SoK: Polynomial Multiplications for Lattice-Based Cryptosystems

Vincent Hwang

Max Planck Institute for Security and Privacy, Bochum, Germany

Abstract. We survey various mathematical tools used in software works multiplying polynomials in

$$\frac{\mathbb{Z}_q[x]}{\langle x^n - \alpha x - \beta \rangle}.$$

In particular, we survey implementation works targeting polynomial multiplications in lattice-based cryptosystems Dilithium, Kyber, NTRU, NTRU Prime, and Saber with instruction set architectures/extensions Armv7-M, Armv7E-M, Armv8-A, and AVX2.

There are three emphases in this paper: (i) modular arithmetic, (ii) homomorphisms, and (iii) vectorization. For modular arithmetic, we survey Montgomery, Barrett, and Plantard multiplications. For homomorphisms, we survey (a) various homomorphisms such as Cooley–Tukey FFT, Bruun's FFT, Rader's FFT, Karatsuba, and Toom–Cook; (b) various algebraic techniques for adjoining nice properties to the coefficient rings, including injections, Schönhage's FFT, Nussbaumer's FFT, and localization; and (c) various algebraic techniques related to the polynomial moduli, including twisting, composed multiplication, evaluation at ∞, Good–Thomas FFT, truncation, incomplete transformation, and Toeplitz matrix-vector product. For vectorization, we survey the relations between homomorphisms and the support of vector arithmetic. We then go through several case studies: We compare the implementations of modular multiplications used in Dilithium and Kyber, explain how the matrix-to-vector structure was exploited in Saber, and review the design choices of transformations for NTRU and NTRU Prime with vectorization. Finally, we outline several interesting implementation projects.

Keywords: Lattice-based cryptography · Polynomial multiplication · Modular arithmetic · Homomorphism · Vectorization

1 Introduction

This paper surveys various ways multiplying polynomials in the ring

$$\frac{\mathbb{Z}_q[x]}{\langle x^n - \alpha x - \beta \rangle}.$$

We aim at surveying various mathematical tools that are frequently used in highly optimized assembly implementations. In particular, we survey how to multiply polynomials in Dilithium, Kyber, NTRU, NTRU Prime, and Saber on the processors Cortex-M3, Cortex-M4, Cortex-A72, and Haswell. All of the polynomial multiplications fall into the case $\mathbb{Z}_q[x]/\langle x^n - \alpha x - \beta \rangle$: We have $\mathbb{Z}_{8380417}[x]/\langle x^{256} + 1 \rangle$ in Dilithium, $\mathbb{Z}_{3329}[x]/\langle x^{256} + 1 \rangle$ in Kyber, $\mathbb{Z}_{2^k}[x]/\langle x^n - 1 \rangle$ with prime n in NTRU, $\mathbb{Z}_q[x]/\langle x^p - x - 1 \rangle \cong \mathbb{F}_{q^p}$ in NTRU Prime, and $\mathbb{Z}_{2^{13}}[x]/\langle x^{256} + 1 \rangle$ in Saber. We refer to [ABD+20a, ABD+20b, CDH+20, BBC+20, DKRV20] for the specifications.

E-mail: vincentvbh7@gmail.com (Vincent Hwang)

1.1 Emphases

This paper is written with three emphases: (i) modular arithmetic, (ii) homomorphisms of algebraic structures, and (iii) vectorization.

1.1.1 Modular Arithmetic

We first survey various modular arithmetic computing representatives of elements in $\mathbb{Z}_q$. Let $\mathtt{R}$ be a power of two with exponent a power of two and $q \leq \mathtt{R}$. We call $\log_2 \mathtt{R}$ the width or precision of arithmetic. We only need the cases $\mathtt{R} = 2^{16}, 2^{32}$ in this paper. If q is a power of two, then reduction modulo q can be implemented as reduction modulo $\mathtt{R}$ and logical and $\mathtt{\&}$.

If q is not a power of two, then there are two cases: q is an even number with an odd factor, or q is an odd number. We leave the discussion of even q with an odd factor to future work since it is not used in the interested implementations of this paper.

Let's assume q is odd. For $a, b \in \mathbb{Z}_{\mathtt{R}}$, there are many ways to compute $c \in \mathbb{Z}_{\mathtt{R}}$ with $c \equiv ab \pmod{q}$. The requirement $c \in \mathbb{Z}_{\mathtt{R}}$ is to ensure that everything we have at the end can be passed to successive computations with the same width of arithmetic. In practice, we prefer $c \in \mathbb{Z}_{\mathtt{B}}$ with $q \leq \mathtt{B} \leq \mathtt{R}$ for $\mathtt{B}$ reasonably close to q. Montgomery multiplication [Mon85] achieves $\mathtt{B} = 2q$ and Plantard multiplication [Pla21] achieves $\mathtt{B} = q + 1$. Both modular multiplications come with multiplicative forms by design. Barrett reduction [Bar86] effectively achieves $\mathtt{B} = 2q$ with $b = 1$ for the same $\mathtt{R}$, and $\mathtt{B} = q$ while replacing $\mathtt{R}$ with sufficiently large $2^k\mathtt{R}$. [BHK+22b] later introduced Barrett multiplication – a multiplicative form of Barrett reduction – and showed that its range is the same as Montgomery multiplication. They introduced the notion "integer approximation" $[\![\,]\!]$ mapping a real number to an integer with a difference bounded by 1, and defined $\bmod^{[\![]\!]}$ as

$$\forall a \in \mathbb{Z}, q \left[\!\!\left[\frac{a}{q}\right]\!\!\right] = a - a \bmod^{[\![]\!]} q.$$

[BHK+22b] established a correspondence between Montgomery and Barrett multiplications. While Montgomery multiplication is considered exact, Barrett multiplication encompasses various multiplication instructions by interpreting them as high-products (multiplication instructions returning the high parts) with integer approximations. Recently, [HKS23] showed that relaxing the condition on $[\![]\!]$ allows efficient Barrett multiplication on microcontrollers with limited multiplication instructions.

1.1.2 Homomorphisms of Algebraic Structures

This paper involves several notions of algebraic structures and their homomorphisms. An algebraic structure is a set $\mathcal{A}$ of elements equipped with finitely many operations on $\mathcal{A}$. In this paper, there are always identity elements for the operations. Homomorphisms are structure-preserving maps between two algebraic structures – a homomorphism $\eta : \mathcal{A} \to \mathcal{B}$ must satisfy that

$$\forall a, b \in \mathcal{A}, \eta(a \cdot_{\mathcal{A}} b) = \eta(a) \cdot_{\mathcal{B}} \eta(b)$$

for $\cdot_{\mathcal{A}}$ and $\cdot_{\mathcal{B}}$ same type of operations. We call η a monomorphism if it is injective. Common algebraic structures are rings, modules, and associative algebras. Associative algebras are algebraic structures that are modules and rings at the same time. For simplicity, we call associative algebra an algebra.

Let R be a unital commutative ring. This paper surveys various algebra homomorphisms implementing the polynomial ring multiplication of $R[x]/\langle x^n - \alpha x - \beta \rangle$ as an algebra. $R = \mathbb{Z}_q$ is a special case, but the survey of homomorphisms considers arbitrary unital commutative rings. Since algebra homomorphisms are ring and module homomorphisms by definition, we can view them in both ways. In this paper, we always view algebra

homomorphisms as module homomorphisms. Suppose we find a way to decompose an algebra homomorphism η into a composition of module homomorphisms:

$$\cdots \circ \eta_{i+2} \circ \eta_{i+1} \circ \eta \circ \cdots \circ \eta_{j+2} \circ \eta_{j+1} \circ \eta_j \circ \cdots .$$

We now identify the series of module homomorphisms resulting in ring homomorphisms. Such series allow us to multiply the homomorphic images of multiplicands. In practice, this is an interactive process with the target platform – we first write an algebra homomorphism as a composition of module homomorphisms, implement a series of module homomorphisms giving a ring homomorphism, and decide if we want to implement the remaining module homomorphisms, or halt and multiply the images. Therefore, thoroughly examining the efficiency of module homomorphisms in practice is crucial.

We first survey various homomorphisms and their definability on $R[x]/\langle x^n - \beta \rangle$ (the case $\alpha = 0$), including Cooley–Tueky, Bruun's, Good–Thomas, and Rader's fast Fourier transforms (FFTs). In practice, $R[x]/\langle x^n - \beta \rangle$ does not always exhibit nice properties for FFTs. One usually embed $R[x]/\langle x^n - \beta \rangle$ and generally $R[x]/\langle x^n - \alpha x - \beta \rangle$ into $R'[x]/\langle \boldsymbol{g} \rangle$ admitting fast computations. We survey various techniques for deciding the pair $(R', \boldsymbol{g})$.

For the coefficient ring injection $R \hookrightarrow R'$, we consider the integer ring and polynomial ring cases. For the integer ring case, we search for an R' containing suitable roots of unity or allowing us to adjoin the inverses of some integers. For the polynomial ring case, we assume $\alpha = 0$ and start by replacing R with a polynomial ring $R' = R[x]/\langle \boldsymbol{h} \rangle$. Schönhage's FFT converts the relation $\boldsymbol{h} \sim 0$ into an identity defining roots of unity and splits $R'[x]/\langle x^n - \beta \rangle$ accordingly, and Nussbaumer's FFT factors $R[x]/\langle \boldsymbol{h} \rangle$ using the roots of unity already defined by the relation $x^n \sim \beta$.

For the choice of $\boldsymbol{g}$, the most straightforward case when $\alpha = 0$ is twisting and composed multiplication replacing $x^n \pm \beta$ with $\boldsymbol{g} = x^n \pm 1$. This requires the invertibility of β and the existence of $\beta^{\frac{1}{n}}$. The most complicated case is to choose a $\boldsymbol{g}$ with $\deg(\boldsymbol{g}) \geq 2n-1$ in order to have a "forgetful map[1]" from $R[x]/\langle x^n - \alpha x - \beta \rangle$ to $R[x]/\langle \boldsymbol{g} \rangle$. A straightforward approach is to embed $R[x]/\langle x^n - \alpha x - \beta \rangle$ into $R[x]/\langle \boldsymbol{g} \rangle$ as an algebra, multiply in $R[x]/\langle \boldsymbol{g} \rangle$, and reduce modulo $x^n - \alpha x - \beta$. Frequent choices of $\boldsymbol{g}$ for such an approach are $x^n - 1$ and its factors for FFTs, and $\prod_i (x - s_i)$ for Karatsuba and Toom–Cook where $\{s_i\} \subset \mathbb{Q} \cup \{\infty\}$. An alternative approach is the Toeplitz transformation via dual modules. [IKPC20, IKPC22] showed that reduction modulo $x^n - \beta$ becomes free, and [CCHY23] demonstrated its benefit on register scheduling and permutation whenever there are vector-by-scalar multiplication instructions.

1.1.3 Vectorization

Vectorization is another important topic for highly-optimized assembly implementations. Common vector instruction sets are Neon on Arm Cortex-A processors and SSE/AVX/AVX2/AVX512 on Intel processors. Usually, vector instructions perform a wide variety of vector-by-vector arithmetic, including additions, subtractions, multiplications, shift operations, and variants. Let v be the number of algebraic elements contained in a vector register. In this paper, an algebraic element is an element in the coefficient ring $\mathbb{Z}_q$ where $q < 2^{16}$ or $q < 2^{32}$. An obvious way to design an easily vectorizable polynomial transformation f is to define it in $R[x]/\langle \boldsymbol{g}(x^v) \rangle \cong R[y]/\langle x^v - y, \boldsymbol{g}(y) \rangle$. This leads to the first question:

- How to identify a suitable $R[x]/\langle \boldsymbol{g}(x^v) \rangle$ defining an easily vectorizable f?

After applying such an f, one usually interleaves v computing tasks of the same kind and computes accordingly to fully exploit the vectorization feature. We can vectorize

[1] We borrow this usage from category theory.

everything if the number of computing tasks is a multiple of v. On the other hand, if there are some leftover computing tasks, we can still interleave the leftovers with don't-cares and extract the results. This comes with the expense of computing the results of don't-cares. Our second question is the following:

- How to identify a suitable f giving v' subproblems of the same kind where $v|v'$?

In addition to vector-by-vector instructions, there are also vector-by-scalar counterparts in Neon[2]. [CCHY23] showed the benefit of vector-by-scalar multiplication instructions for small-dimensional Toeplitz matrix-vector products and explained how to turn arbitrary polynomial transformations defined in $R[x]$ into a computation for $R[x]/\langle x^n - \beta \rangle$ resulting small-dimensional Toeplitz matrix-vector products. The case $R[x]/\langle x^n - \alpha x - 1 \rangle$ was implied by [FH07], and the case $R[x]/\langle x^n - \alpha x - \beta \rangle$ was discovered by [Yan23].

1.2 Artifact

We are preparing C implementations for each of the ideas reviewed in this paper and will make them pubicly available soon for referential purposes (we believe the material shown in the paper is self-contained but additional examples with actual programs will be helpful).

1.3 Related Works

There are many survey works targeting polynomial multiplications. We recommand [Win80, Nus82, DV90, Ber01, Ber08] for the underlying mathematical ideas, and [LZ22] for the applications to lattice-based cryptography.

1.4 Assumed Knowledge

This paper assumes that readers have some basic understandings of commutative algebra. We list the following key words and corresponding references: rings from [Jac12a, Section 2] and [Bou89, Section 8, Chapter I], modules from [Jac12a, Section 3] and [Bou89, Section 1, Chapter II], dual modules from [Jac12b, Example 11, Section 1.3] and [Bou89, Section 2, Chapter II], tensor products of modules from [Jac12b, Section 3.7] and [Bou89, Section 3, Chapter II], associative algebras from [Jac12a, Section 7], [Jac12b, Section 3.9], and [Bou89, Sections 1 and 2, Chapter III], and tensor products of algebras from [Jac12b, Section 3.9] and [Bou89, Section 4].

1.5 Structure of This Paper

This paper is structured as follows: Section 2 reviews the modular arithmetic Montgomery, Barrett, and Plantard multiplications. Section 3 reviews several techniques requiring the Chinese remainder theorem for polynomial rings.

We then gradually review techniques for addressing technical limitations arising from the polynomial rings used in practice and the target platform. Section 4 reviews several techniques for replacing the coefficient rings with the ones containing suitable roots of unity and inverses of integers. Section 5 turns the focus to the polynomial moduli. Section 6 relates vector-by-vector multiplication instructions to the notions "vectorization-friendliness" and "permutation-friendliness", and vector-by-scalar multiplication instructions to small-dimensional Toeplitz matrix-vector products.

Section 7 goes through several case studies: We compare the deployment of Montgomery and Barrett multiplication for Dilithium and Montgomery and Plantard multiplications

[2]Additionally, RISC-V "V" vector extension also includes vector-by-vector and vector-by-scalar instructions.

for Kyber. We also survey homomorphism caching and its application to Saber. For vectorization, we survey the application of Toeplitz matrix-vector products to NTRU and the designs of vectorization- and permutation-friendly polynomial multipliers for NTRU Prime. Section 8 gives a brief overview of the recent advances. Finally, Section 9 outlines several possible implementation projects for future works.

2 Modular Arithmetic

We first survey various modular arithmetic. Section 2.1 generalizes integer approximations for unifying the modular arithmetic used in relevant works. Section 2.2 reviews Montgomery multiplication, Section 2.3 reviews Barrett multiplication, and Section 2.4 reviews Plantard multiplication.

2.1 Integer Approximations

For a real number $\delta > 0$ and an integer-valued function $[\![]\!] : \mathbb{R} \to \mathbb{Z}$, we call $[\![]\!]$ a δ-integer-approximation [BHK+22b, HKS23] if

$$\forall r \in \mathbb{R}, |[\![r]\!] - r| \leq \delta.$$

To avoid clutter, we call $[\![]\!]$ an integer approximation as long as there is a δ such that $[\![]\!]$ is a δ-integer-approximation. Furthermore, for a positive integer $q \in \mathbb{Z}_{>0}$, we define the corresponding modular reduction $\bmod^{[\![]\!]} q : \mathbb{Z} \to \mathbb{Z}$ as

$$\forall z \in \mathbb{Z}, z \bmod^{[\![]\!]} q = z - \left[\!\left[\frac{z}{q}\right]\!\right] q$$

and $\left|\bmod^{[\![]\!]} q\right| = \max_{z \in \mathbb{Z}} \left|z \bmod^{[\![]\!]} q\right|$. By definition, we have

$$\forall z \in \mathbb{Z}, \begin{cases} \left[\!\left[\frac{z}{q}\right]\!\right] q = z - z \bmod^{[\![]\!]} q, \\ z \equiv z \bmod^{[\![]\!]} q \pmod q. \end{cases}$$

We illustrate the idea with two examples: the floor function $\lfloor\rfloor$ and the rounding function $\lfloor\rceil := r \mapsto \lfloor r + \frac{1}{2} \rfloor$.

The floor function $\lfloor\rfloor$. The floor function $\lfloor\rfloor$ maps a real number to the largest integer lower-bounding the real number. Therefore, for an $r \in \mathbb{R}$, we have $r - 1 < \lfloor r \rfloor \leq r \longrightarrow |\lfloor r \rfloor - r| \leq 1$ and find $\lfloor\rfloor$ a 1-integer-approximation. This function is commonly accompanied by unsigned arithmetic. We denote the corresponding modulo reduction as $\bmod^{\lfloor\rfloor} = \bmod^{+}$ in this case.

The rounding function $\lfloor\rceil$. For the round function $\lfloor\rceil$ and an $r \in \mathbb{R}$, since $\lfloor r \rceil = \lfloor r + \frac{1}{2} \rfloor$ and $r - \frac{1}{2} < \lfloor r + \frac{1}{2} \rfloor \leq r + \frac{1}{2}$, we find $|\lfloor r \rceil - r| \leq \frac{1}{2}$ and $\lfloor\rceil$ a $\frac{1}{2}$-integer-approximation. If $\lfloor\rceil$ is used for signed arithmetic, we denote the corresponding modulo reduction as $\bmod^{\lfloor\rceil} = \bmod^{\pm}$.

In this paper, we provide a unified view of Montgomery, Barrett, and Plantard multiplication using the pair $\left([\![]\!], \bmod^{[\![]\!]} q\right)$. Usually, two pairs of integer approximations $\left([\![]\!]_0, \bmod^{[\![]\!]_0} q\right)$ and $\left([\![]\!]_1, \bmod^{[\![]\!]_1 \mathtt{R}}\right)$ are involved where $\left([\![]\!]_0, \bmod^{[\![]\!]_0} q\right)$ refers to the one we really want and $\left([\![]\!]_1, \bmod^{[\![]\!]_1 \mathtt{R}}\right)$ refers to the practically efficient one.

2.2 Montgomery Arithmetic

Let a, b be integers. We wish to compute $ab \bmod [\![]_0 q$ for a $\bmod [\![]_0 q$ with odd q. Montgomery multiplication [Mon85, Sei18] computes a representative of $ab \bmod [\![]_1 q$ with possible scaling. Observe that $ab + \left(ab\left(-q^{-1}\right) \bmod [\![]_1 \mathtt{R}\right) q$ is equivalent to 0 modulo $\mathtt{R}$ and ab modulo q, we have

$$\frac{ab + \left(ab\left(-q^{-1}\right) \bmod [\![]_1 \mathtt{R}\right) q}{\mathtt{R}} \equiv ab\mathtt{R}^{-1} \pmod{q}.$$

To see why this is a reduction, we bound the range as follows:

$$\left|\frac{ab + \left(ab\left(-q^{-1}\right) \bmod [\![]_1 \mathtt{R}\right) q}{\mathtt{R}}\right| \leq \frac{|ab| + \left|\bmod [\![]_1 \mathtt{R}\right| q}{\mathtt{R}}.$$

There are many ways to mitigate the scaling. A generic way is to perform an additional Montgomery multiplication with $b = \mathtt{R}^2 \bmod [\![]_0 q$ for some $\bmod [\![]_0 q$. If b is known in prior, we can precompute $b\mathtt{R} \bmod [\![]_0 q$ and compute

$$\frac{a\left(b\mathtt{R} \bmod [\![]_0 q\right) + \left(a\left(b\mathtt{R} \bmod [\![]_0 q\right)\left(-q^{-1}\right) \bmod [\![]_1 \mathtt{R}\right) q}{\mathtt{R}} \equiv ab \pmod{q}.$$

Since $b\mathtt{R} \bmod [\![]_0 q$ is now bounded by $\left|\bmod [\![]_0 q\right|$, we have the following bound:

$$\begin{aligned} & \left|\frac{a\left(b\mathtt{R} \bmod [\![]_0 q\right) + \left(a\left(b\mathtt{R} \bmod [\![]_0 q\right)\left(-q^{-1}\right) \bmod [\![]_1 \mathtt{R}\right) q}{\mathtt{R}}\right| \\ \leq \quad & \frac{|a|\left|\bmod [\![]_0 q\right| + \left|\bmod [\![]_1 \mathtt{R}\right| q}{\mathtt{R}}. \end{aligned}$$

For unsigned arithmetic with $\bmod [\![]_1 \mathtt{R} = \bmod^+ \mathtt{R}$ and $\bmod [\![]_0 q = \bmod^+ q$, the range is

$$\frac{|a|\left|\bmod [\![]_0 q\right| + \left|\bmod [\![]_1 \mathtt{R}\right| q}{\mathtt{R}} \leq q\left(1 + \frac{|a|}{\mathtt{R}}\right).$$

For signed arithmetic with $\bmod [\![]_1 \mathtt{R} = \bmod^{\pm} \mathtt{R}$ and $\bmod [\![]_0 q = \bmod^{\pm} q$, the resulting range is

$$\frac{|a|\left|\bmod [\![]_0 q\right| + \left|\bmod [\![]_1 \mathtt{R}\right| q}{\mathtt{R}} \leq \frac{q}{2}\left(1 + \frac{|a|}{\mathtt{R}}\right).$$

Historical review. [Mon85] proposed the unsigned Montgomery multiplication, and [Sei18] later proposed the signed variant along with the subtractive variant:

$$\frac{ab - \left(abq^{-1} \bmod {}^{\pm}\mathtt{R}\right) q}{\mathtt{R}}.$$

The benefit of the subtractive variant is that $(ab \bmod {}^{\pm}\mathtt{R}) - \left(\left(abq^{-1} \bmod {}^{\pm}\mathtt{R}\right) q \bmod {}^{\pm}\mathtt{R}\right) = 0$ whereas $(ab \bmod {}^{\pm}\mathtt{R}) + \left(\left(ab - q^{-1} \bmod {}^{\pm}\mathtt{R}\right) q \bmod {}^{\pm}\mathtt{R}\right) = 0$ or $\mathtt{R}$ as integers. The former allows us to compute as follows:

$$\left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{\left(abq^{-1} \bmod {}^{\pm}\mathtt{R}\right) q}{\mathtt{R}} \right\rfloor.$$

This replaces double-size products with high-products. See [KAK96, KA98] for the multi-limb versions.

2.3 Barrett Arithmetic

Let $[\![]\!]_0, [\![]\!]_1$ be integer approximations. Barrett multiplication computes

$$ab - \left[\!\!\left[\frac{a \left[\!\!\left[\frac{b\mathtt{R}}{q} \right]\!\!\right]_0}{\mathtt{R}} \right]\!\!\right]_1 q \equiv ab \pmod{q}.$$

Obviously, this is a representative of $ab \bmod q$. The only question is if the resulting range falls into the data width. [BHK+22b] showed the following correspondence

$$ab - \left[\!\!\left[\frac{a \left[\!\!\left[\frac{b\mathtt{R}}{q} \right]\!\!\right]_0}{\mathtt{R}} \right]\!\!\right]_1 q = \frac{a\left(b\mathtt{R} \bmod^{[\![]\!]_0} q\right) + \left(a\left(b\mathtt{R} \bmod^{[\![]\!]_0} q\right)\left(-q^{-1}\right) \bmod^{[\![]\!]_1} \mathtt{R}\right) q}{\mathtt{R}}$$

and obtained the bound

$$\left| ab - \left[\!\!\left[\frac{a \left[\!\!\left[\frac{b\mathtt{R}}{q} \right]\!\!\right]_0}{\mathtt{R}} \right]\!\!\right]_1 q \right| \leq \frac{|a| \left|\bmod^{[\![]\!]_0} q\right| + \left|\bmod^{[\![]\!]_1} \mathtt{R}\right| q}{\mathtt{R}}.$$

In Appendix A, we prove the correspondence for principal ideal domains. This captures the polynomial ring case with coefficient ring a finite field and is of independent interest.

Comparing Montgomery and Barrett multiplications. Since the absolute value of the result is smaller than $\frac{\mathtt{R}}{2}$ for signed arithmetic ($\mathtt{R}$ for unsigned arithmetic) in practice, we only need to compute $ab \bmod^{\pm}\mathtt{R}$ ($ab \bmod^{+}\mathtt{R}$ for unsigned arithmetic) instead of the full product. Same observation holds for $\left[\!\!\left[\frac{a \left[\!\!\left[\frac{b\mathtt{R}}{q} \right]\!\!\right]_0}{\mathtt{R}} \right]\!\!\right]_1 q$. Therefore, Barrett multiplication only requires one to compute a high-product implementing $\left[\!\!\left[\frac{a \left[\!\!\left[\frac{b\mathtt{R}}{q} \right]\!\!\right]_0}{\mathtt{R}} \right]\!\!\right]_1$ and two low-products multiplying in $\bmod^{\pm}\mathtt{R}$ or $\bmod^{+}\mathtt{R}$. On the other hand, one has to compute two full products (or high-products for the subtractive variant) and one low-product for Montgomery multiplication. [BHK+22b] saved one subtraction with Barrett multiplication since there is a subtractive variant for low-product and not high-product.

Historical review. For unsigned arithmetic, [Bar86] proposed the case $b = 1$, and [Sho] proposed Barrett multiplication for generic b. The signed version and its correspondence to Montgomery multiplication was discovered by [BHK+22b]. Interestingly, [Dhe03] proposed the finite field version. Appendix A proves the correspondence for principal ideal domains, and the impact for finite fields is left for future investigation. Recently, [BHK+22a, Section 2.4] improved the output range for $b \neq 1$ while replacing $\mathtt{R}$ for some $2^k\mathtt{R}$, and [HKS23] furthered the approximation nature of $[\![]\!]_1$ and improved the modular multiplications on microcontrollers.

2.4 Plantard Arithmetic

Recently, [Pla21] proposed an unsigned modular multiplication essentially with precision $2\log_2 \mathtt{R}$. The signed versions were later proposed by [HZZ+22, AMOT22]. For multiplying an integer a by a constant b known in prior, Montgomery multiplication results in the bound $\frac{|a|\left| \bmod^{[\![]\!]_0} q\right| + \left| \bmod^{[\![]\!]_1} \mathtt{R}\right| q}{\mathtt{R}}$. If we replace the precision $\log_2 \mathtt{R}$ with $2\log_2 \mathtt{R}$ and compute with

$$\frac{a\left(b\mathtt{R}^2 \bmod^{[\![]\!]_0} q\right) + \left(a\left(b\mathtt{R}^2 \bmod^{[\![]\!]_0} q\right)\left(-q^{-1}\right) \bmod^{[\![]\!]_1} \mathtt{R}^2\right) q}{\mathtt{R}^2},$$

we have the bound

$$\frac{|a|\left|\bmod^{[\![]\!]_0} q\right| + \left|\bmod^{[\![]\!]_1} \mathtt{R}^2\right| q}{\mathtt{R}^2}.$$

For signed arithmetic with $\left|\bmod^{[\![]\!]_1} \mathtt{R}^2\right| \leq \frac{\mathtt{R}^2}{2}$ and $\left|\bmod^{[\![]\!]_0} q\right| \leq \frac{q}{2}$, the bound is $\frac{q}{2}\left(1 + \frac{|a|}{\mathtt{R}^2}\right)$. In practice, since $|a| \leq \mathtt{R}$ and $q < \mathtt{R}$, the result is stricly smaller than $\frac{q}{2}$, and hence an integer in $\left\{-\frac{q-1}{2}, \ldots, 0, \ldots, \frac{q-1}{2}\right\}$.

We borrow the integer-approximation view from [HKS23] and proceed with [Pla21]'s innovation for implementing the above observation. Suppose we find two integer approximations $[\![]\!]_2$ and $[\![]\!]_3$ implementing:

$$\begin{aligned} & \frac{c + \left(c\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2\right) q}{\mathtt{R}^2} \\ = & \left[\!\!\left[\frac{c + \left(c\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2\right) q}{\mathtt{R}^2} - \frac{c + \left(c\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2 \bmod {}^{[\![]\!]_2} \mathtt{R}\right) q}{\mathtt{R}^2} \right]\!\!\right]_3 \end{aligned}$$

for all $c \in \mathbb{Z}_{\mathtt{R}^2}$, we claim the following:

$$\frac{c + \left(c\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2\right) q}{\mathtt{R}^2} = \left[\!\!\left[\frac{\left[\!\!\left[\frac{c\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2}{\mathtt{R}} \right]\!\!\right]_2 q}{\mathtt{R}} \right]\!\!\right]_3.$$

The proof is left as an exercise[3]. If $c = ab$, we can instead precompute $b\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2$, and apply only two high-products. While $z \mapsto \left[\!\!\left[\frac{zq}{\mathtt{R}}\right]\!\!\right]_3$ is the usual high-product multiplying numbers of precision $\log_2 \mathtt{R}$, the high-product $z \mapsto \left[\!\!\left[\frac{zb\left(-q^{-1}\right) \bmod {}^{[\![]\!]_1} \mathtt{R}^2}{\mathtt{R}}\right]\!\!\right]_2$ requires one to multiply a by a number with precision $2\log_2 \mathtt{R}$. [HZZ+22] identified the use case in Armv7E-M implementing the multiplication instructions `smulw{b, t}`[4], and [AMOT22, Source code 1] implemented the idea when only multiplication instructions with precision $2\log_2 \mathtt{R}$ are available.

3 Basic Algebraic Techniques

We survey several techniques that fall into the paradigm – the Chinese remainder theorem for polynomial rings. We will go through isomorphisms of the form

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

with different choices of $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$. Section 3.1 reviews Cooley–Tukey FFT, Section 3.2 reviews Bruun's FFT, Section 3.3 reviews Rader's FFT, and Section 3.4 reviews Karatsuba and Toom–Cook.

3.1 Cooley–Tukey Fast Fourier Transform

For a positive integer n and an element $\omega \in R$, we call ω an n-th root of unity if $\omega^n = 1$. Furthermore, we call ω a principal n-th root of unity if

$$\forall i \in \{0, \ldots, n-1\}, \sum_j \omega^{ij} = n\delta_{0,i}$$

[3] Hint: cancel out the terms $\frac{c}{\mathtt{R}}$, write the remaing as a multiple of $\frac{q}{\mathtt{R}}$, and rewrite the rest with $[\![]\!]_2$.

[4] `w` stands for a word and `{b, t}` stands for the bottum or the top half-word.

where δ is the Kronecker delta. We denote ω_n for a principal n-th root of unity. For an invertible $\zeta \in R$, the isomorphism $R[x]/\langle x^n - \zeta^n \rangle \cong \prod_{i=0}^{n-1} R[x]/\langle x - \zeta\omega_n^i \rangle$ is called a discrete weighted transform (DWT) [CF94]. If $\zeta = 1$, it is called a discrete Fourier transform (DFT).

Let $n_j = |\mathcal{I}_j|$, $n = \prod_j n_j$. Cooley–Tukey FFT [CT65, CF94] chooses

$$\boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x - \zeta\omega_n^{\sum_j i_j \prod_{l<j} n_l}$$

for $i_j \in \mathcal{I}_j$. Since $\prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x^n - \zeta^n$, we now have a fast transformation for the ring $R[x]/\langle x^n - \zeta^n \rangle$. We call it a cyclic FFT if $\zeta^n = 1$ and a negacyclic FFT if $\zeta^n = -1$.

Conditions for an invertible DWT. There are three defining conditions for an invertible $R[x]/\langle x^n - \zeta^n \rangle \cong \prod_i R[x]/\langle x - \zeta\omega_n^i \rangle$: (i) ζ must be invertible, (ii) there must exist a principal n-th root of unity ω_n, and (iii) n must be invertible in R[5]. For conditions (ii) and (iii), [Pol71] showed that n must be a divisor of $q-1$ if $R = \mathbb{F}_q$ and $p-1$ if $R = \mathbb{Z}_{p^k}$ for a prime p. The latter says that for $R = \mathbb{Z}_m$ with prime factorization $m = \prod_i p_i^{d_i}$, n must divide $\gcd(p_i - 1)$ [Pol71, AB74]. [DV78b, Theorem 4] gave the condition when R is a product of local rings[6], and [Für09, Section 2] explained the conditions for rings and named ω_n a principal n-th root of unity.

Real-world example(s). In Dilithium, one is asked to implement the radix-2 FFT defined on $\mathbb{Z}_{8380417}[x]/\langle x^{256} + 1 \rangle$. Since $x^{256} + 1 = \Phi_{512}(x)$, the defining condition is the same for $\mathbb{Z}_{8380417}[x]/\langle x^{512} - 1 \rangle$. Observe that $8380417 = 2^{13} \cdot 3 \cdot 11 \cdot 31 + 1$, we can define a cyclic FFT with transformation size a divisor of $2^{13} \cdot 3 \cdot 11 \cdot 31$. This gives the isomorphism $\mathbb{Z}_{8380417}[x]/\langle x^{512} - 1 \rangle \cong \prod_i \mathbb{Z}_{8380417}[x]/\langle x - \omega_{512}^i \rangle$ and hence $\mathbb{Z}_{8380417}[x]/\langle x^{256} + 1 \rangle \cong \prod_i \mathbb{Z}_{8380417}[x]/\langle x - \omega_{512}^{2i+1} \rangle$ by choosing $\zeta = \omega_{512}$ (any odd power of ω_{512} works) and $\omega_{256} = \omega_{512}^2$.

3.2 Bruun-Like Fast Fourier Transforms

After the introduction of Cooley–Tukey FFT over complex numbers, many works proposed several optimizations if the input coefficients are real. [Bru78] proposed Bruun's FFT for the power-of-two case, [DH84] proposed split-radix FFT, [Bra84] proposed fast Hartley transform for the discrete Hartley transform (DHT) [Har42][7], [Mur96] generalized Bruun's FFT to arbitrary even sizes, and [JF07, Ber07, LVB07] improved the split-radix FFT.

This section reviews the works [Bru78, Mur96] over complex numbers for historical reasons. However, the actual use case relevant to us are the factorization of cyclotomic polynomials over finite fields [BC87, BGM93, Mey96]. See [TW13, BMGVdO15, WYF18, WY21] for recent progresses on this topic.

The complex case. Let $n_j = |\mathcal{I}_j|$, $n = \prod_j n_j$. Bruun's FFT [Bru78, Mur96] chooses $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$ as follows:

$$\boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x^2 - \left(\zeta\omega_n^{\sum_j i_j \prod_{l<j} n_l} + \zeta^{-1}\omega_n^{-\sum_j i_j \prod_{l<j} n_l}\right) x + 1.$$

[5] Since rings are $\mathbb{Z}$-algebras, a positive integer n is encoded as $\underbrace{1 + \cdots + 1}_{n}$.

[6] A ring with a unique maximal left/right-ideal.

[7] One can derive DFT and DHT from each other with linearly number of arithmetic during post-processing.

Since $\prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} = x^{2n} - (\zeta^n + \zeta^{-n})\, x^n + 1$, we now have a fast transformation for the ring $R[x]/\langle x^{2n} - (\zeta^n + \zeta^{-n})\, x^n + 1\rangle$. For $\zeta = \omega_{4n} \in \mathbb{C}$, this implements the isomorphism $\mathbb{C}[x]/\langle x^{2n}+1\rangle \cong \prod_i \mathbb{C}[x]/\langle x - \omega_{4n}^{1+2i}\rangle$ if we further split into linear factors.

The finite field cases. In this paper, we are interested in the case $R = \mathbb{F}_q$ with $q \equiv 3 \pmod 4$ which relies on the following theorem from [BGM93]:

Theorem 1 ([BGM93])**.** *Let $q \equiv 3 \pmod 4$ be a prime and 2^w be the highest power of $q+1$. For $k < w$, $x^{2^k}+1$ factors into irreducible trinomials $x^2 + \gamma x + 1 \in \mathbb{F}_q[x]$. For $k \geq w$, $x^{2^k}+1$ factors into irreducible trinomials $x^{2^{k-w+1}} + \gamma x^{2^{k-w}} - 1 \in \mathbb{F}_q[x]$.*

Real-world example(s). For the NTRU Prime parameter sets `ntrulpr761/sntrup761`, [HLY24] introduced a fast transformation (`Good-Schönhage-Bruun`) leading to computing in $\mathbb{Z}_{4591}[x]/\langle x^{32}+1\rangle$. Since $4591 \equiv 3 \pmod 4$ and $4591 + 1 = 287 \cdot 2^4$, we can split $\mathbb{Z}_{4591}[x]/\langle x^{32}+1\rangle$ into polynomial rings modulo trinomials of the form $x^4 + \gamma x^2 - 1$. [HLY24] splitted into rings of the form $\mathbb{Z}_{4591}[x]/\langle x^8 + \alpha x^4 + 1\rangle$ for efficiency reasons.

3.3 Rader's Fast Fourier Transform

Let n be a positive integer, $\mathcal{I} = \{0,\ldots,n-1\}$, and $\omega_n \in R$ be a principal n-th root of unity. If n is an odd prime power, Rader's FFT computes the map $\boldsymbol{a} \mapsto \left(\boldsymbol{a}(\omega_n^i)\right)_{i\in\mathcal{I}}$ via a size-$\lambda(n)$ cyclic convolution where λ is Carmichael's lambda function. [Rad68] introduced the idea of n an odd prime, and [Win78, Section IV] generalized it to n an odd prime power.

We explain the idea for an odd prime n. Let $\mathcal{I}^* \coloneqq \{1,\ldots,n-1\}$ be an index set, $(a_j)_{j\in\mathcal{I}} \coloneqq \boldsymbol{a}$, and $(\hat{a}_i)_{i\in\mathcal{I}} \coloneqq \left(\boldsymbol{a}(\omega_n^i)\right)_{i\in\mathcal{I}}$. Since n is prime, there is a $g \in \mathcal{I}$ with $\left\{g^k \in \mathcal{I} | k \in \mathbb{Z}_{\lambda(n)}\right\} = \mathcal{I}^*$ where the powers g^k are reduced modulo n. We introduce the reindexing $j \in \mathcal{I}^* \mapsto -\log_g j \in \mathbb{Z}_{\lambda(n)}$ and $i \in \mathcal{I}^* \mapsto \log_g i \in \mathbb{Z}_{\lambda(n)}$ where $\log_g$ is the discrete logarithm, and split the computation $(a_j)_{j\in\mathcal{I}} \mapsto (\hat{a}_i)_{i\in\mathcal{I}}$ into $\hat{a}_0 = \sum_{j\in\mathcal{I}} a_j$ and $\hat{a}_i = a_0 + \sum_{j\in\mathcal{I}^*} a_j \omega_n^{ij}$ for $i \in \mathcal{I}^*$. For the cases $i \in \mathcal{I}^*$, we move a_0 to the left-hand side, and rewrite it as

$$\hat{a}_{g^{\log_g i}} - a_0 = \sum_{j\in\mathcal{I}^*} a_j \omega_n^{ij} = \sum_{-\log_g j \in \mathbb{Z}_{\lambda(n)}} a_{g^{\log_g j}} \omega_n^{g^{\log_g i + \log_g j}}.$$

We can now compute $\left(\hat{a}_{g^k} - a_0\right)_{k\in\mathbb{Z}_{\lambda(n)}}$ as the size-$\lambda(n)$ cyclic convolution of $\left(a_{g^{-k}}\right)_{k\in\mathbb{Z}_{\lambda(n)}}$ and $\left(\omega_n^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(n)}}$. We give an example for $n = 5$ and $g = 2$:

$$(\hat{a}_{2^k} - a_0)_{k\in\mathbb{Z}_5^*} = \begin{pmatrix} a_1\omega_5^2 + a_2\omega_5^4 + a_3\omega_5^1 + a_4\omega_5^3 \\ a_1\omega_5^4 + a_2\omega_5^3 + a_3\omega_5^2 + a_4\omega_5^1 \\ a_1\omega_5^3 + a_2\omega_5^1 + a_3\omega_5^4 + a_4\omega_5^2 \\ a_1\omega_5^1 + a_2\omega_5^2 + a_3\omega_5^3 + a_4\omega_5^4 \end{pmatrix} = \begin{pmatrix} a_{2^4}\omega_5^{2^1} + a_{2^3}\omega_5^{2^4} + a_{2^2}\omega_5^{2^3} + a_{2^1}\omega_5^{2^2} \\ a_{2^4}\omega_5^{2^2} + a_{2^3}\omega_5^{2^1} + a_{2^2}\omega_5^{2^4} + a_{2^1}\omega_5^{2^3} \\ a_{2^4}\omega_5^{2^3} + a_{2^3}\omega_5^{2^2} + a_{2^2}\omega_5^{2^1} + a_{2^1}\omega_5^{2^4} \\ a_{2^4}\omega_5^{2^4} + a_{2^3}\omega_5^{2^3} + a_{2^2}\omega_5^{2^2} + a_{2^1}\omega_5^{2^1} \end{pmatrix}.$$

If n is an odd prime power, we define $\mathcal{I}^* \coloneqq \{z \in \mathbb{Z}_n | z \perp n\}$ and find a g satisfying $\left\{g^k | k \in \mathbb{Z}_{\lambda(n)}\right\} = \mathcal{I}^*$[8]. We can now split the map $(a_j)_{j\in\mathcal{I}} \mapsto (\hat{a}_i)_{i\in\mathcal{I}}$ into $i \in \mathcal{I}^*$ and $i \in \mathcal{I} - \mathcal{I}^*$. For $i \in \mathcal{I}^*$, we move $\sum_{j\in\mathcal{I}-\mathcal{I}^*} a_j\omega_n^{ij}$ to the left-hand side and find

$$\hat{a}_{g^{\log_g i}} - \sum_{j\in\mathcal{I}-\mathcal{I}^*} a_j\omega_n^{ij} = \sum_{j\in\mathcal{I}^*} a_j\omega_n^{ij} = \sum_{-\log_g j\in\mathbb{Z}_{\lambda(n)}} a_{g^{\log_g j}} \omega_n^{g^{\log_g i + \log_g j}}.$$

Obviously, collecting the right-hand side forms a system of equations implementing a size-$\lambda(n)$ cyclic convolution.

[8] There is always such a g since n is an odd prime.

Real-world example(s). The case $(n, g) = (17, 3)$ was used for multiplying over $\mathbb{Z}_{4591}$. [ACC+21] multiplied in $\mathbb{Z}_{4591}[x]/\langle x^{1530} - 1\rangle$ on Cortex-M4, [HLY24] multiplied in $\mathbb{Z}_{4591}[x]/\langle x^{1632} - 1\rangle$ with Armv8.0-A Neon, and [Hwa23] multiplied in $\mathbb{Z}_{4591}[x]/\langle \Phi_{17}(x^{96})\rangle$ with Armv8.0-A Neon and Intel AVX2. Observe $1530 = 17{\cdot}90$ and $1632 = 17{\cdot}96$, their implementations relied on the size-17 cyclic FFT $\mathbb{Z}_{4591}[x]/\langle x^{17} - 1\rangle \cong \prod_i \mathbb{Z}_{4591}[x]/\langle x - \omega_{17}^i\rangle$, and are implemented with Rader's FFT. We will shortly review how [Hwa23] applied Rader's FFT for $\mathbb{Z}_{4591}[x]/\langle \Phi_{17}(x^{96})\rangle$ in Section 5.4.

3.4 Karatsuba and Toom–Cook

Let $\mathcal{I} = \{0, \ldots, 2k-2\}$ and $\{s_i\}_{i\in\mathcal{I}} \subset \mathbb{Z}$ be a finite set. Karatsuba [KO62] and Toom–Cook [Too63] compute $R[x]/\langle \prod_{i\in\mathcal{I}}(x - s_i)\rangle \cong \prod_{i\in\mathcal{I}} R[x]/\langle x - s_i\rangle$. [KO62] proposed the case $k = 2$ with the point set $\{0, 1, \infty\}$, [Too63] chose $k \geq 2$ and $\{s_i\} \subset \mathbb{Z}$, and [Win80] extended the choice of $\{s_i\}$ to $\mathbb{Q} \cup \{\infty\}$. We will review the idea of evaluating at ∞ in Section 5.2. Let $c \in \mathbb{Z}$. Evaluating x at c^{-1} means mapping a polynomial $\boldsymbol{a}(x)$ to $c^{\deg(\boldsymbol{a})}\boldsymbol{a}(c^{-1})$ instead of $\boldsymbol{a}(c^{-1})$. This convention allows us to operate entirely on integers during evaluation and worry about "invertibility" later. We will review how to adjoin the inverses of integers in Section 4.3.

3.5 Homomorphism Caching

Let $f : \mathcal{A} \to \mathcal{B}$ be an algebra monomorphism, and $\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{b} \in \mathcal{A}$. Suppose we want to implement $\boldsymbol{a}_0\boldsymbol{b}$ and $\boldsymbol{a}_1\boldsymbol{b}$. We can compute with $f^{-1}\left(f(\boldsymbol{a}_0)f(\boldsymbol{b})\right)$ and $f^{-1}\left(f(\boldsymbol{a}_1)f(\boldsymbol{b})\right)$ using only three applications of f and two applications of f^{-1}. This is called homomorphism caching and FFT-caching if f is an FFT. [Ber08, Section 2.9] said this was widely known in 1992. Section 5.3 will show historical evidence that caching was used implicitly in [Goo71] dating back to 1971.

4 Coefficient Rings

This section reviews existing techniques and benefits of a coefficient ring injection:

$$R \hookrightarrow R'.$$

Section 4.1 reviews the case when R and R' are integer rings, Section 4.2 reviews Schönhage's and Nussbaumer's FFTs choosing R' as a polynomial ring over R, and Section 4.3 reviews localization for adjoining inverses of integers.

4.1 Coefficient Ring Injection

Let $\boldsymbol{g} = x^n \pm 1$. For multiplying polynomials in $\mathbb{Z}_q[x]/\langle \boldsymbol{g}\rangle$ with $q \in \mathbb{N}$, we can always multiply in $\mathbb{Z}[x]/\langle \boldsymbol{g}\rangle$ and reduce to $\mathbb{Z}_q$ at the end. There are many ways to compute the desired result as if the coefficient ring is $\mathbb{Z}$. For simplicity, let's assume we want to multiply two polynomials. Since we know that the result over $\mathbb{Z}$ has coefficients with absolute values bounded by $\frac{nq^2}{4}$ for signed arithmetic, we choose a q' admitting a suitable FFT over $\boldsymbol{g}$ with $\frac{q'}{2} > \frac{nq^2}{4}$ and compute in $\mathbb{Z}_{q'}[x]/\langle \boldsymbol{g}\rangle$ with signed arithmetic. For unsigned arithmetic, the condition is replaced by $q' > nq^2$. Obviously, $\mathbb{Z}_q \hookrightarrow \mathbb{Z}_{q'}$ is an injection. If arithmetic defined over q' is too large for efficient implementations, one can also choose coprime integers q_i's as long as their product $q' := \prod_i q_i$ fulfills the same conditions. The tuple of coprime integers is called a residue number system (RNS). Multiplying over $\mathbb{Z}_{q'}$ and $\prod_i \mathbb{Z}_{q_i}$ is used in many contexts, including lattice-based cryptography [BBC+20, ACC+21, CHK+21, ACC+22], and also before public key cryptography [Nic71, Pol71].

In the literature, one usually sticks to $\mathbb{Z}_{q'}$ or $\prod_i \mathbb{Z}_{q_i}$ for the transformations and small-degree polynomial multiplications. However, [ACC+22, Section 4.2.1] showed how to balance computing time and memory by operating in $\mathbb{Z}_{\prod_i q_i}$ and $\prod_i \mathbb{Z}_{q_i}$ for different operands.

4.2 Schönhage's and Nussbaumer's Fast Fourier Transforms

This section surveys Schönhage's [Sch77] and Nussbaumer's FFTs [Nus80]. Let $\boldsymbol{g}(x^{n_1}) \in R[x]$ be a degree-$n_0 n_1$ monic polynomial. Both FFTs exploit the structure of $\boldsymbol{g}(x^{n_1})$ by introducing $x^{n_1} \sim y$ (so $R[x]/\langle \boldsymbol{g}(x^{n_1})\rangle \cong R[x,y]/\langle x^{n_1}-y, \boldsymbol{g}(y)\rangle$). Schönhage splits the structure into small structures by adjoining the defining condition. On the other hand, Nussbaumer adjoins a structure for splitting and uses $\boldsymbol{g}(x^{n_1})$ as the defining condition. We start by replacing $x^{n_1}-y$ with $\boldsymbol{h}(x)$ satisfying $\deg(\boldsymbol{h}) \geq 2n_1 - 1$.

Schönhage requires $\boldsymbol{g}(y)|(y^n-1)$ and $\boldsymbol{h}(x)|\Phi_n$ for an n, and treats $R[x]/\langle \boldsymbol{h}(x)\rangle$ as the coefficient ring. We then split as follows:

$$\frac{R[x]}{\langle \boldsymbol{g}(x^{n_1})\rangle} \cong \frac{R[x,y]}{\langle x^{n_1}-y, \boldsymbol{g}(y)\rangle} \hookrightarrow \frac{R[x,y]}{\langle \boldsymbol{h}(x), \boldsymbol{g}(y)\rangle} \cong \frac{\left(\frac{R[x]}{\langle \boldsymbol{h}(x)\rangle}\right)[y]}{\langle \boldsymbol{g}(y)\rangle} \cong \prod_{i\in\mathcal{I}} \frac{\left(\frac{R[x]}{\langle \boldsymbol{h}(x)\rangle}\right)[y]}{\langle y-\boldsymbol{\omega}_i\rangle}$$

for $\boldsymbol{\omega}$ a principal n-th root of unity in $R[x]/\langle \boldsymbol{h}(x)\rangle$.

On the other hand, Nussbaumer requires $\boldsymbol{g}(y)|\Phi_n$ and $\boldsymbol{h}(x)|(x^n-1)$ for an n. This allows us to split as follows:

$$\frac{R[x]}{\langle \boldsymbol{g}(x^{n_1})\rangle} \cong \frac{R[x,y]}{\langle x^{n_1}-y, \boldsymbol{g}(y)\rangle} \hookrightarrow \frac{R[x,y]}{\langle \boldsymbol{h}(x), \boldsymbol{g}(y)\rangle} \cong \frac{\left(\frac{R[y]}{\langle \boldsymbol{g}(y)\rangle}\right)[x]}{\langle \boldsymbol{h}(x)\rangle} \cong \prod_{i\in\mathcal{I}} \frac{\left(\frac{R[y]}{\langle \boldsymbol{g}(y)\rangle}\right)[x]}{\langle x-\boldsymbol{\omega}_i\rangle}$$

for $\boldsymbol{\omega}$ a principal n-th root of unity in $R[y]/\langle \boldsymbol{g}(y)\rangle$.

We briefly compare Schönhage and Nussbaumer when n is a power of two in two aspects: (i) definability and (ii) problem size reduction. For the definability, Schönhage only requires $\boldsymbol{g}(y)|(y^n-1)$ while Nussbaumer requires $\boldsymbol{g}(y)|\Phi_n$. Since $\Phi_n|(y^n-1)$, Schönhage applies to more scenarios. The benefit of Nussbaumer is a broader choice for $\boldsymbol{h}$. This results in smaller subproblem sizes or simpler subproblems in some cases [Ber01, Section 9]. For a more detailed comparison, see Appendix B.

Schönhage and Nussbaumer can be used together, with the former covering a wide range of polynomial modulus and the latter exploiting the newly introduced factor of a cyclotomic polynomial.

Real-world example(s). [BBCT22] transformed $\mathbb{Z}_{4591}[x]/\langle (x^{1024}+1)(x^{512}-1)\rangle$ as follows. They started with Schönhage for

$$\frac{\mathbb{Z}_{4591}[x]}{\langle (x^{1024}+1)(x^{512}-1)\rangle} \cong \frac{\mathbb{Z}_{4591}[x,y]}{\langle x^{32}-y, (y^{32}+1)(y^{16}-1)\rangle} \hookrightarrow \frac{\left(\frac{\mathbb{Z}_{4591}[x]}{\langle x^{64}+1\rangle}\right)[y]}{\langle (y^{32}+1)(y^{16}-1)\rangle}$$

and applied Nussbaumer to $\mathbb{Z}_{4591}[x]/\langle x^{64}+1\rangle$.

For generalizations on introducing a polynomial $\boldsymbol{\omega}$ as a principal root of unity, see [MV83a, MV83b, Ber01].

4.3 Localization

Let $n \in \mathbb{Z}$ be non-invertible in R. Localization allows us to formulate "division by an integer n^k in R." We quote the following from [Jac12b, Section 7.2] for the propose of localization:

> Given a (commutative) ring R and a subset S of R, to construct a ring R_S and a homomorphism λ_S of R into R_S such that every $\lambda_S(s), s \in S$, is invertible in R_S, and the pair (R_S, λ_S) is universal for such pairs in the sense that if η is any homomorphism of R into a ring R' such that every $\eta(s)$ is invertible, then there exists a unique homomorphism $\tilde{\eta} : R_S \to R'$ such that $\eta = \tilde{\eta} \circ \lambda_S$[9].

The ring R_S is also commonly denoted as $S^{-1}R$. We leave the formal treatment to Appendix C and explain with a small example.

Suppose we want to compute $c_0 + c_1 x = (a_0 + a_1 x)(b_0 + b_1 x)$ in $\mathbb{Z}_{2^{15}}[x]/\langle x^2 - 1\rangle$ with "Cooley–Tukey FFT". We compute $a_0 + a_1 x \mapsto (a_0 + a_1, a_0 - a_1)$ and $b_0 + b_1 x \mapsto (b_0 + b_1, b_0 - b_1)$, point-multiply them, and perform an add-sub pair. The result is $((a_0 + a_1)(b_0 + b_1) \pm (a_0 - a_1)(b_0 - b_1)) = 2(a_0 b_0 + a_1 b_1, a_0 b_1 + a_1 b_0)$. It remains to "divide by two". Localization means the following monomorphisms:

$$\frac{\mathbb{Z}_{2^{15}}[x]}{\langle x^2 - 1\rangle} \hookrightarrow \prod \frac{\mathbb{Z}_{2^{16}}[x]}{\langle x \pm 1\rangle} \hookrightarrow \frac{\mathbb{Z}_{2^{16}}[x]}{\langle x^2 - 1\rangle}.$$

Since we know that the result is a 2-multiple of the desired one, we can extract the result by maintaining the set of 2-multiples as in $\mathbb{Z}_{2^{16}}$.

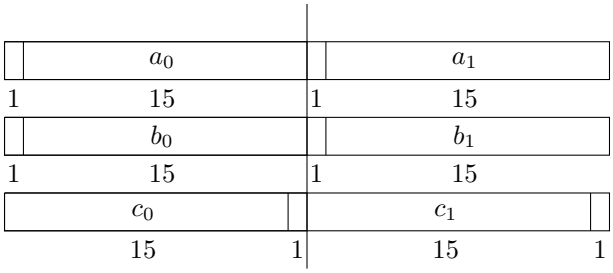

Figure 1: Localization for $\mathbb{Z}_{2^{15}}$ in $\mathbb{Z}_{2^{16}}$. We store the 15-bit values a_0, a_1, b_0, b_1 as halfwords. For the 15-bit values c_0, c_1, we compute the 16-bit values $2c_0$ and $2c_1$ and extract the c_0 and c_1 by shifting.

Real-world example(s). Recall that for Toom-3 with the point set $\{0, \pm 1, 2, \infty\}$, we have to apply

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & -\frac{1}{3} & -\frac{1}{6} & 2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{6} & \frac{1}{6} & -2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

If we work over the ring $\mathbb{Z}_{2^{11}}[x]$ used in `ntruhps2048677`, then we have to maintain the values in $\mathbb{Z}_{2^{12}}$ for adjoining 2^{-1}. Another example is Toom-5. If we choose $\{0, \pm 1, \pm 2, \pm 3, 4, \infty\}$ as the point set for evaluation, we must adjoin 16^{-1}. [CCHY23] showed that one can instead switch to $\{0, \pm 1, \pm 2, \pm \frac{1}{2}, 3, \infty\}$ requiring only 8^{-1}.

It should be noted that localization need not to adjoin the inverses uniformly in practice. For example, if we apply Toom-4 with the point set $\{s_i\} = \{0, \pm 1, \pm 2, 3, \infty\}$, then we only need to implement the following monomorphism:

$$\frac{\mathbb{Z}_{2^k}[x]}{\langle \prod_i s_i\rangle} \hookrightarrow \frac{\mathbb{Z}_{2^{k+2}}[x]}{\langle x\rangle} \times \frac{\mathbb{Z}_{2^{k+2}}[x]}{\langle x - 1\rangle} \times \frac{\mathbb{Z}_{2^{k+3}}[x]}{\langle x + 1\rangle} \times \frac{\mathbb{Z}_{2^{k+3}}[x]}{\langle x - 2\rangle} \times \frac{\mathbb{Z}_{2^{k+3}}[x]}{\langle x + 2\rangle} \times \frac{\mathbb{Z}_{2^{k+3}}[x]}{\langle x - 3\rangle} \times \frac{\mathbb{Z}_{2^k}[x]}{\langle x - \infty\rangle}.$$

[9] The last sentence actually ends with "such that the diagram [Figure] is commutative". We replace the description with the desired composition.

This allows one to apply more transformations for some subproblems by working over $\mathbb{Z}_{2^{k+2}}$ and $\mathbb{Z}_{2^k}$ instead of $\mathbb{Z}_{2^{k+3}}$. The non-uniform property of localization with Toom–Cook does not seem to appear in the literature, but we believe there are practical benefits for implementations.

5 Polynomial Modulus

This section reviews several techniques related to the polynomial modulus $\boldsymbol{g}$ of $R[x]/\langle \boldsymbol{g}(x)\rangle$. Section 5.1 reviews twisting and composed multiplication converting $R[x]/\langle \boldsymbol{g}(x)\rangle$ into a polynomial ring of the form $R[y]/\langle \boldsymbol{g}(\zeta y)\rangle$, Section 5.2 reviews embedding and evaluation at ∞ for choosing a polynomial $\boldsymbol{h}$ admitting the monomorphism $R[x]/\langle \boldsymbol{g}(x)\rangle \hookrightarrow R[x]/\langle \boldsymbol{h}(x)\rangle$. Section 5.3 reviews the group algebra view for $\boldsymbol{g}(x) = x^n - 1$ and Good–Thomas FFT. Section 5.4 reviews truncation computing products in $R[x]/\langle \prod_{i\in\mathcal{I}'} \boldsymbol{g}_i\rangle$ with an isomorphism derived from an isomorphism for $R[x]/\langle \prod_{i\in\mathcal{I}} \boldsymbol{g}_i\rangle$ with $\mathcal{I}' \subset \mathcal{I}$. Section 5.5 reviews incomplete transformations and striding, and Section 5.6 reviews the Toeplitz matrix-vector product for $R[x]/\langle x^n - \alpha x - \beta\rangle$ from the dual module view of algebra homomorphisms multiplying two size-n polynomials in $R[x]$.

5.1 Twisting and Composed Multiplication

5.1.1 Twisting

Let $\zeta \in R$ be an invertible element. Twisting is an isomorphism from $R[x]/\langle \boldsymbol{g}(x)\rangle$ to $R[y]/\langle \boldsymbol{g}(\zeta y)\rangle$ by introducing $x \sim \zeta y$. We have the isomorphism $R[x]/\langle \boldsymbol{g}(x)\rangle \cong R[x,y]/\langle x - \zeta y, \boldsymbol{g}(\zeta y)\rangle$ and treat $R[x]/\langle x - \zeta y\rangle$ as the coefficient ring. Let $n = \deg(\boldsymbol{g})$. In order to change the basis from $(1, x, \ldots, x^{n-1})$ to $(1, y, \ldots, y^{n-1}) = (1, \zeta x, \ldots, \zeta^{n-1}x^{n-1})$, we have to multiply the coefficients with the powers $\zeta, \ldots, \zeta^{n-1}$. This usually amounts to $n-1$ multiplications in R, and allows us to operate in an easier case. However, if n is odd and $\zeta = -1$, we do not need any multiplication for the isomorphism $R[x]/\langle x^n + 1\rangle \cong R[x,y]/\langle x + y, y^n - 1\rangle$. We will shortly see how this insight can be systemized in Section 5.4.

Twisting was introduced in [GS66] for computing FFTs with $R[x]/\langle x^{n_0 n_1} - 1\rangle \cong \prod_i R[x]/\langle x^{n_1} - \omega_{n_0}^i\rangle \cong \prod_i R[x]/\langle x^{n_1} - 1\rangle$. See [DH84, Für09] for more insights on the choices of n_0 and n_1.

5.1.2 Composed Multiplication

We go through a specialized approach when $R = \mathbb{F}_q$. Given $\boldsymbol{f}_0, \boldsymbol{f}_1 \in \mathbb{F}_q[x]$, we defined their composed multiplication [BC87] 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 are elements from an extension field of $\mathbb{F}_q$. Composed multiplication allows us to generalize twisting to the polynomial modulus of the form $(x-\zeta) \odot \boldsymbol{f}(x)$. In particular, we have $\mathbb{F}_q[x]/\langle (x-\zeta) \odot \boldsymbol{f}(x)\rangle \cong \mathbb{F}_q[y]/\langle x - \zeta y, \boldsymbol{f}(y)\rangle$.

Another benefit of composed multiplication is systematically deriving transformations based on (presumably much simpler) coprime factorizations. Let $\boldsymbol{f}_0 = \prod_{i_0} \boldsymbol{f}_{0,i_0}$ and $\boldsymbol{f}_1 = \prod_{i_1} \boldsymbol{f}_{1,i_1}$ be coprime factorizations in $\mathbb{F}_q[x]$. We have $\boldsymbol{f}_0 \odot \boldsymbol{f}_1 = \prod_{i_0} \left(\boldsymbol{f}_{0,i_0} \odot \boldsymbol{f}_1\right) = \prod_{i_0,i_1} \left(\boldsymbol{f}_{0,i_0} \odot \boldsymbol{f}_{1,i_1}\right)$. A practically important example is $\boldsymbol{f}_0 = x^r - 1 = \prod_{i_0} \left(x - \omega_r^{i_0}\right) \in \mathbb{F}_q[x]$ and $\boldsymbol{f}_1 = x^{2^k} - 1$. Given a factorization $x^{2^k} - 1 = \prod_{i_1} \boldsymbol{f}_{1,i_1}$ in $\mathbb{F}_q[x]$, we have

$$x^{2^k r} - 1 = \prod_{i_0} \left(x^{2^k} - \omega_r^{2^k i_0}\right) = \prod_{i_0,i_1} \omega_r^{\deg(\boldsymbol{f}_{1,i_1})} \boldsymbol{f}_{1,i_1}(\omega_r^{-i_0} x).$$

Real-world example(s). For an odd number r with $r|(4591-1)$, we have the size-r transformation $R[x]/\langle x^r-1\rangle \cong \prod_i R[x]/\langle x-\omega_r^i\rangle$. We extend it to a size-$2^k r$ transformation for the ease of vectorization. [HLY24] implemented the isomorphism $\mathbb{Z}_{4591}[x]/\langle x^{1632}-1\rangle \cong \prod_i \mathbb{Z}_{4591}[x]/\langle x^{16}\pm\omega_{51}^i\rangle$, and factor $\mathbb{Z}_{4591}[x]/\langle x^{16}\pm\omega_{51}^i\rangle$ into polynomial rings modulo the composed multiplications of $x-\omega_{51}^i$ and factors of $x^{16}\pm 1$.

5.2 Embedding (Polynomial Modulus) and Evaluation at ∞

Let $\boldsymbol{g} \in R[x]$ be a polynomial with $\deg(\boldsymbol{g}) \le n$. An obvious approach for multiplying polynomials in $R[x]/\langle \boldsymbol{g}\rangle$ is multiplying in $R[x]$ followed by reducing modulo $\boldsymbol{g}$. This is the embedding technique for ignoring the structure of $\boldsymbol{g}$. For $R[x]$, one further applies an identity map from $R[x]$ to $R[x]/\langle \boldsymbol{h}\rangle$ where $\boldsymbol{h}$ is a polynomial with degree larger than the product in $R[x]$. $\boldsymbol{h}$ is usually a polynomial with a very nice structure for fast transformations.

Evaluation at ∞ is an optimization for choosing $\boldsymbol{h}$ [Win80]. Suppose $\boldsymbol{r}$ is the product in $R[x]$, d the degree, and r_d the leading term of $\boldsymbol{r}$. Instead of computing $\boldsymbol{r}$, we compute $\boldsymbol{r}-r_d\boldsymbol{h}$ by embedding into $R[x]/\langle \boldsymbol{h}\rangle$ with $\deg(\boldsymbol{h})=d$. The term $r_d\boldsymbol{h}$ is computed individually and added back. In the literature, the idea is commonly presented as allowing $\boldsymbol{h}$ to contain the polynomial $x-\infty$. Historically, evaluation at ∞ was first used by [KO62]. [Too63] chose small integers for evaluation, and [Win80, Page 31] replaced a point with ∞ for unifying Karatsuba and Toom–Cook. [Win80]'s idea was already as general as this section and applied to other choices of $\boldsymbol{h}$.

In [KO62], they computed $(a_0+a_1x)(b_0+b_1x)$ with $(a_0+a_1x)(b_0+b_1x) = a_0b_0 + ((a_0+a_1)(b_0+b_1)-a_0b_0-a_1b_1)x + a_1b_1x^2$. If we choose $\boldsymbol{h}=x^2+x$, the polynomial $(a_0+a_1x)(b_0+b_1x)-a_1b_1(x^2+x) = a_0b_0+(a_0b_1+a_1b_0-a_1b_1)x$ can be computed in $R[x]/\langle x^2+x\rangle$. Applying $R[x]/\langle x^2+x\rangle \cong R[x]/\langle x\rangle \times R[x]/\langle x-1\rangle$ gives us (a_0, a_0+a_1) and (b_0, b_0+b_1). After point-multiplying and inverting, we have $a_0b_0+((a_0+a_1)(b_0+b_1)-a_0a_1)x$. Adding $a_1b_1(x^2+x)$ derives the desired result.

It doesn't seem that people have ever chosen $\boldsymbol{h}$ with $x-\infty$ for FFT in the literature. We believe the reason is that one usually splits $\boldsymbol{h}$ into a large number of small factors for FFT, and the benefit of replacing one of them with $x-\infty$ is marginal. Nevertheless, we give the following example of multiplying $(a_0+a_1x)(b_0+b_1x)$ for referential purposes. We rewrite $(a_0+a_1x)(b_0+b_1x)$ as $(a_0b_0+a_1b_1)+(a_0b_1+a_1b_0)x+a_1b_1(x^2-1)$, compute $(a_0b_0+a_1b_1)+(a_0b_1+a_1b_0)x$ with the isomorphism $R[x]/\langle x^2-1\rangle \cong \prod R[x]/\langle x\pm 1\rangle$, and finally add $a_1b_1(x^2-1)$ to the result.

5.3 Good–Thomas FFT

This section presents an algebraic view of the Good–Thomas FFT [Goo58, Goo71]. Let $n_0,\ldots,n_{d-1}$ be coprime integers and $n=\prod_j n_j$. We have $R[\mathbb{Z}_n] \cong \bigotimes_j R[\mathbb{Z}_{n_j}]$ as algebras, or equivalently, $R[x]/\langle x^n-1\rangle \cong R[x_0,\ldots,x_{d-1}]/\langle x_0^{n_0}-1,\ldots,x_{d-1}^{n_{d-1}}-1\rangle$ as polynomial rings. We leave the proof as an exercise. For a d-dimensional polynomial $\boldsymbol{a} = \left(a_{i_0,\ldots,i_{d-1}}\right)_{i_0,\ldots,i_{d-1}} \in \bigotimes_j R[\mathbb{Z}_{n_j}]$, we define $\boldsymbol{a}_{i_0,\ldots,i_{h-1}}$ as the $(d-h)$-dimensional tuple $\left(a_{i_0,\ldots,i_{d-1}}\right)_{i_h,\ldots,i_{d-1}}$ for $h>0$ and $\boldsymbol{a}$ otherwise. Multiplying two elements $\boldsymbol{a},\boldsymbol{b}$ is regarded as the following multi-dimensional cyclic convolution $\cdot_h$ defined recursively:

$$\boldsymbol{a}_{i_0,\ldots,i_{h-1}} \cdot_h \boldsymbol{b}_{i_0,\ldots,i_{h-1}} = \begin{cases} a_{i_0,\ldots,i_{d-1}}b_{i_0,\ldots,i_{d-1}} & \text{if } h=d,\\ \sum_k \left(\sum_{k_a+k_b=k} \boldsymbol{a}_{i_0,\ldots,i_{h-1},k_a} \cdot_{h+1} \boldsymbol{b}_{i_0,\ldots,i_{h-1},k_b}\right) x_h^k & \text{otherwise.}\end{cases}$$

Our goal is to implement $\cdot_0$, the multiplication in $R[x_0,\ldots,x_{d-1}]/\langle x_0^{n_0}-1,\ldots,x_{d-1}^{n_{d-1}}-1\rangle$.

We now apply homomorphism caching as follows. Let $f: R[x_{d-1}]/\langle x_{d-1}^{n_{d-1}}-1\rangle \cong R_{d-1}$ be a monomorphism. We naturally have $R[x_0,\ldots,x_{d-1}]/\langle x_0^{n_0}-1,\ldots,x_{d-1}^{n_{d-1}}-1\rangle \cong$

$R[x_0,\ldots,x_{d-2}]/\langle x_0^{n_0}-1,\ldots,x_{d-2}^{n_{d-2}}-1\rangle \otimes R_{d-1}$. This contextualizes $\cdot_{d-2}$ as

$$\boldsymbol{a}_{i_0,\ldots,i_{d-3}} \cdot_{d-2} \boldsymbol{b}_{i_0,\ldots,i_{d-3}} = \sum_k f^{-1}\left(\sum_{k_a+k_b=k} f(\boldsymbol{a}_{i_0,\ldots,i_{d-3},k_a}) \cdot_{R_{d-1}} f(\boldsymbol{b}_{i_0,\ldots,i_{d-3},k_b})\right) x_{d-2}^k.$$

We compute and cache $f(\boldsymbol{a}_{i_0,\ldots,i_{d-3},k_b})$ and $f(\boldsymbol{a}_{i_0,\ldots,i_{d-3},k_b})$, and use them for all the k's. Similarly, the idea applies to all other dimensions. As a side note, arbitrary additive group isomorphism $\mathbb{Z}_n \cong \prod_j \mathbb{Z}_{n_j}$ suffices and there are $\phi(n)$ of them where ϕ is the Euler's totient function. In Appendix F, we survey the vector–radix transform [HMCS77] for optimizing the multi-dimensional transformation directly.

Real-world example(s). Good–Thomas FFT was first used in [BBC+20] for lattice–based cryptography. They implemented $\mathbb{Z}_{7681\cdot 10753}[\mathbb{Z}_{1536}] \cong \mathbb{Z}_{7681}[\mathbb{Z}_{1536}] \times \mathbb{Z}_{10753}[\mathbb{Z}_{1536}] \cong \mathbb{Z}_{7681}[\mathbb{Z}_3 \times \mathbb{Z}_{512}] \times \mathbb{Z}_{10753}[\mathbb{Z}_3 \times \mathbb{Z}_{512}]$ with 16-bit arithmetic using AVX2. [AHY22] proposed to introduce $x^4 \sim y$ first for Good–Thomas FFT and operate in y for vectorization-friendliness, and [HLY24] implemented [AHY22]'s idea with Armv8.0-A Neon. [AHY22] was working with Armv7E-M where a very limited vectorization support is implemented. They observed that vectorization-friendliness implies a flexible code-size optimization while permuting for $R[\mathbb{Z}_n] \cong R\left[\prod_j \mathbb{Z}_{n_j}\right]$ with compact code size. See [AHY22, Sections 3.2 and 3.3] for details.

The notion of homomorphism caching is the actual reason making the multi-dimensional cyclic convolution fast. Historically, Good–Thomas FFT was first presented in [Goo58][10] as a correspondence between a DFT defined on $R[x]/\langle x^n-1\rangle$ and a tensor product of the DFTs defined on $R[x_j]/\langle x_j^{n_j}-1\rangle$. This was cited as a motivation of Cooley–Tukey FFT in [CT65]. [GS66, Sto66] pointed out the use of Cooley–Tukey FFT for cyclic convolutions. [Goo71] explained the differences between Good–Thomas FFT and Cooley–Tukey FFT, and acknowledged the application of multi-dimensional transform to multi-dimensional cyclic convolution. Based on this, we believe that homomorphism caching was already used in [Goo71].

5.4 Truncation

Truncation is a simple and powerful idea. Let $\mathcal{I}' \subset \mathcal{I}$ be index sets and $\{\boldsymbol{g}_i\}_{i\in\mathcal{I}}$ be coprime polynomials in $R[x]$. Suppose we are given the following isomorphism

$$\eta : \begin{cases} R[x]\Big/\left\langle \prod_{i\in\mathcal{I}} \boldsymbol{g}_i \right\rangle & \to \prod_{i\in\mathcal{I}} R[x]/\langle \boldsymbol{g}_i\rangle\ , \\ \boldsymbol{a} & \mapsto (\boldsymbol{a} \bmod \boldsymbol{g}_i)_{i\in\mathcal{I}}\ . \end{cases}$$

We can naturally define an isomorphism $\eta_{\mathcal{I}'}$ as

$$\eta_{\mathcal{I}'} : \begin{cases} R[x]\Big/\left\langle \prod_{i\in\mathcal{I}'} \boldsymbol{g}_i \right\rangle & \to \prod_{i\in\mathcal{I}'} R[x]/\langle \boldsymbol{g}_i\rangle\ , \\ \boldsymbol{a} & \mapsto (\boldsymbol{a} \bmod \boldsymbol{g}_i)_{i\in\mathcal{I}'}\ . \end{cases}$$

$\eta_{\mathcal{I}'}$ is called the truncation of η at $R[x]/\langle\prod_{i\in\mathcal{I}'} \boldsymbol{g}_i\rangle$. Truncation was introduced by [CF94, Section 7]. [Ber08] (according to [vdH04], the work [Ber08] was already online prior to [vdH04]) described the benefit in terms of complexity, and [vdH04] named the technique "truncated Fourier transform" for the FFT case. We call it truncation since it is not restricted to FFTs.

[10] In the literature, people commonly attribute the idea to [Goo58, Tho63]. However, we are unable to locate the work [Tho63], and only find the publication information. If someone finds a copy, we would like to see how general the idea was in [Tho63].

5.4.1 Application I: $R[x]\Big/\left\langle x^{2^{k-1}}+1\right\rangle$ from $R[x]\Big/\left\langle x^{2^k}-1\right\rangle$

We derive FFT for $R[x]\Big/\left\langle x^{2^{k-1}}+1\right\rangle$ from the one for $R[x]\Big/\left\langle x^{2^k}-1\right\rangle$. For a principal 2^k-th root of unity ω_{2^k} realizing $R[x]\Big/\left\langle x^{2^k}-1\right\rangle \cong \prod_{i=0}^{2^k-1} R[x]/\left\langle x-\omega_{2^k}^i\right\rangle$, we have $R[x]\Big/\left\langle x^{2^{k-1}}+1\right\rangle \cong \prod_{i=0}^{2^{k-1}-1} R[x]/\left\langle x-\omega_{2^k}^{1+2i}\right\rangle$. We can generalize the idea to arbitrary transformation size n. Below is a straightforward generalization of [CF94, Section 7]. Let $b=n$ and $\tilde{b}=\sum_j \tilde{b}_j 2^j$ be the 2's complement representation of $-n$ as a $\lceil \log_2 n \rceil$-bit integer. We have $b+\tilde{b}=2^{\lceil \log_2 n \rceil}$ by definition and define a transformation for

$$R[x]\Big/\left\langle \frac{x^{2^{\lceil \log_2 n \rceil}}-1}{\prod_j \left(x^{2^j}+1\right)^{\tilde{b}_j}} \right\rangle.$$

This boils down to transformations for rings of the form $R[x]\Big/\left\langle x^{2^k}\pm 1\right\rangle$. An example is the Schönhage for $R[x]/\left\langle (x^{1024}+1)(x^{512}-1)\right\rangle$ derived from $R[x]/\left\langle x^{2048}-1\right\rangle$.

5.4.2 Application II: $R[x]/\langle x^r+1\rangle$ from $R[x]/\left\langle x^{2r}-1\right\rangle$ for $r\perp 2$

Our second application is to systematically generalize the isomorphism $R[x]/\langle x^r+1\rangle \cong R[x,y]/\langle x+y, y^r-1\rangle$ for an odd r. Let $\psi:\mathbb{Z}_{2r}\cong\mathbb{Z}_2\times\mathbb{Z}_r$ be the additive group isomorphism $1\mapsto(1,1)$. Recall that ψ induces an algebra isomorphism $\psi': R[x]\Big/\left\langle x^{2r}-1\right\rangle \cong R[z]/\left\langle z^2-1\right\rangle \otimes R[y]/\langle y^r-1\rangle$ (cf. Section 5.3). From $R[z]/\left\langle z^2-1\right\rangle \cong \prod R[z]/\langle z\pm 1\rangle$, we have

$$\frac{R[x]}{\langle x^r+1\rangle} \cong \psi'^{-1}\left(\frac{R[z]}{\langle z+1\rangle}\otimes\frac{R[y]}{\langle y^r-1\rangle}\right)\cong\frac{R[x,y]}{\langle x+y,y^r-1\rangle}.$$

Similarly, whenever we are working on a polynomial ring with modulus a factor of $x^{q_0q_1}-1$ for $q_0\perp q_1$, we can always look for transformations for $R[z]/\langle z^{q_0}-1\rangle\otimes R[y]/\langle y^{q_1}-1\rangle$ and pull them back to the desired domain (in our example, we exploit $R[z]/\left\langle z^2-1\right\rangle \cong \prod R[z]/\langle z\pm 1\rangle$). Examples in the literature are the CRT negacyclic/tricyclic transform in [HVDH22, Sections 3.5 and 3.6].

One should notice that we could derive optimizations by exploiting some properties of a factor of $x^{q_0q_1}-1$ and bring the resulting computation back to the isomorphism $R[x]/\langle x^{q_0q_1}-1\rangle \cong R[z]/\langle z^{q_0}-1\rangle\otimes R[y]/\langle y^{q_1}-1\rangle$. The optimization comes from splitting $\Phi_{3\cdot 2^k}=\left(x^{2^k}-\omega_6\right)\left(x^{2^k}-\omega_6^5\right)$ exploiting the identity $\omega_6+\omega_6^5=1$ [LS19] and $\Phi_3=(x-\omega_3)(x-\omega_3^2)$ exploiting the identity $\omega_3+\omega_3^2=-1$ [DV78a, HY22, AHY22]. We leave them as exercises for the readers.

5.4.3 Application III: Nussbaumer from Schönhage

As we know, for arbitrary $\boldsymbol{g}$ a factor of $x^{2^k}-1$, we can derive the corresponding Schönhage's FFT via truncating the Schönhage's FFT for $R[x]\Big/\left\langle x^{2^k}-1\right\rangle$. We now show how to exploit the same idea for Nussbaumer's FFT systematically. An example is to derive the Nussbaumer for $R[x]/\left\langle x^{1536}+1\right\rangle$ from the Schönhage for $R[x]/\left\langle (x^{1024}+1)(x^{512}-1)\right\rangle$. Given polynomials $\boldsymbol{g}(z)|(z^{n'}-1)$ and $\boldsymbol{h}(z)|\Phi_{n'}(z)$ with $\deg(\boldsymbol{g})=2n_0$ and $\deg(\boldsymbol{h})=2n_1$, we have a size-$2n_0n_1$ transformation via Schönhage as follows

$$\frac{R[x]}{\langle \boldsymbol{g}(x^{n_1})\rangle}\cong\frac{R[x,y]}{\langle x^{n_1}-y,\boldsymbol{g}(y)\rangle}\hookrightarrow\frac{R[x,y]}{\langle \boldsymbol{h}(x),\boldsymbol{g}(y)\rangle}\cong\frac{\left(\frac{R[x]}{\langle \boldsymbol{h}(x)\rangle}\right)[y]}{\langle \boldsymbol{g}(y)\rangle}.$$

We exchange x and y in $\left(R[x]/\langle \boldsymbol{h}(x)\rangle\right)[y]/\langle \boldsymbol{g}(y)\rangle$, and invert the derivation of Nussbaumer. This gives us the following transformation for Nussbaumer

$$\frac{R[x]}{\langle \boldsymbol{h}\left(x^{n_0}\right)\rangle} \cong \frac{R[x,y]}{\langle x^{n_0}-y, \boldsymbol{h}(y)\rangle} \hookrightarrow \frac{R[x,y]}{\langle \boldsymbol{g}(x), \boldsymbol{h}(y)\rangle} \cong \frac{\left(\frac{R[y]}{\langle \boldsymbol{h}(y)\rangle}\right)[x]}{\langle \boldsymbol{g}(x)\rangle}.$$

5.4.4 Application IV: Rader's FFT

Let p be an odd prime, $\mathcal{I}=\{0,\ldots,p-1\}$, $\mathcal{I}^*=\{z\in\mathcal{I}|z\perp p\}$, and g be a generator of $\mathcal{I}^*$. For a principal p-th root of unity, we show how Rader's FFT converts the computing task of size-p cyclic FFT into a size-$\lambda(p)$ cyclic convolution in Section 3.3. In this section, we show that the isomorphism $R[x]/\left\langle\prod_{i\in\mathcal{I}^*}\left(x-\omega_p^i\right)\right\rangle \cong \prod_{i\in\mathcal{I}^*} R[x]/\left\langle x-\omega_p^i\right\rangle$ and its inverse can also be converted into size-$\lambda(p)$ cyclic convolutions. For generalization truncating a size-n cyclic DFT to the roots with exponents coprime to n, see [Ber22, Sections 4.12.3 and 4.12.4].

Forward transformation. Given a polynomial $\sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j x^j \in R[x]/\left\langle\prod_{i\in\mathcal{I}^*}\left(x-\omega_p^i\right)\right\rangle$ and its image $\left(\hat{a}_{i-1}\right)_{i\in\mathcal{I}^*} = \sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j x^j \bmod \left(x-\omega_p^i\right)$, we rewrite the definition of $\hat{a}_{g^{\log_g i}-1} = \hat{a}_{i-1}$ as follows:

$$\begin{aligned}\hat{a}_{g^{\log_g i}-1} = \hat{a}_{i-1} &= \sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j\omega_p^{ij} = \omega_p^{-i}\sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j\omega_p^{i(j+1)} = \omega_p^{-i}\sum_{j\in\mathcal{I}^*} a_{j-1}\omega_p^{ij}\\ &= \omega_p^{-g^{\log_g i}}\sum_{-\log_g j\in\mathbb{Z}_{\lambda(p)}} a_{g^{\log_g j}-1}\omega_p^{g^{\log_g i+\log_g j}}.\end{aligned}$$

If we multiply both sides by $\omega_p^{g^{\log_g i}}$, then we find that $\left(\omega_p^{g^k}\hat{a}_{g^k-1}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is a size-$\lambda(p)$ cyclic convolution of $\left(a_{g^{-k}-1}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and $\left(\omega_n^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$.

Inverse transformation. [Ber22, Section 4.8.2] showed that convolution by $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ can be inverted by convolution. By definition, convolution in the polynomial ring $R[x]/\left\langle x^{\lambda(p)}-1\right\rangle$ is the ring multiplication in the group algebra $R[\mathbb{Z}_{\lambda(p)}]$. Therefore, the inversion amounts to multiplying the multiplicative inverse of $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ in the group algebra $R[\mathbb{Z}_{\lambda(p)}]$. The inverse of $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is $\frac{1}{p}\left(\omega_n^{-g^{-k}}-1\right)_{k\in\mathbb{Z}_{\lambda(p)}}$. [Ber22] proved this by showing that the convolution of $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and $\left(\omega_p^{-g^{-k}}-1\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is $(\delta_{0,k}p)_{k\in\mathbb{Z}_{\lambda(p)}}$: For all $k\in\mathbb{Z}_{\lambda(p)}$, we find

$$\sum_{i+j=k}\omega_n^{g^i}\left(\omega_n^{-g^{-j}}-1\right) = \sum_{i+j=k}\omega_n^{g^i\left(1-g^{-(i+j)}\right)} - \sum_{i+j=k}\omega_n^{g^i} = \delta_{0,k}p$$

as desired.

5.5 Incomplete Transformation and Striding

5.5.1 Incomplete Transformation

For a monic polynomial $\boldsymbol{g}(x^v)\in R[x]$, we call a homomorphism $f: R[x]/\langle\boldsymbol{g}(x^v)\rangle \to \mathcal{A}$ "incomplete" if f starts with introducing $x^v\sim y$ and proceed as a polynomial ring in

y with the coefficient ring $R[x]/\langle x^v - y\rangle$. There are several benefits for an incomplete transformation: (i) the definability of fast transformation, (ii) the vectorization-friendliness of $x^v \sim y$, and (iii) the code size for implementing f. We give an example for (i) in this section. For the benefit of vectorization, see Section 6.1. As for (iii), we refer to [AHY22, Sections 3.2 and 3.3] for more details.

Real-world example(s). Let's take the polynomial ring $\mathbb{Z}_{3329}[x]/\langle x^{256}+1\rangle$ used in Kyber as an example. Since 3329 is a prime, we can only define a size-n cyclic FFT for $n|3328$. This doesn't permit splitting the polynomial ring into linear factors since $x^{256}+1 = \Phi_{512}$ and $512 \nmid 3328$. What we can do is introduce $x^2 \sim y$ and split $\left(\mathbb{Z}_{3329}[x]/\langle x^2 - y\rangle\right)[y]/\langle y^{128}+1\rangle$ into linear factors in y.

5.5.2 Striding

A closely related idea is striding – we regard $R[y]/\langle \boldsymbol{g}(y)\rangle$ as the coefficient ring. This is Nussbaumer (cf. Section 4.2) if we replace $x^v - y$ with an $\boldsymbol{h}(x)$, and ask $\boldsymbol{g}(y)|\Phi_{n'}(y)$ and $\boldsymbol{h}(x)|(x^{n'}-1)$ with $n' \geq 2v-1$. We also have striding Toom–Cook [Ber01, BMK+22] if $\boldsymbol{h}(x) = \prod_i (x - s_i)$ for $\{s_i\} \subset \mathbb{Q} \cup \{\infty\}$.

5.6 Toeplitz Matrix-Vector Product

This section goes through a generic technique converting a fast computation for $R[x]$ into a fast computation for $R[x]/\langle x^n - \alpha x - \beta\rangle$. We present the mathematical background in this section and will review the architectural insights in Section 6.2.

5.6.1 Bilinear System

We review a generic technique for bilinear systems adapted from [Win80, Theorem 6].

Theorem 1 ([Win80, Theorem 6] for R commutative)**.** Let R be a ring, $\mathcal{I}, \mathcal{J}, \mathcal{K}$ be finite index sets, and $(a_i)_{i\in\mathcal{I}}, (b_j)_{j\in\mathcal{J}}, (c_k)_{k\in\mathcal{K}}$ be tuples drawn from R. For a bilinear system

$$S_0 : \forall k \in \mathcal{K}, \sum_{i\in\mathcal{I}} \sum_{j\in\mathcal{J}} r_{(i,j,k)} a_i b_j$$

with $r_{(i,j,k)} \in R$, we construct the following bilinear systems:

$$S_1 : \forall j \in \mathcal{J}, \sum_{i\in\mathcal{I}} \sum_{k\in\mathcal{K}} r_{(i,j,k)} a_i c_k,$$

$$S_2 : \forall i \in \mathcal{I}, \sum_{j\in\mathcal{J}} \sum_{k\in\mathcal{K}} r_{(i,j,k)} c_k b_j.$$

Then any bilinear algorithm for one of S_0, S_1 or S_2 leads to algorithms for the other two.

One can prove Theorem 1 by defining a $|\mathcal{K}| \times |\mathcal{I}|$ matrix $B_{k,i} \coloneqq \left(\sum_{j\in\mathcal{J}} r_{(i,j,k)} b_j\right)$, and write S_0 as $B\boldsymbol{a}$ and S_2 as $B^T\boldsymbol{c}$ where $\boldsymbol{a}$ and $\boldsymbol{c}$ are the column representations of $(a_i)_{i\in\mathcal{I}}$ and $(c_k)_{k\in\mathcal{K}}$. See Appendix E for details. If we choose $r_{(i,j,k)} \coloneqq [\![i+j=k]\!]$ where $[\![]\!]$ is the Iverson bracket[11] and $|\mathcal{K}| = |\mathcal{I}| + |\mathcal{J}| - 1$, S_0 represents the coefficients of $\left(\sum_{i\in\mathcal{I}} a_i x^i\right)\left(\sum_{j\in\mathcal{J}} b_j x^j\right)$ in an obvious way. Then, S_2 becomes

$$S_2 : \forall i \in \mathcal{I}, \sum_{j\in\mathcal{J}} \sum_{k\in\mathcal{K}} [\![k-j=i]\!]\, c_k b_j.$$

[11] Iverson bracket is an indicator function for the truthfulness. The image of $[\![]\!]$ is $\{0,1\}$, which can be certainly embedded into a ring.

This is called a transposed multiplication [Sho99, Section 3] or a middle product [HQZ04]. [Fid73, Theorem 4 and 5] proved that transposing an algorithm results in same numbers of constant multiplications (ra_i for a constant r in R), non-constant multiplications (a_ib_j), and additions/subtractions with a linear difference. For the history of transposition principle, see [BCS13, Section 4].

We illustrate with the case $|\mathcal{I}| = |\mathcal{J}| = n$. $S_0 : \forall k \in \mathcal{K}, \sum_{i\in\mathcal{I}} \sum_{j\in\mathcal{J}} [\![i + j = k]\!]\, a_ib_j = \sum_{i\in\mathcal{I}, i\leq k} a_ib_{k-i}$ can be written as:

$$\begin{pmatrix} a_0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots \\ a_{n-1} & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}.$$

And $S_2 : \forall i \in \mathcal{I}, \sum_{j\in\mathcal{J}} \sum_{k\in\mathcal{K}} [\![k - j = i]\!]\, c_kb_j = \sum_{j\in\mathcal{J}} c_{i+j}b_j$ can be written as:

$$\begin{pmatrix} c_0 & \iddots & \iddots \\ \vdots & \iddots & \iddots \\ c_{n-1} & \cdots & c_{2n-2} \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}.$$

S_2 allows us to relate S_0 to polynomial multiplication modulo a polynomial.

5.6.2 Toeplitz Transform for $R[x]/\langle x^n - \alpha x - \beta\rangle$

Let M be an $n \times n$ matrix. We call M a Hankel matrix if $M_{i,j} = M_{i+1,j-1}$ for all possible i, j, and a Toeplitz matrix if $M_{i,j} = M_{i+1,j+1}$ for all possible i, j. Notice that a Hankel matrix can be converted into a Toeplitz matrix by multiplying an anti-diagonal matrix of ones and vice versa.

This section explains how to derive a fast computation for $R[x]/\langle x^n - \alpha x - \beta\rangle$ by looking at an already well-studied algebra monomorphism f multiplying two size-n polynomials in $R[x]$. There are four steps: (i) interpreting multiplication in $R[x]/\langle x^n - \alpha x - \beta\rangle$ as a Toeplitz matrix-vector product with no or little overhead; (ii) interpreting the Toeplitz matrix-vector product as a composition of applying an anti-diagonal matrix of ones and a Hankel matrix–vector product; (iii) rewriting the Hankel matrix–vector product as a bilinear system of the form S_2; and (iv) converting the computing task into a bilinear system of the form S_0. Once we go through all the ideas of (i) – (iv), we can now convert an f into an algorithm for $R[x]/\langle x^n - \alpha x - \beta\rangle$ via module–theoretic view. Notice that steps (ii) and (iii) are sometimes described as a single step. We describe them separately simply for clarity.

Steps (i) – (iii) are already shown in previous paragraphs. We now explain how to interpret the multiplication in $R[x]/\langle x^n - \alpha x - \beta\rangle$ as a Toeplitz matrix-vector product with potential post-processing. We define $\mathbf{Toeplitz}_n$ as the following function mapping a $(2n-1)$-tuple drawn from R to a Toeplitz matrix over R:

$$\mathbf{Toeplitz}_n : (z_{2n-2}, \ldots, z_0) \mapsto \begin{pmatrix} z_{n-1} & \cdots & z_0 \\ \vdots & \ddots & \ddots \\ z_{2n-2} & \ddots & \ddots \end{pmatrix}.$$

Let $\boldsymbol{a} = \sum_i a_i x^i, \boldsymbol{b} = \sum b_i x^i$ be size-n polynomials. We recall that computing $\sum_i c_i x^i = \boldsymbol{ab}$ in $R[x]$ can be regarded as the following matrix–vector product:

$$\begin{pmatrix} c_0 \\ \vdots \\ c_{2n-2} \end{pmatrix} = \begin{pmatrix} b_0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots \\ b_{n-1} & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Since $(c_0, \ldots, c_{n-1})$ can be computed with a Toeplitz matrix-vector product, we only need to convert reduction modulo $x^n - \alpha x - \beta$ into the manipulation of Toeplitz matrices. A standard approach for reducing modulo $x^n - \alpha x - \beta$ is multiplying $(c_n, \ldots, c_{2n-2})$ by α and β and adding the results to $(c_1, \ldots, c_{n-1})$ and $(c_0, \ldots, c_{n-2})$. Based on this, $\boldsymbol{ab} \bmod (x^n - \alpha x - \beta)$ can be written as

$$(M_0 + M_1 + M_2)\,\boldsymbol{a}$$

where

$$\begin{cases} M_0 = \mathbf{Toeplitz}_n(b_{n-1}, \ldots, b_0, 0, \ldots, 0), \\ M_1 = \mathbf{Toeplitz}_n(0, \ldots, 0, \beta b_{n-1}, \ldots, \beta b_1), \end{cases}$$

and

$$M_2 = \alpha \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 0 & b_{n-1} & \cdots & b_0 \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \end{pmatrix}.$$

A specialized approach for $\beta = 1$. We review the case $\beta = 1$ implied by [FH07, Section 3.2]. See Appendix D for an approach handling generic β with some overhead. See [HB95, FD05] for related works when $R = \mathbb{F}_2$. Since $\beta = 1$, $M_0 + M_1$ is the circulant matrix implementing $\boldsymbol{ab} \bmod (x^n - 1)$. Obviously, if we multiply a circulant matrix by a cyclic shift matrix (either left-multipling or right-multipling), we still have a circulant matrix. Let P be the matrix moving the 0-th row of a circulant matrix to the bottom. We find that both $P(M_0 + M_1)$ and PM_2 are Toeplitz matrices. Therefore, $P(M_0 + M_1 + M_2)$ is a Toeplitz matrix and we can implement $(M_0 + M_1 + M_2)\,\boldsymbol{a}$ as

$$(M_0 + M_1 + M_2)\,\boldsymbol{a} = P^{-1}\left(P\,(M_0 + M_1 + M_2)\,\boldsymbol{a}\right).$$

In Section 6.1, we will justify why cyclic shift matrices are practically efficient.

Padding. The last instrument is padding. Suppose we have an $n \times n$ Toeplitz matrix $T = \mathbf{Toeplitz}\,(z_{2n-2}, \ldots, z_0)$. For an $n' \geq n$, we can always pad T to an $n' \times n'$ Toeplitz matrix T' as follows:

$$T' = \mathbf{Toeplitz}\left(\underbrace{0, \ldots, 0}_{n'-n}, z_{2n-2}, \ldots, z_0, \underbrace{0, \ldots, 0}_{n'-n}\right).$$

The point is that if $n \times n$ Toeplitz matrices do not admit efficient applications, we can pad them to slighly larger ones admitting efficient applications [IKPC22, Section 3.1].

6 Vectorization

In this section, we review the formalization of vectorization by [Hwa23]. Since algebra homomorphisms can be characterized as matrix multiplications, their formulation is based on manipulations of matrices with standard bases unless specified otherwise. Section 6.1 formalizes vectorization-friendliness and permutation-friendliness, and Section 6.2 surveys the benefit of vector-by-scalar multiplications.

6.1 Formalization of Vectorization

Throughout this section, we assume there are v elements in a vector register.

6.1.1 Vectorization-Friendliness

We first identify a set `BlockDiag` of matrices that can be implemented efficiently with vector instructions. The set `BlockDiag` is served as a vehicle for defining vectorization-friendliness of algebra homomorphisms. Although `BlockDiag` is definitely platform-dependent, we fix `BlockDiag` to be a union of certain matrices and explain why they are usually suitable for vectorization. We define `BlockDiag` as a set of all possible block diagonal matrices with each block a $v' \times v'$ matrix of the following form for a v-multiple v' [Hwa23]:

1. Diagonal matrix: a matrix with non-diagonal entries all zeros.
2. Cyclic/negacyclic shift matrix: a matrix implementing $(a_i) \mapsto \left(a_{(i+c) \bmod v'}\right)$ (cyclic) or $(a_i) \mapsto \left((-1)^{[\![i+c \geq v']\!]} a_{(i+c) \bmod v'}\right)$ (negacyclic) for a non-negative integer c.

Diagonal matrices are suitable for vectorization since we can load v coefficients, multiply them by v constants, and store them back to memory with vector instructions. For cyclic/negacyclic shift matrices, we discuss how to implement them for the following vector instruction sets/extensions:

- Armv7/8-A Neon: For cyclic shifts, we extract consecutive elements from a pair of vector registers with the instruction `ext`. For negacyclic shifts, we negate one of the registers before applying `ext` [HLY24, Algorithm 5].
- AVX2: For cyclic shifts, we perform unaligned loads, shuffle the last vector register, and store the vectors to memory. Again, the last vector register is negated for negacyclic shifts [BBCT22].

Qualification of vectorization-friendliness. Let f be an algebra homomorphism, and M_f be the matrix form of f. We call f vectorization-friendly if

$$M_f = \prod_i \left(M_{f_i} \otimes I_v\right) S_{f_i}$$

for $S_{f_i} \in S$ and some matrices M_{f_i}. We first observe that vector instruction sets usually provide instructions loading/storing consecutive v coefficients from/to memory. The tensor product $M_{f_i} \otimes I_v$ ensures that each v-chunk is regarded as a whole while applying $M_{f_i} \otimes I_v$. Since S_{f_i}'s are assumed to be vectorization-friendly, the matrix product $\prod_i \left(M_{f_i} \otimes I_v\right) S_{f_i}$ is vectorization-friendly.

6.1.2 Permutation-Friendliness

We introduce the notion "permutation-friendliness". Conceptually, permutation-friendliness stands for vectorization-friendliness after applying a special type of permutation – interleaving.

The set `Interleave` of interleaving matrices. Again, let v' be a multiple of v. We define the transposition matrix $T_{v'^2}$ as the $v'^2 \times v'^2$ matrix permuting the elements as if transposing a $v' \times v'$ matrix. We illustrate the case $v' = 2$ with Algorithm 1 for Neon and Algorithm 2 for AVX2. Now we are ready to specify the set `Interleave` of interleaving matrices. We call a matrix M interleaving matrix with step v' if it takes the form

$$M = (\pi' \otimes I_{v'})\,(I_m \otimes T_{v'^2})\,(\pi \otimes I_{v'})$$

for a positive integer m and permutation matrices π, π' permuting mv' elements. The set `Interleave` consists of interleaving matrices of all possible steps and is closed under inversion.

Algorithm 1 `trn{1, 2}` permuting double words in Armv8.0-A Neon registers.

Inputs: $(\texttt{v0}, \texttt{v1}) = (a_0 \ \texttt{||}\ a_1, b_0 \ \texttt{||}\ b_1)$
Outputs: $(\texttt{v2}, \texttt{v3}) = (a_0 \ \texttt{||}\ b_0, a_1 \ \texttt{||}\ b_1)$

```
1: trn1 v2.2D, v0.2D, v1.2D
2: trn2 v3.2D, v0.2D, v1.2D
```

Algorithm 2 `vperm2i128` permuting double words in AVX2 `%ymm` registers.

Inputs: $(\texttt{\%ymm0}, \texttt{\%ymm1}) = (a_0 \ \texttt{||}\ a_1, b_0 \ \texttt{||}\ b_1)$
Outputs: $(\texttt{\%ymm2}, \texttt{\%ymm3}) = (a_0 \ \texttt{||}\ b_0, a_1 \ \texttt{||}\ b_1)$

```
1: vperm2i128 %ymm2, %ymm0, %ymm1, 0x20
2: vperm2i128 %ymm3, %ymm0, %ymm1, 0x31
```

Qualification of permutation-friendliness. We now define permutation-friendliness formally as follows. We call an algebra homomorphism g permutation-friendly if we can factor its matrix form M_g as

$$M'_g = \prod_i S_{g_i} M_{g_i}$$

for $S_{g_i} \in$ `Interleave` and vectorization-friendly M_{g_i}'s. The permutation-friendliness of g^{-1} follows immediately from g.

6.1.3 Vectorization with Vector-By-Vector Instructions

Generally, while computing with vector-by-vector instructions, we choose algebra homomorphisms f and g such that f is vectorization-friendly and g is permutation-friendly. Their composition $g \circ f$ then admits a suitable mapping to our target vector instruction set. Concretely, we vectorize f, transpose the coefficients, and vectorize g.

6.2 Vector–By–Scalar Multiplication Instructions

For an $m \times n$ Toeplitz matrix $M = \mathbf{Toeplitz}_{m\times n}(a_{m-1}, \ldots, a_0, a'_1, \ldots, a'_{n-1})$ over the ring R, [CCHY23] demonstrated the benefit of vector-by-scalar multiplication instructions when applying M to a vector $v = (b_0, \ldots, b_{n-1})$. For simplicity, we demonstrate with the case $m = n = 4$ and $R = \mathbb{Z}_{2^{32}}$:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_0 & a'_1 & a'_2 & a'_3 \\ a_1 & a_0 & a'_1 & a'_2 \\ a_2 & a_1 & a_0 & a'_1 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

For deploying vector-by-scalar multiplications, the key is to identify the reuses of the scalar operands. Obviously, we find that each of $b_0, \ldots, b_3$ is involved in four multiplications in R: we compute a_0b_0, a_1b_0, a_2b_0, a_3b_0 for the operand b_0, etc. Therefore, an obvious choice is to map each columns to a vector and apply vector-by-scalar multiplications. For a given Toeplitz matrix $\mathbf{Toeplitz}(a_3, \cdots, a_0, a'_1, \cdots, a'_3)$, the construction of such vectors are usually instantiated in two ways: either loading from the addresses pointing $a_0, \ldots, a_3$, or loading the first column and first row and combining them with special instructions. Algorithm 3 constructs the columns with only memory loads, and Algorithm 4 replaces some memory instructions with permutation instructions.

Algorithm 3 Constructing the columns of a Toeplitz matrix from its array form with memory loads [CCHY23].

Inputs: Array `M[8] = {`0`,` a'_3`,` a'_2`,` a'_1`,` a_0`,` a_1`,` a_2`,` a_3`}`.
Outputs: Vector registers

$$\begin{cases} \texttt{t0} = a_3 \ \texttt{||}\ a_2 \ \texttt{||}\ a_1 \ \texttt{||}\ a_0, \\ \texttt{t1} = a_2 \ \texttt{||}\ a_1 \ \texttt{||}\ a_0 \ \texttt{||}\ a'_1, \\ \texttt{t2} = a_1 \ \texttt{||}\ a_0 \ \texttt{||}\ a'_1 \ \texttt{||}\ a'_2, \\ \texttt{t3} = a_0 \ \texttt{||}\ a'_1 \ \texttt{||}\ a'_2 \ \texttt{||}\ a'_3. \end{cases}$$

```
1: t0 = M[4-7]
2: t1 = M[3-6]
3: t2 = M[2-5]
4: t3 = M[1-4]
5:                                                    ▷ Memory load.
```

Algorithm 4 Constructing the columns of a Toeplitz matrix from its array form with memory loads and permutations [Hwa23].

Inputs: Array `M[8] = {`0`,` a'_3`,` a'_2`,` a'_1`,` a_0`,` a_1`,` a_2`,` a_3`}`.
Outputs: Vector registers

$$\begin{cases} \texttt{t0} = a_3 \ \texttt{||}\ a_2 \ \texttt{||}\ a_1 \ \texttt{||}\ a_0, \\ \texttt{t1} = a_2 \ \texttt{||}\ a_1 \ \texttt{||}\ a_0 \ \texttt{||}\ a'_1, \\ \texttt{t2} = a_1 \ \texttt{||}\ a_0 \ \texttt{||}\ a'_1 \ \texttt{||}\ a'_2, \\ \texttt{t3} = a_0 \ \texttt{||}\ a'_1 \ \texttt{||}\ a'_2 \ \texttt{||}\ a'_3. \end{cases}$$

```
1: t0 = M[4-7]
2: t3 = M[0-3]
3:                                                    ▷ Memory load.
4: t1 = ext(t3, t0, 3)
5: t2 = ext(t3, t0, 2)
6: t3 = ext(t3, t0, 1)
```

After constructing the matrix column-wise, we now identify the column vector c as the sum of columns scaled by the corresponding elements in v. In other words,

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = b_0 \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} + b_1 \begin{pmatrix} a'_1 \\ a_0 \\ a_1 \\ a_2 \end{pmatrix} + b_2 \begin{pmatrix} a'_2 \\ a'_1 \\ a_0 \\ a_1 \end{pmatrix} + b_3 \begin{pmatrix} a'_3 \\ a'_2 \\ a'_1 \\ a_0 \end{pmatrix}.$$

Algrotihm 5 is an illustration.

Algorithm 5 Applying a 4×4 Toeplitz matrix with vector-by-scalar multiplication instructions [CCHY23].

Inputs: **Toeplitz**$(a_3, a_2, a_1, a_0, a'_1, a'_2, a'_3)$, (b_0, b_1, b_2, b_3).
Outputs: **Toeplitz**$(a_3, a_2, a_1, a_0, a'_1, a'_2, a'_3)(b_0, b_1, b_2, b_3)$.

1: $\texttt{t0} = a_3||a_2||a_1||a_0$
2: $\texttt{t1} = a_2||a_1||a_0||a'_1$
3: $\texttt{t2} = a_1||a_0||a'_1||a'_2$
4: $\texttt{t3} = a_0||a'_1||a'_2||a'_3$
5: ▷ Applying Algorithms 3 or 4.
6: $\texttt{c} = \texttt{mul}(\texttt{t0}, b_0)$
7: $\texttt{c} = \texttt{mla}(\texttt{c}, \texttt{t1}, b_1)$
8: $\texttt{c} = \texttt{mla}(\texttt{c}, \texttt{t2}, b_2)$
9: $\texttt{c} = \texttt{mla}(\texttt{c}, \texttt{t3}, b_3)$

7 Case Studies

We go through several case studies in this section. Section 7.1 compares Barrett and Montgomery multiplications with Dilithium implementations for modular arithmetic, and Section 7.2 compares Montgomery and Plantard multiplications with Kyber implementations. We then go through several algebraic techniques and vectorization. Section 7.3 explains how to exploit the matrix-to-vector structure with Saber as an example, Section 7.4 reviews the benefit of Toeplitz matrix-vector multiplication with NTRU as an example, and Section 7.5 details the design choices for vectorization with NTRU Prime as an example.

7.1 Dilithium : Barrett vs Montgomery Modular Arithmetic

This section reviews the modular arithmetic used in Dilithium. In Dilithium, we want to multiply polynomials in $\mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ for $q = 2^{23} - 2^{13} + 1$. Since q is a prime supporting a size-2^{13} cyclic FFT, we can split $x^{256}+1$ into linear factors (recall that $x^{256}+1 = \Phi_{512}|(x^{512}-1)$ and $512|2^{13}$). The choice of FFT is already determined by the specification – one of the operands is assumed to be transformed. The remaining question is to compute the modular arithmetic efficiently. For a 32-bit value a, modular reduction is fairly simple. Since q is fairly close to 2^{23}, $a - \left\lfloor \frac{a}{2^{23}} \right\rceil q$ is a representative of $a \bmod^{\pm} q$ within an acceptable range[12].

How about modular multiplications? In Section 2, we have three classes of modular multiplications – Montgomery, Barrett, and Plantard. We compare Montgomery and Barrett multiplications for Dilithium in this section. Essentially, if we briefly categorize multiplication operations into three groups: (i) low products `{mul, mla, mls}lo`, (ii) high products `{mul, mla, mls}hi`, and (iii) double-size products `{mul, mla, mls}l`, Barrett multiplication requires two low products and one high product whereas Montgomery multiplication require one low product and two high/double-size products. Table 1 is a summary. In practice, low products are fairly common, while high products and double-size products usually lack additive or subtractive variants. See Table 2 for a summary. For the actual instructions, see [ARM21b, Section A4.4.3], [ARM21a, Sections C3.5.14, C3.5.16, and C3.5.18], and [Ora14, Section 3.7].

[12]The actual range for $-2^{31} \le a < 2^{31}$ is $[-4186113, 4194303]$ by brute-force testing.

Table 1: Overview of required multiplication instructions of Barrett and Montgomery multiplications.

	`mullo`	`mlslo`	`mulhi`	`mlshi`	`mull`	`mlal`
Barrett	1	1	1	0	0	0
Montgomery	1	0	0	0	1	1
Montgomery	1	0	1	1	0	0

Table 2: Overview of the available forms of input-independent signed multiplication instructions.

Low products			
ISA	`mullo`	`mlalo`	`mlslo`
Armv7-M	✓($\mathtt{R} = 2^{32}$)	✓($\mathtt{R} = 2^{32}$)	✓($\mathtt{R} = 2^{32}$)
Armv7E-M	✓($\mathtt{R} = 2^{32}$)	✓($\mathtt{R} = 2^{32}$)	✓($\mathtt{R} = 2^{32}$)
Armv8.0-A	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)
Armv8.1-A	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)
AVX2	✓($\mathtt{R} = 2^{16}, 2^{32}$)	-	-
High products			
ISA	`mulhi`	`mlahi`	`mlshi`
Armv7-M	-	-	
Armv7E-M	✓($\mathtt{R} = 2^{32}$)	-	-
Armv8.0-A	✓($\mathtt{R} = 2^{16}, 2^{32}$)	-	-
Armv8.1-A	✓($\mathtt{R} = 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{16}, 2^{32}$)
AVX2	✓($\mathtt{R} = 2^{16}$)	-	-
Double-size products			
ISA	`mull`	`mlal`	`mlsl`
Armv7-M	-	-	-
Armv7E-M	✓($\mathtt{R} = 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{16}, 2^{32}$)	-
Armv8.0-A	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)
Armv8.1-A	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)	✓($\mathtt{R} = 2^{8}, 2^{16}, 2^{32}$)
AVX2	✓($\mathtt{R} = 2^{32}$)	-	-

7.1.1 Armv8-A Neon Implementations

For vectorized implementations, [BHK+22b] implemented Barrett multiplication and the subtractive variant of Montgomery multiplication with Armv8.0-A Neon. For Armv8.0-A, there are multiplication instructions `sq{, r}dmulh` computing and doubling the high-products – For two values a and b, `sqdmulh` computes $\left\lfloor \frac{2ab}{\mathtt{R}} \right\rfloor$ with saturations, and `sqrdmulh` applies rounding $\lfloor \rceil$ instead of flooring $\lfloor \rfloor$. For Montgomery multiplication, [BHK+22b] implemented

$$\frac{1}{2}\left(\left\lfloor \frac{2ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{2\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \right\rfloor\right)$$

as shown in Algorithm 6. One can show that $\frac{1}{2}\left(\left\lfloor \frac{2ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{2\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right)q}{\mathtt{R}} \right\rfloor\right) = \left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right)q}{\mathtt{R}} \right\rfloor$. We leave the justification to readers. For Barrett multiplication,

[BHK+22b] implemented

$$ab - \left\lfloor \frac{a \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q$$

for $\lfloor \rceil_2 \coloneqq r \mapsto 2\left\lfloor \frac{r}{2} \right\rceil$ as shown in Algorithm 7. Since there is a subtractive form for low products only, Barrett multiplication saves one addition/subtraction compared to Montgomery multiplication. Additionally, [HLY24, Algorithms 3 and 4] proposed the additive and subtractive variants computing representatives of $d \pm ab \bmod q$ and [Bo-22] found their benefits for computing radix-2 Cooley–Tukey butterflies on platforms implementing barrel shift (for example, Cortex-M4). We leave the exploration of the additive/subtractive Barrett multiplication to the readers.

Algorithm 6 Single-width Montgomery multiplication [BHK+22b, Algorithm 12].

Inputs: Values $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right)$.

Output: $\mathtt{c} = \frac{1}{2}\left(\left\lfloor \frac{2ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{2\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right)q}{\mathtt{R}} \right\rfloor\right)$.

1: `sqdmulh c, a, b` ▷ $\mathtt{c} = \left\lfloor \frac{2ab}{\mathtt{R}} \right\rfloor$.
2: `mul t, a,` $bq^{-1} \bmod^{\pm} \mathtt{R}$ ▷ $\mathtt{t} = abq^{-1} \bmod^{\pm} \mathtt{R}$.
3: `sqdmulh t, t, q` ▷ $\mathtt{t} = \left\lfloor \frac{2\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right)q}{\mathtt{R}} \right\rfloor$.
4: `shsub c, c, t` ▷ $\mathtt{c} = \frac{1}{2}\left(\left\lfloor \frac{2ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{2\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right)q}{\mathtt{R}} \right\rfloor\right)$.

Algorithm 7 Single-width Barrett multiplication [BHK+22b, Algorithm 10].

Inputs: Values $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right)$.

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

1: `mul lo, a, b` ▷ $\mathtt{lo} = ab$.
2: `sqrdmulh hi, a,` $\frac{\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{2}$ ▷ $\mathtt{hi} = \left\lfloor \frac{a\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil$.
3: `mls lo, hi, q` ▷ $\mathtt{lo} = ab - \left\lfloor \frac{a\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q$.

7.1.2 Armv7-M Implementations

This section reviews [HKS23]'s observation of Barrett multiplication on Cortex-M3. Cortex-M3 implements the ISA Armv7-M where `mul/mla/mls`, `{u, s}{mul, mla, mls}l` are the only multiplication instructions. However, double-size products `{u, s}{mul, mla, mls}l` take input-dependent time [ARM10] and can only be used for computing public data. For computing the 32-bit NTTs of secret data in Dilithium, [GKS21] implemented 32-bit Montgomery multiplication while emulating the double-size products with `mul/mla/mls` as shown in Algorithm 8.

[HKS23] proposed using Barrett multiplication for 32-bit modular multiplications on Cortex-M3. They observed the following:

B1 While Montgomery multiplication requires two double-size/high products and one low product, Barrett multiplication requires one high product and two low products.

B2 In Barrett multiplication, the high product only needs to be approximately correct.

Algorithm 8 Constant-time 32-bit Montgomery multiplication [GKS21, Listing 7]

Inputs: $\mathtt{al} + \mathtt{ah} \cdot \mathtt{R} = a, \mathtt{bl} + \mathtt{bh} \cdot \mathtt{R} = b$.

Output: $\mathtt{tmph} = \frac{ab + \left(-abq^{-1} \bmod^{\pm} \mathtt{R}\right)q}{\mathtt{R}}$.

```
1: SBSMULL tmpl, tmph, al, ah, bl, bh      ▷ tmpl + tmph · R = ab.
2: mul       ah, tmpl, −q^{-1} mod± R      ▷ ah = abq^{-1} mod± R.
3: ubfx      al,   ah,  #0, #16
4: asr       ah,   ah, #16                 ▷ al + ah · R = −abq^{-1} mod± R.
5: SBSMLAL tmpl, tmph, al, ah, ql, qh      ▷ tmph = (ab + (−abq^{-1} mod± R)q)/R.
```

Observation B1 saves one emulation of the double-size/high product, and observation B2 allows one to deploy a faster emulation with tolerable errors.

Let's consider $[\![]\!]$ the following integer approximation:

$$\forall r \in \mathbb{R}, [\![r]\!] = a_{r,h} b_h + \left\lfloor \frac{a_{r,l} b_h}{\sqrt{\mathtt{R}}} \right\rceil + \left\lfloor \frac{a_{r,h} b_l}{\sqrt{\mathtt{R}}} \right\rceil$$

for $a_{r,l} + a_{r,h}\sqrt{\mathtt{R}} = \frac{r\mathtt{R}}{\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}$ and $b_l + b_h\sqrt{\mathtt{R}} = \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$. For $-\frac{q}{2} \leq b < \frac{q}{2}$ and $-\frac{\mathtt{R}}{2} \leq a_{r,l} + a_{r,h}\sqrt{\mathtt{R}} < \frac{\mathtt{R}}{2}$, [HKS23] showed that $|r - [\![r]\!]| \leq 3$ and $\left|\bmod^{[\![]\!]} \mathtt{R}\right| \leq \frac{7\mathtt{R}}{2}$, and computed

$$ab - \left[\!\!\left[\frac{a \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right]\!\!\right] q$$

as a representative of $ab \bmod^{\pm} q$ with range bounded by

$$\frac{|a| \left|\bmod^{\pm} q\right| + \left|\bmod^{[\![]\!]} \mathtt{R}\right| |q|}{\mathtt{R}} \leq \frac{|a| \frac{q}{2} + \frac{7}{2}\mathtt{R}q}{\mathtt{R}} = \frac{q}{2}\left(7 + \frac{|a|}{\mathtt{R}}\right).$$

Algorithm 9 is an illustration.

Algorithm 9 Constant-time 32-bit Barrett multiplication with approximated high product [HKS23].

Inputs: $\mathtt{a} = a, \mathtt{b} = b$.

Output: $\mathtt{t3} = ab - \left[\!\!\left[\frac{a \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right]\!\!\right] q$.

```
1: mul           t3,  a,   b                ▷ t3 = ab mod± R.
2: ubfx          t0,  a,  #0, #16
3: asr            a,  a, #16                ▷ t0 + a · R = a.
4: smmulr_approx t1,  a, bhi, t0, blo, t2  ▷ t1 = [[ a⌊bR/q⌉ / R ]].
5: mls           t3, t1,   q, t3           ▷ t3 = ab − [[ a⌊bR/q⌉ / R ]] q.
```

7.2 Kyber : Montgomery vs Plantard Modular Arithmetic

In this section, we compare the applications of Montgomery and Plantard multiplications to Kyber, which requires modular multiplication for the coefficient ring $\mathbb{Z}_{3329}$. We assume $\mathtt{R} = 2^{16}$ in this section. [HZZ+22] and [AMOT22] independently found the signed Plantard multiplications. [HZZ+22] identified the benefit of 16-bit modular arithmetic if there

are 16×32-bit multiplication instructions, and [AMOT22] identified the benefit of 32-bit modular arithmetic using only 64-bit multiplication instructions. [HZZ+] later implemented 16-bit Plantard multiplication following [AMOT22]'s insights when there are no 16×32-bit multiplication instructions.

7.2.1 Armv7-M Implementations

In Armv7-M, since all the registers contain 32-bit values, we can compute the 16-bit Montgomery multiplication with `mul` and `mla` in an obvious way (cf. Algorithm 10). [HZZ+] implemented 16-bit Plantard multiplication with [AMOT22]'s insights. For 16-bit values $a \in \left[-\frac{\mathtt{R}}{2}, -\frac{\mathtt{R}}{2}\right)$ and $b \in \left[-\frac{q}{2}, \frac{q}{2}\right)$, we compute

$$\left\lfloor \frac{\left\lfloor \frac{-abq^{-1} \bmod {}^{\pm}\mathtt{R}^2}{\mathtt{R}} \right\rfloor q + 2^{\alpha} q}{\mathtt{R}} \right\rfloor$$

as a representative of $-ab\mathtt{R}^{-2} \bmod {}^{\pm}q$. See 11 for an illustration. If b is known in prior, we skip the computation for $-bq^{-1} \bmod {}^{\pm}\mathtt{R}^2$ and cancel out the scaling $-\mathtt{R}^2 \bmod {}^{\pm}q$ by precomputing $-\left(-b\mathtt{R}^2 \bmod {}^{\pm}q\right) q^{-1} \bmod {}^{\pm}\mathtt{R}^2$.

Algorithm 10 16-bit Montgomery multipliation with Armv7-M [GKS21].

Inputs: Values $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right)$.
Output: $\mathtt{t0} = ab + \left(-abq^{-1} \bmod {}^{\pm}\mathtt{R}\right) q$.

1: `mul t0,` a`,` b ▷ $\mathtt{t0} = ab$.
2: `mul t1, t0,` $-q^{-1} \bmod {}^{\pm}\mathtt{R}$
3: `sxth t1, t1, #0, #16` ▷ $\mathtt{t1} = -abq^{-1} \bmod {}^{\pm}\mathtt{R}$.
4: `mla t0, t1,` q`, t0` ▷ $\mathtt{t0} = ab + \left(-abq^{-1} \bmod {}^{\pm}\mathtt{R}\right) q$.
5: ▷ The desired result is stored in the upper half.

Algorithm 11 16-bit Plantard multiplication with Armv7-M [HZZ+].

Inputs: Values $a \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right), -bq^{-1} \in \left[-\frac{\mathtt{R}^2}{2}, \frac{\mathtt{R}^2}{2}\right)$.
Output: $\mathtt{t} = \left(\left\lfloor \frac{-abq^{-1} \bmod {}^{\pm}\mathtt{R}^2}{\mathtt{R}} \right\rfloor + 2^{\alpha}\right) q$.

1: `mul t,` b`,` $-q^{-1} \bmod {}^{\pm}\mathtt{R}^2$ ▷ $\mathtt{t} = -bq^{-1} \bmod {}^{\pm}\mathtt{R}^2$.
2: `mul t, t,` a ▷ $\mathtt{t} = -abq^{-1} \bmod {}^{\pm}\mathtt{R}^2$.
3: `add t,` 2^{α}`, t, asr #16` ▷ $\mathtt{t} = \left\lfloor \frac{-abq^{-1} \bmod {}^{\pm}\mathtt{R}^2}{\mathtt{R}} \right\rfloor + 2^{\alpha}$.
4: `mul t, t,` q ▷ $\mathtt{t} = \left(\left\lfloor \frac{-abq^{-1} \bmod {}^{\pm}\mathtt{R}^2}{\mathtt{R}} \right\rfloor + 2^{\alpha}\right) q$.
5: ▷ The desired result is stored in the upper half.

7.2.2 Armv7E-M Implementations

We briefly compare Montgomery and Plantard multiplications with the Digital Signal Processing extension in Armv7E-M where "E" stands for "extension". [ABCG20] showed that 16-bit Montgomery multiplication can be implemented with three 16-bit multiplication instructions from the extension as shown in Algorithm 12. Recently, [HZZ+22] found that the multiplication instruction `smulwb` fits for 16-bit Plantard multiplication. Algorithm 13 is an illustration. If one of the multiplicands is known in prior, we can remove one multiplication and cancel out the scaling with precomputation as shown in previous section.

Algorithm 12 16-bit Montgomery multiplication with Armv7E-M [ABCG20].

Inputs: `lo(a)` $= a_l$, `lo(b)` $= b_l$.

Outputs: `hi(th)` $= \frac{a_l b_l + \left(-a_l b_l q^{-1} \bmod {}^{\pm}\mathtt{R}\right) q}{\mathtt{R}}$.

1: `smulbb th, a, b` ▷ `hi` $= a_l b_l$.

2: `smulbb tl, th,` $-q^{-1} \bmod {}^{\pm}\mathtt{R}$ ▷ `lo` $= (a_l b_l \bmod {}^{\pm}\mathtt{R})\left(-q^{-1} \bmod {}^{\pm}\mathtt{R}\right)$.

3: `smlabb th, tl, q, th` ▷ `th` $= a_l b_l + \left(-a_l b_l q^{-1} \bmod {}^{\pm}\mathtt{R}\right) q$.

4: ▷ The desired result is stored in the upper half.

Algorithm 13 16-bit Plantard multiplication with Armv7E-M [HZZ+22]

Inputs: `lo(a)` $= a_l, b \in \left[-\frac{q}{2}, \frac{q}{2}\right)$.

Outputs: `hi(t)` $= \left\lfloor \frac{\left\lfloor \frac{-a_l b q^{-1} \bmod {}^{\pm}\mathtt{R}^2}{\mathtt{R}} \right\rfloor q + 2^{\alpha} q}{\mathtt{R}} \right\rfloor$.

1: `mul t, b,` $-q^{-1} \bmod {}^{\pm}\mathtt{R}^2$ ▷ `t` $= -bq^{-1} \bmod {}^{\pm}\mathtt{R}^2$.

2: `smulwb t, t, a` ▷ `t` $= \left\lfloor \frac{a_l\left(-bq^{-1} \bmod {}^{\pm}\mathtt{R}^2\right)}{\mathtt{R}} \right\rfloor$.

3: `smlabb t, t, q,` $2^{\alpha} q$ ▷ `t` $= \left\lfloor \frac{-a_l b q^{-1} \bmod {}^{\pm}\mathtt{R}^2}{\mathtt{R}} \right\rfloor q + 2^{\alpha} q$.

4: ▷ The desired result is stored in the upper half.

7.3 Saber : Homomorphism Caching

In Saber, the most performance-critical polynomial operation is multiplying $l \times l$ matrix by an $l \times 1$ vector over the polynomial ring $\mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle$. We review the benefit of caching algebra and module homomorphisms.

Algebra homomorphism caching. Let $f : \mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle \to S$ be an algebra monomorphism, $\cdot_S$ be the multiplication in S, and $+_S$ be the addition in S. We denote $\mathcal{C}(-)$ as the cost function of a map. If we apply f to all the polynomials, compute matrix–vector multiplication over S, and transform back to a vector over $\mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle$, the total cost is

$$(l^2+l)\mathcal{C}(f) + l^2\mathcal{C}(\cdot_S) + (l^2-l)\mathcal{C}(+_S) + l\mathcal{C}(f^{-1}).$$

Optimizations for the matrix–vector multiplication over $\mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle$ should base the comparisons on the dominating term $\mathcal{C}(f)+\mathcal{C}(\cdot_S)+\mathcal{C}(+_S)$. [KRS19] chose f as Toom–Cook but didn't exploit the homomorphic property. [MKV20] exploited the homomorphic property for Toom–Cook, and [CHK+21] chose f as an FFT. The FFT-type approaches for Saber remain the fastest [CHK+21, ACC+22, BHK+22b].

Module homomorphism caching. In the previous paragraph, we have seen the importance of caching algebra homomorphisms. [BHK+22b] introduced "asymmetric multiplication" which falls into module homomorphism caching. For a polynomial $\boldsymbol{a} \in \mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle$ and an algebra homomorphism $f : \mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle \to S$, we first regard $f(\boldsymbol{a})$ as a module homomorphism mapping $f(\boldsymbol{b})$ to $f(\boldsymbol{a})f(\boldsymbol{b})$ for $\boldsymbol{b} \in \mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle$. If $f(\boldsymbol{a})$ amounts to polynomial multiplications modulo $x^v - \zeta$, we can turn $f(\boldsymbol{a})$ into a special kind of module homomorphism – Toeplitz matrix-vector multiplication (cf. Section 5.6). In practice, the Toeplitz matrix conversion of $f(\boldsymbol{a})$ is cached. This is called **asymmetric multiplication** [BHK+22b, Section 4.2].

7.4 NTRU : Toeplitz matrix-vector product

Section 5.6 discusses how to turn arbitrary algebra homomorphisms multiplying size-n polynomials in $R[x]$ into a Toeplitz matrix-vector product for multiplying in $R[x]/\langle x^n - \alpha x - \beta \rangle$. In Section 6.2, we discuss the benefit of computing Toeplitz matrix-vector products with vector-by-scalar multiplications. We review the Toeplitz matrix-vector product approach for multiplying polynomials in $\mathbb{Z}_{2048}[x]/\langle x^{677} - 1 \rangle$ used by the NTRU parameter set `ntruhps2048677`.

7.4.1 Armv7E-M Implementation

[IKPC22] applied Toeplitz matrix-vector product with Karatsuba and Toom–Cook. They first considered the following sequence of Karatsuba and Toom–Cook multiplying two size-720 polynomials:

$$\mathbf{TC}\text{-}4 \to \mathbf{K}\text{-}3 \to \mathbf{K}\text{-}3 \to \mathbf{K}\text{-}2$$

where $\mathbf{K}$-2 is the usual Karatsuba in Section 3.4 and $\mathbf{K}$-3 is the subtractive variant of 3-way Karatsuba[13]. They then took the dual of Toom–Cook, Karatsuba, and their inverses, and formed Toeplitz matrix-vector products as shown in Section 5.6. [IKPC22] identified that one no longer needs to reduce modulo a polynomial since it is merged with the polynomial multiplication itself (cf. Section 5.6.2).

7.4.2 Armv8-A Implementation

Shortly after, [CCHY23] explored the vectorization of Toeplitz matrix-vector products with Armv8-A. They started with the following sequence of Karatsuba and Toom–Cook multiplying two size-720 polynomials:

$$\mathbf{TC}\text{-}5 \to \mathbf{TC}\text{-}3 \to \mathbf{TC}\text{-}3 \to \mathbf{K}\text{-}2$$

and took the dual of all the homomorphisms. They showed that small-dimensional power-of-two Toeplitz matrix-vector product can be implemented efficiently for the following reasons: (i) one can construct Toeplitz matrices efficiently from its first row and column (cf. Section 6.2) and (ii) the existence of vector-by-scalar multiplication instructions allow one to apply the outer-product-based matrix-vector multiplication while avoiding permutations and reducing register pressure significantly [CCHY23]. See [CCHY23, Section 5.1] for more details on memory optimizations while inverting Karatsuba and Toom–Cook.

7.5 NTRU Prime : Vectorized FFTs

In this section, we go through a detailed analysis of vectorized polynomial multipliers in NTRU Prime. Our central objective is to answer the following question:

How FFT-, vectorization-, and permutation-friendly the coefficient ring is?

For simplicity, we focus on the polynomial ring $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1 \rangle$ used in the parameter sets `ntrulpr761` and `sntrup761`. We first discuss a generic approach using Schönhage and Nussbaumer for maintaining the friendliness while exploiting no algebraic properties of the polynomial ring. Schönhage and Nussbaumer usually adjoin algebraic structures for friendliness with expenses. We then systematically analyze how to exploit the algebraic structure endowed in $\mathbb{Z}_{4591}$, showing that $\mathbb{Z}_{4591}$ actually admits FFT, vectorization, and permutation-friendly transformations. Observe that $4591 - 1 = 2 \cdot 3^3 \cdot 5 \cdot 17$ and $4591^2 - 1 = 2^5 \cdot 3^3 \cdot 5 \cdot 7 \cdot 17 \cdot 41$, we summarize the following findings from the works [HLY24, Hwa23].

[13] In principle, we compute all possible $a_i b_i$ and $(a_i - a_j)(b_i - b_j)$ for $i \neq j$ so arbitrary $a_i b_j$ can be derived by only additions and subtractions, see [WP06, Section 3.2] for details.

- We qualify $\mathbb{Z}_{4591}$ as an FFT-friendly prime by considering the application of Good–Thomas and Rader's FFTs based on the factorization of $4591-1$ and Bruun's FFT based on the factorization of 4591^2-1 [HLY24].
- We qualify $\mathbb{Z}_{4591}$ as a vectorization-friendly prime since the product $2^5\cdot 3^3\cdot 5\cdot 17 = 73440$ ($73440 = 16(4591-1)|(4591^2-1)$) allows a wide range of FFTs resulting small-dimensional power-of-two polynomial multiplications [HLY24].
- We qualify $\mathbb{Z}_{4591}$ as a permutation-friendly prime since 3, 5, and 17 are Fermat primes, and truncating Fermat-prime-size Rader's FFTs gives power-of-two cyclic convolutions [Hwa23].

We review two AVX2-optimized implementations in this section: (i) [BBCT22]'s approach with truncated Schönhage and Nussbaumer FFTs, and (ii) [Hwa23]'s approach with truncated Rader, Good–Thomas, and Bruun FFTs.

7.5.1 A Generic Approach with Truncated Schönhage and Nussbaumer FFTs

Let's recall the AVX2-optimized polynomial multiplication for `ntrulpr761/sntrup761` from [BBCT22]. For multiplying polynomials in $\mathbb{Z}_{4591}[x]/\langle x^{761}-x-1\rangle$, [BBCT22] computed the product in $\mathbb{Z}_{4591}[x]/\langle (x^{512}-1)(x^{1024}+1)\rangle$ as follows. They first applied Schönhage replacing $x^{32}-y$ with $x^{64}+1$:

$$\frac{\mathbb{Z}_{4591}[x]}{\langle (x^{512}-1)(x^{1024}+1)\rangle} \overset{\eta_0}{\cong} \frac{\frac{\mathbb{Z}_{4591}[x]}{\langle x^{32}-y\rangle}[y]}{\langle (y^{16}-1)(y^{32}+1)\rangle} \overset{\eta_1}{\hookrightarrow} \frac{\frac{\mathbb{Z}_{4591}[x]}{\langle x^{64}+1\rangle}[y]}{\langle (y^{16}-1)(y^{32}+1)\rangle}.$$

Since x^2 is a principal 64-th root of unity in $\mathbb{Z}_{4591}[x]/\langle x^{64}+1\rangle$, we have $(y^{16}-1)(y^{32}+1) = \prod_{i\not\equiv 2 \pmod 4}(y-x^{2i})$ over $\mathbb{Z}_{4591}[x]/\langle x^{64}+1\rangle$. We find Schönhage's FFT vectorization-friendly since $64 = 4\cdot 16$. After splitting the polynomial ring in y, the remaining problem is multiplying in $\mathbb{Z}_{4591}[x]/\langle x^{64}+1\rangle$. [BBCT22] interleaved the polynomials with no leftovers and applied Nussbaumer as follows:

$$\frac{\mathbb{Z}_{4591}[x]}{\langle x^{64}+1\rangle} \overset{\eta_2}{\cong} \frac{\frac{\mathbb{Z}_{4591}[z]}{\langle z^8+1\rangle}[x]}{\langle x^8-z\rangle} \overset{\eta_3}{\hookrightarrow} \frac{\frac{\mathbb{Z}_{4591}[z]}{\langle z^8+1\rangle}[x]}{\langle x^{16}-1\rangle}.$$

Since z is a principal 16-th root of unity in $\mathbb{Z}_{4591}[z]/\langle z^8+1\rangle$, we can factor $x^{16}-1$ into $\prod_j (x-z^j)$ over $\mathbb{Z}_{4591}[z]/\langle z^8+1\rangle$. In summary, we are left with $\frac{1536\cdot 2\cdot 2}{8} = 768$ polynomial multiplications in $\mathbb{Z}_{4591}[z]/\langle z^8+1\rangle$. For multiplying polynomials in $\mathbb{Z}_{4591}[z]/\langle z^8+1\rangle$ for details.

7.5.2 A Specialized Approach with Truncated Rader, Good–Thomas, and Bruun FFTs

We briefly review the friendliness measures found by [HLY24, Hwa23]. The state-of-the-art AVX2 implementation [Hwa23] computed the products in $\mathbb{Z}_{4591}[x]/\langle \Phi_{17}(x^{96})\rangle$. [Hwa23] first applied truncated Rader's FFT to the isomorphism:

$$\frac{\mathbb{Z}_{4591}[x]}{\langle \Phi_{17}(x^{96})\rangle} \cong \prod_{i\neq 0} \frac{\mathbb{Z}_{4591}[x]}{\langle x^{96}-\omega_{17}^i\rangle}$$

and twisted each of $\mathbb{Z}_{4591}[x]/\langle x^{96}-\omega_{17}^i\rangle$ into $\mathbb{Z}_{4591}[x]/\langle x^{96}-1\rangle$. They then applied Good–Thomas FFT implementing the isomorphism:

$$\frac{\mathbb{Z}_{4591}[x]}{\langle x^{96}-1\rangle} \cong \prod_{j} \frac{\mathbb{Z}_{4591}[x]}{\left\langle x^{16}-\omega_6^j\right\rangle}$$

and twisted into $\mathbb{Z}_{4591}[x]/\langle x^{16} \pm 1\rangle$. Since each vector registers in AVX2 contains sixteen 16-bit values, all of the above are vectorization-friendly. The remaining problems are 48 polynomial multiplications in $\mathbb{Z}_{4591}[x]/\langle x^{16} - 1\rangle$ and 48 in $\mathbb{Z}_{4591}[x]/\langle x^{16} + 1\rangle$. Since 48 is a multiple of 16, we can interleave the polynomials with no leftovers. This implies permutation-friendliness. For multiplying polynomials in $\mathbb{Z}_{4591}[x]/\langle x^{16} \pm 1\rangle$, see [Hwa23] for details.

8 Overview of Advances

We give overviews of the advances of polynomial multiplications used in lattice-based cryptosystem implementations with emphases on modular arithmetic, algebraic techniques, and vectorization. Table 3 gives an overview of existing works for Dilithium, Kyber, and Saber, and Table 4 gives an overview of existing works for NTRU and NTRU Prime.

Table 3: Target architectures/extensions of existing works for Dilithium, Kyber, and Saber.

	Dilithium	Kyber	Saber
[BKS19]	-	Armv7E-M	-
[KRS19]	-	-	Armv7E-M
[ABD+20a]	AVX2	-	-
[ABD+20b]	-	AVX2	-
[DKRV20]	-	-	AVX2
[ABCG20]	-	- Armv7E-M	-
[MKV20]	-	-	Armv7E-M, AVX2
[IKPC20]	-	-	Armv7E-M
[CHK+21]	-	-	Armv7-M, AVX2
[GKS21]	Armv7-M	-	-
[SKS+21]	-	Armv8-A	-
[NG21]	-	Armv8-A	Armv8-A
[BHK+22b]	Armv8-A	Armv8-A	Armv8-A
[AHKS22]	Armv7-M	Armv7E-M	-
[HZZ+22]	-	Armv7E-M	-
[AMOT22]	-	-	RISC-V
[HKS23]	Armv7-M	-	-

8.1 Modular Arithmetic

We first give an overview of modular arithmetic. See Table 5 for a summary of existing works on 8-bit AVR, Armv7-M, Armv7E-M, Armv8.0-A, MVE, and AVX2.

8.1.1 Vector architecture implementations

[Sei18] was the first work proposing signed Montgomery multiplication. They applied the idea to the vectorized 16-bit NTT used in the Ring-LWE scheme NewHope. Their idea nicely captured the availability of 16-bit multiplication instructions in AVX2, and it was applied to Kyber [ABD+20b] and NTRU Prime [BBC+20]. The subtractive variant was also implemented by [SKS+21] in Armv8-A. The "unsigned Barrett multiplication" was implemented in [Sho].

Table 4: Target architectures/extensions of existing works for NTRU and NTRU Prime.

	NTRU	NTRU Prime
[KRS19]	Armv7E-M	-
[BBC+20]	AVX2	-
[CDH+20]	-	AVX2
[ACC+21]	-	Armv7-M
[CHK+21]	Armv7-M, AVX2	-
[NG21]	Armv8-A	-
[IKPC22]	Armv7E-M	-
[AHY22]	Armv7-M	Armv7-M
[BBCT22]	-	AVX2
[CCHY23]	Armv8-A	-
[Hwa23]	-	Armv8-A, AVX2
[HLY24]	-	Armv8-A

Table 5: Summary of existing works of modular multiplications relavent to our target architectures and extensions.

	Barrett	Montgomery	Plantard
[Sho]	✓	-	-
[Sei18]	AVX2	AVX2	-
[BKS19]	Armv7E-M	Armv7E-M	-
[ABCG20]	-	Armv7E-M	-
[ACC+21]	Armv7E-M	Armv7-M	-
[GKS21]	-	Armv7-M	-
[SKS+21]	Armv8.0-A	Armv8.0-A	-
[BHK+22b]	Armv8.0-A	Armv8.0-A	-
[AHKS22]	Armv7E-M	-	-
[BHK+22a]	MVE	-	-
[HZZ+22]	-	-	Armv7E-M
[AMOT22]	-	-	✓
[HKS23]	Armv7-M, 8-bit AVR	-	-

[BHK+22b] independently[14] found the signed Barrett multiplication, the correspondence between Montgomery and Barrett multiplication, and their variants and implemented them with Armv8-A. [BHK+22a] later demonstrated that if one increases the precision of the arithmetic, then we have the canonical representations of the products for some special moduli, and implemented the idea with M-profile vector extension (MVE).

8.1.2 Microcontroller Implementations

[BKS19] implemented Barrett reduction and the subtractive variant of Montgomery multiplication with the SIMD instruction `smul{b, t}{b, t}` in Armv7E-M. [ABCG20] switched to the additive variant of Montgomery multiplication and absorbed the addition by replacing a `smul{b, t}{b, t}` with `smla{b, t}{b, t}` [ABCG20, Algorithm 11]. The signed

[14] [BHK+22b] cited the eprint version of [SKS+21]. The subtractive variant of Montgomery multiplication was shown in the published version but not the eprint one. We are informing the authors of [BHK+22b] for this miscontribution.

Barrett reduction was later improved by [ACC+21] with instructions `smlaw{b, t}`[15], but it was not reported (we found this while carefully examining the assembly programs). In [ACC+21], they also proposed the uses of `s{mul, mla}l` in Armv7-M for 32-bit Montgomery multiplication and `smmulr` in Armv7E-M for 32-bit Barrett reduction. The 32-bit Montgomery multiplication was independently proposed by [GKS21].

The improvement of signed Barrett reduction with Armv7E-M was later reported in [AHKS22]. [HZZ+22] and [AMOT22] independently found the signed versions of Plantard multiplication. [HZZ+22] applied the idea to 16-bit modular arithmetic with Armv7E-M instructions `s{mul, mla}w{b, t}` while [AMOT22] applied the idea to 32-bit modular arithmetic using $64 = 64 \times 64$ arithmetic on K210 (64-bit) [AMOT22, Section V-B]. Shortly after, [HZZ+] applied signed Plantard arithmetic to Armv7-M with essentially the same idea from [AMOT22]. Recently, [HKS23] proposed the uses of Barrett multiplication when long/high multiplication instructions are slow, unusable, or unavailable and implemented the ideas with Armv7-M and 8-bit AVR.

8.2 Algebraic Techniques

In Dilithium and Kyber, most optimizations are about modular arithmetic, memory footprint, and instructions scheduling, so we exclude them unless specified otherwise in this section.

8.2.1 Vector Architecture Implementations

We first give an overview of AVX2-optimized implementations. For the big-by-small polynomial multiplication, [BBC+20] implemented 16-bit Good–Thomas FFT with permutations instantiated as logical operations for `ntrulpr761/sntrup761` and applied radix-2 FFT to the power-of-two dimension. [CHK+21] applied 16-bit size-256 negacyclic FFT to Saber and size-1024, size-1536, and size-1728 cyclic FFTs to NTRU. For the big-by-big polynomial multiplication, [MKV20, CDH+20] applied Toom–Cook and Karatsuba to NTRU and Saber. For NTRU Prime, [BBCT22] implemented truncated Schönhage's and Nussbaumer's FFTs (cf. Section 7.5.2), and [Hwa23] applied truncated Rader's, Good–Thomas, and Bruun's FFTs following [HLY24]'s Armv8-A work.

For the Armv8-A Neon implementations, [NG21] implemented 16-bit size-256 negacyclic FFT for Saber, and 3- and 4-way Toom–Cook for NTRU. Shortly after, [BHK+22b] demonstrated 32-bit negacyclic FFT is more performant for Saber [16]. [CCHY23] deployed 5-way Toom–Cook to NTRU and showed that Toeplitz transformation with Toom–Cook was more favorable due to the presence of vector-by-scalar multiplication instructions on Armv8-A, and [HLY24] applied Rader's, Good–Thomas, and Bruun's FFTs. Finally, [Hwa23] applied truncated Rader's FFT, Good–Thomas FFT, and Toeplitz matrix-vector products to small-dimensional cyclic/negacyclic convolutions.

8.2.2 Microcontroller Implementations

[KRS19] applied Toom–Cook and Karatsuba to NTRU and Saber. [MKV20] later cached the homomorphisms in the case of Saber and [IKPC20] applied the Toeplitz matrix-vector product to Saber with Toom–Cook and Karatsuba as the underlying monomorphisms. [ACC+21] proposed three implementations for NTRU Prime parameter sets `ntrulpr761/sntrup761`: (i) a Good–Thomas FFT computing the big-by-small polynomial multiplication with 32-bit arithmetic over $\mathbb{Z}$, (ii) an FFT using radix-2, radix-3, and radix-5 butterflies with 16-bit arithmetic over $\mathbb{Z}_{4591}$, and (iii) an FFT using

[15] `smlaw{b, t}` multiplies a 32-bit value by a certain half of a 32-bit value, accumulates the result to the accumulator, and returns the most significant 32-bit value.

[16] This doesn't say that we should do the same thing for AVX2-optimized implementation since there are no native 32-bit multiplication instructions in AVX2.

radix-3 and Rader's radix-17 butterflies with 16-bit arithmetic over $\mathbb{Z}_{4591}$. The big-by-small polynomial multiplication came from [BBC+20] and was later adapted to NTRU and Saber [CHK+21]. [IKPC22] extended [IKPC20]'s work to NTRU and [AHY22] improved [ACC+21, CHK+21]'s NTRU and NTRU Prime implementations by proposing vector-radix butterflies for speed [AHY22, Section 4.1] and vectorization-friendly Good–Thomas for code size [AHY22, Section 3.3].

9 Directions for Future Works

We point out several possible future works as follows.

Non-uniform property of localization in Toom–Cook. In Section 4.3, we explain that localization does not need to be uniform among subproblems and illustrate the idea with Toom–Cook. In practice, one usually applies Toom–Cook recursively. Since the required localization for subproblems is not uniform, applying more aggressive divide-and-conquer strategies for some subproblems is possible. We want to know the practical impact of this observation of Toom–Cook and its Toeplitz version for NTRU and Saber.

Schönhage and Nussbaumer for NTRU and Saber. In lattice-based cryptosystem implementations, Schönhage and Nussbaumer FFTs were only applied to NTRU Prime where the coefficient ring is $\mathbb{Z}_q$ for an odd q. We want to know the practical impact of Schönhage's FFT, Nussbaumer's FFT, and their Toeplitz versions for NTRU and Saber where q is a power of two.

Barrett multiplication for finite fields. The finite field versions of Montgomery multiplication [KA98] and Barrett reduction [Dhe03] were known in the literature. Appendix A extends the correspondence between Montgomery multiplication and Barrett multiplication [BHK+22b] to principal ideal domains. For a finite field $\mathbb{F}_p$, since $\mathbb{F}_p[x]$ is a principal ideal domain, the correspondence implies the finite field version of Barrett multiplication. The Barrett reduction found by [Dhe03] is then a special case in this regard. We want to know the practical impact of the deployment of Barrett multiplication for finite fields.

Toeplitz matrix-vector product for NTRU Prime. Section 5.6.2 explains that polynomial multiplication modulo $x^n - \alpha x - \beta$ can be turned into a Toeplitz matrix-vector product. Explore the Toeplitz approach for NTRU Prime.

A Modular Arithmetic for Principal Ideal Domains

Let R be a principal ideal domain, $e_0, e_1 \in R$ be elements with $\gcd(e_0, e_1) = 1$. We assume implicitly that $R/\langle e_0 \rangle$, $R/\langle e_1 \rangle \subset R$ by first fixing the representatives for each equivalence classes. We define Montgomery multiplication as:

$$\frac{ab + \left(ab \left(-e_0^{-1} \bmod \langle e_1 \rangle\right) \bmod \langle e_1 \rangle\right) e_0}{e_1} \equiv abe_1^{-1} \bmod \langle e_0 \rangle .$$

If b is known in prior, we compute the following instead:

$$\frac{a \left(be_1 \bmod \langle e_0 \rangle\right) + \left(a \left(be_1 \bmod \langle e_0 \rangle\right) \left(-e_0^{-1} \bmod \langle e_1 \rangle\right) \bmod \langle e_1 \rangle\right) e_0}{e_1} \equiv ab \bmod \langle e_0 \rangle .$$

We prove the equivalence as follows.

Proof. Let $\texttt{term} = ab + \left(ab\left(-e_0^{-1} \bmod \langle e_1\rangle\right) \bmod \langle e_1\rangle\right) e_0$ be an abbreviation. By definition, we have

$$\texttt{term} \bmod \langle e_1\rangle = \left(ab + \left(ab\left(-e_0^{-1} \bmod \langle e_1\rangle\right) \bmod \langle e_1\rangle\right) e_0\right) \bmod \langle e_1\rangle = 0.$$

Therefore, $\texttt{term}$ is a multiple of e_1 and $\frac{\texttt{term}}{e_1} \in R$. It remains to show that $\frac{\texttt{term}}{e_1} \equiv abe_1^{-1} \bmod \langle e_0\rangle$. This bolds down to the fact that $\texttt{term} \equiv ab \bmod \langle e_0\rangle$ and $e_0 \perp e_1$.

□

Suppose we are given an ideal $\langle e\rangle$ and a quotient ring $R/\langle e\rangle$ with a choice function for the representatives. For an $a \in R$, we define $\left[\!\left[\frac{a}{e}\right]\!\right]$ as the element in R satisfying:

$$e\left[\!\left[\frac{a}{e}\right]\!\right] = a - a \bmod \langle e\rangle.$$

We claim the following equation:

$$ab - \left[\!\left[\frac{a\left[\!\left[\frac{be_1}{e_0}\right]\!\right]}{e_1}\right]\!\right] e_0 = \frac{a\left(be_1 \bmod \langle e_0\rangle\right) + \left(a\left(be_1 \bmod \langle e_0\rangle\right)\left(-e_0^{-1}\right) \bmod \langle e_1\rangle\right) e_0}{e_1}.$$

Proof. We first find the following:

$$\left[\!\left[\frac{be_1}{e_0}\right]\!\right] \bmod \langle e_1\rangle = \left(be_1 \bmod \langle e_0\rangle\right)\left(-e_0^{-1}\right) \bmod \langle e_1\rangle$$

Then, we have:

$$\begin{aligned}
& ab - \left[\!\left[\frac{a\left[\!\left[\frac{be_1}{e_0}\right]\!\right]}{e_1}\right]\!\right] e_0 \\
= \; & \frac{abe_1 - a\left[\!\left[\frac{be_1}{e_0}\right]\!\right] e_0 + \left(a\left[\!\left[\frac{be_1}{e_0}\right]\!\right] \bmod \langle e_1\rangle\right) e_0}{e_1} \\
= \; & \frac{abe_1 - a\left[\!\left[\frac{be_1}{e_0}\right]\!\right] e_0 + \left(a\left(be_1 \bmod \langle e_0\rangle\right)\left(-e_0^{-1}\right) \bmod \langle e_1\rangle\right) e_0}{e_1} \\
= \; & \frac{abe_1 - a\left(be_1 - \left(be_1 \bmod \langle e_0\rangle\right)\right) + \left(a\left(be_1 \bmod \langle e_0\rangle\right)\left(-e_0^{-1}\right) \bmod \langle e_1\rangle\right) e_0}{e_1} \\
= \; & \frac{a\left(be_1 \bmod \langle e_0\rangle\right) + \left(a\left(be_1 \bmod \langle e_0\rangle\right)\left(-e_0^{-1}\right) \bmod \langle e_1\rangle\right) e_0}{e_1}.
\end{aligned}$$

□

[KAK96, KA98] demonstrated the benefit of unsigned Montgomery multiplication for multi-precision arithmetic. We leave the principal-ideal-domain view of $\left\langle e_0^k\right\rangle$ and $\langle e_1\rangle$ and its relation to Barrett multiplication as a future work.

B Comparing Radix-2 Schönhage and Nussbaumer

We compare the most commonly used radix-2 Schönhage and Nussbaumer.

Let's start with Nussbaumer. Suppose the original polynomial modulus is $x^{2^{2^k}} + 1$. We introduce $x^{2^{2^{k-1}}} \sim y$ and replace the relation with $x^{2^{2^{k-1}+1}} \sim 1$. This defines the FFT modulo $x^{2^{2^{k-1}+1}} - 1$ by regarding $R[y]\Big/\left\langle y^{2^{2^{k-1}}} + 1\right\rangle$ as the coefficient ring. Since the

remaining problem is again a power-of-two with exponent a power-of-two, we apply the same idea recursively down to x^2+1. This leads to the following problem sizes:

$$x^{2^{2^k}}+1, x^{2^{2^{k-1}}}+1, \ldots, x^2+1.$$

If $x^{2^{2^k+1}}+1$ is the polynomial moduls, Nussbaumer leads to the following problem sizes:

$$x^{2^{2^k+1}}+1, x^{2^{2^{k-1}+1}}+1, \ldots, x^8+1, x^4+1, x^2+1.$$

For recursive Schönhage, one can show that if the polynomial modulus is $x^{2^{2^k+1}}+1$, we have the same sequence of problem sizes as Nussbaumer. On the other hand, if $x^{2^{2^k}}+1$ is the polynomial modulus, we have to introduce $x^{2^{2^{k-1}}} \sim y$ and replace it with $x^{2^{2^{k-1}+1}}+1$, returning to the power-of-two case with exponent a power-of-two plus one. This leads to the following problem sizes:

$$x^{2^{2^k}}+1, x^{2^{2^{k-1}+1}}+1, \ldots, x^8+1, x^4+1, x^2+1.$$

This supports the statement "Nussbaumer's trick has the advantage of using slightly smaller ring extensions" in [Ber01, Section 9].

C A Formal Treatment of Localization

We refer to [Jac12b, Sections 7.2 and 7.3] for the localization of rings and modules. Let $\mathcal{A}$ be an R-algebra, $z \in \mathbb{Z}$, and $z^{-1}R$ be the localization of R at the multiplicative set $\mathcal{Z} = \{1, z, z^2, \ldots\}$. Naturally, we have $\mathcal{Z}^{-1}\mathcal{A}$ as a $\mathcal{Z}^{-1}R$-module. We turn it into a $\mathcal{Z}^{-1}R$-algebra by defining:

$$\forall z^{-k_0}\boldsymbol{a}, z^{-k_1}\boldsymbol{b} \in \mathcal{Z}^{-1}\mathcal{A}, z^{-k_0}\boldsymbol{a}z^{-k_1}\boldsymbol{b} \coloneqq z^{-k_0-k_1}\boldsymbol{a}\boldsymbol{b} \in \mathcal{Z}^{-1}\mathcal{A}.$$

Since rings and $\mathbb{Z}$-algebra are exactly the same thing, for a ring R, we can pick a $z \in \mathbb{Z}$ and define $\mathcal{Z}^{-1}R$ as a $\mathcal{Z}^{-1}\mathbb{Z}$-algebra (and hence a ring).

Let $\mathcal{A}, \mathcal{B}$ be R-algebras, and $\eta : \mathcal{A} \to \mathcal{B}$ be an algebra monomorphism. For an integer $z \in \mathbb{Z} - \{0\}$, suppose we find a map $\psi_z : \eta(\mathcal{A}) \to \mathcal{A}$ such that

$$\forall \boldsymbol{a} \in \mathcal{A}, (\psi_z \circ \eta)(\boldsymbol{a}) = z\boldsymbol{a}.$$

We define an algebra homomorphism $\xi : \mathcal{Z}^{-1}\eta(\mathcal{A}) \to \mathcal{Z}^{-1}\mathcal{A}$ as

$$\forall z^{-k}\eta(\boldsymbol{a}) \in \mathcal{Z}^{-1}\eta(\mathcal{A}), \xi\left(z^{-k}\eta(\boldsymbol{a})\right) \coloneqq z^{-1-k}\psi_z(\eta(\boldsymbol{a})).$$

If we restrict the image of ξ to $\eta(\mathcal{A})$, we find $\xi|_{\eta(\mathcal{A})} \coloneqq \left(\eta(\boldsymbol{a}) \mapsto z^{-1}\psi_z(\eta(\boldsymbol{a}))\right) = \eta^{-1}$. In summary, to invert η while given ψ_z with $z \in \mathbb{Z} - \{0\}$ non-invertible in $\mathcal{A}$, it suffices to define $\xi : \mathcal{Z}^{-1}\eta(\mathcal{A}) \to \mathcal{Z}^{-1}\mathcal{A}$ and apply $\xi|_{\eta(\mathcal{A})}$.

Notice that applying ξ assumes an already existing approach for multiplying z^{-1}. An alternative way is to find ψ_{z_0} and ψ_{z_1} with $z_0 \perp z_1$ and integers e_0, e_1 satisfying $e_0z_0 + e_1z_1 = 1$, and define η^{-1} as

$$\eta^{-1} \coloneqq e_0\psi_{z_0} + e_1\psi_{z_1}.$$

Since e_0 and e_1 are integers, η^{-1} can be implemented entirely with arithmetic in R. [CK91] used localization and Schönhage's [Sch77] radix-2 and radix-3 FFTs for multiplying polynomials over arbitrary unital (possibly-noncommutative) rings.

D Interprating Multiplications in $R[x]/\langle x^n - \alpha x - \beta\rangle$ as TMVPs

We outline [Yan23]'s ideas interprating multiplications in $R[x]/\langle x^n - \alpha x - \beta\rangle$ as Toeplitz matrix-vector products for generic β as follows. For reducing modulo $x^n - \alpha x$, if we additionally add the element $\sum_{j=0}^{n-2} \alpha b_{n-1-j} a_j$ to c_0, then the resulting computation is compatible with a Toeplitz matrix-vector product. This gives us the following transformation matrix mapping $\boldsymbol{a}$ to $\boldsymbol{ab} \in R[x]/\langle x^n - \alpha x - \beta\rangle$:

$$M_0 + M_1 + M_2' - \begin{pmatrix} \alpha b_{n-1} & \cdots & \alpha b_1 & 0 \\ 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \end{pmatrix}$$

where M_0 and M_1 are the same as previous paragraph and

$$M_2' = \mathbf{Toeplitz}_n(0, \ldots, 0, \alpha b_{n-1}, \ldots, \alpha b_1, 0).$$

Since $M_0 + M_1 + M_2'$ is the Toeplitz matrix

$$\mathbf{Toeplitz}_n(b_{n-1}, \ldots, b_1, b_0 + \alpha b_{n-1}, \beta b_{n-1} + \alpha b_{n-2}, \ldots, \beta b_2 + \alpha b_1, \beta b_1),$$

$\boldsymbol{ab} \in R[x]/\langle x^n - \alpha x - \beta\rangle$ can be written as a Toeplitz matrix-vector product with post-processing as follows:

$$\boldsymbol{ab} \bmod (x^n - \alpha x - \beta) = (M_0 + M_1 + M_2')\,\boldsymbol{a} - \begin{pmatrix} \alpha b_{n-1} & \cdots & \alpha b_1 & 0 \\ 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \end{pmatrix} \boldsymbol{a}.$$

E A Formal Treatment of Bilinear Systems

Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be modules over R. We call a map $\eta : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$ a bilinear map if

- $\forall \boldsymbol{a} \in \mathcal{A}, \eta(\boldsymbol{a}, -) : \mathcal{B} \to \mathcal{C}$ is a module homomorphism.
- $\forall \boldsymbol{b} \in \mathcal{B}, \eta(-, \boldsymbol{b}) : \mathcal{A} \to \mathcal{C}$ is a module homomorphism.

Suppose we have maps $\psi : \mathcal{A}^* \to \mathcal{A}', \iota : \mathcal{C}' \to \mathcal{C}^*$, and a bilinear map $\xi : \mathcal{C}' \times \mathcal{B}' \to \mathcal{A}'$ satisfying

$$\forall \boldsymbol{b} \in \mathcal{B}' \cap \mathcal{B}, \xi(-, \boldsymbol{b}) = \psi \circ \eta(-, \boldsymbol{b})^* \circ \iota.$$

If $\eta(-, \boldsymbol{b}) = f_{\boldsymbol{b}} \circ g_{\boldsymbol{b}}$ for some $f_{\boldsymbol{b}}$ and $g_{\boldsymbol{b}}$, we have the corresponding factorization for $\xi(-, \boldsymbol{b})$:

$$\forall \boldsymbol{b} \in \mathcal{B}' \cap \mathcal{B}, \xi(-, \boldsymbol{b}) = \psi \circ g_{\boldsymbol{b}}^* \circ f_{\boldsymbol{b}}^* \circ \iota.$$

In Section 5.6, we present the ideas with bilinear systems. We now rephrase the core idea as follows: Let's assume $\mathcal{A}' = \mathcal{A}, \mathcal{B}' = \mathcal{B}$, $\mathcal{C}' = \mathcal{C}$, $\psi = \boldsymbol{a}^* \mapsto \boldsymbol{a}$, and $\iota = \boldsymbol{c} \mapsto \boldsymbol{c}^*$. For finite index sets $\mathcal{I}, \mathcal{J}, \mathcal{K}$ and $(r_{(i,j,k)})_{(i,j,k)\in\mathcal{I}\times\mathcal{J}\times\mathcal{K}}$, define $\boldsymbol{a} = (a_i)_{i\in\mathcal{I}}, \boldsymbol{b} = (b_j)_{j\in\mathcal{J}}, \boldsymbol{c} = (b_k)_{k\in\mathcal{K}}$. Then, we write

$$\begin{cases} \left(\sum_{i\in\mathcal{I}} \sum_{j\in\mathcal{J}} r_{(i,j,k)} a_i b_j\right)_{k\in\mathcal{K}} & = \eta(-, \boldsymbol{b})(\boldsymbol{a}) \\ \left(\sum_{j\in\mathcal{J}} \sum_{k\in\mathcal{K}} r_{(i,j,k)} c_k b_j\right)_{i\in\mathcal{I}} & = \xi(-, \boldsymbol{b})(\boldsymbol{c}) \end{cases}$$

and find $\xi(-, \boldsymbol{b}) = (\psi \circ \eta(-, \boldsymbol{b})^* \circ \iota)$.

F Vector-Radix Transform

In Section 5.3, we know that one-dimensional size-n cyclic convolution can be turned into a multi-dimensional cyclic convolution of dimensionals based on a coprime factorization of n. If we apply isomorphism for each dimension and cache the results, then we save the cost of transformation significantly. This section explains how one can save more by directly optimizing a multi-dimensional transform $\otimes_j f_j$ with vector-radix transformation [HMCS77].

Frequently, f_j is a composition of one-dimensional isomorphisms shown in Section 3. Let's write $f_j = f_{j,0} \circ \cdots \circ f_{j,h-1}$. A crucial property while tensoring two compositions $f_{0,0} \circ f_{0,1}$ and $f_{1,0} \circ f_{1,1}$ is that $(f_{0,0} \circ f_{0,1}) \otimes (f_{1,0} \circ f_{1,1}) = (f_{0,0} \otimes f_{1,0}) \circ (f_{0,1} \otimes f_{1,1})$. Usually, f_j can be characterized as a composition of multiplicative steps and additive steps. During the multiplicative steps, we only multiply coefficients by some constants. For the additive steps, we perform additions and subtractions. One observation is that multiplicative steps are faster if we apply their composition directly. Suppose we have two multiplicative steps represented as matrix multiplications by $\begin{pmatrix} 1 & 0 \\ 0 & \zeta_0 \end{pmatrix} \otimes I_2$ and $I_2 \otimes \begin{pmatrix} 1 & 0 \\ 0 & \zeta_1 \end{pmatrix}$. Since $\left(\begin{pmatrix} 1 & 0 \\ 0 & \zeta_0 \end{pmatrix} \otimes I_2\right)\left(I_2 \otimes \begin{pmatrix} 1 & 0 \\ 0 & \zeta_1 \end{pmatrix}\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \zeta_1 & 0 & 0 \\ 0 & 0 & \zeta_0 & 0 \\ 0 & 0 & 0 & \zeta_0\zeta_1 \end{pmatrix}$, we only need three multiplications on the right-hand side. If we compute with the left-hand side, then we need four multiplications. The high-dimensional generalization and f_j's as series of compositions are obivous. See [AHY22] for applications.

References

[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. https://ieeexplore.ieee.org/abstract/document/1162555. 9

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8593. 29, 30, 33, 34

[ABD+20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/dilithium/. 1, 33

[ABD+20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/kyber/. 1, 33

[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. 11, 34, 35, 36

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292. 11, 12, 30

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Dann Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, pages 853–871, 2022. https://link.springer.com/chapter/10.1007/978-3-031-09234-3_42. 33, 34, 35

[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*, 2022(4):349–371, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9823. 16, 17, 19, 34, 36, 40

[AMOT22] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi. Efficient Word Size Modular Multiplication over Signed Integers. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 94–101. IEEE, 2022. https://ieeexplore.ieee.org/document/9974215. 7, 8, 28, 29, 33, 34, 35

[ARM10] ARM. *Cortex-M3 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0337/h. 27

[ARM21a] ARM. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*, 2021. https://developer.arm.com/documentation/ddi0487/gb/?lang=en. 25

[ARM21b] ARM. *Armv7-M Architecture Refernce Manual*, 2021. https://developer.arm.com/documentation/ddi0403/ed. 25

[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. https://link.springer.com/chapter/10.1007/3-540-47721-7_24. 2, 7

[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/. 1, 11, 16, 33, 34, 35, 36

[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. https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein. 12, 22, 32, 34, 35

[BC87] J. V. Brawley and L. Carlitz. Irreducibles and the composed product for polynomials over a finite field. *Discrete Mathematics*, 65(2):115–139, 1987. https://www.sciencedirect.com/science/article/pii/0012365X8790135X. 9, 14

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In *Cryptographic Hardware and Embedded Systems-CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings 15*, pages 250–272. Springer, 2013. https://link.springer.com/chapter/10.1007/978-3-642-40349-1_15#:~:text=Abstract,a%20single%20Ivy%20Bridge%20core. 20

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 2001. https://cr.yp.to/papers.html#m3. 4, 12, 19, 38

[Ber07] Daniel J. Bernstein. The tangent FFT. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 17th International Symposium, AAECC-17*, pages 291–300, 2007. https://link.springer.com/chapter/10.1007/978-3-540-77224-8_34. 9

[Ber08] Daniel J. Bernstein. Fast multiplication and its applications. *Algorithmic number theory*, 44:325–384, 2008. https://cr.yp.to/papers.html#multapps. 4, 11, 16

[Ber22] Daniel J. Bernstein. Fast norm computation in smooth-degree abelian number fields. 2022. https://eprint.iacr.org/2022/980. 18

[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. https://link.springer.com/article/10.1007/BF01386832. 9, 10

[BHK+22a] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient Multiplication of Somewhat Small Integers using Number–Theoretic Transforms. *Cryptology ePrint Archive*, 2022. https://eprint.iacr.org/2022/439. 7, 34

[BHK+22b] 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. 2, 5, 7, 26, 27, 30, 33, 34, 35, 36

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In *Progress in Cryptology - AFRICACRYPT 2019*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019. https://doi.org/10.1007/978-3-030-23696-0_11. 33, 34

[BMGVdO15] F.E. Brochero Martínez, C. R. Giraldo Vergaraand, and L. Batista de Oliveira. Explicit factorization of $x^n - 1 \in \mathbb{F}_q[x]$. *Designs, Codes and Cryptography*, 77:277–286, 2015. https://link.springer.com/article/10.1007/s10623-014-0005-y. 9

[BMK+22] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhedeg. Polynomial multiplication on embedded

vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9305. 19

[Bo-22] Bo-Yin Yang. 2022. Personal communication. 27

[Bou89] Nicolas Bourbaki. *Algebra I*. Springer, 1989. 4

[Bra84] Ronald N. Bracewell. The Fast Hartley Transform. *Proceedings of the IEEE*, 72(8):1010–1018, 1984. https://ieeexplore.ieee.org/document/1457236. 9

[Bru78] Georg Bruun. z-transform DFT Filters and FFT's. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):56–63, 1978. https://ieeexplore.ieee.org/document/1163036. 9

[CCHY23] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU. 2023. To appear at Indocrypt 2023, currently available at https://eprint.iacr.org/2023/1637. 3, 4, 13, 23, 24, 25, 31, 34, 35

[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/. 1, 34, 35

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994. https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/?active=current. 9, 16, 17

[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. 11, 30, 33, 34, 35, 36

[CK91] David G. Cantor and Erich Kaltofen. On Fast Multiplication of Polynomials over Arbitrary Algebras. *Acta Informatica*, 28(7):693–701, 1991. https://link.springer.com/article/10.1007/BF01178683. 38

[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. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/. 9, 16

[DH84] Pierre Duhamel and Henk Hollmann. 'Split Radix' FFT Algorithm. *Electronics letters*, 20(1):14–16, 1984. https://digital-library.theiet.org/content/journals/10.1049/el_19840012. 9, 14

[Dhe03] Jean-François Dhem. Efficient Modular Reduction Algorithm in $\mathbb{F}_q[x]$ and Its Application to "Left to Right" Modular Multiplication in $\mathbb{F}_2[x]$. pages 203–213, 2003. https://link.springer.com/chapter/10.1007/978-3-540-45238-6_17. 7, 36

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/. 1, 33

[DV78a] Eric Dubois and Anastasios N. Venetsanopoulos. A New Algorithm for the Radix-3 FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(3):222–225, 1978. https://ieeexplore.ieee.org/document/1163084. 17

[DV78b] Eric Dubois and Anastasios N. Venetsanopoulos. The discrete Fourier transform over finite rings with application to fast convolution. *IEEE Computer Architecture Letters*, 27(07):586–593, 1978. https://ieeexplore.ieee.org/document/1675158. 9

[DV90] Pierre Duhamel and Martin Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal processing*, 19(4):259–299, 1990. https://www.sciencedirect.com/science/article/pii/016516849090158U. 4

[FD05] Haining Fan and Yiqi Dai. Fast Bit-Parallel GF(2/sup n/) Multiplier for All Trinomials. *IEEE Transactions on Computers*, 54(4):485–490, 2005. https://ieeexplore.ieee.org/document/1401867. 21

[FH07] Haining Fan and M. Anwar Hasan. A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields. *IEEE Transactions on Computers*, 56(2):224–233, 2007. https://ieeexplore.ieee.org/document/4042682. 4, 21

[Fid73] Charles M. Fiduccia. On the Algebraic Complexity of Matrix Multiplication. 1973. https://cr.yp.to/bib/entries.html#1973/fiduccia-matrix. 20

[Für09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. https://doi.org/10.1137/070711761. 9, 14

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8725. 27, 28, 29, 33, 34, 35

[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. https://www.jstor.org/stable/2983896. 15, 16

[Goo71] I. J. Good. The relationship between two fast Fourier transforms. *IEEE Transactions on Computers*, 100(3):310–317, 1971. https://ieeexplore.ieee.org/document/1671829. 11, 15, 16

[GS66] W. M. Gentleman and G. Sande. Fast Fourier Transforms: For Fun and Profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578. Association for Computing Machinery, 1966. https://doi.org/10.1145/1464291.1464352. 14, 16

[Har42] Ralph VL Hartley. A More Symmetrical Fourier Analysis Applied to Transmission Problems. *Proceedings of the IRE*, 30(3):144–150, 1942. https://ieeexplore.ieee.org/document/1694454. 9

[HB95] M.A. Hasan and V.K. Bhargava. Architecture for a low complexity rate-adaptive Reed-Solomon encoder. *IEEE Transactions on Computers*, 44(7):938–942, 1995. https://ieeexplore.ieee.org/document/392853. 21

[HKS23] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Barrett Multiplication for Dilithium on Embedded Devices. 2023. https://eprint.iacr.org/2023/1955. 2, 5, 7, 8, 27, 28, 33, 34, 35

[HLY24] Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime. 2024. To appear at ACNS 2024, currently available at https://eprint.iacr.org/2023/1580. 10, 11, 15, 16, 22, 27, 31, 32, 34, 35

[HMCS77] David B. Harris, James H. McClellan, David S. K. Chan, and Hans W. Schuessler. Vector Radix Fast Fourier Transform. In *ICASSP'77. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 548–551, 1977. https://ieeexplore.ieee.org/document/1170349. 16, 40

[HQZ04] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. The Middle Product Algorithm I. *Applicable Algebra in Engineering, Communication and Computing*, 14(6):415–438, 2004. https://link.springer.com/article/10.1007/s00200-003-0144-2. 20

[HVDH22] David Harvey and Joris Van Der Hoeven. Polynomial Multiplication over Finite Fields in time O (n log n). *Journal of the ACM*, 69(2):1–40, 2022. https://dl.acm.org/doi/full/10.1145/3505584. 17

[Hwa23] Vincent Hwang. Pushing the Limit of Vectorized Polynomial Multiplication for NTRU Prime. 2023. https://eprint.iacr.org/2023/604. 11, 22, 24, 31, 32, 33, 34, 35

[HY22] Chenar Abdulla Hassan and Oğuz Yayla. Radix-3 NTT-Based Polynomial Multiplication for Lattice Based Cryptography. *Cryptology ePrint Archive*, 2022. https://eprint.iacr.org/2022/726. 17

[HZZ+] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Cetin Kaya Koc, and Donglong Chen. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. https://github.com/UIC-ESLAS/Kyber_RV_M3. 29, 35

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9833. 7, 8, 28, 29, 30, 33, 34, 35

[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. 3, 33, 35, 36

[IKPC22] Írem Keskinkurt Paksoy and Murat Cenk. Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication. 2022. https://ia.cr/2022/300. 3, 21, 31, 34, 36

[Jac12a] Nathan Jacobson. *Basic Algebra I.* Courier Corporation, 2012. 4

[Jac12b] Nathan Jacobson. *Basic Algebra II.* Courier Corporation, 2012. 4, 12, 38

[JF07] Steven G. Johnson and Matteo Frigo. A Modified Split-Radix FFT With Fewer Arithmetic Operations. *IEEE Transactions on Signal Processing*, 55(1):111–119, 2007. https://ieeexplore.ieee.org/document/4034175. 9

[KA98] Cetin K Koc and Tolga Acar. Montgomery Multiplication in GF (2k). *Designs, Codes and Cryptography*, 14:57–69, 1998. https://link.springer.com/article/10.1023/A:1008208521515. 6, 36, 37

[KAK96] C. Kaya Koc, Tolga Acar, and Burton S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996. https://ieeexplore.ieee.org/document/502403. 6, 37

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145(2), pages 293–294, 1962. http://cr.yp.to/bib/1963/karatsuba.html. 11, 15

[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. https://link.springer.com/chapter/10.1007/978-3-030-21568-2_14. 30, 33, 34, 35

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly Fast NTRU Using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8293. 17

[LVB07] T. Lundy and James Van Buskirk. A new matrix approach to real FFTs and convolutions of length 2^k. *Computing*, 80:23–45, 2007. https://link.springer.com/article/10.1007/s00607-007-0222-6. 9

[LZ22] Zhichuang Liang and Yunlei Zhao. Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey. *arXiv preprint arXiv:2211.13546*, 2022. https://arxiv.org/abs/2211.13546. 4

[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. https://www.sciencedirect.com/science/article/pii/S107157979690026X. 9

[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. 30, 33, 35

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/?active=current. 2, 6

[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. https://ieeexplore.ieee.org/document/543667. 9

[MV83a] Jean-Bernard Martens and Marc C. Vanwormhoudt. Convolution Using a Conjugate Symmetry Property for Number Theoretic Transforms Over Rings of Regular Integers. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(5):1121–1125, 1983. https://ieeexplore.ieee.org/document/1164198. 12

[MV83b] Jean-Bernard Martens and Marc C. Vanwormhoudt. Convolutions of Long Integer Sequences by Means of Number Theoretic Transforms Over Residue Class Polynomial Rings. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(5):1125–1134, 1983. https://ieeexplore.ieee.org/abstract/document/1164201. 12

[NG21] Duc Tri Nguyen and Kris Gaj. Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists. In *Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings*, pages 234–254, 2021. https://link.springer.com/chapter/10.1007/978-3-030-81293-5_13. 33, 34, 35

[Nic71] Peter J. Nicholson. Algebraic Theory of Finite Fourier Transforms. *Journal of Computer and System Sciences*, 5(5):524–547, 1971. https://www.sciencedirect.com/science/article/pii/S0022000071800144. 11

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography. 40, 41, 43, 44

[Nus80] Henri Nussbaumer. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980. https://ieeexplore.ieee.org/document/1163372. 12

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer Berlin, Heidelberg, 2nd edition, 1982. https://doi.org/10.1007/978-3-642-81897-4. 4

[Ora14] Oracle. *x86 Assembly Language Reference Manual*, 2014. https://docs.oracle.com/cd/E36784_01/html/E36859/enmzx.html#scrolltoc. 25

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, 2021. https://ieeexplore.ieee.org/document/9416314. 2, 7, 8

[Pol71] John M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of computation*, 25(114):365–374, 1971. https://www.ams.org/journals/mcom/1971-25-114/S0025-5718-1971-0301966-0/?active=current. 9, 11

[Rad68] Charles M. Rader. Discrete Fourier Transforms When the Number of Data Samples Is Prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968. https://ieeexplore.ieee.org/document/1448407. 10

[Sch77] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977. https://link.springer.com/article/10.1007/bf00289470. 12, 38

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. 2018. https://eprint.iacr.org/2018/039. 6, 33, 34

[Sho] Victor Shoup. NTL: a Library for Doing Number Theory. http://www.shoup.net/ntl/. 7, 33, 34

[Sho99] Victor Shoup. Efficient Computation of Minimal Polynomials in Algebraic Extensions of Finite Fields. In *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, pages 53–58, 1999. https://dl.acm.org/doi/10.1145/309831.309859. 20

[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. In *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II*, pages 424–440. Springer, 2021. https://link.springer.com/chapter/10.1007/978-3-030-90022-9_23. 33, 34

[Sto66] Thomas G. Stockham, Jr. High-Speed Convolution and Correlation. In *Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 229–233, 1966. https://dl.acm.org/doi/10.1145/1464182.1464209. 16

[Tho63] Llewellyn Hilleth Thomas. Using a computer to solve problems in physics. *Applications of digital computers*, pages 44–45, 1963. 16

[Too63] Andrei L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. http://toomandre.com/my-articles/engmat/MULT-E.PDF. 11, 15

[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. https://link.springer.com/article/10.1007/s10623-012-9647-9. 9

[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. https://dl.acm.org/doi/10.1145/1005285.1005327. 16

[Win78] Shmuel Winograd. On Computing the Discrete Fourier Transform. *Mathematics of computation*, 32(141):175–199, 1978. https://www.ams.org/journals/mcom/1978-32-141/S0025-5718-1978-0468306-4/?active=current. 10

[Win80] Shmuel Winograd. *Arithmetic Complexity of Computations*, volume 33. Society for Industrial and Applied Mathematics, 1980. https://epubs.siam.org/doi/10.1137/1.9781611970364. 4, 11, 15, 19

[WP06] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. 2006. https://eprint.iacr.org/2006/224. 31

[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. https://www.worldscientific.com/doi/10.1142/S1793830921500701. 9

[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. https://www.sciencedirect.com/science/article/pii/S1071579718300996. 9

[Yan23] Bo-Yin Yang, 2023. Personal communication. 4, 39