Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime

Vincent Hwang[1,3], Chi-Ting Liu[2], and Bo-Yin Yang[3]

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

Abstract. In this paper, we explore the cost of vectorization for multiplying polynomials with coefficients in $\mathbb{Z}_q$ for an odd prime q, as exemplified by NTRU Prime, a postquantum cryptosystem that found early adoption due to its inclusion in OpenSSH.

If there is a large power of two dividing $q-1$, we can apply radix-2 Cooley–Tukey fast Fourier transforms to multiply polynomials in $\mathbb{Z}_q[x]$. The radix-2 nature admits efficient vectorization. Conversely, if 2 is the only power of two dividing $q-1$, we can apply Schönhage's and Nussbaumer's FFTs to craft radix-2 roots of unity, but these double the number of coefficients.

We show how to avoid the doubling while maintaining the vectorization friendliness with Good–Thomas, Rader's, and Bruun's FFTs. In particular, in `sntrup761`, the most common instance of NTRU Prime we have $q = 4591$, and we exploit the existing Fermat-prime factor of $q-1$ for Rader's FFT and power-of-two factor of $q+1$ for Bruun's FFT.

Polynomial multiplications in $\mathbb{Z}_{4591}[x]/\left\langle x^{761}-x-1\right\rangle$ is still a worthwhile target because while out of the NIST PQC competition, `sntrup761` is still going to be used with OpenSSH by default in the near future.

Our polynomial multiplication outperforms the state-of-the-art vector-optimized implementation by 6.1×. For `ntrulpr761`, our keygen, encap, and decap are 2.98×, 2.79×, and 3.07× faster than the state-of-the-art vector-optimized implementation. For `sntrup761`, we outperform the reference implementation significantly.

Keywords: Good–Thomas FFT · Rader's FFT · Bruun's FFT · NTRU Prime · Vectorization

1 Introduction

At PQCrypto 2016, the National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization Process for replacing existing standards for public-key cryptography with quantum-resistant cryptosystems. For lattice-based cryptosystems, polynomial multiplications have been the most time-consuming operations. Recently standardized [AAC$^+$22] Dilithium, Kyber, and Falcon wrote number–theoretic transforms (NTTs) into their specifications in response.

OpenSSH 9.0 defaults to NTRU Prime[4]. However, in NTRU Prime the polynomial ring doesn't allow NTT-based multiplications naturally. State-of-the-art vectorized implementations introduced various techniques extending coefficient rings, or computed the results over $\mathbb{Z}$. In each of these approaches, empirically small-degree polynomial multiplications is always an important bottleneck. We study the compatibility of vectorization and various algorithmic techniques in the literature and choose the ARM Cortex-A72 implementing the Armv8-A architecture [5] for this work. We are interested in vectorized polynomial multiplications for NTRU Prime. [BBCT22] showed that a vectorized generic polynomial multiplication takes $\sim 1.5\times$ time of a "generic by small (ternary coefficients)" one with AVX2. [BBCT22] applied Schönhage and Nussbaumer to ease vectorization. Schönhage and Nussbaumer double the sizes of the coefficient rings and lead to a larger number of small-degree polynomial multiplications. We explain how to avoid the doubling with Good–Thomas, Rader's, and Bruun's FFTs.

We implement our ideas on Cortex-A72 implementing Armv8.0-A with the vector instruction set Neon. However, we emphasize that our approaches are built around the notion of vectorization and not a specific architecture.

1.1 Contributions

We summarize our contributions as follows.

- We formalize the needs of vectorization commonly involved in vectorized implementations.
- We propose vectorized polynomial multipliers essentially quartering and halving the number of small-dimensional polynomial multiplications after FFTs.
- We propose novel accumulative (subtractive) variants of Barrett multiplication absorbing the follow up addition (subtraction).
- We implement the ideas with the SIMD technology Neon in Armv8.0-A on a Cortex-A72. Our fastest polynomial multiplier outperforms the state-of-the-art optimized implementation by a factor of 6.1×.
- In addition to the polynomial multiplication, we vectorize the sorting network, polynomial inversions, encoding, and decoding subroutines used in `ntrulpr761` and `sntrup761`. For `ntrulpr761`, our key generation, encapsulation, and decapsulation are 2.98×, 2.79×, and 3.07× faster than the state-of-the-art optimized implementation. For `sntrup761`, we outperform the reference implementation significantly.

---

[4] `https://marc.info/?l=openssh-unix-dev&m=164939371201404&w=2`.

[5] ARMv8-A, which naturally comes with the SIMD technology Neon, is currently the most prevalent architecture for mobile devices and is used for all Apple hardware.

1.2 Code

Our source code can be found at `https://github.com/vector-polymul-ntru-ntrup/NTRU_Prime` under the CC0 license.

1.3 Structure of this Paper

Section 2 goes through the preliminaries. Section 3 surveys FFTs. Section 4 describes our implementations. We show the performance numbers in Section 5.

2 Preliminaries

Section 2.1 describes the polynomials rings in NTRU Prime, Section 2.2 describes our target platform Cortex-A72, and Section 2.3 describes the modular arithmetic.

2.1 Polynomials in NTRU Prime

The NTRU Prime submission comprises two families: Streamlined NTRU Prime and NTRU LPRime. Both operate on the polynomial ring $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ where q and p are primes such that the ring is a finite field. We target the polynomial multiplications for parameter sets `sntrup761` and `ntrulpr761` where $q = 4591$ and $p = 761$. One should note that `sntrup761`, which is used by OpenSSH, uses a (Quotient) NTRU structure, and requires inversions in $\mathbb{Z}_3[x]/\left\langle x^{761} - x - 1\right\rangle$ and $\mathbb{Z}_{4591}[x]/\left\langle x^{761} - x - 1\right\rangle$. We refer the readers to the specification [BBC$^+$20] for more details. With no other assumptions on the inputs, we call a polynomial multiplication "big by big". If one of the inputs is guaranteed to be ternary, we call it "big by small". We optimize both although the former is required only if we apply the fast constant-time GCD [BY19] to the inversions in the key generation of `sntrup761`. The fast constant-time GCD is left as a future work.

2.2 Cortex-A72

Our target platform is the ARM Cortex-A72, implementing the 64-bit Armv8.0-A instruction set architecture. It is a superscalar Central Processing Unit (CPU) with an in-order frontend and an out-of-order backend. Instructions are first decoded into μops in the frontend and dispatched to the backend, which contains these eight pipelines: `L` for loads, `S` for stores, `B` for branches, `I0/I1` for integer instructions, `M` for multi-cycle integer instructions, and `F0/F1` for Single-Instruction-Multiple-Data (SIMD) instructions. The frontend can only dispatch at most three μops per cycle. Furthermore, in a single cycle, the frontend dispatches at most one μop using `B`, at most two μops using `I0/I1`, at most two μops using `M`, at most one μop using `F0`, at most one μop using `F1`, and at most two μops using `L/S` [ARM15, Section 4.1].

We mainly focus on the pipelines `F0`, `F1`, `L`, and `S` for performance. `F0/F1` are both capable of various additions, subtractions, permutations, comparisons, minimums/maximums, and table lookups [6]. However, multiplications can only be dispatched to `F0`, and shifts to `F1`. The most heavily-loaded pipeline is clearly the critical path. If there are more multiplications than shifts, we much prefer instructions that can use either pipeline to go to `F1` since the time spent in `F0` will dominate our runtime. Conversely, with more shifts than multiplications, we want to dispatch most non-shifts to `F0`. In practice, we interleave instructions dispatched to the pipeline with the most workload with other pipelines (or even `L/S`) — and pray. Our experiment shows that this approach generally works well. In the case of `chacha20` implementing `randombytes` for benchmarking [BHK$^+$22], we even consider a compiler-aided mixing of `I0/I1`, `F0/F1`, *and* `L/S`[7]. The idea also proved valuable for `Keccak` on some other Cortex-A cores [BK22, Table 1].

SIMD registers. The 64-bit Armv8-A has 32 architectural 128-bit SIMD registers with each viewable as packed 8-, 16-, 32-, or 64-bit elements ([ARM21, Fig. A1-1]), denoted by suffixes `.16B` `.8H`, `.4S`, and `.2D` on the register name, respectively.

Armv8-A vector instructions.

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

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

Additions/subtractions. For basic arithmetic, the usual `add/sub` adds/subtracts the corresponding elements. Long variants `s{add,sub}l{,2}` add or subtract the

---

[6] There are some exceptions, including `addv`, `smaxv`, `sadalp`. We are not using them in this paper and refer to [ARM15] for more details.

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

corresponding elements from the lower or upper 64-bit halves and signed-extend into double-width results[8].

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

2.3 Modular Arithmetic

Algorithm 1 Barrett reduction.

This is [BHK+22, Algorithm 11].
Input: $\mathtt{a} = a$.
Output: $\mathtt{a} = a - \left\lfloor \frac{a \left\lfloor \frac{2^e \mathtt{R}}{q} \right\rceil}{2^e \mathtt{R}} \right\rceil q \equiv a \bmod {}^{\pm}q$.

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

Algorithm 2 Barrett multiplication.

This is [BHK+22, Algorithm 10].
Input: $\mathtt{a} = a$.
Output: $\mathtt{a} = ab - \left\lfloor \frac{a \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q \equiv ab \bmod {}^{\pm}q$.

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

Let q be an odd modulus, and $\mathtt{R}$ be the size of the arithmetic. We describe the modular reductions and multiplications for computing in $\mathbb{Z}_q$. Barrett reduction [Bar86] reduces a value a by approximating $a \bmod {}^{\pm}q$ with $a - \left\lfloor \frac{a \cdot \left\lfloor \frac{2^e \mathtt{R}}{q} \right\rceil}{2^e \mathtt{R}} \right\rceil$ (cf. Algorithm 1). For multiplying an unknown a with a fixed value b, we compute $ab - \left\lfloor \frac{a \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil_2}{\mathtt{R}} \right\rceil q \equiv ab \bmod {}^{\pm}q$ (Barrett multiplication [BHK+22]) where $\lfloor \rceil_2$ is the function mapping a real number r to $2 \left\lfloor \frac{r}{2} \right\rceil$ (cf. Algorithm 2). We give novel multiply-add/sub variants of Barrett multiplication in Algorithms 3–4. Algorithm 3 (resp. 4) computes a representation of $a + bc$ (resp. $a - bc$) by merging a `mul` with an `add` (resp. a `sub`) into an `mla` (resp. `mls`), saving 1 instruction.

---

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

Algorithm 3 `Barrett_mla`.

```
Input: a = $a$.
Output: a = $a + bc - \left\lfloor \frac{b \left\lfloor \frac{cR}{q} \right\rceil_2}{R} \right\rceil q$.
sqrdmulh t, $b$, $\frac{\left\lfloor \frac{cR}{q} \right\rceil_2}{2}$
mla      a, $b$, $c$
mls      a, t, $q$
```

Algorithm 4 `Barrett_mls`.

```
Input: a = $a$.
Output: a = $a - bc + \left\lfloor \frac{b \left\lfloor \frac{cR}{q} \right\rceil_2}{R} \right\rceil q$.
sqrdmulh t, $b$, $\frac{\left\lfloor \frac{cR}{q} \right\rceil_2}{2}$
mls      a, $b$, $c$
mla      a, t, $q$
```

3 Fast Fourier Transforms

We go through the mathematics behind various fast Fourier transforms (FFTs) and emphasize their defining conditions. This section is structured as follows. Section 3.1 reviews the Chinese remainder theorem for polynomial rings and discrete Fourier transform (DFT). We then survey various FFTs, including Cooley–Tukey in Section 3.2, Bruun and its finite field counterpart in Section 3.3, Good–Thomas in Section 3.4, Rader in Section 3.5, and Schönhage and Nussbaumer in Section 3.6. We use number–theoretic transform (NTT) as a synonym of FFT.

3.1 The Chinese Remainder Theorem (CRT) for Polynomial Rings

Let $n = \prod_l n_l$, and $\boldsymbol{g}_{i_0,\ldots,i_{h-1}} \in R[x]$ be coprime polynomials for all indices $(i_l)_{l=0\cdots h-1}$ where $0 \le i_l < n_l$. The CRT gives us a chain of isomorphisms

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

Multiplying in $\prod_{i_0,\ldots,i_{h-1}} R[x] \Big/ \left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle$ is cheap if the polynomial modulus is small. If the isomorphism chain is also cheap, we improve the polynomial multiplications in $R[x] \Big/ \left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle$. For small n_l's, it is usually cheap to decompose a polynomial ring into a product of n_l polynomial rings.

Transformations will be described with the words "radix", "split", and "layer". We demonstrated below for $h = 2$. Suppose we have isomorphisms

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

where $i_0 \in \{0, \ldots, n_0 - 1\}$ and $i_1 \in \{0, \ldots, n_1 - 1\}$. We call η_0 a *radix-n_0 split* and an implementation of η_0 a radix-n_0 computation, and similarly for η_1. Usually, we implement several isomorphisms together to minimize memory operations. The resulting computation is called a *multi-layer* computation. Suppose we implement η_0 and η_1 with a single pair of loads and stores, and η_0 and η_1 both rely on X, a shape of computations, then the resulting multi-layer computation is called a *2-layer* X. If additionally $n_0 = n_1$, the computation is a *2-layer radix-*n_0 X, and similarly for more layers.

3.2 Cooley–Tukey FFT

In a Cooley–Tukey FFT [CT65], we have $\zeta \in R$, $\omega_n \in R$ a principal nth root of unity, n coprime to $\mathrm{char}(R)$, and $\boldsymbol{g}_{i_0,\dots,i_{h-1}} = x - \zeta\omega_n^{\sum_l i_l \prod_{j<l} n_j} \in R[x]$. Since $\prod_{i_0,\dots,i_{h-1}} \boldsymbol{g}_{i_0,\dots,i_{h-1}} = x^n - \zeta^n$, the efficiency of multiplying polynomials in $R[x]/\langle x^n - \zeta^n\rangle$ boils down to the efficiency of the isomorphisms indexed by i_l's. Furthermore, it is a *cyclic* NTT if $\zeta^n = 1$.

3.3 Bruun-Like FFTs

[Bru78] first introduced the idea of factoring into trinomials $\boldsymbol{g}_{i_0,\dots,i_{h-1}}$ when n is a power of two — to reduce the number of multiplications in R while operating over $\mathbb{C}$. [Mur96] generalized this to arbitrary even n. For our implementations, we need the results on factoring $x^{2^k} + 1 \in \mathbb{F}_q[x]$ when $q \equiv 3 \pmod 4$ [BGM93] and composed multiplications of polynomials in $\mathbb{F}_q[x]$ [BC87]. Factoring $x^n - 1$ over $\mathbb{F}_q$ is actively researched [BGM93,Mey96,TW13,MVdO14,WYF18,WY21].

Review: the original Bruun's FFT ($\boldsymbol{R} = \mathbb{C}$). We choose $\boldsymbol{g}_{i_0,\dots,i_{h-1}} = x^2 - \left(\zeta\omega_n^{\sum_l i_l \prod_{j<l} n_j} + \zeta^{-1}\omega_n^{-\sum_l i_l \prod_{j<l} n_j}\right) x + 1$ so $x^{2n} - (\zeta^n + \zeta^{-n})\, x^n + 1 = \prod_{i_0,\dots,i_{h-1}} \boldsymbol{g}_{i_0,\dots,i_{h-1}}$. This provides us an alternative factorization for $x^{4n} - 1 = (x^{2n} - 1)(x^{2n} + 1)$ by choosing $\zeta^n = \omega_4$. For a complex number with norm 1, since the sum of its inverse and itself is real, we only need arithmetic in $\mathbb{R}$ to reach $\prod_{i_0,\dots,i_{h-1}} \mathbb{C}[x] \Big/ \Big\langle \boldsymbol{g}_{i_0,\dots,i_{h-1}(x)} \Big\rangle$.

$\boldsymbol{R} = \mathbb{F}_{\boldsymbol{q}}$ where $\boldsymbol{q} \equiv 3 \pmod 4$. We need Theorem 1 for our implementations.

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

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

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

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

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

Proof. First observe $x^{2^k r} - 1 = (x^r - 1) \odot \left(x^{2^k} - 1 \right)$[9]. By Lemma 1, this equals $\prod_{i_0} \left((x - \omega_r^{i_0}) \odot \left(x^{2^k} - 1 \right) \right) = \prod_{i_0, i_1} \left((x - \omega_r^{i_0}) \odot \boldsymbol{f}_{i_1} \right)$. According to Lemma 2, $(x - \omega_r^{i_0}) \odot \left(x^{2^k} - 1 \right) = x^{2^k} - \omega_r^{2^k i_0}$ and $(x - \omega_r^{i_0}) \odot \boldsymbol{f}_{i_1} = \omega_r^{i_0 \deg(\boldsymbol{f}_{i_1})} \boldsymbol{f}_{i_1}(\omega_r^{-i_0} x)$ as desired.

In summary, by Lemma 3 we have the following isomorphisms:

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

Radix-2 Bruun's butterflies and inverses. Define $\mathbf{Bruun}_{\alpha,\beta}$ as follows:

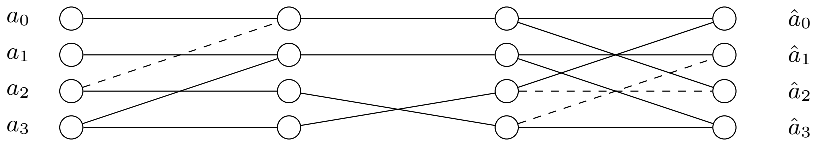
$$\mathbf{Bruun}_{\alpha,\beta} : \begin{cases} \frac{R[x]}{\langle x^4 + (2\beta - \alpha^2)x^2 + \beta^2 \rangle} & \to \frac{R[x]}{\langle x^2 + \alpha x + \beta \rangle} \times \frac{R[x]}{\langle x^2 - \alpha x + \beta \rangle} \\ a_0 + a_1 x + a_2 x^2 + a_3 x^3 & \mapsto ((\hat{a}_0 + \hat{a}_1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

where

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

We compute $(a_0 - \beta a_2, a_1 + (\alpha^2 - \beta) a_3, \alpha a_2, \alpha\beta a_3)$, swap the last two values implicitly, and do an addition-subtraction (cf. Figure 1). Notice that we can use `Barrett_mla` and `Barrett_mls` whenever a product is followed by only one accumulation $(a_1 + \left(\alpha^2 - \beta\right) a_3)$ or subtraction $(a_0 - \beta a_2)$.

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



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

---

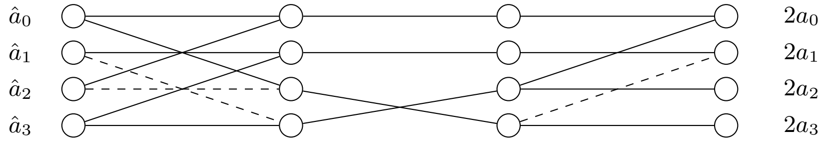
[9] $\forall$ coprime q_0, q_1, $\left\{ \omega_{q_0}^{i_0} \omega_{q_1}^{i_1} | 0 \le i_0 < q_0, 0 \le i_1 < q_1 \right\} = \left\{ \omega_{q_0 q_1}^{i} | 0 \le i < q_0 q_1 \right\}$ in the splitting field of $x^{q_0 q_1} - 1$.

correspondingly defines the inverse, where

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

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

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



We will use three special cases of Bruun's butterflies.

$\mathbf{Bruun}_{\sqrt{2},1}$: The initial split of $x^{2^k} + 1$ is $\mathbf{Bruun}_{\sqrt{2},1}$. Since $\beta = \alpha^2 - \beta = 1$, we only need two multiplications by $\times\sqrt{2}$.

$\mathbf{Bruun}_{\alpha,\pm 1}$: We avoid multiplying with $\beta = \pm 1$ in $\mathbf{Bruun}_{\alpha,\pm 1}$ and $2\mathbf{Bruun}^{-1}_{\alpha,\pm 1}$.

$\mathbf{Bruun}_{\alpha,\frac{\alpha^2}{2}}$: We save no multiplications, but only use 2 constants α and $\frac{\alpha^2}{2}$ instead of 4. It is used in the split of $x^{2^k} + \omega_r^{2^k i}$ for an odd r.

3.4 Good–Thomas FFTs

A Good–Thomas FFT [Goo58] converts cyclic FFTs and convolutions into multi-dimensional ones for coprime n_l's. For the polynomial ring $R[x]/\langle x^n - 1\rangle$, we implement $R[x]/\langle x^n - 1\rangle \cong \prod_{i_0,\dots,i_{h-1}} R[x]/\left\langle x - \prod_l \omega_{n_l}^{i_l}\right\rangle$ with a multi-dimensional FFT induced by the equivalences $x \sim \prod_l u_l$ and $\forall l, u_l^{n_l} \sim 1$. Formally, we have

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

We illustrate the idea for $h = 2, n_0 = 2$, and $n_1 = 3$. Let $P_{(14)}$ be the permutation matrix exchanging the 1st and the 4th rows. We write the size-6

FFT matrix as follows:

$$P_{(14)} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6 & \omega_6^2 & \omega_6^3 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & \omega_6^3 & 1 & \omega_6^3 & 1 & \omega_6^3 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^5 & \omega_6^4 & \omega_6^3 & \omega_6^2 & \omega_6 \end{pmatrix} P_{(14)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & 1 & 1 & \omega_6^3 & \omega_6^3 & \omega_6^3 \\ 1 & \omega_6^4 & \omega_6^2 & \omega_6^3 & \omega_6 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & \omega_6^3 & \omega_6^5 & \omega_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^2 & \omega_6^4 \end{pmatrix}.$$

3.5 Rader's FFT for Odd Prime p

Suppose $\omega_p \in R$ for an odd prime p. [Rad68] introduced how to map a polynomial $\sum_i a_i x^i \in R[x]/\langle x^p - 1\rangle$ to the tuple $(\hat{a}_j) := \left(\sum_i a_i \omega_p^{ij}\right) \in \prod_i R[x]/\left\langle x - \omega_p^i \right\rangle$ with a size-$(p-1)$ cyclic convolution. Let g be a generator of $\mathbb{Z}_p^*$ and write $j = g^k$ and $i = g^{-\ell}$. Then $\hat{a}_{g^k} - a_0 = \hat{a}_j - a_0 = \sum_{i=1}^{p-1} a_i \omega_p^{ij} = \sum_{\ell=0}^{p-2} a_{g^{-\ell}} \omega_p^{g^{k-\ell}}$ for $k = 0, \dots, p-2$.

The sequence $\left(\sum_{\ell=0}^{p-2} a_{g^{-\ell}} \omega_p^{g^{k-\ell}}\right)_{j=0,\dots,p-2}$ is the size-$(p-1)$ cyclic convolution of sequences $\left(a_{g^{-i}}\right)_{i=0,\dots,p-2}$ and $\left(\omega_p^{g^i}\right)_{i=0,\dots,p-2}$. For example, let $p = 5$. We have $(1, 2, 3, 4) = (2^4, 2, 2^3, 2^2)$ and

$$\begin{pmatrix} \hat{a}_2 - a_0 \\ \hat{a}_4 - a_0 \\ \hat{a}_3 - a_0 \\ \hat{a}_1 - a_0 \end{pmatrix} = \begin{pmatrix} \omega_5 & \omega_5^2 & \omega_5^4 & \omega_5^3 \\ \omega_5^3 & \omega_5 & \omega_5^2 & \omega_5^4 \\ \omega_5^4 & \omega_5^3 & \omega_5 & \omega_5^2 \\ \omega_5^2 & \omega_5^4 & \omega_5^3 & \omega_5 \end{pmatrix} \begin{pmatrix} a_3 \\ a_4 \\ a_2 \\ a_1 \end{pmatrix}.$$

3.6 Schönhage's and Nussbaumer's FFTs

Instead of isomorphisms based on CRT, we sometimes compute chains of monomorphisms and determine the unique inverse image from the product of two images. Given polynomials $\boldsymbol{a}, \boldsymbol{b} \in R[x]/\langle \boldsymbol{g} \rangle$ where $\boldsymbol{g}$ is a degree-$n_0 n_1$ polynomial, we introduce $y = x^{n_1}$, and write $\boldsymbol{a}$ and $\boldsymbol{b}$ as polynomials in $R[x, y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle$ where $\boldsymbol{g}_0|_{y = x^{n_1}} = \boldsymbol{g}(x)$. In other words, $\boldsymbol{a}(y) := \sum_{i_0=0}^{n_0-1} \left(\sum_{i=0}^{n_1-1} a_{i+i_0 n_1} x^i\right) y^{i_0} \in R[x, y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle$. We recap transforms when $R[x, y]/\langle x^{n_1} - y, \boldsymbol{g}_0 \rangle$ does not naturally split.

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

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

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

$$\frac{R[x]}{\langle x^{n_0n_1}+1\rangle} \cong \frac{R[x,y]}{\langle x^{n_1}-y, y^{n_0}+1\rangle} \hookrightarrow \frac{\frac{R[y]}{\langle y^{n_0}+1\rangle}[x]}{\langle x^{2n_1}-1\rangle} \triangleq \frac{\tilde{R}[x]}{\langle x^{2n_1}-1\rangle} \cong \prod_i \frac{\tilde{R}[x]}{\langle x-y^i\rangle}.$$

Our presentation is motivated by [Ber01, Section 9, Paragraph "High–radix variants"] and [vdH04, Section 3].

4 Implementations

In this section, we discuss our ideas for multiplying polynomials over $\mathbb{Z}_{4591}$. For brevity, we assume $R=\mathbb{Z}_{4591}$ in this section. The state-of-the-art vectorized "big by big" polynomial multiplication in NTRU Prime [BBCT22] computed the product in $R[x]/\big\langle (x^{1024}+1)(x^{512}-1)\big\rangle$ with Schönhage and Nussbaumer. This leads to 768 size-8 base multiplications where all of them are negacyclic convolutions. [BBCT22] justified the choice as follows:

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

We propose transformations (essentially) *quartering* and *halving* the number of coefficients involved in base multiplications for vectorization. Our first transformation computes the result in $R[x]/\big\langle x^{1536}-1\big\rangle$. We apply Good–Thomas with $\omega_3\in R$ for a more rapid decrease of the sizes of polynomial rings, Schönhage for radix-2 butterflies, and Bruun over $R[x]/\big\langle x^{32}+1\big\rangle$. This leads to 384 size-8 base multiplications defined over trinomial moduli. Our second transformation computes the result in $R[x]/\big\langle x^{1632}-1\big\rangle$. We show how to incorporate Rader for radix-17 butterflies and Good–Thomas for the coprime factorization $17\cdot 3\cdot 2$. For computing the size-16 weighted convolutions, we split with Cooley–Tukey and Bruun for $R[x]/\big\langle x^{16}\pm\omega_{102}^i\big\rangle$. Since no coefficient ring extensions are involved, this leads to 96 size-8 base multiplication with binomial moduli, 96 size-8 base multiplications with trinomial moduli, and six size-16 base multiplications with binomial moduli.

Section 4.1 formalizes the needs of vectorization, and Section 4.2 goes through our implementation `Good--Thomas` for big-by-small polynomial multiplications. We then go through big-by-big polynomial multiplications. Section 4.3 goes through our implementation `Good--Schönhage--Bruun`, and Section 4.4 goes through our implementation `Good--Rader--Bruun`.

4.1 The Needs of Vectorization

We formalize "the needs of vectorization" to justify how we choose among transformations. In the literature, power-of-two-sized FFTs are oftenly described as easily vectorizable. In this paper, we explicitly state and relate them to the designs of vectorization-friendly polynomial multiplications. Our definition is based on our programming experience.

We assume that a reasonable vector instruction set should provide the following features accessible to programmers:

- Several vector registers each holding a large number of bits of data. Commonly, each register holds 2^k bits.
- Several vector arithmetic instructions computing 2^k-bit data from 2^k-bit data while regarding each 2^k-bit data as packed elements.
  - If input and output are regarded as packed $2^{k'}$-bit data, we call the instruction a single-width instruction.
  - If input is regarded as packed $2^{k'-1}$-bit data and output is regarded as packed $2^{k'}$-bit data, we call the instruction a widening instruction.
  - If input is regarded as packed $2^{k'}$-bit data and output is regarded as packed $2^{k'-1}$-bit data, we call the instruction a narrowing instruction.

The terminologies "widening" and "narrowing" come from [ARM21]. For a $k' \leq k$, we are interested in the number of elements $v = 2^{k-k'}$ contained in a vector register. Intuivitely, we want to compute with minimal number of data shuffling while maintaining the vectorization feature: if we want to add up several pairs (a_i, b_i) of elements, we assign (a_i) to one vector register and (b_i) to another one and issue a vector addtion, similarly for subtractions, multiplications, and bitwise operations. We formalize this intuition for algebra homomorphisms.

Let π be a platform-dependent set of module homomorphisms. We'll specify $\pi = \pi(\texttt{neon})$ in the case of Neon shortly. Let f be an algebra homomorphism. We call f "vectorization friendly" if f is a composition of homomorphisms of the form $g \otimes \mathrm{id}_v \otimes d$ for g an algebra homomorphism, d a composition of elements from π. Since $g \otimes \mathrm{id}_v$ operates over several chunks of v-sets, we need no permutations for this part. For the set π, we define it with the matrix view for simplicity. π is defined as the set of module homomorphisms representable as a $v' \times v'$ diagonal matrix or a size-v' cyclic/negacyclic shift for v' a multiple of v.

In this paper, we start with $R[x]\Big/\Big\langle \boldsymbol{g}\left(x^{v'}\right)\Big\rangle \cong R[y]\Big/\Big\langle x^{v'} - y, \boldsymbol{g}(y)\Big\rangle$ for v' a multiple of v and transform accordingly.

4.2 Good–Thomas FFT in "Big×Small" Polynomial Multiplications

We recall below the design principle of vectorization–friendly Good–Thomas from [AHY22], and describe our implementation `Good--Thomas` for the "big by small" polynomial multiplications. For a cyclic convolution $R[x]/\langle x^{vn_0n_1} - 1\rangle$ where n_0 and n_1 coprime, and v a multiple of the number of coefficients in a vector, one introduces the equivalences $x^v \sim uw$, $u^{n_0} \sim w^{n_1} \sim 1$. Usually, one picks n_0 and n_1 carefully for fast computations. In the simplest form, one picks n_0 as a power of 2 and $n_1 = 3$. Our `Good--Thomas` computes the polynomial multiplication in $\mathbb{Z}[x]/\big\langle x^{1536} - 1\big\rangle$ with $(v, n_0, n_1) = (4, 128, 3)$ where $v = 4$ comes from the fact that each Neon SIMD register holds four 32-bit values. After reaching $\mathbb{Z}[x, u, w]/\big\langle x^4 - uw, u^3 - 1, w^{128} - 1\big\rangle$, we want to compute size-3 NTT over $u^3 - 1$ and size-128 NTT over $w^{128} - 1$. It suffices to choose a large modulus q' with a principal 384-th root of unity. We choose q' as a 32-bit modulus bounding the maximum value of the product in $\mathbb{Z}[x]/\big\langle x^{1536} - 1\big\rangle$. Obviously, our `Good--Thomas` supports any "big-by-small" polynomial multiplications with size less than or equal to 1536.

4.3 Good–Thomas, Schönhage's, and Bruun's FFT

This section describes our `Good--Schönhage--Bruun`. We briefly recall the AVX2-optimized "big by big" polynomial multiplication by [BBCT22]. They computed the product in $R[x]/\big\langle (x^{512} - 1)(x^{1024} + 1)\big\rangle$. They first applied Schönhage as follows.

$$\begin{aligned}\frac{R[x]}{\langle (x^{512} - 1)(x^{1024} + 1)\rangle} &\cong \frac{\frac{R[x]}{\langle x^{32} - y\rangle}[y]}{\langle (y^{16} - 1)(y^{32} + 1)\rangle}\\ \hookrightarrow \quad \frac{\frac{R[x]}{\langle x^{64} + 1\rangle}[y]}{\langle (y^{16} - 1)(y^{32} + 1)\rangle} &\cong \prod_{i=0,1,3,j=0,\dots,15} \frac{\frac{R[x]}{\langle x^{64} + 1\rangle}[y]}{\langle y - x^{2i+8j}\rangle}.\end{aligned}$$

They then applied Nussbaumer for multiplying in $\frac{R[x]}{\langle x^{64} + 1\rangle}$ as follows.

$$\frac{R[x]}{\langle x^{64} + 1\rangle} \cong \frac{\frac{R[x]}{\langle x^8 - z\rangle}[z]}{\langle z^8 + 1\rangle} \hookrightarrow \frac{\frac{R[x]}{\langle x^{16} - 1\rangle}[z]}{\langle z^8 + 1\rangle} \cong \frac{\frac{R[z]}{\langle z^8 + 1\rangle}[x]}{\langle x^{16} - 1\rangle} \cong \prod_{k=0,\dots,15} \frac{\frac{R[z]}{\langle z^8 + 1\rangle}[x]}{\langle x - z^k\rangle}.$$

The vectorization-friendliness of Schönhage is obvious. In principle, Nussbaumer is vectorization-friendly since it shares the same computation as Schönhage after transposing.

Truncated Schönhage vs Good–Thomas and Schönhage. We first discuss an optimization of Schönhage if there is a principal root of unity with order coprime to the one defining Schönhage.

How it works, mathematically. In $R = \mathbb{Z}_{4591}$, we know that there is a principal 3rd root of unity $\omega_3 \in R$. Instead of computing in $R[x]/\big\langle (x^{512} - 1)(x^{1024} + 1)\big\rangle$, we apply Schönhage and Good–Thomas FFTs to $R[x]/\big\langle x^{1536} - 1\big\rangle$. By definition, if $\boldsymbol{\omega}$ is a principal 2^k-th root of unity, then $\omega_3\boldsymbol{\omega}$ is a principal $3 \cdot 2^k$-th root of unity. Let's define $\bar{R} = R[x]/\big\langle x^{32} + 1\big\rangle$. We introduce a principal 32-th root of unity $\boldsymbol{\omega}_{32} = x^2$ as follows:

$$\frac{R[x]}{\langle x^{1536} - 1\rangle} \cong \frac{\frac{R[x]}{\langle x^{16} - y\rangle}[y]}{\langle y^{96} - 1\rangle} \hookrightarrow \frac{\bar{R}[y]}{\langle y^{96} - 1\rangle}.$$

Then $\omega_3\boldsymbol{\omega}_{32}$ is a principal 96-th root of unity implementing $\bar{R}[y]/\langle y^{96}-1\rangle \cong \prod_{i=0,1,2,j=0,\ldots,31} \bar{R}[y]\Big/\Big\langle y-\omega_3^i\boldsymbol{\omega}_{32}^j\Big\rangle$. However, one should not implement this isomorphism with Cooley–Tukey FFT. Observe that multiplication by $\boldsymbol{\omega}_{32} = x^2$ requires negating and permuting whereas multiplication by ω_3 requires actual modular multiplication. Cooley–Tukey FFT requires one to multiply $\omega_3^i\boldsymbol{\omega}_{32}^j$ which is unreasonably complicated while optimizing for $i, j \neq 0$. We apply Good–Thomas FFT implementing $\bar{R}[y]/\langle y^{96}-1\rangle \cong \bar{R}[y]/\langle y-uw, u^3-1, w^{32}-1\rangle$. Obviously, we only need multiplications by powers of ω_3 and $\boldsymbol{\omega}_{32}$ and not $\omega_3\boldsymbol{\omega}_{32}$. See Table 1 for an overview of available approaches.

Table 1: Approaches for computing the size-1536 product of two polynomials drawn from $R[x]/\langle x^{761}-x-1\rangle$.

Approach	Domain	Image	Twiddle factors
Truncated Schönhage [BBCT22]	$\frac{R[x]}{\langle (x^{1024}+1)(x^{512}-1)\rangle}$	$\left(\frac{R[x]}{\langle x^{64}+1\rangle}\right)^{48}$	x^{2i}
Cooley–Tukey and Schönhage	$\frac{R[x]}{\langle x^{1536}-1\rangle}$	$\left(\frac{R[x]}{\langle x^{32}+1\rangle}\right)^{96}$	$\omega_3^i x^{2j}$
Good–Thomas and Schönhage	$\frac{R[x]}{\langle x^{1536}-1\rangle}$	$\left(\frac{R[x]}{\langle x^{32}+1\rangle}\right)^{96}$	ω_3^i, x^{2j}

How it works, concretely. We detail the implementation as follows.

- We transform the input array `in[761]` into a temporary array `out[3][32][32]`, where `out[`i`][`j`][0-31]` is the size-32 polynomial in $\frac{R[x]}{\langle x^{32}+1, u-\omega_3^i, w-x^{2j}\rangle}$. Concretely, we combine the permutations of Good–Thomas and Schönhage as `out[`i`][`j`][`k`] = in[`$(16(64i+33j) \bmod 96)+k$`]` if $(16(64i+33j) \bmod 96)+k < 761$ and zero otherwise. This step is the foundation of the implicit permutations [ACC+21].
- For input `small`, we start with the 8-bit form of the polynomial. Since coefficients are in $\{\pm 1, 0\}$, we first perform five layers of radix-2 butterflies without any modular reductions. The initial three layers of radix-2 butterflies are combined with the implicit permutations. For the last two layers of radix-2 butterflies, we use `ext` if the root is not a power of x^{16}. For the last layer of radix-2 butterflies, we merge the sign-extension and add-sub pairs into the sequence `saddl`, `saddl2`, `ssubl`, `ssubl2`. We then apply one layer of radix-3 butterflies based on the improvement of [DV78, Equation 8]. We compute the radix-3 NTT $(\hat{v}_0, \hat{v}_1, \hat{v}_2)$ of size-32 polynomials (v_1, v_2, v_3) as:

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

Algorithm 5 Radix-2 butterfly with symbolic root x^2.

```
Input: Size-32 8-bit polynomials $a = \texttt{a0} + \texttt{a1}x^{16}, b = \texttt{b0} + \texttt{b1}x^{16}$, where $\texttt{a0}, \texttt{a1}, \texttt{b0}, \texttt{b1}$
are SIMD registers containing:
    $\begin{cases} \texttt{a0} = a_7 || \cdots || a_0, \\ \texttt{a1} = a_{15} || \cdots || a_8, \\ \texttt{b0} = b_7 || \cdots || b_0, \\ \texttt{b1} = b_{15} || \cdots || b_8. \end{cases}$
Output: $\texttt{a0} + \texttt{a1}x^{16} = (a + bx^2) \bmod (x^{32} + 1), \texttt{b0} + \texttt{b1}x^{16} = (a - bx^2) \bmod (x^{32} + 1)$
ext    v0.16b, b0.16b, b1.16b, #14      ▷ $\texttt{v0} = b_{29} || \cdots || b_{14}$
neg    b1.16b, b1.16b
ext    v1.16b, b1.16b, b0.16b, #14      ▷ $\texttt{v1} = b_{13} || \cdots || b_0 || (-b_{31}) || (-b_{30})$
sub    b0.16b, a0.16b, v0.16b
sub    b1.16b, a1.16b, v1.16b           ▷ $\texttt{b0} + \texttt{b1}x^{16} = (a - x^2 b) \bmod (x^{32} + 1)$
add    a0.16b, a0.16b, v0.16b
add    a1.16b, a1.16b, v1.16b           ▷ $\texttt{a0} + \texttt{a1}x^{16} = (a + x^2 b) \bmod (x^{32} + 1)$
```

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

Nussbaumer vs Bruun. Next, we discuss efficient polynomial multiplications in $R[x]/\langle x^{32} + 1 \rangle$. [BBCT22] applied Nussbaumer to $R[x]/\langle x^{64} + 1 \rangle$. We state without proof that applying Nussbaumer to $R[x]/\langle x^{32} + 1 \rangle$ results in 8 polynomial multiplications in $R[z]/\langle z^8 + 1 \rangle$. We instead apply Brunn's FFT resulting in multiplications in rings $R[x]/\langle x^8 + \alpha x^4 + 1 \rangle$ for 4 different α. Since

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

we apply $\mathbf{Bruun}_{1229,1}$ followed by $\mathbf{Bruun}_{58,1}$ and $\mathbf{Bruun}_{2116,1}$. We have slower FFT and base multiplications, but we do only half as many as in [BBCT22]. See Table 2 for comparisons.

Table 2: Approaches for multiplying in $R[x]/\langle x^{64}+1\rangle$ and $R[x]/\langle x^{32}+1\rangle$.

Approach	Domain	Image	Twiddle factors
Nussbaumer [BBCT22]	$\frac{R[x]}{\langle x^{64}+1\rangle}$	$\left(\frac{R[z]}{\langle z^{8}+1\rangle}\right)^{16}$	z^{i}
Nussbaumer	$\frac{R[x]}{\langle x^{32}+1\rangle}$	$\left(\frac{R[z]}{\langle z^{8}+1\rangle}\right)^{8}$	z^{2i}
Bruun	$\frac{R[x]}{\langle x^{32}+1\rangle}$	$\prod_{i=0,1}\prod\frac{R[x]}{\langle x^{8}\pm\alpha_i x^{4}+1\rangle}$	Elements in R.

Then, we perform $96 \cdot 4 = 384$ size-8 base multiplications and compute the inverses of Bruun's, Schönhage's, and Good–Thomas FFT.

4.4 Good–Thomas, Rader's, and Bruun's FFT

In the previous section, we replace Nussbaumer with Bruun. This section shows how to replace Schönhage with Rader while computing in $R[x]/\langle x^{1632}-1\rangle$. We name the resulting computation `Good--Rader--Bruun`.

Schönhage vs Rader-17. We first observe that the Schönhage in [BBCT22] reduced a size-1536 problem to several size-64 problems. We are looking for a multiple of 17 close to $\frac{1536}{64} = 48$. We choose 51 since one can define a size-51 cyclic NTT nicely over $\mathbb{Z}_q$ and optimize further by extending the size-51 cyclic NTT to size-102. For the size-102 cyclic NTT, we apply the 3-dimensional Good–Thomas FFT by identifying $(\omega_{17}, \omega_3, \omega_2) = (\omega_{102}^{e_0}, \omega_{102}^{e_1}, \omega_{102}^{e_2})$ as the principal roots of unity where (e_0, e_1, e_2) is the unique tuple satisfying $\forall a \in \mathbb{Z}_{102}, a \equiv e_0(a \bmod 17) + e_1(a \bmod 3) + e_2(a \bmod 2) \pmod{102}$. Algorithm 6 is an illustration. Radix-2 and radix-3 computations are straightforward. For the radix-17 cyclic FFT, we apply Rader's FFT. Algorithm 7 illustrates the multi-dimensional cyclic FFT. Obviously, the above computation is vectorization–friendly.

Algorithm 6 Good–Thomas, in practice merged with Algorithm 7.

Inputs: `src[1632]`.

Outputs: `poly_NTT[17][3][2][16]`.

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

Algorithm 7 FFTs over chunks of 16 coefficients.

```
Inputs: poly_NTT[17][3][2][16].
Outputs: poly_NTT[17][3][2][16].
for $i_3 \in \{0, \dots, 15\}$ do
    for $i_1 \in \{0, 1, 2\}, i_2 \in \{0, 1\}$ do
        rader-17(poly_NTT[0-16][$i_1$][$i_2$][$i_3$]).
    end for
    for $i_0 \in \{0, \dots, 16\}$ do
        radix-$(3, 2)$(poly_NTT[$i_0$][0-2][0-1][$i_3$]).
    end for
end for
```

Generalize Bruun over $x^{2^k} + c$ for $c \neq \pm 1$. The composed multiplication over a finite field shows that the remaining factorization follows the same pattern of factorizing $R[x]/\langle x^{16} \pm 1 \rangle$. The isomorphism $R[x]/\langle x^{16} - \omega_{102}^{2i} \rangle \cong \prod R[x]/\langle x^8 \pm \omega_{102}^{i} \rangle$ is obvious. Since we also have $\prod_i R[x]/\langle x^{16} - \omega_{102}^{2i+1} \rangle \cong \prod_i R[x]/\langle x^{16} + \omega_{102}^{2i} \rangle$ by permuting, it suffices to understand the isomorphisms defined on $R[x]/\langle x^{16} + \omega_{102}^{2i} \rangle$. Applying Lemma 3, we have $R[x]/\langle x^{16} + \omega_{102}^{2i} \rangle \cong \prod R[x]/\langle x^8 \pm \sqrt{2}\omega_{102}^{128i} x^4 + \omega_{102}^{256i} \rangle$.

Finally, the remaining computing task is multiplication in $R[x]/\langle x^8 + \alpha x^4 + \beta \rangle$ for some $\alpha, \beta \in R$. We extend the idea of [CHK+21, Algorithm 17] by altering between multiplying in $R[x]$ and reducing modulo $x^8 + \alpha x^4 + \beta$.

5 Results

We present the performance numbers in this section. We focus on polynomial multiplications, leaving the fast constant-time GCD [BY19] as future work.

5.1 Benchmark Environment

We use the Raspberry Pi 4 Model B featuring the quad-core Broadcom BCM2711 chipset. It comes with a 32 kB L1 data cache, a 48 kB L1 instruction cache, and a 1 MB L2 cache and runs at 1.5 GHz. For hashing, we use the `aes`, `sha2`, and `fips202` from PQClean [KSSW] without any optimizations due to the lack of corresponding cryptographic units. For the randombytes, [BHK+22] used the `randombytes` from SUPERCOP which in turn used `chacha20`. We extract the conversion from `chacha20` into `randombytes` from SUPERCOP and replace `chacha20` with our optimized implementations using the pipelines `I0/I1`, `F0/F1`. We use the cycle counter of the PMU for benchmarking. Our programs are compilable with GCC 10.3.0, GCC 11.2.0, Clang 13.1.6, and Clang 14.0.0. We report numbers for the binaries compiled with GCC 11.2.0.

Table 3: Overview of polynomial multiplications in `ntrulpr761/sntrup761`.

Armv8-A Neon		x86 AVX2	
Implementation	Cycles	Implementation	Cycles
Big-by-small polynomial multiplications			
`Good--Thomas`	47 696	[BBCT22]	16 992
[Haa21]	242 585		
Big-by-big polynomial multiplications			
`Good--Rader--Bruun`	39 788	[BBCT22]	25 113
`Good--Schönhage--Bruun`	50 398		

5.2 Performance of Vectorized Polynomial Multiplications

Table 3 summarizes the performance of vectorized polynomial multiplications.

Table 4: Detailed `Good--Schönhage--Bruun` cycle counts including reducing to $\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}$.

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

For NTRU Prime, our `Good--Rader--Bruun` performs the best. It is followed by `Good--Thomas` and `Good--Schönhage--Bruun`. Notice that `Good--Rader--Bruun` requires no extensions or changes of coefficient rings. The closest instances in the literature regarding vectorization are the `Good--Thomas` and `Schönhage--Nussbaumer` by [BBCT22], and `Good--Thomas` by [Haa21]. [BBCT22]'s, [Haa21], and our `Good--Thomas` compute “big by small” polynomial multiplications. We outperform [Haa21] `Good--Thomas` by a factor of 6.1× since they implemented the base multiplications with scalar code using the C

% operator. On the other hand, [BBCT22]'s `Schönhage--Nussbaumer` and our `Good--Schönhage--Bruun` compute "big by big" polynomial multiplications. Regarding the impact of switching "big by small" to "big by big", [BBCT22]'s `Schönhage--Nussbaumer` takes $\frac{25113}{16992} \approx 147.79\%$ cycles of their own `Good--Thomas` [BBCT22, Section 3.4.2] while our `Good--Schönhage--Bruun` takes only $\frac{50398}{47696} \approx 105.67\%$ cycles of our own `Good--Thomas`. Essentially, this demonstrates the benefit of vectorization-friendly Good–Thomas and Bruun over truncated [vdH04] Schönhage and Nussbaumer.

Table 5: Detailed cycle counts of `Good--Rader--Bruun`, excluding reductions to $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$.

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

We also provide the detailed cycle counts of the polynomial multiplications. For the "big by big" polynomial multiplications in `sntrup761/ntrulpr761`, Table 5 details the numbers of `Good--Rader--Bruun` and Table 4 details the numbers of `Good--Schönhage--Bruun`.

5.3 Performance of Schemes

Before comparing the overall performance, we first illustrate the performance numbers of some other critical subroutines. Most of our optimized implementations of these subroutines are not seriously optimized except for parts involving polynomial multiplications. We simply translate existing techniques and AVX2-optimized implementations into Neon. Table 6 summarizes the performance of inversions, encoding, and decoding.

Table 6: Performance of inversions, encoding, and decoding in NTRU Prime.

Operation	Ref	Ours
`sntrup761/ntrulpr761`		
`Rq_recip3`	116 353 545	5 811 777
`R3_recip`	127 578 811	587 407
`Rq_encode`	17 753	2 084
`Rq_decode`	31 715	3 914
`Rounded_encode`	14 707	3 145
`Rounded_decode`	31 832	3 445
`crypto_sort_uint32`	186 867	21 659

Inversions, sorting network, encoding, and decoding. For `sntrup761`, we need one inversion over $\mathbb{Z}_{4591}$ and one inversion over $\mathbb{Z}_3$. We bitslice the inversion over $\mathbb{Z}_3$, and identify and vectorize the hottest loop in the inversion over $\mathbb{Z}_{4591}$. Additionally, we translate AVX2-optimized sorting network, encoding, and decoding into Neon. Notice that inversions over $\mathbb{Z}_2$, $\mathbb{Z}_3$, and $\mathbb{Z}_{4591}$, sorting networks, encoding, and decoding are implemented in a generic sense. With fairly little effort, they can be used for other parameter sets.

Performance of `sntrup761/ntrulpr761`. Table 7 summarizes the overall performance. For `ntrulpr761`, our key generation, encapsulation, and decapsulation are 2.98×, 2.79×, and 3.07× faster than [Haa21]. For `sntrup761`, we outperform the reference implementation significantly. Finally, Table 8 details the performance.

Constant-time concerns. There are no input-dependent branches in our code. Our program is constant-time only if one believes the documentation [ARM15]. The source code from [Haa21] and Armv8-A works [NG21, BHK+22], indicate the requirement of the same assumption. In the most relevant documented Neon implementations, our code is constant-time, but this is never strictly guaranteed[10] even with Data-Independent Timing (DIT). If ARM decides to extend the domain of DIT to relevant multiplication instructions used in this paper, our code is guaranteed to be constant-time once the DIT flag is set. Furthermore, literally all the lattice-based post-quantum cryptosystems will be benefit from this since the constant-time concerns arise from the basic building blocks implementing modular multiplications.

---

[10] ARM's DIT flag, according to `https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT--Data-Independent-Timing`, does not guarantee the high half multiplications `sqrdmulh` and `sqdmulh` to be constant-time.

Table 7: Overall cycles of `sntrup761`/`ntrulpr761`.

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

Acknowledgments

This work was supported in part by the Academia Sinica Investigator Award AS-IA-109-M01, and Taiwan's National Science and Technology Council grants 112-2634-F-001-001-MBK and 112-2119-M-001-006.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Detailed Performance Numbers

Table 8: Detailed performance numbers of `sntrup761` and `ntrulpr761` with `Good--Rader--Bruun`. Only performance-critical subroutines are shown.

`sntrup761`		`ntrulpr761`	
Operation	Cycles	Operation	Cycles
`crypto_kem_keypair`	6 333 403	`crypto_kem_keypair`	260 606
`ZKeyGen`	6 248 089	`ZKeyGen`	247 919
		`XKeyGen`	243 332
`KeyGen`	6 194 194	`KeyGen`	112 496
`Rq_recip3`	5 811 777		
`R3_recip`	587 407		
`Rq_mult_small`	39 829	`Rq_mult_small`	39 829
`sort`	22 369	`sort`	21 243
`randombytes`	86 932	`randombytes`	44 713
		`aes`	127 203
`Rq_encode`	2 084	`Rounded_encode`	3 145
`sha2`	13 207	`sha2`	16 386
`crypto_kem_enc`	147 977	`crypto_kem_enc`	412 629
`ZEncrypt`	48 639	`ZEncrypt`	383 991
		`XEncrypt`	374 695
`Encrypt`	40 650	`Encrypt`	83 487
`Rq_mult_small`	39 829	`Rq_mult_small` (2×)	2× 39 829
		`aes`	253 597
		`sort`	21 773
		`sha2`	2 914
`Rq_decode`	3 914	`Rounded_decode`	3 445
`Rounded_encode`	3 145	`Rounded_encode`	3 145
`randombytes`	45 109		
`sha2`	29 713	`sha2`*	26 548
`sort`	21 659		
`crypto_kem_dec`	158 233	`crypto_kem_dec`	461 250
`ZDecrypt`	88 054	`ZDecrypt`	47 573
`Decrypt`	83 892	`XDecrypt` (defined as `Decrypt`)	43 799
`Rq_mult_small`	39 829	`Rq_mult_small`	39 829
`R3_mult`	42 059		
`Rounded_decode`	3 445	`Rounded_decode`	3 445
`ZEncrypt`	48 639	`ZEncrypt`	383 991
`sha2`	18 111	`sha2`*	16 982

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