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

Han-Ting Chen[1], Yi-Hua Chung[2], Vincent Hwang[2,3], Chi-Ting Liu[1,2] and Bo-Yin Yang[2]

[1] National Taiwan University, Taipei, Taiwan
r10922073@csie.ntu.edu.tw,gting906@gmail.com
[2] Academia Sinica, Taipei, Taiwan
yhchiara@gmail.com,vincentvbh7@gmail.com,by@crypto.tw
[3] Max Planck Institute for Security and Privacy, Bochum, Germany

Abstract.
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 `ntrulpr761`, 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

Contents

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 [AAC+22] Dilithium, Kyber, and Falcon wrote number-theoretic transforms (NTTs) into their specifications in response.

NTRU is still being used by Google [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 [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 [Sch77] and Nussbaumer [Nus80] while maintaining the vectorization friendliness? [BBCT22] showed that vectorized "big by big" polynomial multiplication takes $\sim 1.5\times$ cycles of the "big by small" one with AVX2. [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 $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ where q and p are primes such that the ring is a finite field. We target the polynomial multiplications for parameter sets `sntrup761` and `ntrulpr761` where $q = 4591$ and $p = 761$. One should note that `sntrup761`, which is used by OpenSSH, uses a (Quotient) NTRU structure, and requires inversions in $\mathbb{Z}_3[x]/\langle x^{761} - x - 1\rangle$ and $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$. We refer the readers to the specification [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 $\mathbb{Z}_3[x]/\langle\Phi_n\rangle$, $\mathbb{Z}_q[x]/\langle\Phi_n\rangle$, and $\mathbb{Z}_q[x]/\langle x^n-1\rangle$ where q is a power of 2, n is a prime, and Φ_n is the nth cyclotomic polynomial, which for prime n is $\frac{x^n-1}{x-1} = \sum_{i<n} x^i$. We target the parameter 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 $\mathbb{Z}_3[x]/\langle\Phi_n\rangle$ and $\mathbb{Z}_q[x]/\langle x^n-1\rangle$, 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 $\boldsymbol{g} \in R[x]$ defines an ideal $\langle\boldsymbol{g}\rangle \coloneqq \boldsymbol{g}R[x]$ and a quotient ring $R[x]/\langle\boldsymbol{g}\rangle$. We often multiply in $R[x]/\langle\boldsymbol{g}\rangle$ for a monic $\boldsymbol{g}$ of degree n with the convention that $\deg(\boldsymbol{g}) = -1 \implies \boldsymbol{g} = 0$. We regard $R[x]/\langle\boldsymbol{g}\rangle$ as an (associative) R-algebra [Jac12, Chapter 7] over R. This allows us to characterize $R[x]/\langle\boldsymbol{g}\rangle$ as a ring and a module. We use $R[x]/\langle\boldsymbol{g}\rangle$ while focusing on the ring or R-algebra view, and R^n for the module view. For arbitrary R-algebra monomorphism with $R[x]/\langle\boldsymbol{g}\rangle$ as the domain, one can derive optimizations via a series of ring or module monomorphisms.

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

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

$$\frac{R[x]}{\left\langle\prod_{i_0,\dots,i_{h-1}} \boldsymbol{g}_{i_0,\dots,i_{h-1}}\right\rangle} \cong \prod_{i_0} \frac{R[x]}{\left\langle\prod_{i_1,\dots,i_{h-1}} \boldsymbol{g}_{i_0,\dots,i_{h-1}}\right\rangle} \cong \cdots \cong \prod_{i_0,\dots,i_{h-1}} \frac{R[x]}{\left\langle \boldsymbol{g}_{i_0,\dots,i_{h-1}}\right\rangle}.$$

Cooley–Tukey FFT [CT65] chooses $\boldsymbol{g}_{i_0,\dots,i_{h-1}}$'s as binomials and Bruun's FFT [Bru78, Mur96] chooses $\boldsymbol{g}_{i_0,\dots,i_{h-1}}$'s as trinomials of the form $x^{2m}+\alpha x^m+1$. The finite field version of Bruun's FFT enables the choices $\boldsymbol{g}_{i_0,\dots,i_{h-1}} = x^{2m} + \alpha x^m + \beta$ [BC87]. For $\boldsymbol{g} = x^p - 1$ with prime p, Rader's FFT [Rad68] computes the map $R[x]/\langle x^p-1\rangle \cong \prod_i R[x]/\langle x-\omega_p^i\rangle$ via a polynomial multiplication modulo $x^{p-1}-1$ where ω_p is a principal p-th root of unity. A k-way Toom–Cook computes via the monomorphism $R[x]\Big/\left\langle\prod_{i=0}^{2k-2}(x-s_i)\right\rangle \cong \prod_{i=0}^{2k-2} R[x]/\langle x-s_i\rangle$ 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 $\boldsymbol{g} = x^n - 1$ for an n with the coprime factorization $n = \prod_d q_d$, Good–Thomas FFT [Goo58] converts $R[x]/\langle x^n-1\rangle$ into $\bigotimes_d R[x_d]/\langle x_d^{n_d}-1\rangle$. We then choose our favorite transformations for each of $R[x_d]/\langle x_d^{n_d}-1\rangle$ and tensor them. For a factorization $n = n_0 n_1$, Schönhage's and Nussbaumer's FFTs first start with writing $\boldsymbol{g}_0|_{y=x^{n_1}} = \boldsymbol{g}(x)$. Schönhage requires $\boldsymbol{g}_0|(y^{n_0}-1)$ and an injection $R[x]/\langle x^{n_1}-y\rangle \hookrightarrow R[x]/\langle\boldsymbol{h}\rangle$

with $\boldsymbol{h}|\Phi_{n_0}(x)$. Nussbaumer requires $\boldsymbol{g}_0|\Phi_{2n_1}$ and an injection $R[x,y]/\langle x^{n_1}-y, \boldsymbol{g}_0\rangle \hookrightarrow R[x,y]/\langle \boldsymbol{h}, \boldsymbol{g}_0\rangle$ with $h|(x^{2n_1}-1)$. *The injections usually double the sizes of polynomial rings.* For finite index sets $\mathcal{J} \subset \mathcal{I}$, truncation [CF94, vdH04] computes the product by restricting the domain and image of a monomorphism. For an $\eta_{\mathcal{I}} : R[x]/\langle \prod_{i\in\mathcal{I}} \boldsymbol{g}_i\rangle \cong \prod_{i\in\mathcal{I}} R[x]/\langle \boldsymbol{g}_i\rangle$, we define $\eta_{\mathcal{J}}$ as $R[x]\Big/\Big\langle \prod_{j\in\mathcal{J}} \boldsymbol{g}_j\Big\rangle \cong \prod_{j\in\mathcal{J}} R[x]/\langle \boldsymbol{g}_j\rangle$. $\eta_{\mathcal{J}}$ now computes a product with size less than or equal to $\deg\left(\prod_{j\in\mathcal{J}} \boldsymbol{g}_j\right)$ while requiring R the special structure defining $\eta_{\mathcal{I}}$.

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.

	`pqm4`	[ACC+21](**MR***)	[ACC+21](**GT**)	[AHY22]
Idea				
Toom-4	✓	-	-	-
GT (I, VUNF)	-	-	✓	✓
GT (I, VF)	-	-	-	✓
Rader	-	✓	-	-
Applicability				
Big by small	✓	✓	✓	✓
Big by big	✓	✓	-	-
Coefficient ring	$\mathbb{Z}_{4591}$	$\mathbb{Z}_{4591}$	$\mathbb{Z}$	$\mathbb{Z}$
Performance				
	223 871	152 177	159 176	151 374

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

[BBC+20] was already using Good–Thomas FFT for computing big-by-small polynomial multiplications in NTRU Prime on Intel Haswell processor with AVX2. [BBC+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 $\mathbb{Z}_q[x]/\langle x^{1620}-1\rangle$ with one radix-2, three radix-3, and one radix-5 butterflies; and (ii) FFT for $\mathbb{Z}_q[x]/\langle x^{1530}-1\rangle$ 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.

Platform	Haswell			Cortex-A72
Work	[BBC+20]	[BBCT22](**GT**)	[BBCT22](**SN**)	[Haa21]
Idea				
GT (E, VUNF)	✓	✓	-	✓
Schönhage	-	-	✓	-
Nussbaumer	-	-	✓	-
Truncation	-	-	✓	-
Applicability				
Big by small	✓	✓	✓	✓
Big by big	-	-	✓	-
Coefficient ring	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}_{4591}$	$\mathbb{Z}$
Performance				
	18 080	16 992	25 113	242 585

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 $\mathbb{Z}_q[x]/\langle(x^{512}-1)(x^{1024}+1)\rangle$. 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 $\mathbb{Z}[x]\Big/\left\langle x^{2^k}+1\right\rangle$ 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 $\frac{1536}{64}=48$. 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 $x^{2^k}+c$ for $c\neq\pm1$. 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 $\mathbb{Z}_q[x]/\langle x^{16}\pm\omega_{51}^i\rangle$. The composed multiplication over a finite field shows that the factorization follows the same pattern of factorizing $\mathbb{Z}_q[x]/\langle x^{16}\pm1\rangle$ since $51\perp16$.

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`.

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

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.

	[KRS19]	[CHK+21]	[IKPC22]	[AHY22]
Idea				
Toom-4	✓	-	-	-
Toeplitz	-	-	✓(**TC**-4)	-
GT (implicity, VUNF)	-	✓	-	✓
GT (implicity, VF)	-	-	-	✓
Applicability				
Big by small	✓	✓	✓	✓
Big by big	✓	✓	✓	-
Coefficient ring	$\mathbb{Z}_{2^{16}}$	$\mathbb{Z}$	$\mathbb{Z}_{2^{16}}$	$\mathbb{Z}$
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.

Platform	Skylake		Cortex-A72
Work	[CDH+20]***	[CHK+21]	[NG21]
Idea			
Toom-4	✓	-	✓
Cooley–Tukey	-	✓	-
Applicability			
Big by small	✓	✓	✓
Big by big	✓	-	✓
Coefficient ring	$\mathbb{Z}_{2^{16}}$	$\mathbb{Z}$	$\mathbb{Z}_{2^{16}}$
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].

	NTT	dim × base_mul	InnerProduct
lightsaber	1 529	Unknown	7 038
saber	1 529	2 689	9 284

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 $\mathbb{Z}_{q'}[x]/\langle x^{1536}-1\rangle$ where q' is a 32-bit NTT-friendly modulus. We choose $(\tilde{q}, v) = (1, 4)$ 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 $\mathbb{Z}_{q'}[x]/\langle x^{256}+1\rangle$ (for Saber) takes 1529 cycles, we estimate a size-384 NTT for the size-1536 cyclic polynomial ring to be no worse than $1529 \cdot \frac{1536}{256} \cdot \frac{9}{6} \approx 13761$[1]. Next, we estimate that 64 size-4 base multiplications will take $2689 - 2 \cdot (9284 - 7038 - 1529) = 1255$ cycles[2]. In summary, we believe that the size-384 NTT approach for $\mathbb{Z}_{q'}[x]/\langle x^{1536}-1\rangle$ will take at most $3 \cdot 1376112 + 6 \cdot 1255 = 48813$ 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 $\mathbb{Z}_{2^{11}}$, 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 $\left\{0, \pm 1, \pm 2, \pm \frac{1}{2}, 3, \infty\right\}$ which doesn't seem to be used for Toom–5 in the literature.

[1] We 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.

[2] In [BHK+22], the inner product in the encryption of Saber consists of l NTTs, one `dim` × `basemul`, and one inverse NTT. `dim` × `basemul` 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` × `basemul` 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).

NTRU strategies			
	`Toeplitz-TC`	`Toom-Cook`	`Good-Thomas`
Idea			
Toeplitz (Section 4)	✓(**TC**-5)	-	-
Toom–5 (Section 3.6)	-	✓	-
Good–Thomas (Section 3.4)	-	-	✓
Vector-by-scalar (`basemul`, Section 4.2)	✓	-	-
Vector-by-vector (`basemul`, Section 4.2)	-	✓	✓****
Applicability			
Big by small	✓	✓	✓
Big by big	✓	✓	-
Coefficient ring	$\mathbb{Z}_{2^{16}}$	$\mathbb{Z}_{2^{16}}$	$\mathbb{Z}$

**** 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 $i_0 \to i_1 \to i_2$ is called inner-product approach and $i_2 \to i_0 \to i_1$ is called outer-product approach. When `B` is a vector, there are two ways for iterating: $i_0 \to i_2$ and $i_2 \to i_0$. Since $i_0 \to i_2$ is a special case of $i_0 \to i_1 \to i_2$ and $i_2 \to i_0$ is a special case of $i_2 \to i_0 \to i_1$, we call $i_0 \to i_2$ inner-product-based and $i_2 \to i_0$ outer-product-based approaches. For vectorization, $i_0 \to i_2$ means interleaving, applying vector-by-vector multiplications, and deinterleaving. We show that $i_2 \to i_0$ 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 $f : R[x]_{<n} \to S$ 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 `ntrulpr761/sntrup761`, we pick f as an FFT implementing $\mathbb{Z}_{4591}[x]/\langle x^{1632} - 1\rangle \cong \prod_i \mathbb{Z}_{4591}[x]/\langle x^{16} \pm \omega_{51}^i\rangle$. Since we have weighted convolutions after applying f, we compute with vector-by-scalar multiplications. This is our `Good-Rader-outer`. For `ntruhps2048677`, we regard the polynomial multiplication in $\mathbb{Z}_{2048}[x]/\langle x^{677} - 1\rangle$ 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 $\frac{720}{5\cdot 3^2\cdot 2} = 8$, we eventually have several Toeplitz matrix–vector products of dimension 8. We compute them with vector-by-scalar multiplications. This is our `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 μops 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 μops per cycle. Furthermore, in a single cycle, the frontend dispatches at most one μop using `B`, at most two μops using `I0/I1`, at most two μops using `M`, at most one μop using `F0`, at most one μop using `F1`, and at most two μops 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 `addv`, `smaxv`, `sadalp`. We are not using them in this paper and refer to [ARM15] for more details.

to F1 since the time spent in F0 will dominate our runtime. Conversely, with more shifts than multiplications, we want to dispatch most non-shifts to F0. 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 chacha20 implementing randombytes for benchmarking [BHK+22], we even consider a compiler-aided mixing of I0/I1, F0/F1, *and* L/S[4]. The idea also proved valuable for 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 suffices .16B .8H, .4S, and .2D respectively on the register name. For referencing a certain lane, we use the annotation .H[5] for the 5th (zero-based) halfword of the register and similarly for other lanes and data widths. We refer readers to [ARM21, Figure A1-1].

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

There are many variants of multiplications: mla/mls computes the same product vector and accumulates to or subtracts from the destination. There are high-half products sqdmulh and sqrdmulh. 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 s{mul,mla,mls}l{,2}. 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 smull2 using the upper 64-bit instead. Their accumulating and subtracting variants are s{mla,mls}l{,2}. We will not use the unsigned counterparts u{mul,mla,mls}l{,2}.

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

For basic arithmetic, the usual add/sub adds/subtracts the corresponding elements. Long variants 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[5].

Then we have permutations — uzp{1,2} extracts the even and odd positions respectively from a pair of vectors and concatenates the results into a vector. 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. 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 smull and smull2 to the same pair of vectors for the double-width products. We denote (lo, hi) = (smull, smull2)(a, b) where lo is the destination register of smull and hi is the destination register of smull2. For the accumulating/subtracting variants s{mla, mls}{, 2}, we denote (lo, hi) = (lo, hi)(smlal, smlal2)(a, b) where the pairs of destinations and the accumulators must be the same.

[4]We write some assembly and only obtain comparable performance. So we keep the implementations with intrinsics instead for readability.

[5]There are several options for signed-extending vector elements — saddl{,2} and ssubl{,2} which go to either F0/F1, sxtl{,2} to F1, and smull{,2} going to F0.

2.2 Modular Reductions and Multiplications in Armv8-A.

Algorithm 1 Barrett reduction.

This is [BHK+22, Algorithm 11].

Input: $\mathtt{a} = a$.

Output: $\mathtt{a} = a - \left\lfloor \frac{a\left\lfloor \frac{2^e\mathtt{R}}{q} \right\rceil}{2^e\mathtt{R}} \right\rceil q \equiv a \bmod^{\pm} q$.

1: `sqdmulh t, a,` $\left\lfloor \frac{2^e\mathtt{R}}{q} \right\rceil$
2: `srshr t, t, #`$(e+1)$
3: `mls a, t,` q

Algorithm 2 Barrett multiplication.

This is [BHK+22, Algorithm 10].

Input: $\mathtt{a} = a$.

Output: $\mathtt{a} = ab - \left\lfloor \frac{a\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q \equiv ab \bmod^{\pm} q$.

1: `sqrdmulh t, a,` $\frac{\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{2}$
2: `mul a, a,` b
3: `mls a, t,` q

Let q be an odd modulus, and $\mathtt{R}$ 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 $a \bmod^{\pm} q$ with $a - \left\lfloor \frac{a \cdot \left\lfloor \frac{2^e\mathtt{R}}{q} \right\rceil}{2^e\mathtt{R}} \right\rceil$ (cf. Algorithm 1). For multiplying an unknown a with a fixed value b, we compute $ab - \left\lfloor \frac{a\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q \equiv ab \bmod^{\pm} q$ (Barrett multiplication [BHK+22]) where $\lfloor \rceil_2$ is the function mapping a real number r to $2\left\lfloor \frac{r}{2} \right\rceil$ (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 `mul` and `add` into `mla`. Algorithm 4 computes $a - bc$ by toggling `mls` into `mla` and merging `mul` and `sub` into `mls`. For accumulating several products, we use Montgomery multiplication [Mon85] with long arithmetic as shown in Algorithm 5.

Algorithm 3 `Barrett_mla`.

Input: $\mathtt{a} = a$.

Output: $\mathtt{a} = a + bc - \left\lfloor \frac{b\left\lfloor \frac{c\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q$.

1: `sqrdmulh t, b,` $\frac{\left\lfloor \frac{c\mathtt{R}}{q} \right\rceil_2}{2}$
2: `mla a, b, c`
3: `mls a, t,` q

Algorithm 4 `Barrett_mls`.

Input: $\mathtt{a} = a$.

Output: $\mathtt{a} = a - bc + \left\lfloor \frac{b\left\lfloor \frac{c\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q$.

1: `sqrdmulh t, b,` $\frac{\left\lfloor \frac{c\mathtt{R}}{q} \right\rceil_2}{2}$
2: `mls a, b, c`
3: `mla a, t,` q

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

Input: $(\mathtt{a0}, \mathtt{a1}, \mathtt{b0}, \mathtt{b1}) = (a_0, a_1, b_0, b_1)$.

Output: $\mathtt{c} = \frac{\sum_i a_i b_i + (((\sum_i a_i b_i) \bmod^{\pm} \mathtt{R})(-q^{-1} \bmod^{\pm} \mathtt{R}) \bmod^{\pm} \mathtt{R})}{\mathtt{R}} \equiv (\sum_i a_i b_i)\,\mathtt{R}^{-1} \bmod^{\pm} q$.

1: $(\mathtt{lo}, \mathtt{hi}) = (\mathtt{smull}, \mathtt{smull2})(\mathtt{a0}, \mathtt{b0})$
2: $(\mathtt{lo}, \mathtt{hi}) = (\mathtt{lo}, \mathtt{hi})(\mathtt{smlal}, \mathtt{smlal2})(\mathtt{a1}, \mathtt{b1})$
3: `uzp1 t, lo, hi`
4: `mul t, t,` $-q^{-1} \bmod^{\pm} \mathtt{R}$
5: $(\mathtt{lo}, \mathtt{hi}) = (\mathtt{lo}, \mathtt{hi})(\mathtt{smlal}, \mathtt{smlal2})(\mathtt{t}, q)$
6: `uzp2 c, lo, hi`
7: ▷ Steps 3-6 (written `c = 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 $n = \prod_l n_l$ and $\boldsymbol{g}_{i_0,\ldots,i_{h-1}} \in R[x]$ 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

$$\frac{R[x]}{\left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \prod_{i_0} \frac{R[x]}{\left\langle \prod_{i_1,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \cdots \cong \prod_{i_0,\ldots,i_{h-1}} \frac{R[x]}{\left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle}.$$

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

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

$$R[x] \Big/ \left\langle \prod_{i_0,i_1} \boldsymbol{g}_{i_0,i_1} \right\rangle \overset{\eta_0}{\cong} \prod_{i_0} R[x] \Big/ \left\langle \prod_{i_1} \boldsymbol{g}_{i_0,i_1} \right\rangle \overset{\eta_1}{\cong} \prod_{i_0,i_1} R[x]/\langle \boldsymbol{g}_{i_0,i_1} \rangle$$

where i_0 ranges over $0, \ldots, n_0 - 1$ and i_1 ranges over $0, \ldots, n_1 - 1$. We call the isomorphism η_0 a radix-n_0 split and an implementation of η_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 η_0 and η_1 in a single load-store pair. If η_0 and η_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, $\zeta \in R$, ω_n a principal nth root of unity in R, $n \perp \mathrm{char}(R)$, and $\boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x^v - \zeta \omega_n^{\sum_l i_l \prod_{j<l} n_j} \in R[x]$. Since $\prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x^{nv} - \zeta^n$, 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_l'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 $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$ when n is a power of two — as a way to reduce the number of real multiplications when we are operating over $\mathbb{C}$. [Mur96] later generalized it to arbitrary even n. For our implementations, we need the results of factorizing $x^{2^k} + 1 \in \mathbb{F}_q[x]$ when $q \equiv 3 \pmod 4$ [BGM93] and the composed multiplication of polynomials in $\mathbb{F}_q[x]$ [BC87]. Factorizing $x^n - 1$ over $\mathbb{F}_q$ is an active research area [BGM93, Mey96, TW13, MVdO14, WYF18, WY21].

3.3.1 Review: Bruun's Original Complex ($R = \mathbb{C}$) Case

We choose $\boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x^{2v} - \left(\zeta\omega_n^{\sum_l i_l \prod_{j<l} n_j} + \zeta^{-1}\omega_n^{-\sum_l i_l \prod_{j<l} n_j}\right) x^v + 1$ so $x^{2n} - (\zeta^n + \zeta^{-n})\, x^n + 1 = \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}}$. This provides us an alternative factorization for $x^{4n} - 1 = (x^{2n} - 1)(x^{2n} + 1)$ 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 $\prod_{i_0,\ldots,i_{h-1}} \mathbb{C}[x] \Big/ \left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}(x)} \right\rangle$. For $v = 1$, if we split each $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$ into the degree-1 factors, we have $\mathbb{C}[x]/\left\langle x^{2n}+1 \right\rangle \cong \prod_{i_0,\ldots,i_{h-1}} \left(\mathbb{C}[x] \Big/ \left\langle x - \omega_{4n}^{1+\sum_l i_l \prod_{j<l} n_j} \right\rangle \times \mathbb{C}[x] \Big/ \left\langle x - \omega_{4n}^{-1-\sum_l i_l \prod_{j<l} n_j} \right\rangle \right)$, which is $\prod_i \mathbb{C}[x]/\left\langle x - \omega_{4n}^{1+2i} \right\rangle$ 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 $x^{2^k} + 1$ factors into irreducible trinomials $x^2 + \gamma x + 1$ in $\mathbb{F}_q[x]$. Else (i.e., $k \geq w$) $x^{2^k} + 1$ factors into irreducible trinomials $x^{2^{k-w+1}} + \gamma x^{2^{k-w}} - 1$ in $\mathbb{F}_q[x]$.

We need Theorem 1 for our implementations. Also, given $\boldsymbol{f}_0, \boldsymbol{f}_1 \in \mathbb{F}_q[x]$, we define their "composed multiplication" as $(\boldsymbol{f}_0 \odot \boldsymbol{f}_1) \coloneqq \prod_{\boldsymbol{f}_0(\alpha)=0} \prod_{\boldsymbol{f}_1(\beta)=0} (x - \alpha\beta)$ where α, β run over all the roots of $\boldsymbol{f}_0, \boldsymbol{f}_1$ in an extension field of $\mathbb{F}_q$.We need the following from [BC87]:

Lemma 1 ([BC87, Equation 8])**.** $\left(\prod_{i_0} \boldsymbol{f}_{0,i_0}\right) \odot \left(\prod_{i_1} \boldsymbol{f}_{1,i_1}\right) = \prod_{i_0,i_1} \left(\boldsymbol{f}_{0,i_0} \odot \boldsymbol{f}_{1,i_1}\right)$ holds for any sequences of polynomials $\boldsymbol{f}_{0,i_0}, \boldsymbol{f}_{1,i_1} \in \mathbb{F}_q[x]$.

Lemma 2 ([BC87, Equation 5])**.** If $\boldsymbol{f}_0 = \prod_\alpha (x - \alpha) \in \mathbb{F}_q[x]$, then for any $\boldsymbol{f}_1 \in \mathbb{F}_q[x]$, we have $\boldsymbol{f}_0 \odot \boldsymbol{f}_1 = \prod_\alpha \alpha^{\deg(\boldsymbol{f}_1)} \boldsymbol{f}_1(\alpha^{-1}x) \in \mathbb{F}_q[x]$.

Lemma 3. Let r be odd, $x^r - 1 = \prod_{i_0} (x - \omega_r^{i_0}) \in \mathbb{F}_q[x]$, and $x^{2^k} - 1 = \prod_{i_1} \boldsymbol{f}_{i_1} \in \mathbb{F}_q[x]$. We have $x^{2^k r} - 1 = \prod_{i_0} \left(x^{2^k} - \omega_r^{2^k i_0}\right) = \prod_{i_0,i_1} \omega_r^{i_0 \deg(\boldsymbol{f}_{i_1})} \boldsymbol{f}_{i_1}(\omega_r^{-i_0} x)$.

Proof. First observe $x^{2^k r} - 1 = (x^r - 1) \odot \left(x^{2^k} - 1\right)$[6]. By Lemma 1, this equals $\prod_{i_0} \left((x - \omega_r^{i_0}) \odot \left(x^{2^k} - 1\right)\right) = \prod_{i_0,i_1} \left((x - \omega_r^{i_0}) \odot \boldsymbol{f}_{i_1}\right)$. Then, according to Lemma 2. we write $(x - \omega_r^{i_0}) \odot \left(x^{2^k} - 1\right) = x^{2^k} - \omega_r^{2^k i_0}$ and $(x - \omega_r^{i_0}) \odot \boldsymbol{f}_{i_1} = \omega_r^{i_0 \deg(\boldsymbol{f}_{i_1})} \boldsymbol{f}_{i_1}(\omega_r^{-i_0} x)$. □

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

$$\frac{\mathbb{F}_q[x]}{\left\langle x^{2^k r} - 1 \right\rangle} \cong \frac{\mathbb{F}_q[x]}{\left\langle \prod_{i_0} \left(x^{2^k} - \omega_r^{2^k i_0}\right) \right\rangle} \cong \frac{\mathbb{F}_q[x]}{\left\langle \prod_{i_0,i_1} \omega_r^{i_0 \deg(\boldsymbol{f}_{i_1})} \boldsymbol{f}_{i_1}(\omega_r^{-i_0} x) \right\rangle}.$$

3.3.3 Bruun's Butterflies and Inverses

We illustrate radix-2 Bruun's butterflies. Define $\mathbf{Bruun}_{\alpha,\beta}$ as follows:

$$\mathbf{Bruun}_{\alpha,\beta} : \begin{cases} \frac{R[x]}{\langle x^4 + (2\beta - \alpha^2)x^2 + \beta^2 \rangle} & \to \frac{R[x]}{\langle x^2 + \alpha x + \beta \rangle} \times \frac{R[x]}{\langle x^2 - \alpha x + \beta \rangle} \\ a_0 + a_1 x + a_2 x^2 + a_3 x^3 & \mapsto ((\hat{a}_0 + \hat{a}_1 x), (\hat{a}_2 + \hat{a}_3 x)) \end{cases}$$

[6] $\forall q_0 \perp q_1, \left\{\omega_{q_0}^{i_0} \omega_{q_1}^{i_1} | 0 \leq i_0 < q_0, 0 \leq i_1 < q_1\right\} = \left\{\omega_{q_0 q_1}^{i} | 0 \leq i < q_0 q_1\right\}$ in the splitting field of $x^{q_0 q_1} - 1$.

where

$$\begin{cases}(\hat{a}_0, \hat{a}_1) = & \left(a_0 - \beta a_2 + \alpha\beta a_3, a_1 + (\alpha^2 - \beta)a_3 - \alpha a_2\right), \\ (\hat{a}_2, \hat{a}_3) = & \left(a_0 - \beta a_2 - \alpha\beta a_3, a_1 + (\alpha^2 - \beta)a_3 + \alpha a_2\right). \end{cases}$$

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 `Barrett_mla` and `Barrett_mls` whenever a product is followed by only one accumulation or subtraction.

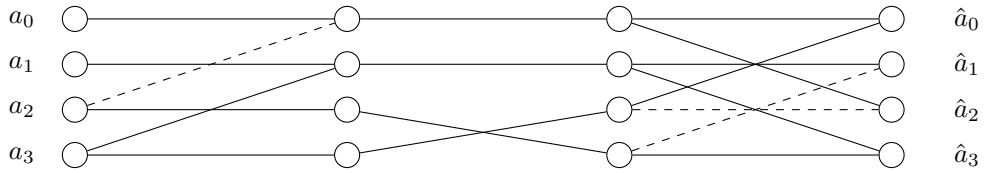


Figure 1: Bruun's butterfly. $(\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3) = \mathbf{Bruun}_{\alpha,\beta}(a_0, a_1, a_2, a_3)$.

$$2\mathbf{Bruun}^{-1}_{\alpha,\beta} : \begin{cases} \frac{R[x]}{\langle x^2+\alpha x+\beta\rangle} \times \frac{R[x]}{\langle x^2-\alpha x+\beta\rangle} & \to \frac{R[x]}{\langle x^4+(2\beta-\alpha^2)x^2+\beta^2\rangle} \\ ((\hat{a}_0 + \hat{a}_1 x), (\hat{a}_2 + \hat{a}_3 x)) & \mapsto 2a_0 + 2a_1 x + 2a_2 x^2 + 2a_3 x^3 \end{cases}$$

correspondingly defines the inverse, where

$$\begin{cases}2(a_0, a_1) = & (\hat{a}_0 + \hat{a}_2 + (\hat{a}_3 - \hat{a}_1)\,\alpha^{-1}\beta, \hat{a}_1 + \hat{a}_3 - (\hat{a}_0 - \hat{a}_2)\,\alpha^{-1}\beta^{-1}\left(\alpha^2 - \beta\right)), \\ 2(a_2, a_3) = & ((\hat{a}_3 - \hat{a}_1)\,\alpha^{-1}, (\hat{a}_0 - \hat{a}_2)\,\alpha^{-1}\beta^{-1}). \end{cases}$$

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 $\left(\alpha^2 - \beta\right)$, and add-sub (cf. Figure 2). Both $\mathbf{Bruun}_{\alpha,\beta}$ and $2\mathbf{Bruun}^{-1}_{\alpha,\beta}$, takes 4 multiplications.

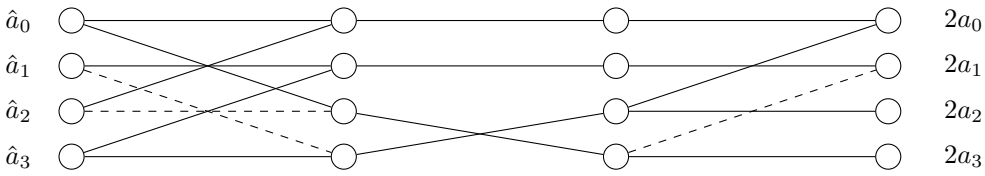


Figure 2: Inverse of Bruun's butterfly. $(2a_0, 2a_1, 2a_2, 2a_3) = 2\mathbf{Bruun}^{-1}_{\alpha,\beta}(\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3)$.

3.3.4 Special Cases of Bruun's Butterflies

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

$\mathbf{Bruun}_{\alpha,\pm1}$. We only need 3 multiplications for each of $\mathbf{Bruun}_{\alpha,\pm1}$ and $2\mathbf{Bruun}^{-1}_{\alpha,\pm1}$ 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 $\mathbf{Bruun}_{\alpha,\pm1}$ for further factorizations by Theorem 1.

$\mathbf{Bruun}_{\alpha,\frac{\alpha^2}{2}}$. We save no multiplications, but only use 2 constants α and $\frac{\alpha^2}{2}$ instead of 4. It is used in the split of $x^{2^k} + \omega_r^{2^k i}$ 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_l's. We first regard $R[x]/\langle x^{nv}-1\rangle \cong (R[x]/\langle x^v-y\rangle)[y]/\langle y^n-1\rangle$ as a polynomial in y and define $\bar{R} \coloneqq R[x]/\langle x^v-y\rangle$ [FP07]. Instead of transforming into $\prod_i \bar{R}[y]/\langle y-\omega_n^i\rangle$ with Cooley–Tukey, we choose $\omega_{n_l} \coloneqq \omega_n^{e_l}$ and write $\omega_n = \prod_l \omega_{n_l}$ where (e_l) is the unique tuple satisfying $a \equiv \sum_l e_l(a \bmod n_l) \pmod n$. Now we rewrite the image as $\prod_i \bar{R}[y]/\langle y-\prod_l \omega_{n_l}^{i \bmod n_l}\rangle$. Rewriting the indices gives $\prod_{i_0,\ldots,i_{h-1}} \bar{R}[y]/\langle y-\prod_l \omega_{n_l}^{i_l}\rangle$. By introducing the equivalences $y \sim \prod_l u_l$ and $\forall l, u_l^{n_l} \sim 1$, we transform $\bar{R}[y]/\langle y^n-1\rangle$ into $\prod_{i_0,\ldots,i_{h-1}} \bar{R}[y]/\langle y-\prod_l \omega_{n_l}^{i_l}\rangle$ with a multi-dimensional FFT. Formally, we have

$$\begin{aligned}\frac{\bar{R}[y]}{\langle y^n-1\rangle} &\cong \frac{\bar{R}[y,u_0,\ldots,u_{h-1}]}{\langle y-\prod_l u_l, u_0^{n_0}-1,\ldots,u_{h-1}^{n_{h-1}}-1\rangle} \\ \cong \prod_{i_0,\ldots,i_{h-1}} \frac{\bar{R}[y,u_0,\ldots,u_{h-1}]}{\left\langle y-\prod_l u_l, u_0-\omega_{n_0}^{i_0},\ldots,u_{h-1}-\omega_{n_{h-1}}^{i_{h-1}}\right\rangle} &\cong \prod_{i_0,\ldots,i_{h-1}} \frac{\bar{R}[y]}{\langle y-\prod_l \omega_{n_l}^{i_l}\rangle}.\end{aligned}$$

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 $\sum_{i=0}^{p-1} a_i x^i \in R[x]/\langle x^p-1\rangle$ to the tuple $(\hat{a}_j) \coloneqq \left(\sum_{i=0}^{p-1} a_i\omega_p^{ij}\right) \in \prod_i R[x]/\langle x-\omega_p^i\rangle$. Let $g \in \mathbb{Z}_p^*$ be a generator. Now $\{1,\ldots,p-1\} = \left\{g,\ldots,g^{p-2},g^{p-1}=1\right\}$. Write $j=g^k$ and $i=g^{-\ell}$. Then $\hat{a}_{g^k}-a_0 = \hat{a}_j - a_0 = \sum_{i=1}^{p-1} a_i\omega_p^{ij} = \sum_{\ell=1}^{p-1} a_{g^{-\ell}}\omega_p^{g^{k-\ell}}$.

The sequence $\left(\sum_{i=1}^{p-1} a_{g^{-i}}\omega_p^{g^{j-i}}\right)_{j=1,\ldots,p-1}$ is the size-$(p-1)$ cyclic convolution of sequences $\left(a_{g^i}\right)_{i=1,\ldots,p-1}$ and $\left(\omega_p^{g^i}\right)_{i=1,\ldots,p-1}$. 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 $R[x]$. We can also use them for multiplying polynomials in $R[x]_{<n}$. We introduce $y \sim x^{\frac{n}{k}}$ (zero-pad so that $k|n$) [Ber01], and map $R[x]_{<n} \hookrightarrow R[x]\big/\left\langle x^{\frac{n}{k}}-y\right\rangle [y]_{<k} \hookrightarrow R'[y]_{<k}$ for $R' = R[x]/\langle \boldsymbol{g}\rangle$ with $\deg \boldsymbol{g} \geq \frac{2n}{k}-1$.

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

If one of the *evaluation points* is $s_i=\infty$, the corresponding map into $R'[y]/\langle y-s_i\rangle$ takes the highest degree coefficient (deg-$(k-1)$ for $\boldsymbol{a},\boldsymbol{b}$, deg-$(2k-2)$ for $\boldsymbol{ab}$). [KO62] chose $k=2$ at $\{s_i\}_i=\{0,1,\infty\}$; [Too63] chose $\{s_i\}_i=\{0,\pm1,\ldots,\pm(k-1)\}$; and [Win80, Page 31] replaced $-k+1$ with ∞. We write $\mathbf{TC}_{(2k-1)\times k}$ for the matrix mapping the coefficients of a deg $< k$ polynomial into $\prod_{i=0}^{2k-2} R'[y]/\langle y-s_i\rangle$ and $\mathbf{TC}^{-1}_{(2k-1)\times(2k-1)}$ for the matrix mapping $\prod_{i=0}^{2k-2} R'[y]/\langle y-s_i\rangle$ into $R[y]\Big/\left\langle\prod_{i=0}^{2k-2}(y-s_i)\right\rangle$.

A key observation is that for $k=5$ and $\{s_i\} = \left\{0,\pm1,\pm2,\pm\frac{1}{2},3,\infty\right\}$, $\mathbf{TC}^{-1}_{9\times9}$ only requires "division by 8". This implies 3-bit losses when working over $\mathbb{Z}_{2^{16}}$. The matrix $\mathbf{TC}^{-1}_{9\times9}$ 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 $\boldsymbol{a}, \boldsymbol{b} \in R[x]/\langle \boldsymbol{g} \rangle$ where $\boldsymbol{g}$ is a degree-$n_0 n_1$ polynomial, we introduce with $y = x^{n_1}$, and write $\boldsymbol{a}$ and $\boldsymbol{b}$ as polynomials in $R[x,y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle$, where $\boldsymbol{g}_0|_{y=x^{n_1}} = \boldsymbol{g}(x)$. In other words, $\boldsymbol{a}_0(y) := \sum_{i_0=0}^{n_0-1} \left(\sum_{i=0}^{n_1-1} a_{i+i_0 n_1} x^i \right) y^{i_0} \in R[x,y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle$ (ditto for $\boldsymbol{b}_0$). We recap transforms when $R[x,y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle$ does not naturally split. Our presentation is motivated by [Ber01, Section 9, Paragraph "High–radix variants"] and [vdH04, Section 3].

We want an injection $R[x]/\langle x^{n_1} - y \rangle \hookrightarrow \bar{R}$ such that $R[x,y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle \hookrightarrow \bar{R}[y]/\langle \boldsymbol{g}_0 \rangle$ is a monomorphism with $\bar{R}[y]/\langle \boldsymbol{g}_0 \rangle \cong \prod_j \bar{R}[y]/\langle \boldsymbol{g}_{0,j} \rangle$. A Schönhage FFT [Sch77] is when $\boldsymbol{g}_0 | (y^{n_0} - 1)$, and $\bar{R} = R[x]/\langle \boldsymbol{h} \rangle$ with $\boldsymbol{h} | \Phi_{n_0}(x)$ (the n_0-th cyclotomic polynomial). E.g., "cyclic Schönhage" for powers of two n_0, $n_1 = \frac{n_0}{4}$, $\boldsymbol{g}_0 = y^{n_0} - 1$, and $\boldsymbol{h} = x^{2n_1} + 1$ is:

$$\frac{R[x]}{\langle x^{n_0 n_1} - 1 \rangle} \cong \frac{\frac{R[x]}{\langle x^{n_1} - y \rangle}[y]}{\langle y^{n_0} - 1 \rangle} \hookrightarrow \frac{\frac{R[x]}{\langle x^{2n_1} + 1 \rangle}[y]}{\langle y^{n_0} - 1 \rangle} \triangleq \frac{\bar{R}[y]}{\langle y^{n_0} - 1 \rangle} \cong \prod_i \frac{\bar{R}[y]}{\langle y - x^i \rangle}.$$

We can also exchange the roles of x and y and get Nussbaumer's FFT [Nus80]. We map $R[x,y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle \hookrightarrow R[x,y]/\langle \boldsymbol{h}, \boldsymbol{g}_0 \rangle$ such that $\boldsymbol{g}_0 | \Phi_{2n_1}(y)$ and $\boldsymbol{h} | (x^{2n_1} - 1)$. This can be illustrated for powers of two $n_0 = n_1$, $\boldsymbol{h} = x^{2n_1} - 1$, and $\boldsymbol{g}_0 = y^{n_0} + 1$:

$$\frac{R[x]}{\langle x^{n_0 n_1} + 1 \rangle} \cong \frac{R[x,y]}{\langle x^{n_1} - y, y^{n_0} + 1 \rangle} \hookrightarrow \frac{\frac{R[y]}{\langle y^{n_0} + 1 \rangle}[x]}{\langle x^{2n_1} - 1 \rangle} \triangleq \frac{\tilde{R}[x]}{\langle x^{2n_1} - 1 \rangle} \cong \prod_i \frac{\tilde{R}[x]}{\langle x - y^i \rangle}.$$

3.8 Enlarging Coefficient Rings

To multiply in $R[x]/\langle x^n - 1 \rangle$ while R lacks a principal n-th root ω_n of unity, we map $R \hookrightarrow R'$ and induce a multiplication in $R'[x]/\langle x^n - 1 \rangle$ 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 $p | q'$, $p \equiv 1 \pmod{n}$ [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 $2^\epsilon r \in \mathbb{Z}_{2^{k+\epsilon}}$, and right-shift $2^\epsilon r$ by ϵ 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 $\mathbb{Z}_{2^{16}}$ so $r = \frac{2^{16-k} r}{2^{16-k}} \in \mathbb{Z}_{2^k}$ can be derived by right-shifting $2^{16-k} r \in \mathbb{Z}_{2^{16}}$ by $16 - k$ bits.

4 Toeplitz Matrix–Vector Product

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

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

4.1 Module and Associative Algebra

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 $r \cdot_M \boldsymbol{a}$ for $(\cdot_M)(r, \boldsymbol{a})$) satisfying the following:

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

We call $(M, +, \cdot_M)$ an R-module. For elements $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1} \in M$, if they are linearly independent and every element in M can be expressed as a linear combination of $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}$, we call $\{\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}\}$ a basis of M and n the dimension. We denote by R^n for an n-dimensional free module. Notice that a ring R and a polynomial ring $R[x]/\langle \boldsymbol{g} \rangle$ are free modules, and the matrix ring $M_{n\times n}(R)$ is an R-module.

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

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

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

4.1.2 Associative Algebra

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

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

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

Obviously, a polynomial ring is an R-algebra and all the ring monomorphisms in 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

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

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

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

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

$\mathbf{Toeplitz}_{n\times n}\left(\texttt{Expand}_{n\to n,\zeta}(-)\right)(-)$ is exactly the `asymmetric_mul` by [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 $x^4 - \zeta$, we apply $\texttt{Expand}_{4\to 4,\zeta}$ with `ext` instructions, and accumulate vector-by-scalar products. Algorithm 6 is an illustration.

Algorithm 6 Outer product approach for $R[x]/\langle x^4 - \zeta\rangle$.

Inputs: $\boldsymbol{a} = a_0 + a_1 x + a_2 x^2 + a_3 x^3$, $\boldsymbol{b} = b_0 + b_1 x + b_2 x^2 + b_3 x^3$.
Outputs: $\boldsymbol{c} = \boldsymbol{ab} \bmod (x^4 - \zeta)$.

1: $\texttt{b} = b_3||b_2||b_1||b_0$
2: $\texttt{t0} = a_3||a_2||a_1||a_0$
3: Compute $\texttt{t} = \zeta a_3||\zeta a_2||\zeta a_1||\zeta a_0$ with Barrett multiplication.
4: ▷ [BHK+22] proposed an interleaved version of this; others [SKS+21, NG21] reduced the interleaved partial results instead.
5: ▷ The remaining steps are different from [BHK+22].
6: `ext t1, t, t0, #3 · 4` ▷ $\texttt{t1} = a_2||a_1||a_0||\zeta a_3$
7: `ext t2, t, t0, #2 · 4` ▷ $\texttt{t2} = a_1||a_0||\zeta a_3||\zeta a_2$
8: `ext t3, t, t0, #1 · 4` ▷ $\texttt{t3} = a_0||\zeta a_3||\zeta a_2||\zeta a_1$
9: $(\texttt{lo}, \texttt{hi}) = (\texttt{smull}, \texttt{smull2})(\texttt{t0}, b_0)$
10: $(\texttt{lo}, \texttt{hi}) = (\texttt{lo}, \texttt{hi})(\texttt{smlal}, \texttt{smlal2})(\texttt{t1}, b_1)$
11: $(\texttt{lo}, \texttt{hi}) = (\texttt{lo}, \texttt{hi})(\texttt{smlal}, \texttt{smlal2})(\texttt{t2}, b_2)$
12: $(\texttt{lo}, \texttt{hi}) = (\texttt{lo}, \texttt{hi})(\texttt{smlal}, \texttt{smlal2})(\texttt{t3}, b_3)$
13: $\boldsymbol{c} = \texttt{Montgomery\_long}(\texttt{lo}, \texttt{hi})$

Generally speaking, once the Toeplitz matrix is constructed via `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 $R[x]/\langle x^4 - \zeta \rangle$. 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 $\log_2 4 \cdot \frac{4}{2} = 4$ pairs of (`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 (`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.

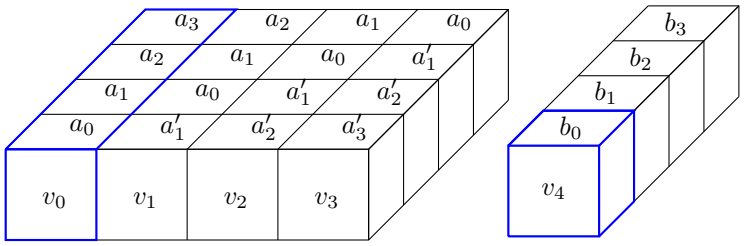


Figure 3: Outer-product-based Toeplitz matrix–vector multiplication via vector-by-scalar multiplication. No permutations are required once we have data in registers $v_0, \ldots, v_3$. We only need 5 registers $v_0, \ldots, v_4$ for holding the operands and 1 or 2 registers for the partial results.

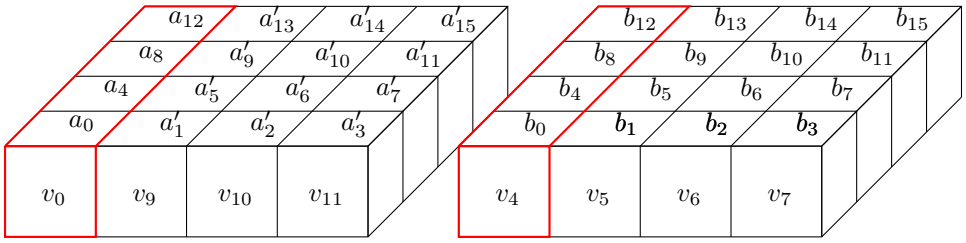


Figure 4: Inner-product-based Toeplitz matrix–vector multiplication via vector-by-vector multiplication. One load $(a_0, \ldots, a_3), \ldots, (a_{12}, \ldots, a_{15})$ into registers $(v_0, \ldots, v_3)$, and transpose the registers as a 4×4 matrix. Same for $(v_4, \ldots, v_7)$ holding $(b_0, \ldots,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

4.5 Large Dimensional Toeplitz Transformation

There are several benefits when working on Toeplitz matrices. Firstly, we only need to store $m+n-1$ coefficients $M_{m-1,0},\ldots,M_{0,0},\ldots,M_{0,n-1}$ of the matrix. Secondly, additions/subtractions of two Toeplitz matrices require only $m+n-1$ additions/subtractions in R. Finally, submatrices from adjacent rows and columns are also Toeplitz matrices. These properties enable efficient divide-and-conquer computations when the dimension is large.

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

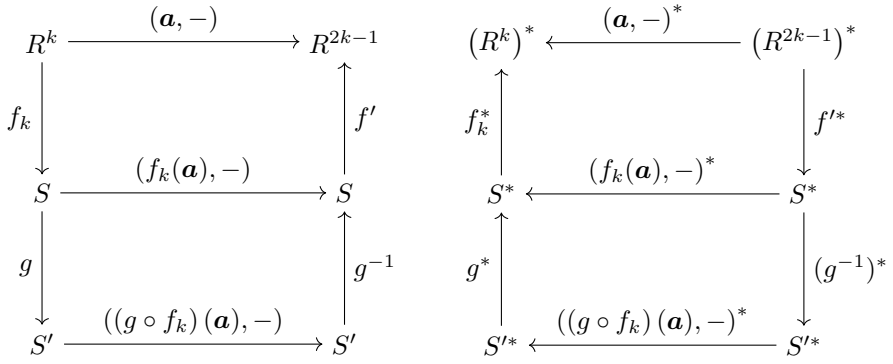
Definition 1. For arbitrary R-algebra monomorphism $f:R[x]_{<n}\to S$, we define $f_k=f|_{R[x]_{<k}}$. Furthermore, let $\mathrm{rev}_{k\to k}:R^k\to R^k$ be the index reversal map and $\mathrm{id}_{m\to n}:R^m\to R^n$ be the inclusion (pad 0's) map for $m\leq n$. The TMVP associated with f refers to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

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

We call $(f^{-1})^*\circ\mathrm{id}_{(2k-1)\to n}$ split–matrix, $f_k(\boldsymbol{a})$ split–vector, $(f_k(\boldsymbol{a}),-)^*$ base multiplication, and f_k^* interpolation. If $n=2k-1$, $f=\mathbf{TC}_{(2k-1)\times(2k-1)}$, then this is the k-way Toeplitz-TC matrix–vector product [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 $(\boldsymbol{a},-)$ as $\mathrm{id}_{n\to(2k-1)}\circ f^{-1}\circ(f_k(\boldsymbol{a}),-)\circ f_k$, take the transpose of $(\boldsymbol{a},-)$, and relate $(\boldsymbol{a},-)^*$ to the Toeplitzation $\mathbf{Toeplitz}_{k\times k}(-)$ and the right-vector-multiplication $(-)(\boldsymbol{a})$. This allows us to convert any fast computation for $(\boldsymbol{a},-)$ into something for $\left(\mathbf{Toeplitz}_{k\times k}(-)\right)(\boldsymbol{a})$. Concretely, we suppose we have $(\boldsymbol{a},-)=f'\circ g^{-1}\circ((g\circ f_k)(\boldsymbol{a}),-)\circ g\circ f_k$ as shown in Figure 5a where $f'=\mathrm{id}_{n\to(2k-1)}\circ f^{-1}$. We transpose all the objects and maps and revert all the arrows as shown in Figure 5b. Since $(\mathbf{Toeplitz}(-))(\boldsymbol{a})=\mathrm{rev}_{k\times k}\circ(\boldsymbol{a},-)^*$, and $(-,\boldsymbol{a})_{R[x]/\langle x^k-\zeta\rangle}=\mathrm{id}_{k\to k}\circ(\mathbf{Toeplitz}(-))(\boldsymbol{a})\circ\mathtt{Expand}_{k\to k,\zeta}$ as shown in Figure 6, we eventually have the following fast computation:

$$(-,\boldsymbol{a})_{R[x]/\langle x^k-\zeta\rangle}=\mathrm{id}_{k\to k}\circ\mathrm{rev}_{k\to k}\circ f_k^*\circ g^*\circ((g\circ f_k)(\boldsymbol{a}),-)^*\circ\left(g^{-1}\right)^*\circ f'^*\circ\mathtt{Expand}_{k\to k,\zeta}.$$



(a) Fast computation for $(\boldsymbol{a},-)$. (b) Fast computation for $(\boldsymbol{a},-)^*$.

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

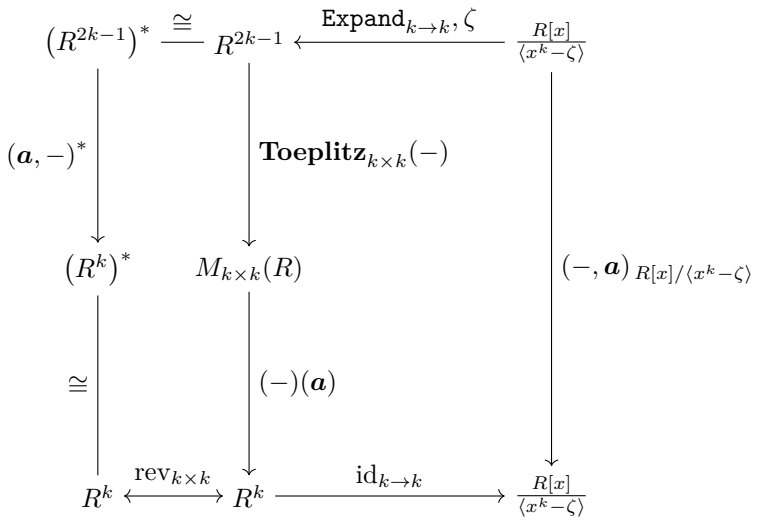


Figure 6: Relations between $(-,\boldsymbol{a})_{R[x]/\langle x^k-1\rangle}$, $(\mathbf{Toeplitz}_{k\times k}(-))(\boldsymbol{a})$, and $(\boldsymbol{a},-)^*$.

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 $R[x]/\langle x^{vn_0n_1}-1\rangle$ 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 as a power of 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 $v = \frac{128}{32} = 4$ (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 $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ for `ntruhps2048677`. We propose two implementations with 16-bit arithmetic mod 65536: (i) `Toom-Cook` implements Toom–Cook with the splitting sequence $5 \to 3 \to 3 \to 2$, and (ii) `Toeplitz-TC` computes the Toeplitz matrix–vector product derived from `Toom-Cook`. Our `Toom-Cook` applies a more aggressive divide-and-conquer than prior works [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 $\mathbf{TC}^{-1}$.

Chosen splitting sequence: Toom–5 → two Toom–3's → Karatsuba. We first zero-pad the size-677 polynomials to size-720 for ease of vectorization and compute in $\mathbb{Z}_{2^{16}}$. Since the coefficient ring of `ntruhps2048677` is $\mathbb{Z}_{2048}$ and $\frac{2^{16}}{2048} = 2^5$, divisions by 2^e for $e = 0, \ldots, 5$ translate into shifting e bits. We choose the splitting sequence $5 \to 3 \to 3 \to 2$. Our `Toom-Cook` consists of one layer of $\mathbf{TC}_{9\times5}$, two layers of $\mathbf{TC}_{5\times3}$'s, one layer of $\mathbf{TC}_{3\times2}$, 675 size-8 schoolbooks, one layer of $\mathbf{TC}^{-1}_{3\times3}$, two layers of $\mathbf{TC}^{-1}_{5\times5}$'s, and one layer of $\mathbf{TC}^{-1}_{9\times9}$. We choose the point sets $\left\{0, \pm1, \pm2, \pm\frac{1}{2}, 3, \infty\right\}$ (cf. Section 3.6) for $\mathbf{TC}_{9\times5}$ and $\{0, \pm1, 2, \infty\}$ for $\mathbf{TC}_{5\times3}$. The interpolation matrices $\mathbf{TC}^{-1}_{9\times9}$, $\mathbf{TC}^{-1}_{5\times5}$, and $\mathbf{TC}^{-1}_{3\times3}$ 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 $3 \to 4 \to 2 \to 2$. The polynomial size goes down to 240 after the Toom-3, 60 after the Toom-4, and 15 after two Karatsuba's. Since 60 is not a multiple of 8, [NG21] basically padded to size-64 polynomials before Karatsuba. In this paper, we instead split via the sequence $5 \to 3 \to 3 \to 2$ down to size-8 schoolbooks. Our evaluation points for Toom-5 has the same precision loss as Toom-4. This is 1 fewer bit than the standard $\{0, \pm1, \pm2, \pm3, 4, \infty\}$. We also avoid zero-padding in vectorization. We merge the two Toom-3 layers (for both $\mathbf{TC}_{5\times3}$ and $\mathbf{TC}^{-1}_{5\times5}$) to reduce memory operations.

Memory optimizations for interpolations. Let $k|n$, $\boldsymbol{g}'$ be a polynomial of degree at least $\frac{2n}{k} - 1$, and $R' = R[x]/\langle \boldsymbol{g}' \rangle$. Recall that $\mathbf{TC}_{(2k-1)\times k}$ computes $R[x]/\left\langle x^{\frac{n}{k}} - y \right\rangle [y] \hookrightarrow R'[y]\Big/\left\langle \prod_{i=0}^{2k-2}(y - s_i) \right\rangle \cong \prod_{i=0}^{2k-2} R'[y]/\langle y - s_i \rangle$ and results in computations in $R[x]/\langle \boldsymbol{g}' \rangle$. After examining the source code, we find that prior works [KRS19, NG21] inverted the steps $\cong$ and $\hookrightarrow$ separately. Algorithm 7 is an illustration. Inverting $\cong$ means applying the interpolation matrix and inverting $\hookrightarrow$ means accumulating the overlapped coefficients while substituting y with $x^{\frac{n}{k}}$ in each of the polynomials in $R[x]/\langle \boldsymbol{g}' \rangle$. We instead alternate between the inversions of $\cong$ and $\hookrightarrow$ to reduce memory operations, *in essence merging two layers of computations.*

Algorithm 7 $\mathbf{TC}^{-1}_{5\times5}$ by [KRS19, NG21].

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

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

Algorithm 8 Our $\mathbf{TC}^{-1}_{5\times5}$.

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

Registers `r[11]`.
`r[0-4]` $= \mathbf{TC}^{-1}_{5\times5}(p_0$`[0]`$, \ldots, p_4$`[0]`$)$
`r[6-10]` $= \mathbf{TC}^{-1}_{5\times5}(p_0$`[2]`$, \ldots, p_4$`[2]`$)$
▷ Memory read.
`r[5]` $= 0$
for $j = \{0, \ldots, 4\}$ **do**
  `r[`$i + 1$`]` $=$ `r[`$i + 1$`]` $+$ `r[`$i + 6$`]`
end for
for $j = \{0, \ldots, 5\}$ **do**
  `c[`$2j$`]` $=$ `r[`j`]`
end for
▷ Memory write.
`r[0-4]` $= \mathbf{TC}^{-1}_{5\times5}(p_0$`[1]`$, \ldots, p_4$`[1]`$)$
▷ Memory read.
for $j \leftarrow 0$ to 4 **do**
  `c[`$2j + 1$`]` $=$ `r[`j`]`
end for
▷ 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 $5 \to 3 \to 3 \to 2$. We call it `Toeplitz-TC`.

Our `Toeplitz-TC` with the splitting sequence $5 \to 3 \to 3 \to 2$. Algorithm 9 describes our `Toeplitz-TC` implementation. Essentially, we regard size-677 polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ as size-720 polynomials. In practice, we zero-pad $\boldsymbol{a}$ and $\boldsymbol{b}$ to length 680 and omit the computations involving the indices $680, \ldots, 719$. Then, we apply one layer of `Toeplitz-TC-5`, two layers of `Toeplitz-TC-3`'s, and one layer of `Toeplitz-TC-2`. Algorithm 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**, $\mathbf{TC}'^{-1*}$, and $\mathbf{TC}'^{*}$, we prefer shifts over multiplications and reuse intermediate values.

Algorithm 9 `Toeplitz-TC` for `ntruhps2048677`.

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

Declare `uint16_t buff_a[9][288], buff_b[9][144], buff_c[9][144]`.
`buff_a[0-8][0-287]` $= \mathbf{TC}^{-1*}_{9\times9}$(`a`)
▷ See Section 4.5 for definition.
`buff_b[0-8][0-143]` $= \mathbf{TC}_{9\times5}$(`b`)
for $i = \{0, \ldots, 8\}$ **do**
  `buff_c[`i`][0-143]` $=$ `Toeplitz-TC-3-3-2(buff_a[`i`][0-287], buff_b[`i`][0-143])`
end for
`c[0-676]` $= \mathbf{TC}^{*}_{9\times5}$(`buff_c[0-8][0-143]`)

Algorithm 10 `Toeplitz-TC-3-3-2`.

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

1: Declare `uint16_t M1[5][96], M2[5][5][3][16].`
2: Declare `uint16_t v1[5][5][16], c1[5][5][16].`
3: $\mathtt{M1[0\text{-}4][0\text{-}95]} = \mathbf{TC}_{5\times5}^{-1*}(\mathtt{M[0\text{-}143][0\text{-}143]})$
4: **for** $i = \{0, \ldots, 4\}$ **do**
5: $\quad \mathtt{M2}[i]\mathtt{[0\text{-}4][0\text{-}2][0\text{-}15]} = \left(\mathbf{TC}_{3\times3}^{-1*} \circ \mathbf{TC}_{5\times5}^{-1*}\right)(\mathtt{M1}[i]\mathtt{[0\text{-}95]})$
6: **end for**
7: $\mathtt{v1[0\text{-}4][0\text{-}4][0\text{-}15]} = (\mathbf{TC}_{5\times3} \circ \mathbf{TC}_{5\times3})(\mathtt{v})$
8: $\mathtt{c1}[i][j]\mathtt{[0\text{-}15]} = \mathbf{TC}_{3\times2}^{*}(\mathtt{M2}[i][j]\mathtt{[0\text{-}2][0\text{-}15]} \cdot \mathbf{TC}_{3\times2}(\mathtt{b1}[i][j]\mathtt{[0\text{-}15]}))$
9: $\mathtt{c[0\text{-}143]} = \left(\mathbf{TC}_{5\times3}^{*} \circ \mathbf{TC}_{5\times3}^{*}\right)(\mathtt{c1[0\text{-}4][0\text{-}4][0\text{-}15]})$

Comparisons to [IKPC22]. [IKPC22] implemented the Toeplitz matrix–vector product with $\mathbf{TC}_{(2k-1)\times k}$ as the underlying monomorphisms on Cortex-M4, but they chose the splitting sequence $4 \to 3 \to 2 \to 2$. We improve the efficiency by applying a more aggressive splitting sequence. For the first layer, we use Toeplitz–TC–5 instead of Toeplitz–TC–4. Both strategies yield 3-bit losses. Although our $\mathbf{TC}_{9\times9}^{-1*}$, $\mathbf{TC}_{9\times5}$, and $\mathbf{TC}_{9\times5}^{*}$ 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 $\mathbf{TC}_{(2k-1)\times(2k-1)}^{-1*}$, $\mathbf{TC}_{(2k-1)\times k}$ and $\mathbf{TC}_{(2k-1)\times k}^{*}$ to compute while we multiply some constants to the precomputed matrices for easier computation. The modified $\mathbf{TC}_{(2k-1)\times(2k-1)}^{-1*}$, $\mathbf{TC}_{(2k-1)\times k}$ and $\mathbf{TC}_{(2k-1)\times k}^{*}$ 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 $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$. 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 $R[x]/\langle (x^{1024}+1)(x^{512}-1)\rangle$ 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 $4591 - 1 = 2 \cdot 3^3 \cdot 5 \cdot 17$, 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 $R[x]/\langle x^{1530} - 1\rangle$). 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 $R[x]/\langle x^{1536} - 1\rangle$. 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 $R[x]/\langle x^{32}+1\rangle$. This leads to 384 size-8 base multiplications defined over trinomial moduli. Our second and third transformations compute the result in $R[x]/\langle x^{1632} - 1\rangle$. We show how to incorporate Rader for radix-17 butterflies and Good–Thomas for the coprime factorization $17 \cdot 3 \cdot 2$. 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 $R[x]/\langle x^{16} \pm \omega_{102}^{i}\rangle$, 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 $x^{32}+1$ into 384 size-8 polynomial rings. In detail, the implementation goes as follows:

1. We first transform the input array `in[761]` into a temporary array `out[3][32][32]`, where `out[`i`][`j`][0-31]` represents the size-32 polynomial in $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{32}+1, u-\omega_3^i, w-x^{2j}\right\rangle}$. Concretely, we combine the permutations of Good–Thomas and Schönhage as `out[`i`][`j`][`k`] = in[`$(16(64i+33j) \bmod 96)+k$`]` if $(16(64i+33j) \bmod 96)+k < 761$ 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 `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 `saddl`, `saddl2`, `ssubl`, `ssubl2`. 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 $(\hat{v}_0, \hat{v}_1, \hat{v}_2)$ of size-32 polynomials (v_1, v_2, v_3) as:

$$\begin{cases} \hat{v}_0 = v_0 + v_1 + v_2, \\ \hat{v}_1 = (v_0 - v_2) + \omega_3(v_1 - v_2), \\ \hat{v}_2 = (v_0 - v_1) - \omega_3(v_1 - v_2). \end{cases}$$

   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 $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{32}+1, u-\omega_3^i, w-x^{2j}\right\rangle}$, we apply Bruun's butterflies. Specifically, the polynomial modulus $x^{32}+1$ splits into:

$$\begin{aligned} x^{32}+1 &= (x^{16}+1229x^2+1)(x^{16}-1229x^2+1) \\ &= (x^8+58x^4+1)(x^8-58x^4+1)(x^8+2116x^4+1)(x^8-2116x^4+1) \end{aligned}$$

[7] Based on https://dougallj.github.io/applecpu/firestorm-simd.html, `smull` has the same throughput as `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 $\mathbf{Bruun}_{1229,1}$ followed by $\mathbf{Bruun}_{58,1}$ and $\mathbf{Bruun}_{2116,1}$ 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 $z^{\frac{64}{8}}+1=z^8+1$.

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 $\forall a\in\mathbb{Z}_{102}, a\equiv e_0(a \bmod 17)+e_1(a \bmod 3)+e_2(a \bmod 2) \pmod{102}$. We rewrite each size-1632 polynomial $\boldsymbol{a}(x)$ as the size-102 polynomial $\boldsymbol{a}'(x^{16})$, introduce $x^{16}\sim y$, and compute FFTs in y. Since $4591-1=102\cdot 45$, there is a principal 102-th root of unity $\omega_{102}\in R$. For evaluating y at powers of ω_{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: `src[1632]`.
Outputs: `poly_NTT[17][3][2][16]`.

1: **for** $i=0,\ldots,1631$ **do**
2: Let $t=i/16$.
3: `poly_NTT[`$t \bmod 17$`][`$t \bmod 3$`][`$t \bmod 2$`][`$i \bmod 16$`]` $=$ `src[`i`]`.
4: **end for**

This gives us $\frac{R'[y,u,w,z]}{\langle y-uwz,u^2-1,w^3-1,z^{17}-1\rangle}$ 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 $\left(x^{16},\omega_2^{i_2},\omega_3^{i_1},\omega_{17}^{i_0}\right)$, we have $\prod_{i_0,i_1,i_2}\frac{R[x]}{\langle x^{16}-\omega_2^{i_2}\omega_3^{i_1}\omega_{17}^{i_0}\rangle}\cong \prod_{i_0,i_1,i_2}\frac{R[x]}{\langle x^{16}-\omega_{102}^{e_0i_0+e_1i_1+e_2i_2}\rangle}$ by choosing $(\omega_2,\omega_3,\omega_{17})=(\omega_{102}^{e_2},\omega_{102}^{e_1},\omega_{102}^{e_0})$. We implement the following two approaches:

- `Good-Rader-outer`. 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.

Algorithm 12 FFTs in $y \sim x^{16}$.
Inputs: `poly_NTT[17][3][2][16]`.
Outputs: `poly_NTT[17][3][2][16]`.
1: **for** $i_3 \in \{0, \ldots, 15\}$ **do**
2: **for** $i_1 \in \{0, 1, 2\}, i_2 \in \{0, 1\}$ **do**
3: `rader-17` (`poly_NTT[0-16]`$[i_1][i_2][i_3]$).
4: **end for**
5: **for** $i_0 \in \{0, \ldots, 16\}$ **do**
6: `radix-(3,2)` (`poly_NTT`$[i_0]$`[0-2][0-1]`$[i_3]$).
7: **end for**
8: **end for**

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.

– Write $x^{16} - \omega_{102}^{e_0 i_0 + e_1 i_1 + e_2 i_2} = x^{16} - (-1)^{i_2} \omega_{51}^{128(e_0 i_0 + e_1 i_1)}$ ($e_2 = 51$, $e_0 i_0 + e_1 i_1$ is even, and $\omega_{102}^2 = \omega_{51} = \omega_{51}^{256}$) and compute the following.
1. Transform $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{16} - \omega_{51}^{128(e_0 i_0 + e_1 i_1)} \right\rangle}$ into $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{8} - \omega_{51}^{64(e_0 i_0 + e_1 i_1)} \right\rangle} \times \frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{8} + \omega_{51}^{64(e_0 i_0 + e_1 i_1)} \right\rangle}$ with radix-2 Cooley–Tukey butterflies. The remaining problems are size-8 weighted convolutions.
2. Transform $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{16} + \omega_{51}^{128(e_0 i_0 + e_1 i_1)} \right\rangle}$ into $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{8} + \omega_{51}^{32(e_0 i_0 + e_1 i_1)} \alpha x^4 + \omega_{51}^{64(e_0 i_0 + e_1 i_1)} \right\rangle}$ and $\frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{8} - \omega_{51}^{32(e_0 i_0 + e_1 i_1)} \alpha x^4 + \omega_{51}^{64(e_0 i_0 + e_1 i_1)} \right\rangle}$ 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 $\boldsymbol{ab}\mathtt{R}^{-1} \in R[x]$ and reductions modulo $x^8 + \alpha x^4 + \beta$.

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 $\frac{6}{16}$ of the computations are discarded at the end. We should store each size-15 polynomials 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 $\frac{1}{16}$ of the computations will be discarded at the end. Our transformations however admit further optimizations via truncation [vdH04], meaning working over $\mathbb{Z}_{4591}[x] \Big/ \left\langle \frac{x^{1632}-1}{x^{96}-1} \right\rangle$ 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`.

`ntruhps2048677`		`ntrulpr761/sntrup761`	
Implementation	Cycles	Implementation	Cycles
[NG21]	58 286	[Haa21]	242 585
`Toeplitz-TC`	26 784	`Good-Rader-outer`	36 266
`Toom-Cook`	37 318	`Good-Rader-Bruun`	39 788
`Good-Thomas`	42 355	`Good-Thomas`	47 696
		`Good-Schönhage-Bruun`	50 398

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 $\frac{25113}{16992} \approx 147.79\%$ cycles of their own `Good-Thomas` [BBCT22, Section 3.4.2] while our `Good-Schönhage-Bruun` takes only $\frac{50398}{47696} \approx 105.67\%$ 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 `ntruhps2048677` and `sntruhp761/ntrulpr761`. Table 15 details the numbers of `Toeplitz-TC` and `Toom-Cook` implementing the "big by big" polynomial multiplications in `ntruhps2048677`. For the "big by big" polynomial multiplications in `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`.

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 $a = a' \diamondsuit a''$ where a' and a'' are called (arithmetic) shares. In higher-order masking, we write it as $a = a' \diamondsuit a'' \diamondsuit \cdots$. We then define all the follow-up computations on the shares. If $\diamondsuit$ is the addtion 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 $\diamondsuit = +$. For correctness, there are constraints on the chosen ring monomorphisms. Suppose the goal is to compute $\boldsymbol{ab} \in \mathbb{Z}_q[x]/\langle \boldsymbol{g} \rangle$ via the secret-sharing $\boldsymbol{ab} = (\boldsymbol{a}' + \boldsymbol{a}'')\boldsymbol{b}$ for some monic polynomial $\boldsymbol{g}$. There are two obvious ways: we compute $\boldsymbol{a}'\boldsymbol{b}$ and $\boldsymbol{a}''\boldsymbol{b}$ in either $\mathbb{Z}_q[x]/\langle \boldsymbol{g} \rangle$ or $\mathbb{Z}[x]/\langle \boldsymbol{g} \rangle$. The former says that any ring monomorphisms from $\mathbb{Z}_q[x]/\langle \boldsymbol{g} \rangle$ can be applied while the latter requires one to carefully bound the maximum values of $\boldsymbol{a}'\boldsymbol{b}$ and $\boldsymbol{a}''\boldsymbol{b}$ in $\mathbb{Z}$. In the context of arithmetic sharing, we can only assume $\boldsymbol{a}', \boldsymbol{a}'' \in \mathbb{Z}_q[x]/\langle \boldsymbol{g} \rangle$ even if we know beforehand that $\boldsymbol{a}' + \boldsymbol{a}'' = \boldsymbol{a}$ 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.

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.

Inversions. For `ntruhps2048677`, we need one inversion in $\mathbb{Z}_{2048}[x]/\langle x^{677} - 1 \rangle$ and one inversion in $\mathbb{Z}_3[x]\Big/\left\langle \frac{x^{677}-1}{x-1} \right\rangle$. The inversion in $\mathbb{Z}_{2048}[x]/\langle x^{677} - 1 \rangle$ consists of one inversion in $\mathbb{Z}_2[x]/\langle x^{677} - 1 \rangle$ and lifting to $\mathbb{Z}_{2048}[x]/\langle x^{677} - 1 \rangle$ with eight polynomial multiplications since the coefficient ring is $\mathbb{Z}_{2048}$. We use the 1-bit form of $\mathbb{Z}_2$ for the inversion over $\mathbb{Z}_2$ without any algorithmic improvements and obtain a 20× speedup, leading to 10.27× overall speedup for the inversion over $\mathbb{Z}_{2048}$. The rest of the improvement for inversion over $\mathbb{Z}_{2048}$ comes from our improved polynomial multiplications (we use `Toeplitz-TC` here). For the inversion in $\mathbb{Z}_3[x]\Big/\left\langle \frac{x^{677}-1}{x-1} \right\rangle$, we use bitsliced implementation and obtain a 8.6× speedup. For `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 $\mathbb{Z}_{4591}$.

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× faster. The main contribution is our optimized inversions in $\mathbb{Z}_2[x]/\langle x^{677}-1\rangle$ and $\mathbb{Z}_3[x]\Big/\left\langle \frac{x^{677}-1}{x-1}\right\rangle$, followed by polynomial multiplications in $\mathbb{Z}_{2048}[x]/\langle x^{677}-1\rangle$ (for lifting) and sorting network. Our encapsulation is 2.48× faster. The main contribution is the sorting network followed by polynomial multiplications. Finally, our decapsulation is 1.77× 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 9: Overall cycles of `ntruhps2048677`.

`ntruhps2048677`			
Operation	Key generation	Encapsulation	Decapsulation
Ref	8 245 039	227 980	331 274
[NG21]	7 686 272	196 526	212 265
`Toeplitz-TC`	1 002 187	79 213	120 208
`Toom-Cook`	1 127 089	88 037	146 422
`Good-Thomas` (with `Toeplitz-TC`)	1 061 870	96 814	152 059
`Good-Thomas` (with `Toom-Cook`)	1 178 983	97 182	165 381

Table 10: Overall cycles of `sntrup761/ntrulpr761`.

`sntrup761`			
Operation	Key generation	Encapsulation	Decapsulation
Ref	273 598 470	29 750 035	89 968 342
`Good-Rader-outer`	6 297 001	144 681	145 777
`Good-Rader-Bruun`	6 333 403	147 977	158 233
`Good-Thomas`	6 340 758	153 465	182 271
`Good-Schönhage-Bruun`	6 345 787	163 305	193 626
`ntrulpr761`			
Operation	Key generation	Encapsulation	Decapsulation
Ref	29 853 635	59 572 637	89 185 030
[Haa21]	775 472	1 150 294	1 417 394
`Good-Rader-outer`	257 382	400 223	435 789
`Good-Rader-Bruun`	260 606	412 629	461 250
`Good-Thomas`	269 590	422 102	471 014
`Good-Schönhage-Bruun`	272 738	436 965	499 559

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 [CHK+21]'s NTTs, and suggest **TC**-4, **TC**-3, 3-way Karatsuba, and Karatsuba. For `ntruhps40961229`, we borrow the transformation size 2560 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.

	`ntruhps2048509`	`ntruhps2048677`	`ntruhrss701`
Target ring	$\frac{\mathbb{Z}_{2048}[x]}{\langle x^{509}-1\rangle}$	$\frac{\mathbb{Z}_{2048}[x]}{\langle x^{677}-1\rangle}$	$\frac{\mathbb{Z}_{8192}[x]}{\langle x^{701}-1\rangle}$
Ring/Toeplitz	$\frac{\mathbb{Z}_{65536}[x]}{\langle x^{1024}-1\rangle}/512$	$\frac{\mathbb{Z}_{65536}[x]}{\langle x^{1440}-1\rangle}/720$	$\frac{\mathbb{Z}_{65536}[x]}{\langle x^{1440}-1\rangle}/720$
Idea			
TC-5	-	✓	✓
TC-4	✓	-	-
TC-3	-	✓	-
3-way **K**	-	-	✓
K	✓	✓	✓
	`ntruhps4096821`	`ntruhps40961229`	`ntruhrss1373`
Target ring	$\frac{\mathbb{Z}_{4096}[x]}{\langle x^{821}-1\rangle}$	$\frac{\mathbb{Z}_{4096}[x]}{\langle x^{1229}-1\rangle}$	$\frac{\mathbb{Z}_{16384}[x]}{\langle x^{1373}-1\rangle}$
Ring/Toeplitz	$\frac{\mathbb{Z}_{65536}[x]}{\langle x^{1728}-1\rangle}/864$	$\frac{\mathbb{Z}_{65536}[x]}{\langle x^{2560}-1\rangle}/1280$	$\frac{\mathbb{Z}_{65536}[x]}{\langle x^{2880}-1\rangle}/1440$
Idea			
TC-5	-	✓	-
TC-4	✓	-	-
TC-3	✓	-	✓
3-way **K**	✓	-	-
K	✓	✓	✓

Next, we go through our suggestions for NTRU Prime parameter sets. We propose four strategies for `ntrulpr857/sntrup857`. The first two strategies operate over $\mathbb{Z}_{5167}[x]/\langle x^{1968}-1\rangle$ 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 $1968 = 41 \cdot 3 \cdot 16$. 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 $\prod_i \mathbb{Z}_{5167}[x]/\langle x^8 \pm \omega_{246}^i\rangle$, 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 $\mathbb{Z}_{5167}[x]/\langle x^{1792}-1\rangle$. We first

pull out the factor 7 with vectorization–friendly Good–Thomas, and applie 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 $\mathbb{Z}_{7177}[x]\Big/\left\langle \frac{x^{2496}-1}{x^{312}+1}\right\rangle$. The remaining problem is the multiplication in the product ring $\prod_i \mathbb{Z}_{7177}[x]/\langle x^{16}-\omega_{312}^i\rangle$. 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 12: Our suggestions for NTRU Prime parameter sets.

	`ntrulpr857/sntrup857`				`ntrulpr1013/sntrup1013`	
Target ring	$\frac{\mathbb{Z}_{5167}[x]}{\langle x^{857}-x-1\rangle}$				$\frac{\mathbb{Z}_{7177}[x]}{\langle x^{1013}-x-1\rangle}$	
Ring	$\frac{\mathbb{Z}_{5167}[x]}{\langle x^{1968}-1\rangle}$		$\frac{\mathbb{Z}_{5167}[x]}{\langle x^{1792}-1\rangle}$		$\frac{\mathbb{Z}_{7177}[x]}{\left\langle \frac{x^{2496}-1}{x^{312}+1}\right\rangle}$	
Idea						
Good–Thomas	✓	✓	✓	✓	✓	✓
Truncation	-	-	-	-	✓	✓
Schönhage	-	-	✓	✓	-	-
Rader	✓	✓	-	-	✓	✓
Bruun	-	-	-	✓	-	-
Vector-by-scalar	✓	-	✓	-	✓	-
Vector-by-vector	-	✓	-	✓	-	✓

	`ntrulpr1277/sntrup1277`	
Target ring	$\frac{\mathbb{Z}_{7879}[x]}{\langle x^{1277}-x-1\rangle}$	
Ring	$\frac{\mathbb{Z}_{7879}[x]}{\langle (x^{2496}+1)(x^{64}-1)\rangle}$	
Idea		
Good–Thomas	✓	✓
Rader	✓	✓
Bruun	-	✓
Toeplitz(**TC**-4)	✓	-
K	-	✓
Vector-by-scalar	✓	-
Vector-by-vector	-	✓

A Proof for the Toeplitz Transformation

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

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

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

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

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

B Examples of Toeplitz Transformations

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

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

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

C Matrix multiplications

Algorithm 13 Inner-product-based matrix–matrix multiplication.

1: **for** $i_0 = 0, \ldots, n_0 - 1$ **do**
2: **for** $i_1 = 0, \ldots, n_1 - 1$ **do**
3: **for** $i_2 = 0, \ldots, n_2 - 1$ **do**
4: $\texttt{C}[i_0]\,[i_1] = \texttt{C}[i_0]\,[i_1] + \texttt{A}[i_0]\,[i_2]\texttt{B}[i_2]\,[i_1]$
5: **end for**
6: ▷ Inner product of the vectors $\texttt{A}[i_0]\,[\texttt{-}]$ and $\texttt{B}[\texttt{-}]\,[i_1]$.
7: **end for**
8: **end for**

Algorithm 14 Outer-product-based matrix–matrix multiplication.

1: **for** $i_2 = 0, \ldots, n_2 - 1$ **do**
2: **for** $i_0 = 0, \ldots, n_0 - 1$ **do**
3: **for** $i_1 = 0, \ldots, n_1 - 1$ **do**
4: $\texttt{C}[i_0]\,[i_1] = \texttt{C}[i_0]\,[i_1] + \texttt{A}[i_0]\,[i_2]\texttt{B}[i_2]\,[i_1]$
5: **end for**
6: **end for**
7: ▷ Outer product of the vectors $\texttt{A}[\texttt{-}]\,[i_2]$ and $\texttt{B}[i_2]\,[\texttt{-}]$.
8: **end for**

Algorithm 15 Inner-product-based matrix–vector multiplication.

1: **for** $i_0 = 0, \ldots, n_0 - 1$ **do**
2: **for** $i_2 = 0, \ldots, n_2 - 1$ **do**
3: $\texttt{C}[i_0] = \texttt{C}[i_0] + \texttt{A}[i_0]\,[i_2]\texttt{B}[i_2]$
4: **end for**
5: ▷ Inner product of the vectors $\texttt{A}[i_0]\,[\texttt{-}]$ and $\texttt{B}[\texttt{-}]$.
6: **end for**

Algorithm 16 Outer-product-based matrix–vector multiplication.

1: **for** $i_2 = 0, \ldots, n_2 - 1$ **do**
2: **for** $i_0 = 0, \ldots, n_0 - 1$ **do**
3: $\texttt{C}[i_0] = \texttt{C}[i_0] + \texttt{A}[i_0]\,[i_2]\texttt{B}[i_2]$
4: **end for**
5: ▷ Outer product of the vectors $\texttt{A}[\texttt{-}]\,[i_2]$ and $\texttt{B}[i_2]$.
6: **end for**

D Detailed Numbers of Polynomial Multiplications

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

`Good-Rader-outer`				`Good-Rader-Bruun`			
Operation	Count	Cycles	Total	Operation	Count	Cycles	Total
`polymul`	-	-	33 218	`polymul`	-	-	37 475
`Good-Rader-17`	24	395	9 480	`Good-Rader-17`	24	407	9 768
`Radix-(3, 2)`	2	2 337	4 674	`Radix-(3, 2)`	2	2 339	4 678
`Weighted-16x16`	1	9 093	9 093	`CT`	2	570	1 140
				`Bruun`	2	838	1 676
				`Weighted-8x8`	12	244	2 928
				`Trinomial-8x8`	12	328	3 936
				$\texttt{CT}^{-1}$	1	592	592
				$\texttt{Bruun}^{-1}$	1	989	989
				`Weighted-16x16`	1	1 019	1 019
$\texttt{Radix-(3, 2)}^{-1}$	1	2 337	2 337	$\texttt{Radix-(3, 2)}^{-1}$	1	2 341	2 341
$\texttt{Good-Rader-17}^{-1}$	12	532	6 360	$\texttt{Good-Rader-17}^{-1}$	12	543	6 516

Table 14: Detailed numbers of `Good-Thomas` for $\mathbb{Z}_{q'}[x]/\langle x^{1536}-1\rangle$.

Operation	`ntruhps2048677`	`sntrup761/ntrulpr761`
`polymul`	42 355	47 696
`NTT` ($\times 2$)	5 502	
`basemul`	21 444	
`iNTT`	8 884	
`Rq_reduce`	547	5 102

Table 15: Detailed cycle counts of `Toeplitz-TC` and `Toom-Cook`, splitting $5 \to 3 \to 3 \to 2$.

`Toeplitz-TC`				`Toom-Cook`			
Operation	Count	Cycles	Total cycles	Operation	Count	Cycles	Total cycles
`polymul`	-	-	26 784	`polymul`	-	-	37 278
`tmvp_ittc5`	1	2 387	2 387	`tc5`	2	586	1 172
`tmvp_tc5`	1	565	565				
`tmvp_ittc3`	9	199	1 791	`tc33`	18	147	2 646
`tmvp_ittc32`	9	559	5 031				
`tmvp_tc33`	9	148	1 332				
`tmvp2_8x8`	9	1 389	12 501	`k2`	2	711	1 422
				`schoolbook_8x8`	1	19 122	19 122
				`ik2`	1	2 777	2 777
				`schoolbook_16x16`	1	511	511
`tmvp_ttc33`	9	209	1 881	`itc33`	9	467	4 203
`tmvp_ttc5`	1	821	821	`itc5`	1	3 303	3 303

Table 16: Detailed `Good-Schönhage-Bruun` cycle counts including reducing to $\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}$.

Operation	Count	Cycles	Total cycles
`polymul`	-	-	50 398
`Good-Schönhage-3-2x2`	1	1 708	1 708
`Schönhage-3x2`	3	1 246	3738
`Good-Schönhage-5x2`	1	1 527	1 527
`Radix-3`	1	2 084	2 084
`Bruun`	24	291	6 984
`Trinomial-8x8`	12	1 115	13 380
`Bruun inverse`	12	409	4 908
`Schönhage-2x4 inverse`	3	1 304	3 912
`Good-Schönhage-2-3 inverse`	1	7 653	7 653

E Performance of Inversions, Encoding, and Decoding

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

Operation	Ref	Ours
`ntruhps2048677`		
`poly_Rq_inv`	3 506 621	341 482
`poly_R2_inv`	2 791 906	136 776
`poly_S3_inv`	4 153 823	482 005
`crypto_sort_int32`	104 691	17 819
`sntrup761/ntrulpr761`		
`Rq_recip3`	116 353 545	5 500 466
`R3_recip`	127 578 811	580 494
`Rq_encode`	17 753	2 084
`Rq_decode`	31 715	3 914
`Rounded_encode`	14 707	3 145
`Rounded_decode`	31 832	3 445
`crypto_sort_uint32`	186 867	19 625

F Detailed Numbers of NTRU Prime

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

sntrup761		ntrulpr761	
Operation	Cycles	Operation	Cycles
crypto_kem_keypair	6 297 001	crypto_kem_keypair	257 382
ZKeyGen	6 241 207	ZKeyGen	245 982
		XKeyGen	236 919
KeyGen	6 190 454	KeyGen	109 574
Rq_recip3	5 500 466		
R3_recip	580 494		
Rq_mult_small	36 266	Rq_mult_small	36 266
sort	197 99	sort	19 207
randombytes	88 678	randombytes	41 193
		aes	127 837
Rq_encode	2 084	Rounded_encode	3 145
sha2	13 626	sha2	12 507
crypto_kem_enc	144 681	crypto_kem_enc	400 223
ZEncrypt	43 978	ZEncrypt	375 021
		XEncrypt	366 654
Encrypt	36 820	Encrypt	76 034
Rq_mult_small	36 266	Rq_mult_small (2×)	2× 36 266
		aes	254 771
		sort	19 478
		sha2	3 610
Rq_decode	3 914	Rounded_decode	3 445
Rounded_encode	3 145	Rounded_encode	3 145
randombytes	43 748		
sha2	31 487	sha2*	27 719
sort	19 625		
crypto_kem_dec	145 777	crypto_kem_dec	435 789
ZDecrypt	79 863	ZDecrypt	44 112
Decrypt	75 517	XDecrypt (defined as Decrypt)	39 702
Rq_mult_small	36 266	Rq_mult_small	36 266
R3_mult	37 652		
Rounded_decode	3 445	Rounded_decode	3 445
ZEncrypt	43 978	ZEncrypt	375 021
sha2	19 901	sha2*	18 793

* The numbers of `sha2` cycles of `XEncrypt` are included.

G Detailed Numbers of NTRU

Table 19: Detailed performance numbers of `ntruhps2048677` with `Toeplitz-TC`. Only performance-critical subroutines are shown.

Operation	Cycles
`crypto_kem_keypair`	1 002 187
`owcpa_keypair`	990 579
`poly_S3_inv`	482 005
`poly_Rq_mul`(×5)	5× 26 784
`poly_Rq_inv`	341 482
`poly_R2_inv`	136 776
`poly_Rq_mul`(×8)	8× 26 784
`sort`	17 819
`randombytes`	12 054
`crypto_kem_enc`	79 213
`owcpa_enc`	32 501
`poly_Rq_mul`	26 784
`randombytes`	13 023
`sort`	18 040
`sha3`	5 148
`crypto_kem_dec`	120 208
`owcpa_dec`	100 842
`poly_Rq_mul`(×2)	2× 26 784
`poly_S3_mul`	28 341
`sha3`	18 867

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. NISTIR8413 – status report on the second round of the nist post-quantum cryptography standardization process, September 2022. https://doi.org/10.6028/NIST.IR.8413-upd1.

[AB74] Ramesh C. Agarwal and Charles S. Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974.

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733.

[AHY22] Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 349–371, 2022.

[ARM15] ARM. *Cortex-A72 Software Optimization Guide*, 2015. https://developer.arm.com/documentation/uan0016/a/.

[ARM21] ARM. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*, 2021. https://developer.arm.com/documentation/ddi0487/gb/?lang=en.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO 1986*, LNCS, pages 311–323. SV, 1986.

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntruprime.cr.yp.to/.

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 845–862, 2022.

[BC87] Joel V Brawley and Leonard Carlitz. Irreducibles and the composed product for polynomials over a finite field. *Discrete Mathematics*, 65(2):115–139, 1987.

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 2001.

[BGM93] Ian F Blake, Shuhong Gao, and Ronald C Mullin. Explicit Factorization of $x^{2^k}+1$ over $\mathbb{F}_p$ with Prime $p \equiv 3 \bmod 4$. *Applicable Algebra in Engineering, Communication and Computing*, 4(2):89–94, 1993.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid scalar/vector implementations of Keccak and SPHINCS+ on AArch64. *Cryptology ePrint Archive*, 2022.

[Bru78] Georg Bruun. z-transform DFT Filters and FFT's. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):56–63, 1978.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8298.

[CA69] Stephen A Cook and Stål O Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntru.org/.

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8791.

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[DV78] Eric Dubois and A Venetsanopoulos. A New Algorithm for the Radix-3 FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(3):222–225, 1978.

[FP07] Franz Franchetti and Markus Puschel. SIMD Vectorization of Non-Two-Power Sized FFTs. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, volume 2, 2007.

[Goo58] I. J. Good. The Interaction Algorithm and Practical Fourier Analysis. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):361–372, 1958.

[Haa21] Jasper Haasdijk. Optimizing NTRU LPRime on the ARM Cortex - A72, 2021. https://github.com/jhaasdijk/KEMobi.

[Hwa22] Vincent Bert Hwang. Case Studies on Implementing Number–Theoretic Transforms with Armv7-M, Armv7E-M, and Armv8-A. Master's thesis, 2022. https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A.

[HY22] Chenar Abdulla Hassan and Oğuz Yayla. Radix-3 NTT-Based Polynomial Multiplication for Lattice Based Cryptography. *Cryptology ePrint Archive*, 2022.

[IKPC20] Írem Keskinkurt Paksoy and Murat Cenk. TMVP-based Multiplication for Polynomial Quotient Rings and Application to Saber on ARM Cortex-M4. *Cryptology ePrint Archive*, 2020. https://eprint.iacr.org/2020/1302.

[IKPC22] Írem Keskinkurt Paksoy and Murat Cenk. Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication. 2022. https://eprint.iacr.org/2022/300.

[Jac12] Nathan Jacobson. *Basic Algebra I*. Courier Corporation, 2012.

[KMS22] Stefan Kölbl, Rafael Misoczki, and Sophie Schmieg. Securing tomorrow today: Why google now protects its internal communications from quantum threats, November 2022. https://cloud.google.com/blog/products/identity-security/why-google-now-uses-post-quantum-cryptography-for-internal-comms.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145(2), pages 293–294, 1962.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates. In *International Conference on Applied Cryptography and Network Security*, pages 281–301. Springer, 2019.

[KSSW] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. PQClean. https://github.com/PQClean.

[Mey96] Helmut Meyn. Factorization of the Cyclotomic Polynomial $x^{2^n}+1$ over Finite Fields. *Finite Fields and Their Applications*, 2(4):439–442, 1996.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8550.

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985.

[Mur96] Hideo Murakami. Real-valued fast discrete Fourier transform and cyclic convolution algorithms of highly composite even length. In *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, volume 3, pages 1311–1314, 1996.

[MVdO14] FE Martínez, CR Vergara, and L Batista de Oliveira. Explicit Factorization of $x^n-1 \in \mathbb{F}_q[x]$. *arXiv preprint arXiv:1404.6281*, 2014.

[NG21] Duc Tri Nguyen and Kris Gaj. Optimized Software Implementations of CRYSTALS-Kyber, NTRU, and Saber Using NEON-Based Special Instructions of ARMv8, 2021. Third PQC Standardization Conference.

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography.

[Nus80] Henri Nussbaumer. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[Rad68] Charles M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977.

[SKS+21] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Kyber on ARM64: compact implementations of Kyber on 64-bit ARM Cortex-A processors. Cryptology ePrint Archive, Report 2021/561, 2021. https://eprint.iacr.org/2021/561.

[Too63] Andrei L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3(4), pages 714–716, 1963.

[TW13] Aleksandr Tuxanidy and Qiang Wang. Composed products and factors of cyclotomic polynomials over finite fields. *Designs, codes and cryptography*, 69(2):203–231, 2013.

[vdH04] Joris van der Hoeven. The truncated Fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 290–296, 2004.

[Win80] Shmuel Winograd. *Arithmetic Complexity of Computations*, volume 33. Siam, 1980.

[WY21] Yansheng Wu and Qin Yue. Further factorization of $x^n - 1$ over a finite field (II). *Discrete Mathematics, Algorithms and Applications*, 13(06):2150070, 2021.

[WYF18] Yansheng Wu, Qin Yue, and Shuqin Fan. Further factorization of $x^n - 1$ over a finite field. *Finite Fields and Their Applications*, 54:197–215, 2018.