.7)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Rader (Section 3.5)</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bruun (Section 3.3)</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Vector-by-scalar (basemul, Section 4.2)</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Vector-by-vector (basemul, Section 4.2)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Big by small</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Big by big</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Coefficient ring</td>
<td>( \mathbb{Z}_{4591} )</td>
<td>( \mathbb{Z}_{4591} )</td>
<td>( \mathbb{Z} )</td>
<td>( \mathbb{Z}_{4591} )</td>
</tr>
</tbody>
</table>

1.3.3 NTRU and Saber: Prior Works

[KRS19] applied Toom-4 to NTRU and Saber on Cortex-M4. They compared multiple combinations of Toom-4, Toom-3, and Karatsuba. [CDH+20] later implemented Toom-4 for NTRU on Haswell, and [MKV20] implemented Toom-4 with lazy interpolation for Saber on Cortex-M4 and Haswell. [IKPC20] introduced the Toeplitz matrix-vector product with Toom–Cook as the underlying monomorphism for Saber on Cortex-M4. Their implementation remains to be the fastest non-NTT-based Saber on Cortex-M4. [CHK+21] introduced the uses of NTTs for NTRU and Saber on Cortex-M4 and Skylake. They improved NTRU polynomial multiplication by 10% – 19% and Saber matrix-to-vector multiplication by > 60%. Their ideas for NTRU were adapted from the works [ACC+21] and [BBC+20].
[NG21] implemented Toom-4 and NTT for Saber, and Toom-4 for NTRU on Cortex-A72. They skipped the NTT-based polynomial multiplications for NTRU without any explanations. We believe the reason is that their Toom–Cook for Saber performed roughly the same as their own NTT. Since the improvement for Saber is much more pronounced than NTRU in [CHK+21], hypothetically, NTT-based NTRU polynomial multiplication won’t outperform Toom–Cook on Cortex-A72. We also believe that [NG21] was not aware of the Toeplitz approach for Saber [IKPC20] even though it was already publicly available prior to [CHK+21]. Shortly after, [BHK+22] proposed several improvements for the NTTs on Cortex-A72. Their NTT-based Saber matrix-to-vector multiplication was about 2× faster than [NG21]’s.

Perhaps more surprisingly, [IKPC22] extended the Toeplitz approach for Saber to NTRU by embedding cyclic convolutions into Toeplitz matrix-vector products of slightly larger sizes. [IKPC22] outperformed [CHK+21] on all NTRU parameter sets. Finally, [AHY22] outperformed [IKPC22] by proposing several NTT optimizations.

Table 4 summarizes prior works on Cortex-M4, and Table 5 summarizes prior vectorized implementations.

Table 4: Prior works of ntruhps2048677 on Cortex-M4.

<table>
<thead>
<tr>
<th>Idea</th>
<th>[KRS19]</th>
<th>[CHK+21]</th>
<th>[IKPC22]</th>
<th>[AHY22]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toom-4</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Toeplitz</td>
<td>-</td>
<td>-</td>
<td>✓ (TC-4)</td>
<td>-</td>
</tr>
<tr>
<td>GT (implicity, VUNF)</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>GT (implicity, VF)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Big by small</th>
<th>Big by big</th>
<th>Coefficient ring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z₂¹₆</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z₂¹₆</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z</td>
</tr>
</tbody>
</table>

| Performance** | 175k | 156k | 144k | 140k |

** We only find numbers with k-cycle as unit for some implementations.

Table 5: Prior works of ntruhps2048677 with vector instructions.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Skylake</th>
<th>Cortex-A72</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>[CDH+20]***</td>
<td>[CHK+21]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idea</th>
<th>[CDH+20]***</th>
<th>[CHK+21]</th>
<th>[NG21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toom-4</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Cooley–Tukey</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Big by small</th>
<th>Big by big</th>
<th>Coefficient ring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z₂¹₆</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z₂¹₆</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Z</td>
</tr>
</tbody>
</table>

| Performance** | 11 103 | 10 373 | 58 286 |

*** From [CHK+21, Table 7].
1.3.4 NTRU: Motivations and Contributions

We propose three implementation strategies (i) Toeplitz–TC, (ii) Toom–Cook, and (iii) Good–Thomas. We go through the motivations of Good–Thomas and Toom–Cook. The Good–Thomas for NTRU is almost the same as the one for NTRU Prime since we compute the results in $\mathbb{Z}[x]$, and reduce them to the polynomial rings in NTRU and NTRU Prime.

Table 6: Performance of [BHK+22]’s Saber. $\dim = 3$ for lightsaber and $\dim = 2$ for saber. Numbers are stripped from Tables 4 and 5 of [BHK+22].

<table>
<thead>
<tr>
<th></th>
<th>NTT</th>
<th>dim × base_mul</th>
<th>InnerProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightsaber</td>
<td>1529</td>
<td>Unknown</td>
<td>7638</td>
</tr>
<tr>
<td>saber</td>
<td>1529</td>
<td>2689</td>
<td>9284</td>
</tr>
</tbody>
</table>

Problem 5: NTT vs Toom–Cook. Our first problem for NTRU implementation is to analyze the NTT on Cortex-A72 after [BHK+22]’s improvement. We roughly estimate the performance of the vectorization–friendly Good–Thomas for $\mathbb{Z}_q[x]/(x^{1536} − 1)$ where $q$ is a 32-bit NTT-friendly modulus. We choose $(\bar{q}, v) = (1, 4)$ where $v$ refers to the number of coefficients in each vector (we follow the suggestions in [AHY22, Section 3.3]) and compute size-384 cyclic NTT defined on 128-bit chunks (each holds a size-4 polynomial). The estimation comes from two parts: size-384 cyclic NTT and size-4 base multiplications. We derive the numbers based on [BHK+22]’s Saber on Cortex-A72. Table 6 extracts the numbers we need. Since a 6-layer 32-bit NTT for $\mathbb{Z}_q[x]/(x^{2048} + 1)$ (for Saber) takes 1529 cycles, we estimate a size-384 NTT for the size-1536 cyclic polynomial ring to be no worse than $1529 \cdot \frac{1536 \cdot 9}{256} - 7038 - 1529 = 48813$ cycles. In summary, we believe that the size-384 NTT approach for $\mathbb{Z}_q[x]/(x^{1536} − 1)$ will take at most $3 \cdot 137612 + 6 \cdot 1255 = 48813$ cycles – outperforming [NG21] (58286 cycles in our benchmark). We expect the cycles to be smaller than 48813 since cyclic NTT is faster than the negacyclic one and we overestimate the size-384 NTT.

Problem 6: register utilization in Toom–Cook. In Armv8-A Neon, there are 32 SIMD registers. [BHK+22] showed that this enables one to compute four layers of radix-2 butterflies for the NTTs without spilling registers (this is obvious). An immediate problem is the register utilization for Toom–Cook. [KRS19, CDH+20, MKV20] applied Karatsuba, Toom–3, and Toom–4. We believe the reason is that there are only 14 general purpose registers on Cortex-M4 and 16 ymm registers in AVX2. Although [NG21] also implemented Toom–Cook on with Armv8-A Neon, they didn’t explore the availability of high-order Toom–Cook. In ntruhps2048677, since the coefficient ring is $\mathbb{Z}_{211}$, we can only afford 5-bit losses for Toom–Cook. Since evaluating Toom–4 at $\{0, \pm 1, \pm 2, 4, \infty\}$ incurs 3-bit losses, we want to find a point set for Toom–5 with 3-bit losses if we wish to replace Toom–4 with Toom–5. A simple analysis shows that evaluating Toom–5 at $\{0, \pm 1, \pm 2, \pm 3, 4, \infty\}$ results in 4-bit losses. We choose the point set $\{0, \pm 1, \pm 2, \pm 3, 4, \infty\}$ which doesn’t seem to be used for Toom–5 in the literature.

1We only need 7 layers of radix-2 and one layer of radix-3 size-384, but overestimate the performance as a 9-layer radix-2 computation.

2In [BHK+22], the inner product in the encryption of Saber consists of $l$ NTTs, one $\dim \times \text{base mul}$, and one inverse NTT. $\dim \times \text{base mul}$ computes the double-width products of $l$ size-4 base multiplications over 256 coefficients, accumulates the products, and performs the modular reductions. We want the performance for the case $l = 1$ which is not shown in their work. $9284 - 7038 - 1529 = 717$ gives a reasonable estimation for the double-width product of one size-4 base multiplication over 256 coefficients without modular reductions. Subtracting $2 \cdot 717$ from 2689 (performance of $\dim \times \text{base mul}$ for saber) gives a reasonable estimation for the size-4 base multiplication with modular reductions.
We propose Good–Thomas based on Problem 5 and the vectorization–friendly Good–Thomas proposed by [AHY22, Section 3.3]. Our Toom–Cook is based on Problem 6.

Table 7: NTRU strategies of this work (Cortex-A72).

<table>
<thead>
<tr>
<th>NTRU strategies</th>
<th>Toeplitz-TC</th>
<th>Toom-Cook</th>
<th>Good-Thomas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idea</td>
<td>✓ (TC-5)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Toeplitz (Section 4)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toom-5 (Section 3.6)</td>
<td>-</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Good–Thomas (Section 3.4)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Vector-by-scalar (basemul, Section 4.2)</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Vector-by-vector (basemul, Section 4.2)</td>
<td>-</td>
<td>✓</td>
<td>✓****</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Big by small</th>
<th>Big by big</th>
<th>Coefficient ring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>$\mathbb{Z}_{2^{16}}$</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>$\mathbb{Z}_{2^{16}}$</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>-</td>
<td>$\mathbb{Z}$</td>
</tr>
</tbody>
</table>

**** We also implemented vector-by-scalar for the base multiplication. There is no observable performance improvement since permutation instructions are not the bottleneck in this case.

1.3.5 Matrix–Vector Multiplications and Vector-by-Scalar Multiplications

Finally, we go through the impact of vector-by-scalar multiplication instructions. The use of vector-by-scalar multiplications leads to the fastest implementations Good–Rader–outer for NTRU Prime and Toeplitz–TC for NTRU in this work.

Outer-product-based matrix-vector multiplications and Toeplitz matrices. In the context of matrix–matrix multiplication in cubic time, there are six ways to iterate the three indices $i,j,k$ for computing $C[i_0][i_1] = C[i_0][i_1] + A[i_0][i_2]B[i_2][i_1]$. Iterating with $i_0 \rightarrow i_1 \rightarrow i_2$ is called inner-product approach and $i_2 \rightarrow i_0 \rightarrow i_1$ is called outer-product approach. When $B$ is a vector, there are two ways for iterating: $i_0 \rightarrow i_2$ and $i_2 \rightarrow i_0$. Since $i_0 \rightarrow i_2$ is a special case of $i_0 \rightarrow i_1 \rightarrow i_2$ and $i_2 \rightarrow i_0$ is a special case of $i_2 \rightarrow i_0 \rightarrow i_1$, we call $i_0 \rightarrow i_2$ inner-product-based and $i_2 \rightarrow i_0$ outer-product-based approaches. For vectorization, $i_0 \rightarrow i_2$ means interleaving, applying vector-by-vector multiplications, and deinterleaving. We show that $i_2 \rightarrow i_0$ performs better when there are vector-by-scalar multiplication instructions. There are no permutations if one already has the matrix in registers and the register pressure is significantly reduced. In this work, although we don’t have the full matrix in registers initially, we exploit the structure of Toeplitz matrices to construct and place the matrix in registers. Since the register pressure is significantly reduced, size-16 Toeplitz matrix–vector multiplications (which cover the weighted convolutions) are now feasible without register spills.

Toeplitz conversions from arbitrary monomorphisms. Our next problem is about deriving fast computations for Toeplitz matrix–vector products when the dimension is large. Let $n \geq 2k - 1$. We show that arbitrary algebra monomorphism $f: R[x]_{<n} \rightarrow S$ gives raise to a computation for a Toeplitz matrix–vector product with the same decreases of subproblem sizes. If $f$ is a composition of cheap algebra monomorphisms, then there is a corresponding composition of Toeplitz matrix–vector products with the same series of subproblem sizes. The resulting computation is then a fast computation for Toeplitz matrix–vector product.
The plan for choosing transformations. In summary, we first pick our favorite algebra monomorphism \( f : R[x]_{<n} \to S \). If \( f \) already results in small Toeplitz matrix–vector products, then we apply vector-by-scalar multiplication instructions. Otherwise, we turn \( f \) into something decomposing a large Toeplitz matrix–vector product into several small ones.

Applications in this work. For \texttt{ntrulpr761/sntrup761}, we pick \( f \) as an FFT implementing \( \mathbb{Z}_{4591}[x]/(x^{1632} - 1) \cong \prod_i \mathbb{Z}_{4591}[x]/(x^{10} \pm \omega_{51}^i) \). Since we have weighted convolutions after applying \( f \), we compute with vector-by-scalar multiplications. This is our \textit{Good–Rader–outer}. For \texttt{ntruhps2048677}, we regard the polynomial multiplication in \( \mathbb{Z}_{2048}[x]/(x^{677} - 1) \) as a Toeplitz matrix–vector product of dimension 720. Then, we choose \( f \) as the composition of one Toom–5, two Toom–3’s, and one Karatsuba, and turn \( f \) into a fast computation for Toeplitz matrix–vector product. Since \( \frac{720}{5 \cdot 3 \cdot 2} = 8 \), we eventually have several Toeplitz matrix–vector products of dimension 8. We compute them with vector-by-scalar multiplications. This is our \textit{Toeplitz–TC}.

1.4 Code.

Our source code can be found at https://github.com/vector-polymul-ntru-ntrup/vector-polymul-ntru-ntrup.

1.5 Structure of this Paper.

This paper is structured as follows: Section 2 describes our target operations and platforms. Section 3 surveys polynomial transformations used for multiplications. Section 5 describes our implementations. We show the performance numbers in Section 6. Section 7 discusses possible strategies based on this work for other parameter sets.

2 Preliminaries

Section 2.1 describes our target platform Cortex-A72, and Section 2.2 describes modular reductions and multiplications.

2.1 ARM 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. Instructions are first decoded into \( \mu \text{ops} \) in the frontend and dispatched to the backend, which contains these eight pipelines: \( L \) for loads, \( S \) for stores, \( B \) for branches, \( I0/I1 \) for integer instructions, \( M \) for multi-cycle integer instructions, and \( F0/F1 \) for Single-Instruction-Multiple-Data (SIMD) instructions. The frontend can only dispatch at most three \( \mu \text{ops} \) per cycle. Furthermore, in a single cycle, the frontend dispatches at most one \( \mu \text{op} \) using \( B \), at most two \( \mu \text{ops} \) using \( I0/I1 \), at most two \( \mu \text{ops} \) using \( M \), at most one \( \mu \text{op} \) using \( F0 \), at most one \( \mu \text{op} \) using \( F1 \), and at most two \( \mu \text{ops} \) using \( L/S \) [ARM15, Section 4.1].

We mainly focus on the pipelines \( F0, F1, L, \) and \( S \) for performance. \( F0/F1 \) are both capable of various additions, subtractions, permutations, comparisons, minimums/maximums, and table lookups \(^3\). However, multiplications can only be dispatched to \( F0 \), and shifts to \( F1 \). The most heavily-loaded pipeline is clearly the critical path. If there are more multiplications than shifts, we much prefer instructions that can use either pipeline to go

\(^3\)There are some exceptions, including \texttt{addv}, \texttt{smaxv}, \texttt{sadalp}. We are not using them in this paper and refer to [ARM15] for more details.
to \( F_1 \) since the time spent in \( F_0 \) will dominate our runtime. Conversely, with more shifts than multiplications, we want to dispatch most non-shifts to \( F_0 \). In practice, we interleave instructions dispatched to the pipeline with the most workload with other pipelines (or even \( L/S \)) — and pray. Our experiment shows that this approach generally works well. In the case of \texttt{chacha20} implementing \texttt{randombytes} for benchmarking [BHK + 22], we even consider a compiler-aided mixing of \( IO/F_1, F_0/F_1, \) and \( L/S \). The idea also proved valuable for \texttt{Keccak} on some other Cortex-A cores [BK22, Table 1].

**SIMD registers.** The 64-bit Armv8-A has 32 architectural 128-bit SIMD registers each viewable as packed 8-, 16-, 32-, or 64-bit elements, denoted by the suffixes .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. We refer readers to [ARM21, Figure A1-1].

**Armv8-A vector instructions.** A plain \texttt{mul} multiplies corresponding vector elements and returns same-sized results. However, \texttt{mul} also refers to another instruction encoding — vector-by-scalar multiplication — if the last operand is a lane of a register. In this case \texttt{mul} multiplies the vector by a scalar (the lane value). This simple feature helps a lot by maximizing register utilization and minimizing permutations.

There are many variants of multiplications: \texttt{mla/mls} computes the same product vector and accumulates to or subtracts from the destination. There are high-half products \texttt{sqdmulh} and \texttt{sqrddmulh}. The former computes the double-size products, doubles the results, and returns the upper halves. The latter first rounds to the upper halves before returning them. There are long multiplications \texttt{s{mul,mla,mls}l{,2}}. \texttt{smull} multiplies the corresponding signed elements from the lower 64-bit of the source registers and places the resulting double-width vector elements in the destination register. It is usually paired with an \texttt{smull2} using the upper 64-bit instead. Their accumulating and subtracting variants are \texttt{s{mla,mls}l{,2}}. We will not use the unsigned counterparts \texttt{u{mul,mla,mls}l{,2}}.

Next, the shifts: \texttt{shl} shifts left; \texttt{sshr} arithmetically shifts right; \texttt{srshr} rounds the results after shifting. We won’t use the unsigned \texttt{ushr} and \texttt{urshr}.

For basic arithmetic, the usual \texttt{add/sub} adds/subtracts the corresponding elements. Long variants \texttt{s{add,sub}l{,2}} add or subtract the corresponding elements from the lower or upper 64-bit halves and signed-extend into double-width results.

Then we have permutations — \texttt{uzp{1,2}} extracts the even and odd positions respectively from a pair of vectors and concatenates the results into a vector. \texttt{ext} extracts the lowest elements (there is an immediate operand specifying the number of bytes) of the second source vector (as the high part) and concatenates to the highest elements of the first source vector. \texttt{zip{1,2}} takes the bottom and top halves of a pair of vectors and riffle-shuffles them into the destination.

Usually instructions extending or narrowing the data width are used in pairs. For example, we frequently apply \texttt{smull} and \texttt{smull2} to the same pair of vectors for the double-width products. We denote \((lo,hi) = (\texttt{smull,smull2})(a,b)\) where \( lo \) is the destination register of \texttt{smull} and \( hi \) is the destination register of \texttt{smull2}. For the accumulating/subtracting variants \texttt{s{mla,mls}l{,2}}, we denote \((lo,hi) = (lo,hi)(\texttt{smlal,smlal2})(a,b)\) where the pairs of destinations and the accumulators must be the same.

---

4We write some assembly and only obtain comparable performance. So we keep the implementations with intrinsics instead for readability.  
5There are several options for signed-extending vector elements — \texttt{saddl{,2}} and \texttt{subl{,2}} which go to either \( F_0/F_1, \texttt{sxtl{,2}} \) to \( F_1 \), and \texttt{smull{,2}} going to \( F_0 \).
2.2 Modular Reductions and Multiplications in Armv8-A.

**Algorithm 1** Barrett reduction.
This is [BHK⁻²², Algorithm 11].

<table>
<thead>
<tr>
<th>Input:</th>
<th>a = a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>a = a − ( \lfloor \frac{a}{2^k} \rfloor \frac{q}{2^k} ) ( q \equiv a \mod \pm q ).</td>
</tr>
</tbody>
</table>

1. sqrdmulh t, a, \( \lfloor \frac{q}{2^k} \rfloor \)
2. sqrdmulh t, t, \#(e+1)
3. mla a, t, q

**Algorithm 2** Barrett multiplication.
This is [BHK⁻²², Algorithm 10].

<table>
<thead>
<tr>
<th>Input:</th>
<th>a = a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>a = ab − ( \lfloor \frac{ab}{2^k} \rfloor \frac{q}{2^k} ) ( q \equiv ab \mod \pm q ).</td>
</tr>
</tbody>
</table>

1. sqrdmulh t, a, \( \lfloor \frac{q}{2^k} \rfloor \)
2. mul a, a, b
3. mla a, t, q

Let q be an odd modulus, and R be the size of the arithmetic. We describe the modular reductions and multiplications for computing in \( \mathbb{Z}_q \). Barrett reduction [Bar86] reduces a value a by approximating a mod ±q with \( a - \lfloor \frac{a\pm1}{2^k} \rfloor \) (cf. Algorithm 1). For multiplying an unknown a with a fixed value b, we compute \( ab - \lfloor \frac{ab}{2^k} \rfloor \) \( q \equiv ab \mod \pm q \) (Barrett multiplication [BHK⁻²²]) where \( \lfloor \cdot \rfloor \) is the function mapping a real number r to \( 2 \lfloor \frac{r}{2} \rfloor \) (cf. Algorithm 2). We propose heretofore unseen multiply-add/sub variants of Barrett multiplication as shown in Algorithms 3 and 4. Algorithm 3 computes the representation of \( a + bc \) by merging \text{mul} and \text{add} into \text{mla}. Algorithm 4 computes \( a - bc \) by toggling \text{mls} into \text{mla} and merging \text{mul} and \text{sub} into \text{mls}. For accumulating several products, we use Montgomery multiplication [Mon85] as shown in Algorithm 5.

**Algorithm 3** Barrett_mla.

<table>
<thead>
<tr>
<th>Input:</th>
<th>a = a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>a = a + bc − ( \lfloor \frac{b\pm1}{2^k} \rfloor \frac{q}{2^k} ) ( q ).</td>
</tr>
</tbody>
</table>

1. sqrdmulh t, b, \( \lfloor \frac{q}{2^k} \rfloor \)
2. mla a, b, c
3. mla a, t, q

**Algorithm 4** Barrett_mls.

<table>
<thead>
<tr>
<th>Input:</th>
<th>a = a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>a = a − bc + ( \lfloor \frac{b\pm1}{2^k} \rfloor \frac{q}{2^k} ) ( q ).</td>
</tr>
</tbody>
</table>

1. sqrdmulh t, b, \( \lfloor \frac{q}{2^k} \rfloor \)
2. mla a, b, c
3. mla a, t, q

**Algorithm 5** Inner product using Montgomery reduction [BHK⁻²², Algorithm 14].

<table>
<thead>
<tr>
<th>Input:</th>
<th>(a0, a1, b0, b1) = (a0, a1, b0, b1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>c = ( \sum a_i b_i + (\sum a_i b_i \mod \pm R) \mod \pm R ) ( (\sum a_i b_i \mod \pm R) \mod \pm R ) \equiv (\sum a_i b_i) R^{-1} \mod \pm q ).</td>
</tr>
</tbody>
</table>

1. (lo, hi) = (smull, smull2)(a0, b0)
2. (lo, hi) = (lo, hi)(smllal, smllal2)(a1, b1)
3. uzp1 t, lo, hi
4. mul t, t, \( -q^{-1} \mod \pm R \)
5. (lo, hi) = (lo, hi)(smllal, smllal2)(t, q)
6. uzp2 c, lo, hi
7. \( \triangleright \) Steps 3-6 (written \( c = \text{Montgomery_long}(lo, hi) \)) form a Montgomery reduction.

3 Polynomial Multiplications

We go through the mathematical background of various transformations. In general we find chains of isomorphisms and monomorphisms gradually decomposing large problems into smaller problems computably quickly. If the polynomial ring splits nicely, we have a
chain of isomorphisms. This is however generally false for NTRU and NTRU Prime and we need to use both monomorphisms and isomorphisms.

This section is structured as follows. Section 3.1 reviews the Chinese remainder theorem for polynomial rings. This forms the basis of various fast polynomial ring transformations. We then survey various FFTs, including Cooley–Tukey in Section 3.2, Bruun and its finite field counterparts in Section 3.3, Good–Thomas in Section 3.4, and Rader in Section 3.5. Section 3.6 reviews Toom–Cook. Section 3.7 reviews Schönhage and Nussbaumer. We use NTT as a synonym for FFT. Section 3.8 reviews the switches of coefficient rings for NTTs, and the bit losses of Toom–Cook.

3.1 The Chinese Remainder Theorem for Polynomial Rings

Let \( n = \prod_{i=0}^{h-1} n_i \) and \( g_{i_0,\ldots,i_{h-1}} \in R[x] \) be coprime polynomials where \( i_0 \in [0, n_0), \ldots, i_{h-1} \in [0, n_{h-1}) \). The CRT gives us the following chain of isomorphisms

\[
\frac{R[x]}{\langle \prod_{i_0,\ldots,i_{h-1}} g_{i_0,\ldots,i_{h-1}} \rangle} \cong \prod_{i_0} \frac{R[x]}{\langle \prod_{i_1,\ldots,i_{h-1}} g_{i_0,\ldots,i_{h-1}} \rangle} \cong \cdots \cong \prod_{i_0,\ldots,i_{h-1}} \frac{R[x]}{\langle g_{i_0,\ldots,i_{h-1}} \rangle}.
\]

Usually, multiplications in \( \prod_{i_0,\ldots,i_{h-1}} R[x]/\langle g_{i_0,\ldots,i_{h-1}} \rangle \) are cheap. If the chain of isomorphisms is also cheap, we have an algorithmic improvement for multiplying polynomials in \( R[x]/\langle \prod_{i_0,\ldots,i_{h-1}} g_{i_0,\ldots,i_{h-1}} \rangle \). If the \( n_i \) is a small constant, then it is usually cheap to decompose a polynomial ring into a product of \( n_i \) polynomial rings.

In this paper, we use the words “radix,” “split,” and “layer” for describing the choices of transformations, demonstrated below for \( h = 2 \). Suppose we have isomorphisms

\[
\frac{R[x]}{\langle \prod_{i_0,i_1} g_{i_0,i_1} \rangle} \cong \prod_{i_0} \frac{R[x]}{\langle \prod_{i_1} g_{i_0,i_1} \rangle} \cong \prod_{i_1} \frac{R[x]}{\langle g_{i_0,i_1} \rangle}
\]

where \( i_0 \) ranges over \( 0, \ldots, n_0 - 1 \) and \( i_1 \) ranges over \( 0, \ldots, n_1 - 1 \). We call the isomorphism \( \eta_0 \) a radix-\( n_0 \) split and an implementation of \( \eta_0 \) a radix-\( n_0 \) computation. Usually, we implement several isomorphisms together to minimize memory operations. We call the resulting computation a multi-layer computation. Suppose we implement \( \eta_0 \) and \( \eta_1 \) in a single load-store pair. If \( \eta_0 \) and \( \eta_1 \) rely on the same shape of computation \( X \), we call the resulting multi-layer computation a 2-layer \( X \). Additionally, if \( n_0 = n_1 \), we call it a 2-layer radix-\( n_0 \) \( X \).

3.2 Cooley–Tukey FFTs

In Cooley–Tukey FFTs [CT65], we have \( v \) a small constant, \( \zeta \in R, \omega_n \) a principal \( n \)th root of unity in \( R, n \mid \text{char}(R) \), and \( g_{i_0,\ldots,i_{h-1}} = x^v - \zeta^n \sum_{i_h} \eta_0^{i_h} n_i \in R[x] \). Since \( \prod_{i_0,\ldots,i_{h-1}} g_{i_0,\ldots,i_{h-1}} = x^{nv} - \zeta^n \), the efficiency of multiplying polynomials in \( R[x]/\langle x^{nv} - \zeta^n \rangle \) boils down to the efficiency of the isomorphisms indexed by \( i_h \)'s. If \( v = 1 \), the NTT is complete. An NTT with \( v > 1 \) is incomplete. Furthermore, the NTT is cyclic if \( \zeta^n = 1 \). We refer to [AHY22, Figures 1 and 2] as illustrations for the radix-2 and radix-3 cases.

3.3 Bruun-Like FFTs

[Bru78] first introduced the idea of factoring into trinomials \( g_{i_0,\ldots,i_{h-1}} \) when \( n \) is a power of two — as a way to reduce the number of real multiplications when we are operating over \( C \). [Mur96] later generalized it to arbitrary even \( n \). For our implementations, we need the results of factoring \( x^2 + 1 \in F_q[x] \) when \( q \equiv 3 \pmod{4} \) [BGM93] and the composed multiplication of polynomials in \( F_q[x] \) [BC87]. Factorizing \( x^n - 1 \) over \( F_q \) is an active research area [BGM93, Mey96, TW13, MVD14, WY18, WY19].
3.3.1 Review: Bruun’s Original Complex \((R = \mathbb{C})\) Case

We choose \(g_{i_0,\ldots,i_{n-1}} = x^{2n} - \left(\zeta \omega_n \sum_{i} u_i \prod_{j \neq i} n_j + \zeta^{-1} \omega_n^{-1} \sum_{i} u_i \prod_{j \neq i} n_j\right) x^n + 1\) so \(x^{2n} - (\zeta^n + \zeta^{-n}) x^n + 1 = \prod_{g_{i_0,\ldots,i_{n-1}}} g_{i_0,\ldots,i_{n-1}}\). This provides us an alternative factorization for \(x^{2n} - 1 = (x^{2n} - 1)(x^{2n} + 1)\) by choosing \(\zeta^n = \omega_4\). Since the sum of a complex number with norm \(1\) and its inverse is real, we only need arithmetic in \(\mathbb{R}\) to reach \(\prod_{g_{i_0,\ldots,i_{n-1}}} \mathbb{C}[x]/\left\langle g_{i_0,\ldots,i_{n-1}(x)}\right\rangle\). For \(v = 1\), if we split each \(g_{i_0,\ldots,i_{n-1}}\) into the degree-1 factors, we have \(\mathbb{C}[x]/\langle x^{2n} + 1 \rangle \cong \prod_{g_{i_0,\ldots,i_{n-1}}} \left(\mathbb{C}[x]/\left\langle x - \omega_4^{1 + \sum_{i} u_i \prod_{j \neq i} n_j}\right\rangle \times \mathbb{C}[x]/\left\langle x - \omega_4^{n - 1} \sum_{i} u_i \prod_{j \neq i} n_j\right\rangle\right)\), which is \(\prod_i \mathbb{C}[x]/\langle x - \omega_4^{1+2i}\rangle\) in disguise.

3.3.2 \(R = \mathbb{F}_q\) Where \(q \equiv 3 \pmod{4}\)

Theorem 1 ([BGM93, Theorem 1]). Let \(q \equiv 3 \pmod{4}\) and \(2^w\) be the highest power of two in \(q + 1\). If \(k < w\), then \(x^{2k} + 1\) factors into irreducible trinomials \(x^{2k} + \gamma x + 1\) in \(\mathbb{F}_q[x]\). Else (i.e., \(k \geq w\) \(x^{2k} + 1\) factors into irreducible trinomials \(x^{2k-w+1} + \gamma x^{2k-w} - 1\) in \(\mathbb{F}_q[x]\).

We need Theorem 1 for our implementations. Also, given \(f_0, f_1 \in \mathbb{F}_q[x]\), we define their “composed multiplication” as \((f_0 \circ f_1) = \prod_{f_0(\alpha) = 0} f_1(\beta) = (x - \alpha \beta)\) where \(\alpha, \beta\) run over all the roots of \(f_0, f_1\) in an extension field of \(\mathbb{F}_q\).

We need the following from [BC87]:

**Lemma 1** ([BC87, Equation 8]). \((\prod_i f_{0,i} \circ f_{1,i}) / (\prod_i f_{0,i} \circ f_{1,i}) = \prod_{0,i} f_{0,i} \circ f_{1,i}\) holds for any sequences of polynomials \(f_{0,i} \circ f_{1,i} \in \mathbb{F}_q[x]\).

**Lemma 2** ([BC87, Equation 5]). If \(f_0 = \prod \alpha (x - \alpha) \in \mathbb{F}_q[x]\), then for any \(f_1 \in \mathbb{F}_q[x]\) we have \(f_0 \circ f_1 = f_1 = \prod \alpha f_{1,i} (x - \alpha^{-1} x) \in \mathbb{F}_q[x]\).

**Lemma 3.** Let \(r\) be odd, \(x^r - 1 = \prod_{\omega r} (x - \omega r^{i_0}) \in \mathbb{F}_q[x]\), and \(x^{2k} - 1 = \prod_i f_i \in \mathbb{F}_q[x]\).

We have \(x^{2k} - 1 = \prod_i \left(x^{2k} - \omega r^{i_0} f_i\right) = \prod_{i_0,i_1} \omega r^{\deg(f_i)} f_{i_1} (\omega r^{i_0} x)\).

**Proof.** First observe \(x^{2k} - 1 = (x^r - 1) \circ (x^{2k} - 1)^6\). By Lemma 1, this equals \(\prod_{\omega r} (x - \omega r^{i_0} \circ (x^{2k} - 1)) = \prod_{i_0,i_1} (x - \omega r^{i_0} \circ f_{i_1})\). Then, according to Lemma 2, we write \((x - \omega r^{i_0} \circ (x^{2k} - 1) = x^{2k} - \omega r^{i_0} f_i\) and \((x - \omega r^{i_0} \circ f_{i_1} = \omega r^{i_0} f_{i_1} (\omega r^{i_0} x)\).

In summary, by Lemma 3 we have the following isomorphisms for \(\mathbb{F}_q[x]/\langle x^{2k} - 1 \rangle\):

\[
\frac{\mathbb{F}_q[x]}{(x^{2k} - 1)} \cong \frac{\mathbb{F}_q[x]}{\prod_{i_0,i_1} \omega r^{\deg(f_i)} f_{i_1} (\omega r^{i_0} x)}.
\]

3.3.3 Bruun’s Butterflies and Inverses

We illustrate radix-2 Bruun’s butterflies. Define \(\text{Bruun}_{\alpha,\beta}\) as follows:

\[
\text{Bruun}_{\alpha,\beta} = \begin{cases}
\frac{R[x]}{(x^{a_0} + a_1 x + a_2 x^2 + a_3 x^3)} \times \frac{R[x]}{(x^{a_0} + a_1 x + a_2 x^2 + a_3 x^3}) & \to (\frac{R[x]}{(x^{a_0} + a_1 x + a_2 x^2 + a_3 x^3)})
\end{cases}
\]

\[\alpha_0 = \{0 \leq i \leq q_0, 0 \leq i \leq q_1\} = \{\omega q_{0,q_1}, 0 \leq i < q_0\}\] in the splitting field of \(x^{n+1} - 1\).
We compute

\[
\begin{align*}
(\hat{a}_0, \hat{a}_1) &= (a_0 - \beta a_2 + \alpha \beta a_3, a_1 + (\alpha^2 - \beta) a_3 - \alpha a_2), \\
(\hat{a}_2, \hat{a}_3) &= (a_0 - \beta a_2 - \alpha \beta a_3, a_1 + (\alpha^2 - \beta) a_3 + \alpha a_2).
\end{align*}
\]

We compute \((a_0 - \beta a_2, a_1 + (\alpha^2 - \beta) a_3, \alpha a_2, \alpha \beta a_3)\), swap the last two values implicitly, and do an addition-subtraction (cf. Figure 1). Notice that we can use \texttt{Barrett\_mla} and \texttt{Barrett\_mls} whenever a product is followed by only one accumulation or subtraction.

Figure 1: Bruun’s butterfly. \((\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3) = \texttt{Bruun}_{\alpha, \beta}(a_0, a_1, a_2, a_3)\).

\[
2\texttt{Bruun}_{\alpha, \beta}^{-1} \left( \frac{R[x]}{(x^2 + \alpha x + \beta)} \times \frac{R[x]}{(x^2 - \alpha x + \beta)} \right) \rightarrow \frac{R[x]}{(x^2 + (\beta - \alpha^2)x^2 + \beta^2)}
\]

correspondingly defines the inverse, where

\[
\begin{align*}
2(a_0, a_1) &= (\hat{a}_0 + \hat{a}_2 + (\hat{a}_3 - \hat{a}_1) \alpha^{-1} \beta, \hat{a}_1 + \hat{a}_3 - (\hat{a}_0 - \hat{a}_2) \alpha^{-1} \beta^{-1} (\alpha^2 - \beta)), \\
2(a_2, a_3) &= ((\hat{a}_3 - \hat{a}_1) \alpha, (\hat{a}_0 - \hat{a}_2) \alpha^{-1} \beta^{-1}).
\end{align*}
\]

We compute \((\hat{a}_0 + \hat{a}_2, \hat{a}_1 + \hat{a}_3, \hat{a}_0 - \hat{a}_2, \hat{a}_3 - \hat{a}_1)\), swap the last two values implicitly, multiply the constants \(\alpha^{-1} \beta, \alpha^{-1} \beta^{-1}, \) and \((\alpha^2 - \beta)\), and add-sub (cf. Figure 2). Both \texttt{Bruun}_{\alpha, \beta} and \texttt{2Bruun}_{\alpha, \beta}^{-1}, takes 4 multiplications.

Figure 2: Inverse of Bruun’s butterfly. \((2a_0, 2a_1, 2a_2, 2a_3) = 2\texttt{Bruun}_{\alpha, \beta}^{-1}(\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3)\).

### 3.3.4 Special Cases of Bruun’s Butterflies

**\texttt{Bruun}_{\sqrt{2}, 1}**. The initial split of \(x^{2^k} + 1\) leads to \texttt{Bruun}_{\sqrt{2}, 1}. Since \(\beta = 1\) and \(\alpha^2 - \beta = 2 - 1 = 1\), we only need two multiplications with the constant \(\sqrt{2}\).

**\texttt{Bruun}_{\alpha, \pm 1}**. We only need 3 multiplications for each of \texttt{Bruun}_{\alpha, \pm 1} and \texttt{2Bruun}_{\alpha, \pm 1} since \(\beta = \pm 1\). After we split \(x^{2^k} + 1 = \left(x^{2^{k-1}} + \sqrt{2} x^{2^{k-2}} + 1\right) \left(x^{2^{k-1}} - \sqrt{2} x^{2^{k-2}} + 1\right)\), we need \texttt{Bruun}_{\alpha, \pm 1} for further factorizations by Theorem 1.

**\texttt{Bruun}_{\alpha, \frac{a^2}{r}}**. We save no multiplications, but only use 2 constants \(\alpha\) and \(\frac{a^2}{r}\) instead of 4. It is used in the split of \(x^{2^k} + \omega_r^{2^{k-1}}\) for an odd \(r\).
3.4 Good–Thomas FFTs

Good–Thomas FFT [Goo58] convert cyclic FFT and convolutions into multi-dimensional ones for coprime $n_i$’s. We first regard $R[x]/(x^{n_i} - 1) \cong (R[x]/(x^r - y))/g^n - 1$ as a polynomial in $y$ and define $\tilde{R} := R[x]/(x^r - y)$ [FP07]. Instead of transforming into $\prod \tilde{R}[y]/\langle y - \omega_i \rangle$ with Cooley–Tukey, we choose $\omega_i := \omega_{t_i}$ and write $\omega_i = \prod \omega_n$, where $(e_i)$ is the unique tuple satisfying $a \equiv \sum e_i (a \mod n_i) \mod n$. Now we rewrite the image as $\prod \tilde{R}[y]/\langle y - \prod \omega_n \rangle$. Rewriting the indices gives $\prod_{i_0, \ldots, i_{h-1}} \tilde{R}[y]/\langle y - \prod \omega_n \rangle$. By introducing the equivalences $y \sim \prod_{i} \omega_i$ and $\forall l, u_i^{\omega_i} \sim 1$, we transform $\tilde{R}[y]/\langle y^n - 1 \rangle$ into $\prod_{i_0, \ldots, i_{h-1}} \tilde{R}[y]/\langle y - \prod \omega_n \rangle$ with a multi-dimensional FFT. Formally, we have

$$\tilde{R}[y] \cong \prod_{i_0, \ldots, i_{h-1}} \tilde{R}[y]/\langle y - \prod \omega_n \rangle.$$ 

3.5 Rader’s FFT for Odd Prime $p$

Suppose $\omega_p \in R$ for an odd prime $p$. [Rad68] introduced how to map a polynomial $\sum_{i=0}^{p-1} a_i x^i \in R[x]/(x^p - 1)$ to the tuple $(\tilde{a}_i) := \left(\sum_{i=0}^{p-1} a_i \omega_p^i\right) \in \prod \tilde{R}[x]/\langle x - \omega_p \rangle$. Let $g \in \mathbb{Z}_p^*$ be a generator. Now $\{1, \ldots, p-1\} = \{g, \ldots, g^{p-2}, g^{p-1} = 1\}$. Write $j = g^k$ and $i = g^{-t}$. Then $\tilde{a}_j - \tilde{a}_i = \tilde{a}_i - a_0 = \sum_{i=1}^{p-1} a_i \omega_p^i = \sum_{t=1}^{p-1} a_{g^{-t}} \omega_p^{h-i}.$

The sequence $(\sum_{i=1}^{p-1} a_{g^{-i}} \omega_p^{g^{-i-1}})_{j=1, \ldots, p-1}$ is the size-$(p-1)$ cyclic convolution of sequences $(a_i)_{i=1, \ldots, p-1}$ and $(\omega_p^i)_{i=1, \ldots, p-1}$. We refer to [ACC+21, Appendix B] and [AHY22, Section 3.1.3] for illustrations.

3.6 Toom–Cook (TC) and Karatsuba

Toom–Cook [CA69, Too63] and Karatsuba [KO62] are divide-and-conquer approaches for multiplying polynomials in $\mathbb{Z}[x]$. We can also use them for multiplying polynomials in $R[x]_{<n}$. We introduce $y \sim x^t$ (zero-pad so that $k|n$) [Ber01], and map $R[x]_{<n} \mapsto R[x]/(x^n - y)/\langle y \rangle_{<k}$ for $R' = R[x]/\langle y \rangle$ with $deg < \frac{n}{k} - 1$.

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

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-$k$) into $\tilde{R}'[y]/\langle y - s_i \rangle$. [KO62] chose $k = 2$ at $\{s_i\}_i = \{0, 1, \infty\}$; [Too63] chose $\{s_i\}_i = \{0, \pm 1, \ldots, \pm (k-1)\}$; and [Win80, Page 31] replaced $-k + 1$ with $k + 1$. We write $\textbf{TC}_{2k-1} \times k$ for the matrix mapping the coefficients of a degree $< k$ polynomial into $\prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$ and $\textbf{TC}_{2k-1} \times (2k-1)$ for the matrix mapping $\prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$ into $R'[y]/\langle \prod_{i=0}^{2k-2} (y - s_i) \rangle$.

A key observation is that for $k = 5$ and $\{s_i\}_i = \{0, \pm 1, \pm \frac{1}{2}, 3, \infty\}$, $\textbf{TC}_{5, 9}^{-1}$ only requires “division by 8”. This implies 3-bit losses when working over $\mathbb{Z}_{216}$. The matrix $\textbf{TC}_{9, 9}^{-1}$ will be stated explicitly in the full version.

3.7 Schönhage’s and Nussbaumer’s FFTs

Instead of isomorphisms based on CRT, we sometimes compute chains of monomorphisms, where we can determine the unique inverse image from the product of two images. Given
polynomials \(a, b \in R[x]/(y)\) where \(g\) is a degree-\(n\) polynomial, we introduce with \(y = x^{n_1}\), and write \(a\) and \(b\) as polynomials in \(R[x]/(x^{n_1} - y, g_0)\), where \(g_0 = y - x^{n_1} = g(x)\). In other words, \(a_0(y) := \sum_{i=0}^{n_1-1} \sum_{j=0}^{n_1-1} a_{i+j+n_1} x^i y^j \in R[x,y]/(x^{n_1} - y, g_0)\) (ditto for \(b_0\)).

We recap transforms when \(R[x,y]/(x^{n_1} - y, g_0)\) does not naturally split. Our presentation is motivated by [Ber01, Section 9, Paragraph “High–radix variants”] and [vdH04, Section 3].

We want an injection \(R[x]/(x^{n_1} - y) \hookrightarrow \tilde{R}\) such that \(R[x,y]/(x^{n_1} - y, g_0) \hookrightarrow \tilde{R}[y]/(g_0)\) is a monomorphism with \(\tilde{R}[y]/(g_0) \cong \prod_i \tilde{R}[y]/(g_{0,j})\). A Schönhage FFT [Sch77] is when \(g_0 = y^{n_0} - 1\), and \(\tilde{R} = R[x]/(h)\) with \(h = \Phi_{n_0}(x)\) (the \(n_0\)-th cyclotomic polynomial). E.g., “cyclic Schönhage” for powers of two \(n_0\), \(n_1 = \frac{n_0}{2}\), \(g_0 = y^{n_0} - 1\), and \(h = x^{2n_1} + 1\):

\[
\frac{R[x]}{(x^{n_0n_1} - 1)} \cong \frac{R[x]}{(x^{n_0} - 1)} \hookrightarrow \frac{\tilde{R}[y]}{(y^{n_0} - 1)} \cong \prod_i \tilde{R}[y]/(g_{0,j})
\]

We can also exchange the roles of \(x\) and \(y\) and get Nussbaum’s FFT [Nus80]. We map \(R[x,y]/(x^{n_1} - y, g_0) \hookrightarrow R[x,y]/(h, g_0)\) such that \(g_0 = \Phi_{2n_1}(y)\) and \(h = (x^{2n_1} - 1)\). This can be illustrated for powers of two \(n_0 = n_1\), \(h = x^{2n_1} - 1\), and \(g_0 = y^{n_0} + 1\):

\[
\frac{R[x]}{(x^{n_0n_1} + 1)} \cong \frac{R[x,y]}{(x^{n_1} - y, y^{n_0} + 1)} \hookrightarrow \frac{\tilde{R}[y]}{(y^{n_0} + 1)} \cong \prod_i \tilde{R}[x]/(x - y')
\]

### 3.8 Enlarging Coefficient Rings

To multiply in \(R[x]/(x^{n_1} - 1)\) while \(R\) lacks a principal \(n\)-th root of unity, we map \(R \leftrightarrow R'\) and induce a multiplication in \(R'[x]/(x^{n_1} - 1)\) with \(\omega_n \in R'\). For \(R = \mathbb{Z}_q\) with signed arithmetic, we pick \(R' = \mathbb{Z}_{q'}\) with any \(q' > \frac{nq^2}{2}\) [ACC+21, CHK+21] such that for each prime \(p\) dividing \(q'\), \(p \equiv 1\) (mod \(n\)) [AB74].

The second scenario is division by 2 when 2 is not invertible, for example in \(\mathbb{Z}_{2^k}\). Suppose we want \(r \in \mathbb{Z}_{2^k}\). We instead compute \(2^er \in \mathbb{Z}_{2^{k+\epsilon}}\), and right-shift \(2^er\) by \(\epsilon\) bits [Ber01, 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 \(\mathbb{Z}_{2^{k+\epsilon}}\) so \(r = \frac{q}{2^k r} \in \mathbb{Z}_{2^k}\) 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

##### 4.1.1 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 \(r \cdot_M a\) for \((\cdot_M)(r,a))\) satisfying the following:
\[ (r + s) \cdot M (a + b) = r \cdot M a + r \cdot M b + s \cdot M a + s \cdot M b. \]

\[ 1 \cdot M a = a. \]

\[ (rs) \cdot M a = r \cdot M (s \cdot M a). \]

We call \((M, +, \cdot_M)\) an \(R\)-module. For elements \(b_0, \ldots, b_{n-1} \in M\), if they are linearly independent and every element in \(M\) can be expressed as a linear combination of \(b_0, \ldots, b_{n-1}\), we call \(\{b_0, \ldots, b_{n-1}\}\) a basis of \(M\) and \(n\) the dimension. We denote by \(R^n\) for an \(n\)-dimensional free module. Notice that a ring \(R\) and a polynomial ring \(R[x]/\langle g \rangle\) 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:

\[ \forall r \in R, \forall a, b \in M, \eta(r \cdot_M a + b) = r \cdot_N \eta(a) + \eta(b). \]

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

### 4.1.2 Associative Algebra

For rings \(R\) and \(A\), we turn \(A\) into an associative \(R\)-algebra by introducing a module structure. One identifies the module addition with the ring addition, and provide a scalar multiplication \(\cdot_A: R \times A \to A\) satisfying

\[ \forall r \in R, \forall a, b \in A, r \cdot_A (ab) = (r \cdot_A a) b = a (r \cdot_A b). \]

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 Section 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. Algorithm 15 computes the result with several inner products of the rows of the matrix and the vector. Algorithm 16 accumulates several products of the columns of the matrix and the corresponding elements of the vector. 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 15 is a special case of the inner product approach (cf. Algorithm 13), and Algorithm 16 is a special case of the outer product approach (cf. Algorithm 14). We also call them accordingly.

### 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
Algorithm 6

Toeplitz matrices for weighted convolutions. Express \( c = \mathbf{a} \mathbf{b} = \left( \sum_i a_i x^i \right) \left( \sum_i b_i x^i \right) \in R[x]/\langle x^n - \zeta \rangle \) as \( c = \sum_i c_i x^i \) where \( c_i = \left( \sum_{j=0}^{i-1} a_j b_{i-j} + \zeta \sum_{j=i+1}^{n-1} a_j b_{n-i-j} \right) \). We choose an \( n' \geq n \), and zero-pad \( \mathbf{a} \) and \( \mathbf{c} \) to size-\( n' \) polynomials \( \mathbf{a}' \) and \( \mathbf{c}' \), respectively, and define

\[
\text{Expand}_{n \to n', \zeta} = \left( \sum_{i<n} b_i x^i, \zeta \right) \mapsto \left( 0, \ldots, 0, b_{n-1}, \ldots, b_0, \zeta b_{n-1}, \ldots, \zeta b_1, 0, \ldots, 0 \right)_{n'-n}. \]

We have

\[
\mathbf{c}' = \text{Toeplitz}_{n' \times n'}(\text{Expand}_{n \to n', \zeta}(\mathbf{b})) \mathbf{a}'.
\]

Toeplitz matrices in \( \text{Expand}_{n \to n, \zeta}(-) \) is exactly the asymmetric_mul by [BHK+22, Section 4.2]. See [Hwa22, Paragraph “A Toeplitz matrix view of asymmetric multiplication”, Section 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 [SKS+21, NG21, BHK+22] took the inner product approach with pre- and post-transposes. The Toeplitz structure admits fast construction of the full matrix. For a weighted convolution over \( x^i - \zeta \), we apply \text{Expand}_{i \to i, \zeta} with \text{ext} instructions, and accumulate vector-by-scalar products. Algorithm 6 is an illustration.

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

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

Algorithm 6 Outer product approach for \( R[x]/\langle x^n - \zeta \rangle \).

Inputs: \( \mathbf{a} = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \), \( \mathbf{b} = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \).

Outputs: \( \mathbf{c} = \mathbf{a} \mathbf{b} \mod (x^n - \zeta) \).

1: \( \mathbf{b} = b_3 || b_2 || b_1 || b_0 \)
2: \( \mathbf{t} = a_3 || a_2 || a_1 || a_0 \)
3: Compute \( t = \zeta a_3 || \zeta a_2 || \zeta a_1 || \zeta a_0 \) with Barrett multiplication.
4: \( \triangleright \) [BHK+22] proposed an interleaved version of this; others [SKS+21, NG21] reduced the interleaved partial results instead.
5: \( \triangleright \) The remaining steps are different from [BHK+22].
6: \text{ext } t_1, t, t_0, \#3 \cdot 4 \quad \triangleright t_1 = a_2 || a_1 || a_0 || \zeta a_3
7: \text{ext } t_2, t, t_0, \#2 \cdot 4 \quad \triangleright t_2 = a_1 || a_0 || \zeta a_3 || \zeta a_2
8: \text{ext } t_3, t, t_0, \#1 \cdot 4 \quad \triangleright t_3 = a_0 || \zeta a_3 || \zeta a_2 || \zeta a_1
9: \( (l_0, h_0) = (\text{smull, smull12})(t_0, b_0) \)
10: \( (l_0, h_0) = (l_0, h_0)(\text{smulal, smulal2})(t_1, b_1) \)
11: \( (l_0, h_0) = (l_0, h_0)(\text{smulal, smulal2})(t_2, b_2) \)
12: \( (l_0, h_0) = (l_0, h_0)(\text{smulal, smulal2})(t_3, b_3) \)
13: \( c = \text{Montgomery\_long}(l_0, h_0) \)
Generally speaking, once the Toeplitz matrix is constructed via \texttt{exts} 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 \( R[x]/(x^4 - \zeta) \). Figure 4 illustrates the register view of inner-product-based Toeplitz matrix–vector multiplication and Figure 3 for the outer-product-based one. For Figure 4, we apply \( \log_2 4 \cdot \frac{1}{2} = 4 \) pairs of \( \text{trn1, trn2} \) 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 \( \text{trn1, trn2} \). On the other hand, Figure 3 requires no additional permutations and avoids the interleaved operands and results. This implies nearly no permutation instructions and very low register pressure.

![Figure 3: Outer-product-based Toeplitz matrix–vector multiplication via vector-by-scalar multiplication](image)

No permutations are required once we have data in registers \( v_0, v_1, v_2, v_3 \). We only need 5 registers \( v_0, \ldots, v_4 \) for holding the operands and 1 or 2 registers for the partial results.

![Figure 4: Inner-product-based Toeplitz matrix–vector multiplication via vector-by-vector multiplication](image)

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 \( 4 \times 4 \) 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 \( v_0, v_1, \ldots, v_{11} \) 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.
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 $a, b \in R[x]_{< k}$ will be considered as $ab \in R[x]_{< 2k-1}$. Given an $a \in R[x]_{< k}$, we write $(a, -) : R^k \to R^{2k-1}$ for the module homomorphism $b \mapsto ab$ and $(a, -)^*$ its transpose. The Toeplitz matrix-vector product (TMVP) can be defined for arbitrary $R$-algebra monomorphisms from $R[x]_{< n}$ to $S$ where $n \geq 2k - 1$.

**Definition 1.** For arbitrary $R$-algebra monomorphism $f : R[x]_{< n} \to S$, we define $f_k = f|_{R[x]_{< k}}$. Furthermore, let $\text{rev}_{k \to k} : R^k \to R^k$ be the index reversal map and $\text{id}_{m \to n} : R^m \to R^n$ be the inclusion (pad 0’s) map for $m \leq n$. The TMVP associated with $f$ refers to the following module homomorphisms:

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

We call $(f^{-1})^* \circ \text{id}_{(2k-1) \to n}$ split–matrix, $f_k(a)$ split–vector, $(f_k(a), -)^*$ base multiplication, and $f_k^*$ base map. If $n = 2k - 1, f = \text{TC}_{(2k-1) \times (2k-1)}$, then this is the $k$-way Toeplitz–TC matrix–vector product [IKPC20, IKPC22]. Generally, any $R$-algebra monomorphism suffices. See Appendices A and B for formal proof and examples. We go through a higher-level overview of the idea.

Since $f$ is a ring monomorphism, we implement the module homomorphism $(a, -)$ as $\text{id}_{n \to (2k-1)} \circ f^{-1} \circ (f_k(a), -) \circ f_k$, take the transpose of $(a, -)$, and relate $(a, -)^*$ to the Toeplitz $\text{Toeplitz}_{k \times k}^{(-)}$ and the right-vector-multiplication $(-)(a)$. This allows us to convert any fast computation for $(a, -)$ into something for $(\text{Toeplitz}_{k \times k}^{(-)}) (a)$. Concretely, we suppose we have $(a, -) = f' \circ g^{-1} \circ ((g \circ f_k)(a), -) \circ g \circ f_k$ as shown in Figure 5a where $f' = \text{id}_{n \to (2k-1)} \circ f^{-1}$. We transpose all the objects and maps and revert all the arrows as shown in Figure 5b. Since $(\text{Toeplitz}^(-))(a) = \text{rev}_{k \times k} \circ (a, -)^*$, and $(-, a)_{R[x]/(x^k - \zeta)} = \text{id}_{k \to k} \circ (\text{Toeplitz}^(-))(a) \circ \text{Expand}_{k \to k, \zeta}$ as shown in Figure 6, we eventually have the following fast computation:

$$(a, -)_{R[x]/(x^k - \zeta)} = \text{id}_{k \to k} \circ \text{rev}_{k \to k} \circ f_k^* \circ g^* \circ ((g \circ f_k)(a), -)^* \circ (g^{-1})^* \circ f_k^* \circ \text{Expand}_{k \to k, \zeta};$$

\[ R^k \xrightarrow{(a, -)} R^{2k-1} \]

\[ f_k \]

\[ S \xrightarrow{(f_k(a), -)} S \]

\[ g \]

\[ S' \xrightarrow{((g \circ f_k)(a), -)} S' \]

\[ (a, -)^* \]

\[ f_k^* \]

\[ S^* \xleftarrow{(f_k(a), -)^*} S^* \]

\[ g^* \]

\[ S'^* \xleftarrow{((g \circ f_k)(a), -)^*} S'^* \]

(a) Fast computation for $(a, -)$. (b) Fast computation for $(a, -)^*$.

Figure 5: Fast computation for $(a, -)$ and $(a, -)^*$. $f' = \text{id}_{n \to (2k-1)} \circ f^{-1}$. 

22 Algorithmic Views of Vectorized Polynomial Multipliers for NTRU and NTRU Prime
5 Implementations

In this section, we describe our implementations. Section 5.1 goes through our Good–Thomas for the “big by small” polynomial multiplications in ntrulpr761/sntrup761 and ntruhps2048677. Section 5.2 describes the “big by big” polynomial multiplications for ntruhps2048677, including Toom–Cook in Section 5.2.1 and Toeplitz–TC in Section 5.2.2. Section 5.3 describes the “big by big” polynomial multiplications for ntrulpr761/sntrup761, including Good–Schönhage–Bruun in Section 5.3.1, Good–Rader–outer and Good–Rader–Bruun in Section 5.3.2.

5.1 Good–Thomas for “Big by Small” Polynomial Multiplications

We recall below the design principal of vectorization–friendly Good–Thomas from [AHY22], and describe our Good–Thomas for the “big by small” polynomial multiplications in ntruhps2048677 and ntrulpr761/sntrup761. For a cyclic convolution \( R[x]/\langle x^{v n_0} - 1 \rangle \) where \( n_0 \perp n_1 \) and \( v \) a multiple of the number of coefficients in a vector, one introduces the relations \( x^v \sim uw, u^{n_0} \sim 1, \) and \( w^{n_1} \sim 1. \) Usually, one picks \( n_0 \) and \( n_1 \) carefully for fast computations. In the simplest form, one picks \( n_0 = 2 \) and \( n_1 = 3. \) Our Good–Thomas computes the polynomial multiplication in \( \mathbb{Z}[x]/\langle x^{1536} - 1 \rangle \) with \( (v, n_0, n_1) = (4, 128, 3). \) After reaching \( \mathbb{Z}[x, u, w]/\langle x^4 - uw, u^3 - 1, w^{128} - 1 \rangle, \) we want to compute size-3 NTT over \( u^3 - 1 \) and size-128 NTT over \( w^{128} - 1. \) It suffices to choose a large modulus \( q' \perp 384 \) with a principal 384-th root of unity since the prime factorization \( 384 = 3 \cdot 128 \) implies the definability for both \( u \) and \( w. \) We choose \( q' \) as a 32-bit modulus bounding the maximum value of the product in \( \mathbb{Z}[x]/\langle x^{1536} - 1 \rangle. \) This implies \( v = \frac{128}{12} = 4 \) (each SIMD register in Neon holds a 128-bit chunk). Obviously, our Good–Thomas supports the “big by small” polynomial multiplications in ntruhps2048677 and ntrulpr761/sntrup761 (generally, any “big by small” with size less than or equal to 1536).
5.2 NTRU Implementations over \( \mathbb{Z}_{65536} \)

We discuss our implementations for multiplying polynomials in \( \mathbb{Z}_{2048}[x]/(x^{677} - 1) \) for ntruhps2048677. We propose two implementations with 16-bit arithmetic mod 65536: (i) Toom–Cook implements Toom–Cook with the splitting sequence 5 \( \rightarrow \) 3 \( \rightarrow \) 3 \( \rightarrow \) 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 [KRS19, NG21] 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 [IKPC22].

5.2.1 Toom–Cook

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

Chosen splitting sequence: Toom–5 \( \rightarrow \) two Toom–3’s \( \rightarrow \) Karatsuba. We first zero-pad the size-677 polynomials to size-720 for ease of vectorization and compute in \( \mathbb{Z}_{216} \). Since the coefficient ring of ntruhps2048677 is \( \mathbb{Z}_{2048} \) and \( \frac{\mathbb{Z}_{2048}}{\mathbb{Z}_{216}} = 2^5 \), divisions by \( 2^e \) for \( e = 0, \ldots, 5 \) translate into shifting \( e \) bits. We choose the splitting sequence 5 \( \rightarrow \) 3 \( \rightarrow \) 3 \( \rightarrow \) 2. Our Toom–Cook consists of one layer of \( TC_{9 \times 5} \), two layers of \( TC_{3 \times 3} \)'s, one layer of \( TC_{3 \times 2} \), 675 size-8 schoolbooks, one layer of \( TC_{3 \times 3}^{-1} \), two layers of \( TC_{5 \times 5}^{-1} \)'s, and one layer of \( TC_{9 \times 9}^{-1} \). We choose the point sets \( \{0, \pm 1, \pm 2, \pm \frac{1}{2}, 3, \infty \} \) (cf. Section 3.6) for \( TC_{9 \times 5} \) and \( \{0, \pm 1, 2, \infty \} \) for \( TC_{5 \times 3} \). The interpolation matrices \( TC_{9 \times 9}^{-1}, TC_{5 \times 5}^{-1} \), and \( TC_{3 \times 3}^{-1} \) 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 [NG21]. [NG21] treated each polynomial as a size-720 polynomial, and applied Toom–Cook with the splitting sequence 3 \( \rightarrow \) 4 \( \rightarrow \) 2 \( \rightarrow \) 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, [NG21] basically padded to size-64 polynomials before Karatsuba. In this paper, we instead split via the sequence 5 \( \rightarrow \) 3 \( \rightarrow \) 3 \( \rightarrow \) 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 \( \{0, \pm 1, \pm 2, \pm 3, 4, \infty \} \). We also avoid zero-padding in vectorization. We merge the two Toom-3 layers (for both \( TC_{5 \times 3} \) and \( TC_{5 \times 5}^{-1} \)) to reduce memory operations.

Memory optimizations for interpolations. Let \( k|n, g' \) be a polynomial of degree at least \( \frac{n}{k} - 1 \), and \( R'[x] \) be \( R[x]/(g') \). Recall that \( TC_{(2k-1) \times k} \) computes \( R'[x]/(x^\tau - y)[y] \to R'[y]/(\prod_{i=0}^{2k-2} (y - s_i)) \cong \prod_{i=0}^{2k-2} R'[y]/(y - s_i) \) and results in computations in \( R[x]/(g') \).

After examining the source code, we find that prior works [KRS19, NG21] inverted the steps \( \cong \) and \( \to \) separately. Algorithm 7 is an illustration. Inverting \( \cong \) means applying the interpolation matrix and inverting \( \to \) means accumulating the overlapped coefficients while substituting \( y \) with \( x^\tau \) in each of the polynomials in \( R[x]/(g') \). We instead alternate between the inversions of \( \cong \) and \( \to \) to reduce memory operations, in essence merging two layers of computations.
Algorithm 7 TC^−1_{5×5} by [KRS19, NG21].

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

1: Declare array \textit{mem} of size-720 polynomials.
2: for \( i = 0, 1, 2 \) do
3: \( \textit{mem}[0-4] = TC^{-1}_{5×5}(p_0[i], \ldots, p_4[i]) \).
4: \( \triangleright\) Memory read and write.
5: for \( j = 0, \ldots, 4 \) do
7: end for \( \triangleright\) Memory read and write.
8: end for
9: end for

Algorithm 8 Our TC^−1_{5×5}.

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

1: Registers \textit{r}[11].
2: \( r[0-4] = TC_{5×5}^{-1}(p_0[0], \ldots, p_4[0]) \).
3: \( r[6-10] = TC_{5×5}^{-1}(p_0[2], \ldots, p_4[2]) \).
4: \( \triangleright\) 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: \( \triangleright\) Memory write.
13: \( r[0-4] = TC_{5×5}^{-1}(p_0[1], \ldots, p_4[1]) \).
14: \( \triangleright\) Memory read.
15: for \( j \leftarrow 0 \) to 4 do
16: \( c[2j + 1] = r[j] \).
17: end for
18: \( \triangleright\) Memory write.

5.2.2 Toeplitz-TC

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

Our Toeplitz-TC with the splitting sequence $5 \rightarrow 3 \rightarrow 3 \rightarrow 2$. Algorithm 9 describes our Toeplitz-TC implementation. Essentially, we regard size-677 polynomials \( a \) and \( b \) as size-720 polynomials. In practice, we zero-pad \( a \) and \( b \) 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 10 describes our Toeplitz-TC-3-3-2 following Toeplitz-TC-5.

Each step of Algorithms 9 and 10 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, TC'−1, and TC', we prefer shifts over multiplications and reuse intermediate values.

Algorithm 9 Toeplitz-TC for ntruhps2048677.

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

1: Declare uint16_t \textit{buff}_a[9][288], \textit{buff}_b[9][144], \textit{buff}_c[9][144].
2: \textit{buff}_a[0-8] = TC_9×9^{-1}(a).
3: \( \triangleright\) See Section 4.5 for definition.
4: \textit{buff}_b[0-8] = TC_9×5(b).
5: for \( i = 0, \ldots, 8 \) do
6: \( \textit{buff}_c[i][0-143] = \text{Toeplitz-TC-3-3-2}(\textit{buff}_a[i][0-287], \textit{buff}_b[i][0-143]) \).
7: end for
8: \( c[0-676] = TC_9×5(\textit{buff}_c[0-8][0-143]) \).
We show how to incorporate Rader for radix-

We focus on the approaches without switching the coefficient ring. In other words, all


Input: a 144 × 144 Toeplitz matrix \( M \), and a size-144 vector \( v \).

Output: the size-144 vector \( c = M \cdot v \).

1: Declare uint16_t \( M_1[5][96] \), \( M_2[5][5][3][16] \).
2: Declare uint16_t \( v_1[5][5][16] \), \( c_1[5][5][16] \).
3: \( M_1[0-4][0-95] = TC_{3 \times 3}^{-1} (M[0-143][0-143]) \)
4: for \( i = \{0, \ldots, 4\} \) do
5: \( M_2[i][0-4][0-2][0-15] = (TC_{5 \times 3}^{-1} \circ TC_{5 \times 5}^{-1}) (M_1[i][0-95]) \)
6: end for
7: \( v_1[0-4][0-4][0-15] = (TC_{5 \times 3} \circ TC_{5 \times 3}) (v) \)
8: \( c_1[i][j][0-15] = TC_{5 \times 2}^{3} (M_2[i][j][0-2][0-15]) \cdot TC_{5 \times 2} (b_1[i][j][0-15]) \)
9: \( c[0-143] = (TC_{5 \times 3} \circ TC_{5 \times 3}) (c_1[0-4][0-4][0-15]) \)

Comparisons to [IKPC22]. [IKPC22] implemented the Toeplitz matrix–vector product with \( TC_{(2k-1)\times k} \) as the underlying monomorphisms on Cortex-M4, but they chose the splitting sequence \( 4 \rightarrow 3 \rightarrow 2 \rightarrow 2 \). 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 \( TC_{9 \times 3}^{-1} \), \( TC_{9 \times 5}^{-1} \), and \( TC_{9 \times 5}^{-1} \) require more multiplications, we have a smaller number of schoolbooks, which is the bottleneck of the computation. Compared to [IKPC22], our Cortex-A72 implementation reaches the best performance with size-8 schoolbooks instead of size-16 ones. Also, [IKPC22] used \( TC_{(2k-1)\times (2k-1)}^{-1} \) \( TC_{(2k-1)\times k} \) and \( TC_{(2k-1)\times k} \) to compute while we multiply some constants to the precomputed matrices for easier computation. The modified \( TC_{(2k-1)\times (2k-1)}^{-1} \), \( TC_{(2k-1)\times k} \) and \( TC_{(2k-1)\times k} \) will be shown in the full version.

5.3 NTRU Prime Implementations over \( \mathbb{Z}_{4591} \)

In this section, we discuss our ideas for multiplying polynomials in \( \mathbb{Z}_{4591}[x]/(x^{761} - x - 1) \).

We focus on the approaches without switching the coefficient ring. In other words, all operations are defined in \( \mathbb{Z}_{4591} \). For brevity, we assume \( R = \mathbb{Z}_{4591} \) in this section. The state-of-the-art vectorized “big by big” polynomial multiplication in NTRU Prime [BBCT22] computed the product in \( R[x]/(x^{1024} + 1)(x^{312} - 1) \) with Schönhage and Nussbaumer. This leads to 768 size-8 base multiplications where all of them are negacyclic convolutions. [BBCT22] justified the choice as follows:

... since 4591 − 1 = 2 · 3^3 · 5 · 17, no simple root of unity is available for recursive radix-2 FFT tricks. ... They ([ACC+21]) performed radix-3, radix-5, and radix-17 NTT stages in their NTT (defined in \( R[x]/(x^{1530} - 1) \)). We instead use a radix-2 algorithm that efficiently utilizes the full ymm registers (for vectorization) in the Haswell architecture.

We propose transformations (essentially) quartering and halving the number of coefficients involved in base multiplications for vectorization. Our first transformation computes the result in \( R[x]/(x^{1536} - 1) \). We apply Good–Thomas with \( \omega_3 \in R \) for a more rapid decrease of the sizes of polynomial rings, Schönhage for radix-2 butterflies, and Bruun over \( R[x]/(x^{32} + 1) \). This leads to 384 size-8 base multiplications defined over trinomial moduli. Our second and third transformations compute the result in \( R[x]/(x^{1632} - 1) \).

We show how to incorporate Rader for radix-17 butterflies and Good–Thomas for the coprime factorization 17 · 3 · 2. For computing the size-16 weighted convolutions, we proposed two approaches: (i) We compute size-16 weighted convolutions with the outer product approach. (ii) We split with Cooley–Tukey and Bruun for \( R[x]/(x^{16} ± \omega_{16}^{102}) \), and compute the products with the inner product approach. Since no extensions of coefficient
rings are involved, (i) leads to 102 size-16 weighted convolutions, and (ii) leads to 96 size-8 base multiplication with binomial moduli, 96 size-8 base multiplications with trinomial moduli, and six size-16 base multiplications with binomial moduli.

5.3.1 Good–Thomas, Schönhage’s, and Bruun’s FFT

We first focus on Good–Schönhage-Bruun combining Good–Thomas, Schönhage’s and Bruun’s FFTs in this section. Conceptually, we load and permute the coefficients for Good–Thomas and Schönhage, perform five layers of radix-2 butterflies and one layer of radix-3 butterflies to split the polynomial ring into 96 size-32 polynomial rings. Finally, we split polynomial rings over \(x^{12} + 1\) into 384 size-8 polynomial rings. In detail, the implementation goes as follows:

1. We first transform the input array \(\text{in}[761]\) into a temporary array \(\text{out}[3][32][32]\), where \(\text{out}[i][j][0-31]\) represents the size-32 polynomial in \(\mathbb{Z}_{2^{32} + 1, u - \omega_2^1, w - x^{27}}\).

Concretely, we combine the permutations of Good–Thomas and Schönhage as follows:

\[
\text{out}[i][j][k] = \text{in}[(16(64i + 33j) \mod 96) + k] \quad \text{if} \quad (16(64i + 33j) \mod 96) + k < 761 \text{ and zero otherwise.}
\]

This step is the foundation of the implicit permutations [ACC+21].

2. For input small, we start with the 8-bit form of the polynomial. Since coefficients are in \{\pm 1, 0\}, we first perform five layers of radix-2 butterflies without any modular reductions. The initial three layers of radix-2 butterflies are combined with the implicit permutations. For the last two layers of radix-2 butterflies, we use \text{ext} if the root is not a power of \(x^{16}\). For the last layer of radix-2 butterflies, we merge the sign-extension and add-sub pairs into the sequence \text{sadd1, sadd2, ssub1, ssub2}.

We then apply one layer of radix-3 butterflies based on the improvements of [DV78, Equation 8] and [HY22, Equation 5]. We compute the radix-3 NTT \((\tilde{v}_0, \tilde{v}_1, \tilde{v}_2)\) of size-32 polynomials \((v_1, v_2, v_3)\) as:

\[
\begin{aligned}
\tilde{v}_0 &= v_0 + v_1 + v_2, \\
\tilde{v}_1 &= (v_0 - v_2) + \omega_4(v_1 - v_2), \\
\tilde{v}_2 &= (v_0 - v_1) - \omega_4(v_1 - v_2).
\end{aligned}
\]

We find it comparable to [AHY22, Algorithm 4] on Cortex-A72 but potentially faster on platforms with fast single-width multiplication instructions\(^7\).

3. For the input big, we use the 16-bit form and perform one layer of radix-3 butterflies followed by five layers of radix-2 butterflies. This implies only 1536 coefficients are involved in radix-3 butterflies instead of 3072 as for the input small. We first apply one layer of radix-3 butterflies and two layers of radix-2 butterflies followed by one layer of Barrett reductions while permuting implicitly for Good–Thomas and Schönhage. Then, we perform three layers of radix-2 butterflies and another layer of Barrett reductions.

4. After splitting the polynomial ring into \(\mathbb{Z}_{2^{32} + 1, u - \omega_2^1, w - x^{27}}\), we apply Bruun’s butterflies. Specifically, the polynomial modulus \(x^{32} + 1\) splits into:

\[
x^{32} + 1 = (x^{16} + 1229x^2 + 1)(x^{16} - 1229x^2 + 1) = (x^8 + 58x^4 + 1)(x^8 - 58x^4 + 1)(x^8 + 2116x^4 + 1)(x^8 - 2116x^4 + 1)
\]

\(^7\)Based on https://dougallj.github.io/applecpu/firestorm-simd.html, \text{smulh} has the same throughput as \text{sqrdmulh} on Apple M1. This implies short sequences of instructions are more favorable. The Armv8-A version of [AHY22, Algorithm 4] has longer sequence than that of [DV78, HY22].
We use Bruun_{1229,1} followed by Bruun_{58,1} and Bruun_{2116,1} to split the polynomials. Then, we perform 384 size-8 base multiplications and compute the inverses of Bruun’s butterflies.

Finally, we invert all the transformations.

**Comparisons to existing uses of symbolic roots of unity.** We compare our transformation with the “big by big” polynomial multiplication for \( R[x]/\langle x^{761} - x - 1 \rangle \) by [BBCT22]. [BBCT22] computed the result in \( R[x]/\langle (x^{1024} + 1)(x^{512} - 1) \rangle \). We divide their transformation into the following steps: (a) Introduce the equivalence \( x^{32} \sim y \) and switch to \( x^{64} \sim 1 \) for Schönhage. (b) Compute FFT over \( (y^{32} + 1)(y^{16} - 1) \) with \( x^2 \) as the principal 64-th root of unity. (c) Introduce the equivalence \( x^{8} \sim z \) and switch to \( x^{16} \sim 1 \) for Nussbaumer. (d) Compute FFT over \( x^{16} - 1 \) with \( z \) as the principal 16-th root of unity. (e) Compute base multiplications defined over \( z^{\frac{512}{16}} + 1 = z^{8} + 1 \).

Compared to (a), we exploit the existing \( \omega_3 \in R \) and introduce \( x^{16} \sim y \) then \( x^{32} \sim 1 \) instead for Schönhage. This implies faster shrinking of polynomial rings after Schönhage. Compared to (c), we simply split \( x^{32} + 1 \) into four polynomials of the form \( x^{8} + \alpha x^4 + 1 \), removing the need to extend coefficient rings using Nussbaumer.

### 5.3.2 Good–Thomas and Rader’s

This section describes our transformation defined on \( R[x]/\langle x^{1632} - 1 \rangle \). Let \( \alpha = 1229, (e_0, e_1, e_2) = (18, 34, 51) \) so \( \alpha^2 = 2 \in R \) and \( \forall a \in \mathbb{Z}_{102}, a \equiv e_0(a \text{ mod } 17) + e_1(a \text{ mod } 3) + e_2(a \text{ mod } 2) \) (mod 102). We rewrite each size-1632 polynomial \( a(x) \) as the size-102 polynomial \( a'(x^{16}) \), introduce \( x^{16} \sim y \), and compute FFTs in \( y \). Since 4591 = 102 \cdot 45, there is a principal 102-th root of unity \( \omega_{102} \in R \). For evaluating \( y \) at powers of \( \omega_{102} \), we perform a 3-dimensional Good–Thomas based on the coprime factorization 102 = 17 \cdot 3 \cdot 2 as shown in Algorithm 11.

**Algorithm 11** Good–Thomas, in practice merged with Algorithm 12.

**Inputs:** \( \text{arc}_{1632} \).

**Outputs:** \( \text{poly}_\text{NTT}_{17}[3][2][16] \).

1. for \( i = 0, \ldots, 1631 \) do
2. \hspace{1em} let \( \bar{t} = i / 16 \).
3. \hspace{1em} \text{poly}_\text{NTT}[(t \text{ mod } 17) [t \text{ mod } 3] [t \text{ mod } 2] [i \text{ mod } 16] = \text{arc}[i] \).
4. end

This gives us \( R'[y, u, w, z]/\langle x^{1632} - 1 \rangle \), where \( R' = R[x]/\langle x^{16} - y \rangle \). For the FFT defined on \( z \), we use Rader’s FFT to convert the size-17 cyclic FFT into a size-16 cyclic convolution. For the FFTs defined on \( u \) and \( w \), we apply the straightforward radix-(3, 2) butterfly [AHY22, Section 2.6]. Algorithm 12 illustrates the FFTs in \( y \sim x^{16} \).

After substituting \((y, u, w, z) \) with \((x^{16}, \omega_2^y, \omega_3^y, \omega_1^y) \), we have \( \prod_{i_0, i_1, i_2} R[x] = \prod_{i_0, i_1, i_2} R[x] / (x^{16} - \omega_{102}^{i_0} \omega_{102}^{i_1} \omega_{102}^{i_2}) \) by choosing \((\omega_2, \omega_3, \omega_1) = (\omega_{102}^{i_2}, \omega_{102}^{i_1}, \omega_{102}^{i_0}) \). We implement the following two approaches:

- **Good–Rader-out**. We compute 102 size-16 weighted convolutions with vector-by-scalar multiplications implementing outer products (cf. Algorithm 6). This is the fastest and most penetrable implementation for the size-16 weighted convolutions.

- **Good–Rader-Bruun**. We split for \( i_2 = 0, \ldots, 15 \) and compute size-16 weighted convolutions for \( i_2 = 16 \). The remaining paragraphs are dedicated to this approach.
We present the performance numbers in this section. We focus on polynomial multiplications, leaving the fast constant-time GCD [BY19] as future work.

Our transformations however admit further optimizations via truncation [vdH04], meaning friendly factorization for polynomial rings. We believe it will be a useful alternative when there are no vector-by-scalar multiplications.

Albeit it is a bit complicated, it gives several algebraic insights on vectorization-friendly factorization for polynomial rings. We believe it will be a useful alternative when there are no vector-by-scalar multiplications.

Finally, we invert all the computations after computing the base multiplications.

\[ x^{16} - \omega^{e_0 i_0 + e_1 i_1 + e_2 i_2} = x^{16} - (-1)^2 \omega^{128 (e_0 i_0 + e_1 i_1)} = x^{16} - \omega^{e_2 (i_0 + i_1)} \]

(a) Write \( x^{16} - \omega^{e_0 i_0 + e_1 i_1 + e_2 i_2} = x^{16} - (-1)^2 \omega^{128 (e_0 i_0 + e_1 i_1)} = x^{16} - \omega^{e_2 (i_0 + i_1)} \) and compute the following.

1. Transform \( x^{16} - \omega^{128 (e_0 i_0 + e_1 i_1)} \) with radix-2 Cooley–Tukey butterflies. The remaining problems are size-8 weighted convolutions.

2. Transform \( x^{16} - \omega^{128 (e_0 i_0 + e_1 i_1)} \) and \( x^{16} - \omega^{128 (e_0 i_0 + e_1 i_1)} \) with radix-2 Bruun’s butterflies. The remaining problems are size-8 base multiplications defined on trinomials.

3. Compute size-8 base multiplications. The size-8 weighted convolutions are obvious. For the trinomial cases, we extend the idea of [CHK+21, Algorithm 17] by altering between the computations of \( abr^{-1} \in R[x] \) and reductions modulo \( x^8 + ax^4 + b \).

Finally, we invert all the computations after computing the base multiplications.

**Comparisons to [ACC+21].** We compare to the size-1530 NTT by [ACC+21]. They computed the result of \( R[x]/\langle x^{1530} - 1 \rangle \) with Rader’s radix-17 butterflies and two layers of size-3 butterflies, resulting size-10 weighted convolutions at the end. This means that the radix-17 and radix-3 butterflies are defined on size-10 polynomials. If we store every size-10 polynomial in a pair of SIMD registers (each holding 8 halfwords), then \( \frac{5}{8} \) of the computations are discarded at the end. We should store each size-15 polynomial in a pair of SIMD registers, and compute radix-17, radix-3, and radix-2 butterflies, and size-15 weighted convolutions. The radix-17, radix-3, and radix2 butterflies are then the same as our computation. This implies \( \frac{1}{15} \) of the computations will be discarded at the end. Our transformations however admit further optimizations via truncation [vdH04], meaning working over \( \mathbb{Z}_{4591}[x]/\langle \frac{x^{1632} - 1}{16} \rangle \) instead of \( \mathbb{Z}_{4591}[x]/\langle x^{1632} - 1 \rangle \). We left the idea as a future work.

6 Results

We present the performance numbers in this section. We focus on polynomial multiplications, leaving the fast constant-time GCD [BY19] 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 [KSSW] without any optimizations due to the lack of corresponding cryptographic units. For the randombytes, [BHK’22] 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 8 summarizes the performance of vectorized polynomial multiplications. For ntruhps2048, all of our implementations outperform the state-of-the-art Toom–Cook [NG21]. Our Toeplitz–TC, Toom–Cook, and Good–Thomas are 2.18×, 1.56×, and 1.38× faster than [NG21]. Comparing Toeplitz–TC and Toom–Cook based on the same splitting sequence, the result is consistent to [IKPC22]. But the most significant reason is the use of vector-by-scalar multiplications. This finding is new. Furthermore, we clarify that Toeplitz matrix–vector product is not privy to Toom–Cook. Comparing our Toom–Cook and Good–Thomas, we find that Toom–Cook is more favorable for vectorization. This contradicts to the AVX2 optimized NTTs by [CHK’21]. We left the investigation as future work.

Table 8: Overview of polymuls.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>ntruhps2048</th>
<th>ntrulpr761/sntrup761</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NG21]</td>
<td>58 286</td>
<td>[Haa21] 242 585</td>
</tr>
<tr>
<td>Toeplitz–TC</td>
<td>26 784</td>
<td>Good–Rader–outer 36 266</td>
</tr>
<tr>
<td>Toom–Cook</td>
<td>37 318</td>
<td>Good–Rader–Bruun 39 788</td>
</tr>
<tr>
<td>Good–Thomas</td>
<td>42 355</td>
<td>Good–Thomas 47 696</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Good–Schönhage–Bruun 50 398</td>
</tr>
</tbody>
</table>

For NTRU Prime, our Good–Rader–outer performs the best. It is followed by Good–Rader–Bruun, Good–Thomas, and Good–Schönhage–Bruun. Notice that Good–Rader–outer and Good–Rader–Rader requires no extensions or changes of coefficient rings. Good–Rader–outer relies on implementing matrix–vector products with vector-by-scalar multiplications. If there are no vector-by-scalar multiplications, Good–Rader–Bruun demonstrates further factorizations. The closest instances in the literature regarding vectorization are the Good–Thomas and Schönhage–Nussbaumer by [BBCT22], and Good–Thomas by [Haa21], [BBCT22]’s, [Haa21], and our Good–Thomas compute “big by small” polynomial multiplications. We outperform [Haa21] Good–Thomas by a factor of 6.7× since they implemented the base multiplications with scalar code using the C % operator. On the other hand, [BBCT22]’s Schönhage–Nussbaumer and our Good–Schönhage–Bruun compute “big by big” polynomial multiplications. Regarding the impact of switching “big by small” to “big by big”, [BBCT22]’s Schönhage–Nussbaumer takes \( \frac{25113}{147.79}\approx 147.79\% \) cycles of their own Good–Thomas [BBCT22, Section 3.4.2] while our Good–Schönhage–Bruun takes only \( \frac{50398}{5757}\approx 105.67\% \) cycles of our own Good–Thomas. Essentially, this demonstrates the benefit of vectorization-friendly Good–Thomas and Bruun over truncated [vdH04] Schönhage and Nussbaumer.
We also provide the detailed cycle counts of the polynomial multiplications. Table 14 details the numbers of Good-Thomas implementing the “big by small” polynomial multiplications in \texttt{ntruhps2048677} and \texttt{sntrup761/ntrulpr761}. Table 15 details the numbers of Toom-Cook and Toom-Cook implementing the “big by big” polynomial multiplications in \texttt{ntruhps2048677}. For the “big by big” polynomial multiplications in \texttt{sntrup761/ntrulpr761}, Table 13 details the numbers of Good-Rader-outer and Good-Rader-Bruun and Table 16 details the numbers of Good-Schönhage-Bruun.

\textbf{Compatibility with arithmetic masking.} We briefly go through the compatibility of our polynomial multiplications with arithmetic masking. Arithmetic masking is a mean of protection from side-channel attacks. For a secret value \( a \), we write it as \( a = a' \Diamond a'' \) where \( a' \) and \( a'' \) are called (arithmetic) shares. In higher-order masking, we write it as \( a = a' \Diamond a'' \Diamond \cdots \). We then define all the follow-up computations on the shares. If \( \Diamond \) is the addition or multiplication in the ring where the shares belong to, any correct ring monomorphisms can be applied to the follow-up computations. For simplicity, we assume \( \Diamond = + \). For correctness, there are constraints on the chosen ring monomorphisms. Suppose the goal is to compute \( ab \in \mathbb{Z}_q[x]/\langle g \rangle \) via the secret-sharing \( ab = (a' + a'')b \) for some monic polynomial \( g \). There are two obvious ways: we compute \( a'b \) and \( a''b \) in either \( \mathbb{Z}_q[x]/\langle g \rangle \) or \( \mathbb{Z}[x]/\langle g \rangle \). The former says that any ring monomorphisms from \( \mathbb{Z}_q[x]/\langle g \rangle \) can be applied while the latter requires one to carefully bound the maximum values of \( a'b \) and \( a''b \) in \( \mathbb{Z} \). In the context of arithmetic sharing, we can only assume \( a', a'' \in \mathbb{Z}_q[x]/\langle g \rangle \) even if we know beforehand that \( a' + a'' = a \) has coefficients in \( \{0, 1, 2\} \) for NTRU and \( \{-1, 0, 1\} \) for NTRU Prime. This implies a large overhead for our Good-Thomas for NTRU and NTRU Prime and any other approaches working over \( \mathbb{Z} \) in Tables 1, 2, 4, and 5. On the other hand, approaches Toeplitz-TC, Toom-Cook, Good-Rader-outer, Good-Rader-Bruun, and Good-Schönhage-Bruun require no modifications and suffer no performance penalty (except for the growth of the number of shares, but any approaches suffer from this) since the coefficient rings are unchanged.

\textbf{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.

\textbf{Inversions.} For \texttt{ntruhps2048677}, we need one inversion in \( \mathbb{Z}_{2048}[x]/\langle x^{677} - 1 \rangle \) and one inversion in \( \mathbb{Z}_3[x]/\langle x^{677} + 1 \rangle \). 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 \( 20 \times \) speedup, leading to \( 10.27 \times \) 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 \( \mathbb{Z}_3[x]/\langle x^{677} + 1 \rangle \), we use bitsliced implementation and obtain a \( 8.6 \times \) speedup. For \texttt{sntrup761}, we need one inversion over \( \mathbb{Z}_{4591} \) and one inversion over \( \mathbb{Z}_3 \). We bitslice the inversion over \( \mathbb{Z}_3 \), and identify and vectorize the hottest loop in the inversion over \( \mathbb{Z}_{4591} \).

\textbf{Sorting network, encoding, and decoding.} We translate AVX2-optimized sorting network, encoding, and decoding into Neon. Notice that inversions over \( \mathbb{Z}_2, \mathbb{Z}_3, \) and \( \mathbb{Z}_{4591}, \)
sorting networks, encoding, and decoding are implemented in a generic sense. With fairly little effort, they can be used for other parameter sets.

Table 17 summarizes the performance of inversions, encoding, and decoding.

**Performance of ntruhps2048677 and sntrup761/ntrulpr761.** Table 9 summarizes our ntruhps2048677 and Table 10 summarizes our sntrup761/ntrulpr761. We compare our fastest ntruhps2048677 implementation to the existing NTRU on Cortex-A72 [NG21]. Our key generation is $7.67 \times$ faster. The main contribution is our optimized inversions in $\mathbb{Z}_2[x]/(x^{677} - 1)$ and $\mathbb{Z}_3[x]/\left\langle x^{677} - 1 \right\rangle$, followed by polynomial multiplications in $\mathbb{Z}_{2048}[x]/(x^{677} - 1)$ (for lifting) and sorting network. Our encapsulation is $2.48 \times$ faster. The main contribution is the sorting network followed by polynomial multiplications. Finally, our decapsulation is $1.77 \times$ faster. The improvement entirely comes from the improved polynomial multiplications. Notice that Good–Thomas can only be applied to “big by small” polynomial multiplications. For concrete evaluations, we pair Good–Thomas with Toeplitz–TC and Toom–Cook, and provide the resulting numbers. For ntrulpr761 we outperform [Haa21] by a factor of 6.7. [Haa21] didn’t implement sntrup761.

Finally, Table 19 details the numbers of ntruhps2048677 with Toeplitz–TC, and Table 18 details the numbers of sntrup761/ntrulpr761 with Good–Rader–outer. Notice that only performance-critical subroutines are shown.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Key generation</th>
<th>Encapsulation</th>
<th>Decapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>8 245 039</td>
<td>227 980</td>
<td>331 274</td>
</tr>
<tr>
<td>[NG21]</td>
<td>7 686 272</td>
<td>196 526</td>
<td>212 265</td>
</tr>
<tr>
<td>Toeplitz–TC</td>
<td>1 002 187</td>
<td>79 213</td>
<td>120 208</td>
</tr>
<tr>
<td>Toom–Cook</td>
<td>1 127 089</td>
<td>88 037</td>
<td>146 422</td>
</tr>
<tr>
<td>Good–Thomas (with Toeplitz–TC)</td>
<td>1 061 870</td>
<td>96 814</td>
<td>152 059</td>
</tr>
<tr>
<td>Good–Thomas (with Toom–Cook)</td>
<td>1 178 983</td>
<td>97 182</td>
<td>165 381</td>
</tr>
</tbody>
</table>

Table 9: Overall cycles of ntruhps2048677.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Key generation</th>
<th>Encapsulation</th>
<th>Decapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>273 598 470</td>
<td>297 500 35</td>
<td>89 968 342</td>
</tr>
<tr>
<td>Good–Rader–outer</td>
<td>6 297 001</td>
<td>144 681</td>
<td>145 777</td>
</tr>
<tr>
<td>Good–Rader–Brunn</td>
<td>6 333 403</td>
<td>147 977</td>
<td>158 233</td>
</tr>
<tr>
<td>Good–Thomas</td>
<td>6 340 758</td>
<td>153 465</td>
<td>182 271</td>
</tr>
<tr>
<td>Good–Schönhage–Brunn</td>
<td>6 345 787</td>
<td>163 305</td>
<td>193 626</td>
</tr>
</tbody>
</table>

Table 10: Overall cycles of sntrup761/ntrulpr761.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Key generation</th>
<th>Encapsulation</th>
<th>Decapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
<td>29 853 635</td>
<td>59 572 637</td>
<td>89 185 030</td>
</tr>
<tr>
<td>[Haa21]</td>
<td>7 75 472</td>
<td>1 150 294</td>
<td>1 417 394</td>
</tr>
<tr>
<td>Good–Rader–outer</td>
<td>257 382</td>
<td>400 223</td>
<td>435 789</td>
</tr>
<tr>
<td>Good–Rader–Brunn</td>
<td>260 606</td>
<td>412 629</td>
<td>461 250</td>
</tr>
<tr>
<td>Good–Thomas</td>
<td>269 590</td>
<td>422 102</td>
<td>471 014</td>
</tr>
<tr>
<td>Good–Schönhage–Brunn</td>
<td>272 738</td>
<td>436 965</td>
<td>499 559</td>
</tr>
</tbody>
</table>
7 Discussions

We briefly discuss possible future work. Platform-wise, we expect significant performance improvement for ntrulpr761/sntrup761 on Haswell with AVX2. For NTRU, we would like to explore the performance of Toom–Cook with Toom–5 on Skylake and see if it is faster than [CHK+21]’s mixed–radix NTT. Parameter-wise, we briefly draft the following approaches.

We first go through the NTRU parameter sets since they are more simple. For ntruhrss701, we choose a transformation with the same series of subproblem sizes as ntruhps2048677 and replace TC–3 with 3-way Karatsuba due to the increase of coefficient ring. For ntruhps2048509, we suggest TC–4 and Karatsuba. For ntruhps4096821, we borrow the transformation size 1728 from BBC+20, Hwa22]’s NTTs for ntrulpr1277/sntrup1277, and suggest TC–5 and Karatsuba. For ntruhrss1373, we suggest TC–3 and Karatsuba. Notice that all the transformations for NTRU can be turned into transformations for Toeplitz matrix–vector products. Table 11 summarizes the choices of transformations.

Table 11: Our suggestions for NTRU parameter sets. Strategies of ntruhps2048677 are implemented in this paper. K = Karatsuba.

<table>
<thead>
<tr>
<th>Target ring</th>
<th>ntruhps2048509</th>
<th>ntruhps2048677</th>
<th>ntruhrss701</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring/Toeplitz</td>
<td>Z_{2048}[x]/(x^{677} - 1)</td>
<td>Z_{2048}[x]/(x^{509} - 1)</td>
<td>Z_{16384}[x]/(x^{701} - 1)</td>
</tr>
<tr>
<td>TC–5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TC–4</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TC–3</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>3-way K</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>K</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Next, we go through our suggestions for NTRU Prime parameter sets. We propose four strategies for ntrulpr857/sntrup857. The first two strategies operate over Z_{5167}[x]/(x^{1968} – 1) and are similar to Good–Rader–outer and Good–Rader–Bruun with Bruun’s FFT excluded. We first apply a 3-dimensional Good–Thomas based on the coprime factorization 1968 = 41 · 3 · 16. We then apply Rader’s FFT for the size-41 NTT, and Cooley–Tukey FFT of sizes 3 and 2. This leaves us with the product ring \( \prod Z_{5167}[x]/(x^6 \pm \omega_2^{346}) \), and we compute the products with vector-by-scalar or vector-by-vector multiplications according to the target architecture. The remaining two strategies are similar to Good–Schönhage–Bruun by operating over Z_{5167}[x]/(x^{1792} – 1). We first
pull out the factor 7 with vectorization-friendly Good–Thomas, and apply Schönhage for the power-of-two dimension. The remaining problem is the multiplication in the product ring $\prod_{i} \mathbb{Z}_{5167}[x]/\langle x^{16} + 1 \rangle$. We apply vector-by-scalar multiplications or Bruun’s with vector-by-vector multiplications. For ntrulpr1013/sntrup1013, we first choose the composition of vectorization-friendly 3-dimensional Good–Thomas and Rader’s FFT for $\mathbb{Z}_{7177}[x]/\langle x^{2496} - 1 \rangle$, and truncate the computation to $\mathbb{Z}_{7177}[x]/\langle x^{16} - \omega_{312}^{1} \rangle$. The remaining problem is the multiplication in the product ring $\prod_{i} \mathbb{Z}_{7177}[x]/\langle x^{16} - \omega_{i}^{312} \rangle$. Again, we choose between vector-by-scalar and vector-by-vector multiplications according to the architecture. Finally, for ntrulpr1277/sntrup1277, we truncate the 3-dimensional Good–Thomas and Rader from $\mathbb{Z}_{7879}[x]/\langle x^{4992} - 1 \rangle$ to $\mathbb{Z}_{7879}[x]/\langle (x^{2496} + 1) (x^{64} - 1) \rangle$, and choose between Toeplitz matrix–vector products with TC-4 and Bruun’s FFT. Table 12 summarizes the choices of transformations.

<table>
<thead>
<tr>
<th></th>
<th>ntrulpr857/sntrup857</th>
<th>ntrulpr1013/sntrup1013</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target ring</td>
<td>$\mathbb{Z}_{5167}[x]/(x^{1968} - 1)$</td>
<td>$\mathbb{Z}_{7177}[x]/(x^{2496} - 1)$</td>
</tr>
<tr>
<td>Ring</td>
<td>$\mathbb{Z}_{5167}[x]/(x^{1792} - 1)$</td>
<td>$\mathbb{Z}_{7177}[x]/(x^{312} + 1)$</td>
</tr>
<tr>
<td>Idea</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Good–Thomas</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Truncation</td>
<td>- - - - ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Schönhage</td>
<td>- - ✓ ✓ - - -</td>
<td></td>
</tr>
<tr>
<td>Rader</td>
<td>✓ ✓ - - ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Bruun</td>
<td>✓ ✓ - - ✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Vector-by-scalar</td>
<td>✓ - ✓ - ✓ - -</td>
<td></td>
</tr>
<tr>
<td>Vector-by-vector</td>
<td>- ✓ - ✓ - - ✓</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ntrulpr1277/sntrup1277</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target ring</td>
<td>$\mathbb{Z}_{7879}[x]/(x^{1277} - 1)$</td>
</tr>
<tr>
<td>Ring</td>
<td>$\mathbb{Z}_{7879}[x]/(x^{2496} - 1) (x^{64} - 1)$</td>
</tr>
<tr>
<td>Idea</td>
<td>✓ ✓ - ✓ ✓</td>
</tr>
<tr>
<td>Good–Thomas</td>
<td>✓ ✓ - ✓ ✓</td>
</tr>
<tr>
<td>Rader</td>
<td>✓ ✓ - ✓ ✓</td>
</tr>
<tr>
<td>Bruun</td>
<td>- ✓ - ✓ ✓</td>
</tr>
<tr>
<td>Toeplitz(TC-4)</td>
<td>✓ - ✓</td>
</tr>
<tr>
<td>K</td>
<td>- ✓ - ✓</td>
</tr>
<tr>
<td>Vector-by-scalar</td>
<td>✓ - ✓</td>
</tr>
<tr>
<td>Vector-by-vector</td>
<td>- ✓ - ✓</td>
</tr>
</tbody>
</table>
A Proof for the Toeplitz Transformation

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

\[
(\text{Toeplitz}_{k \times k}(-))(a) = \text{rev}_{k \times k} \circ f_k^* \circ (f_k(a), -)^* \circ (f^{-1})^* \circ \text{id}_{(2k-1) \to n}.
\]

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

\[
(\text{rev}_{k \times k} \circ \text{Toeplitz}_{k \times k}(z))(a) = \sum_{j \in [k]} (z_{i+j}a_j)_{(i,j) \in [k] \times [k]} = \sum_{j \in [k]} (0_{k,j} \ a_jf_k \ 0_{k,k-j-1}) (z_k)_{(k,0) \in [2k-1] \times [1]} = \text{Toeplitz}_{k \times (2k-1)} (0_{1,k-1}, a_0, \ldots, a_{k-1}, 0_{1,k-1}) (z_k)_{(k,0) \in [2k-1] \times [1]} = (a, -)^*(z).
\]

Applying \( \text{rev}_{k \times k} \) from the left finishes the proof (cf. [Win80, Theorem 6]). \( \square \)

B Examples of Toeplitz Transformations

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

\[
\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} z_1 & z_2 \\ z_0 & z_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \end{pmatrix} = \begin{pmatrix} 1 & \omega_3 \\ \omega_3 & 1 \end{pmatrix} \begin{pmatrix} a_0 + a_1 & 0 \\ 0 & a_0 + \omega_3a_1 \end{pmatrix} \begin{pmatrix} z_0 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 & \omega_4 \omega_2^2 \\ \omega_4 \omega_2^2 & 1 \end{pmatrix} \begin{pmatrix} a_0 + a_1 & 0 \\ 0 & a_0 + \omega_4a_1 \end{pmatrix} \begin{pmatrix} z_0 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 & \omega_4 \omega_2^2 \omega_4^2 \\ \omega_4 \omega_2^2 \omega_4^2 & 1 \end{pmatrix} \begin{pmatrix} a_0 + a_1 & 0 \\ 0 & a_0 + \omega_4a_1 \end{pmatrix} \begin{pmatrix} z_0 \\ z_2 \end{pmatrix}
\]

where \( F_k^{-1} = (F_k^{-1})^T \) is the inverse of the cyclic size-\( k \) FFT.
C Matrix multiplications

1: \textbf{for} $i_0 = 0, \ldots, n_0 - 1$ \textbf{do}
2: \hspace{1em} \textbf{for} $i_1 = 0, \ldots, n_1 - 1$ \textbf{do}
3: \hspace{2em} \textbf{for} $i_2 = 0, \ldots, n_2 - 1$ \textbf{do}
4: \hspace{3em} $c[i_0][i_1] = c[i_0][i_1] + a[i_0][i_2]b[i_2][i_1]$
5: \hspace{2em} \textbf{end for}
6: \hspace{1em} \textbf{end for}
7: \hspace{1em} $\triangleright$ Inner product of the vectors $a[i_0][\cdot]$ and $b[\cdot][i_1]$.
8: \textbf{end for}

Algorithm 14 Outer-product-based matrix–matrix multiplication.
1: \textbf{for} $i_2 = 0, \ldots, n_2 - 1$ \textbf{do}
2: \hspace{1em} \textbf{for} $i_0 = 0, \ldots, n_0 - 1$ \textbf{do}
3: \hspace{2em} \textbf{for} $i_1 = 0, \ldots, n_1 - 1$ \textbf{do}
4: \hspace{3em} $c[i_0][i_1] = c[i_0][i_1] + a[i_0][i_2]b[i_2][i_1]$
5: \hspace{2em} \textbf{end for}
6: \hspace{1em} \textbf{end for}
7: \hspace{1em} $\triangleright$ Outer product of the vectors $a[\cdot][i_2]$ and $b[i_2][\cdot]$.
8: \textbf{end for}

1: \textbf{for} $i_0 = 0, \ldots, n_0 - 1$ \textbf{do}
2: \hspace{1em} \textbf{for} $i_2 = 0, \ldots, n_2 - 1$ \textbf{do}
3: \hspace{2em} $c[i_0] = c[i_0] + a[i_0][i_2]b[i_2]$
4: \hspace{2em} \textbf{end for}
5: \hspace{1em} $\triangleright$ Inner product of the vectors $a[i_0][\cdot]$ and $b[\cdot]$.
6: \textbf{end for}

1: \textbf{for} $i_2 = 0, \ldots, n_2 - 1$ \textbf{do}
2: \hspace{1em} \textbf{for} $i_0 = 0, \ldots, n_0 - 1$ \textbf{do}
3: \hspace{2em} $c[i_0] = c[i_0] + a[i_0][i_2]b[i_2]$
4: \hspace{2em} \textbf{end for}
5: \hspace{1em} $\triangleright$ Outer product of the vectors $a[\cdot][i_2]$ and $b[i_2]$.
6: \textbf{end for}
D Detailed Numbers of Polynomial Multiplications

Table 13: Detailed cycle counts of Good–Rader–outer and Good–Rader–Bruun, excluding reductions to $\mathbb{Z}_{4591}[x]/(x^{761} - x - 1)$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
<th>Cycles</th>
<th>Total</th>
<th>Operation</th>
<th>Count</th>
<th>Cycles</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good-Rader-17</td>
<td>24</td>
<td>395</td>
<td>9480</td>
<td>Good-Rader-17</td>
<td>24</td>
<td>407</td>
<td>9768</td>
</tr>
<tr>
<td>Radix-(3, 2)</td>
<td>2</td>
<td>2337</td>
<td>4674</td>
<td>Radix-(3, 2)</td>
<td>2</td>
<td>2330</td>
<td>4678</td>
</tr>
<tr>
<td>Weighted-16x16</td>
<td>1</td>
<td>9093</td>
<td>9093</td>
<td>CT</td>
<td>2</td>
<td>570</td>
<td>1140</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bruun</td>
<td>2</td>
<td>838</td>
<td>1676</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted-8x8</td>
<td>12</td>
<td>214</td>
<td>2928</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Trinomial-8x8</td>
<td>12</td>
<td>328</td>
<td>3936</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CT^1</td>
<td>1</td>
<td>592</td>
<td>592</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bruun^1</td>
<td>1</td>
<td>989</td>
<td>989</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted-16x16</td>
<td>1</td>
<td>1019</td>
<td>1019</td>
</tr>
<tr>
<td>Radix-(3, 2)^{-1}</td>
<td>1</td>
<td>2337</td>
<td>2337</td>
<td>Radix-(3, 2)^{-1}</td>
<td>1</td>
<td>2341</td>
<td>2341</td>
</tr>
<tr>
<td>Good-Rader-17^{-1}</td>
<td>12</td>
<td>532</td>
<td>6360</td>
<td>Good-Rader-17^{-1}</td>
<td>12</td>
<td>543</td>
<td>6516</td>
</tr>
</tbody>
</table>

Table 14: Detailed numbers of Good–Thomas for $\mathbb{Z}_q[x]/(x^{1536} - 1)$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>ntruhps2048677</th>
<th>sntrup761/ntrulpr761</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymul</td>
<td>42355</td>
<td>47696</td>
</tr>
<tr>
<td>NTT (\times 2)</td>
<td>5502</td>
<td></td>
</tr>
<tr>
<td>basemul</td>
<td>21444</td>
<td></td>
</tr>
<tr>
<td>iNTT</td>
<td>8884</td>
<td></td>
</tr>
<tr>
<td>Rq_reduce</td>
<td>547</td>
<td>5102</td>
</tr>
</tbody>
</table>

Table 15: Detailed cycle counts of Toeplitz–TC and Toom–Cook, splitting 5 → 3 → 3 → 2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
<th>Cycles</th>
<th>Total cycles</th>
<th>Operation</th>
<th>Count</th>
<th>Cycles</th>
<th>Total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymul</td>
<td>-</td>
<td>-</td>
<td>26784</td>
<td>polymul</td>
<td>-</td>
<td>-</td>
<td>37278</td>
</tr>
<tr>
<td>tmvp_ittc5</td>
<td>1</td>
<td>2387</td>
<td>2387</td>
<td>tc5</td>
<td>2</td>
<td>586</td>
<td>1172</td>
</tr>
<tr>
<td>tmvp_tc5</td>
<td>1</td>
<td>565</td>
<td>565</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tmvp_ittc3</td>
<td>9</td>
<td>199</td>
<td>1791</td>
<td>tc33</td>
<td>18</td>
<td>147</td>
<td>2646</td>
</tr>
<tr>
<td>tmvp_ittc32</td>
<td>9</td>
<td>559</td>
<td>559</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tmvp_tc33</td>
<td>9</td>
<td>148</td>
<td>1332</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tmvp_8x8</td>
<td>9</td>
<td>1389</td>
<td>12501</td>
<td>k2</td>
<td>2</td>
<td>711</td>
<td>1422</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>schoolbook_8x8</td>
<td>1</td>
<td>19122</td>
<td>19122</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k2</td>
<td>1</td>
<td>2777</td>
<td>2777</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>schoolbook_16x16</td>
<td>1</td>
<td>511</td>
<td>511</td>
</tr>
<tr>
<td>tmvp_tc33</td>
<td>9</td>
<td>209</td>
<td>1881</td>
<td>itc33</td>
<td>9</td>
<td>467</td>
<td>4203</td>
</tr>
<tr>
<td>tmvp_tc5</td>
<td>1</td>
<td>821</td>
<td>821</td>
<td>itc5</td>
<td>1</td>
<td>3903</td>
<td>3903</td>
</tr>
</tbody>
</table>
Table 16: Detailed Good–Schönhage–Bruun cycle counts including reducing to \( \mathbb{Z}_{4591}[x]/(x^{761} - x - 1) \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
<th>Cycles</th>
<th>Total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymul</td>
<td>-</td>
<td>-</td>
<td>50398</td>
</tr>
<tr>
<td>Good-Schönhage-3-2x2</td>
<td>1</td>
<td>1708</td>
<td>1708</td>
</tr>
<tr>
<td>Schön sage-3x2</td>
<td>3</td>
<td>1246</td>
<td>3738</td>
</tr>
<tr>
<td>Good-Schönhage-5x2</td>
<td>1</td>
<td>1527</td>
<td>1527</td>
</tr>
<tr>
<td>Radix-3</td>
<td>1</td>
<td>2084</td>
<td>2084</td>
</tr>
<tr>
<td>Bruun</td>
<td>24</td>
<td>291</td>
<td>6984</td>
</tr>
<tr>
<td>Trinomial-8x8</td>
<td>12</td>
<td>1115</td>
<td>13380</td>
</tr>
<tr>
<td>Bruun inverse</td>
<td>12</td>
<td>409</td>
<td>4908</td>
</tr>
<tr>
<td>Schön sage-2x4 inverse</td>
<td>3</td>
<td>1304</td>
<td>3912</td>
</tr>
<tr>
<td>Good-Schönhage-2-3 inverse</td>
<td>1</td>
<td>7653</td>
<td>7653</td>
</tr>
</tbody>
</table>

Table 17: Performance of inversions, encoding, and decoding in NTRU and NTRU Prime.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ref</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntruhps2048677</td>
<td></td>
<td></td>
</tr>
<tr>
<td>poly_Rq_inv</td>
<td>3506621</td>
<td>341482</td>
</tr>
<tr>
<td>poly_R2_inv</td>
<td>2791906</td>
<td>136776</td>
</tr>
<tr>
<td>poly_S3_inv</td>
<td>4153823</td>
<td>482005</td>
</tr>
<tr>
<td>crypto_sort_int32</td>
<td>104691</td>
<td>17819</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>snrup761/ntrulp761</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Rq_recip3</td>
<td>116353545</td>
<td>5500466</td>
</tr>
<tr>
<td>R3_recip</td>
<td>127578811</td>
<td>580494</td>
</tr>
<tr>
<td>Rq_encode</td>
<td>17753</td>
<td>2084</td>
</tr>
<tr>
<td>Rq_decode</td>
<td>31715</td>
<td>3914</td>
</tr>
<tr>
<td>Rounded_encode</td>
<td>14707</td>
<td>3145</td>
</tr>
<tr>
<td>Rounded_decode</td>
<td>31832</td>
<td>3445</td>
</tr>
<tr>
<td>crypto_sort_uint32</td>
<td>186867</td>
<td>19625</td>
</tr>
</tbody>
</table>
### Detailed Numbers of NTRU Prime

Table 18: Detailed performance numbers of `sntrup761` and `ntrulpr761` with Good-Rader-outer. Only performance-critical subroutines are shown.

<table>
<thead>
<tr>
<th>sntrup761</th>
<th>Cycles</th>
<th>ntrulpr761</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypto_kem_keypair</td>
<td>6297001</td>
<td>crypto_kem_keypair</td>
<td>257382</td>
</tr>
<tr>
<td>ZKeyGen</td>
<td>6241207</td>
<td>ZKeyGen</td>
<td>245982</td>
</tr>
<tr>
<td>KeyGen</td>
<td>6190454</td>
<td>KeyGen</td>
<td>109574</td>
</tr>
<tr>
<td>Rq_recip3</td>
<td>5500466</td>
<td>Rq_recip3</td>
<td>2084</td>
</tr>
<tr>
<td>R3_recip</td>
<td>580494</td>
<td>R3_recip</td>
<td>3145</td>
</tr>
<tr>
<td>Rq_mult_small</td>
<td>36266</td>
<td>Rq_mult_small</td>
<td>36266</td>
</tr>
<tr>
<td>sort</td>
<td>19799</td>
<td>sort</td>
<td>19207</td>
</tr>
<tr>
<td>randombytes</td>
<td>88678</td>
<td>randombytes</td>
<td>41193</td>
</tr>
<tr>
<td>sha2</td>
<td>13626</td>
<td>sha2</td>
<td>12507</td>
</tr>
<tr>
<td>crypto_kem_enc</td>
<td>144681</td>
<td>crypto_kem_enc</td>
<td>400223</td>
</tr>
<tr>
<td>ZEncrypt</td>
<td>43978</td>
<td>ZEncrypt</td>
<td>375021</td>
</tr>
<tr>
<td>Encrypt</td>
<td>36820</td>
<td>Encrypt</td>
<td>366654</td>
</tr>
<tr>
<td>Rq_mult_small</td>
<td>36266</td>
<td>Rq_mult_small (2×)</td>
<td>2× 36266</td>
</tr>
<tr>
<td>sort</td>
<td>19799</td>
<td>sort</td>
<td>19207</td>
</tr>
<tr>
<td>sha2</td>
<td>13626</td>
<td>sha2</td>
<td>12507</td>
</tr>
<tr>
<td>crypto_kem_dec</td>
<td>145777</td>
<td>crypto_kem_dec</td>
<td>435789</td>
</tr>
<tr>
<td>ZDecrypt</td>
<td>79863</td>
<td>ZDecrypt</td>
<td>44112</td>
</tr>
<tr>
<td>Decrypt</td>
<td>75517</td>
<td>Decrypt</td>
<td>39702</td>
</tr>
<tr>
<td>Rq_mult_small</td>
<td>36266</td>
<td>Rq_mult_small</td>
<td>36266</td>
</tr>
<tr>
<td>R3_mult</td>
<td>37652</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rq_decode</td>
<td>3914</td>
<td>Rq_decode</td>
<td>3445</td>
</tr>
<tr>
<td>Rounded_encode</td>
<td>3145</td>
<td>Rounded_encode</td>
<td>3145</td>
</tr>
<tr>
<td>randombytes</td>
<td>43748</td>
<td>sha2</td>
<td>27719</td>
</tr>
<tr>
<td>sha2</td>
<td>19625</td>
<td>sha2</td>
<td>18793</td>
</tr>
</tbody>
</table>

* The numbers of sha2 cycles of `Encrypt` are included.
G Detailed Numbers of NTRU

Table 19: Detailed performance numbers of \texttt{ntruhps2048677} with Toeplitz-TC. Only performance-critical subroutines are shown.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypto_kem_keypair</td>
<td>1002187</td>
</tr>
<tr>
<td>owcpa_keypair</td>
<td>990579</td>
</tr>
<tr>
<td>poly_S3_inv</td>
<td>482005</td>
</tr>
<tr>
<td>poly_Rq_mul(×5)</td>
<td>5 × 26784</td>
</tr>
<tr>
<td>poly_Rq_inv</td>
<td>341482</td>
</tr>
<tr>
<td>poly_R2_inv</td>
<td>136776</td>
</tr>
<tr>
<td>poly_Rq_mul(×8)</td>
<td>8 × 26784</td>
</tr>
<tr>
<td>sort</td>
<td>17819</td>
</tr>
<tr>
<td>randombytes</td>
<td>12054</td>
</tr>
<tr>
<td>crypto_kem_enc</td>
<td>79213</td>
</tr>
<tr>
<td>owcpa_enc</td>
<td>32501</td>
</tr>
<tr>
<td>poly_Rq_mul</td>
<td>26784</td>
</tr>
<tr>
<td>randombytes</td>
<td>13023</td>
</tr>
<tr>
<td>sort</td>
<td>18040</td>
</tr>
<tr>
<td>sha3</td>
<td>5148</td>
</tr>
<tr>
<td>crypto_kem_dec</td>
<td>120208</td>
</tr>
<tr>
<td>owcpa_dec</td>
<td>100842</td>
</tr>
<tr>
<td>poly_Rq_mul(×2)</td>
<td>2 × 26784</td>
</tr>
<tr>
<td>poly_S3_mul</td>
<td>28341</td>
</tr>
<tr>
<td>sha3</td>
<td>18867</td>
</tr>
</tbody>
</table>

References


