Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4

Erdem Alkim[1], Vincent Hwang[2,3] and Bo-Yin Yang[3]

[1] Dokuz Eylul University, Izmir, Turkey
erdemalkim@gmail.com
[2] National Taiwan University, Taipei, Taiwan
[3] Academia Sinica, Taipei, Taiwan
vincentvbh7@gmail.com,by@crypto.tw

Abstract. We propose NTT implementations with each supporting at least one parameter of NTRU and one parameter of NTRU Prime. Our implementations are based on size-1440, size-1536, and size-1728 convolutions without algebraic assumptions on the target polynomial rings. We also propose several improvements for the NTT computation. Firstly, we introduce dedicated radix-$(2, 3)$ butterflies combining Good–Thomas FFT and vector-radix FFT. In general, there are six dedicated radix-$(2, 3)$ butterflies and they together support implicit permutations. Secondly, for odd prime radices, we show that the multiplications for one output can be replaced with additions/subtractions. We demonstrate the idea for radix-3 and show how to extend it to any odd prime. Our improvement also applies to radix-$(2, 3)$ butterflies. Thirdly, we implement an incomplete version of Good–Thomas FFT for addressing potential code size issues. For NTRU, our polynomial multiplications outperform the state-of-the-art by $2.8\% - 10.3\%$. For NTRU Prime, our polynomial multiplications are slower than the state-of-the-art. However, the SotA exploits the specific structure of coefficient rings or polynomial moduli, while our NTT-based multiplications exploit neither and apply across different schemes. This reduces the engineering effort, including testing and verification.

Keywords: NTT · NTRU · NTRU Prime · Cortex-M4 · NISTPQC · Vector-Radix FFT · Good–Thomas FFT

1 Introduction

Shor's algorithm for integer factorization and discrete logarithm threatens public-key cryptosystems based on RSA and ECC [Sho97]. Since then, researchers have been developing cryptosystems without known weaknesses from quantum computers. This line of research is known as "post-quantum cryptography". In 2016, the National Institute of Standards and Technology (NIST) called for proposals replacing existing standards for public-key cryptosystems with schemes resisting attacks by quantum computers.

Recent research has shown that the number-theoretic transform (NTT) plays an important role in the implementations for lattice-based submissions, including Dilithium [BHK+22], Kyber [BKS19, AHKS22, BHK+22], NTRU [CHK+21], NTRU Prime [ACC+21], and Saber [CHK+21, ACC+22, BHK+22, BMK+22]. If NTT is natively supported, then we can apply it directly. On the other hand, if NTT is not natively supported, we have to choose a large NTT-friendly polynomial ring covering the maximum value of the result in $\mathbb{Z}[x]$. This approach is known as "NTT multiplications for NTT-unfriendly rings" by [CHK+21, FSS20]. On Cortex-M4, the most compelling non-NTT-based approach is the Toeplitz matrix-vector product (TMVP) exploiting the structure of (weighted) convolutions without changing the coefficient rings [IKPC20, IKPC22]. [IKPC20] demonstrated the idea for Saber on the ARM Cortex-M4. Their implementation remains the fastest non-NTT-based approach on Cortex-M4 compared to [BMK+22]. Recently, [IKPC22] showed that TMVP is faster than the NTT-based multiplication by [CHK+21] for NTRU on Cortex-M4.

We propose improvements for NTT-based multiplications balancing between performance and code size without assuming any algebraic properties of the target coefficient rings. Our NTT-based multiplications outperform [IKPC22] for parameters `ntruhps2048677`, `ntruhrss701`, and `ntruhps4096821`. Since no algebraic properties are assumed, our implementations naturally extend to NTRU Prime while TMVP may not apply to NTRU Prime, with its polynomial modulus $x^p - x - 1$, without doubling the size of convolutions

Cortex-M4 implementations targeting the board `STM32F407-DISCOVERY`, [KRS19, BKS19, IKPC20, MKV20, ABCG20, ACC+21, CHK+21, GKS21, AHKS22, ACC+22, IKPC22] reported performance numbers while clocking at 24 MHz to avoid wait states. A much more meaningful way for benchmarking is to report the numbers at the full speed, 168 MHz, of the board. This would illustrate to the users the impact of the performance while adjusting the frequency for their development. We take this under consideration and report at both 24 MHz and 168 MHz. Our implementations are designed with compact code size and negligible performance penalties while raising the frequency.

Contribution. Our contribution is summarized as follows.

1. We implement NTT-based convolutions, each supporting multiple parameters.
   (a) Our NTT-based size-1440 convolution supports `ntruhps2048677`, `ntruhrss701`, and `ntrulpr653/sntrup653`. We present the implementations of all.
   (b) Our NTT-based size-1536 convolution supports (in addition to those instances in (a)) `ntruhps2048677`, `ntruhrss701`, `ntrulpr761/sntrup761`. We present the implementations of `ntruhps2048677`, `ntruhrss701`, and `ntrulpr761/sntrup761`.
   (c) Our NTT-based size-1728 convolution supports (in addition to (a) and (b)) `ntruhps4096821`, `ntrulpr857/sntrup857`. We present the implementations of `ntruhps4096821`, and `ntrulpr857/sntrup857`.
2. We implement dedicated radix-$(2, 3)$ butterflies for combining Good–Thomas FFT by [Goo58] and vector–radix FFT by [HMCS77]. In general, six different dedicated radix-$(2, 3)$ butterflies together support implicit permutations for Good–Thomas FFT when considering radix-2 and -3 transformations. For size-1536 convolutions, they compare favorably to the (dedicated) radix-2 butterflies [ACC+21, Figure 2].
3. We point out an overlooked optimization for all non-radix-2 butterflies. In particular, for a radix-r butterfly with $r \neq 2$, we replace $(r - 1)$ multiplications and 1 addition with $(r - 2)$ additions and 1 subtraction. This extends the existence of subtraction in radix-2 butterflies to all butterflies. Thus, it is applicable to other platforms and other implementations computing non-radix-2 butterflies. We further apply this optimization to our radix-$(2, 3)$ butterflies implementing vector-radix FFT.
4. We reduce code size while permuting coefficients implicitly for Good–Thomas FFT. To enable this, we formally present the original Good–Thomas FFT [Goo58] as an isomorphism from an associative algebra to a tensor product of associative algebras, which justifies the existence of incomplete versions of Good–Thomas FFT different from [Ber01] and [ACC+21][1]. To demonstrate the practical code size advantage, we benchmark our NTT-based polynomial multiplications at the full speed of our board.

[1]They presented a restricted form of Good–Thomas FFT as an isomorphism from a group algebra to a tensor product of group algebras.

Dedicated radix-$(2, 3)$ butterflies and improved non-radix-2 butterflies are algorithmic improvements applicable to other platforms. While our code size optimization for Good–Thomas FFT is specific to our board `STM32F407-DISCOVERY`, our demonstrated approach for addressing code size issues has other benefits, e.g. potential to facilitate vectorization.

We discuss two use cases where multiplications based on our NTT-based convolutions are more favorable: (i) Plural schemes with comparable parameter sets are selected by NIST and other institutions (cf. OpenSSH). (ii) Multiple parameter sets of one scheme are being implemented (here, for the M4). For (i), state-of-the-art multipliers for NTRU [IKPC22] and NTRU Prime [Che21] exploit the special structures of polynomial moduli or coefficient rings. So adapting code for NTRU Prime and NTRU for each other is hard. Since we compute the results in $\mathbb{Z}[x]$, the only distinctions are the reductions to the target polynomial rings. For (ii), each of the multipliers for NTRU Prime by [Che21] only supports one parameter. The multipliers by [IKPC22] (without doubling polynomial degrees) support NTRU parameters up to a fixed degree, and our NTT-based convolutions support NTRU and NTRU Prime parameters up to a fixed degree. Notice our NTT-based multipliers with compact code sizes are faster than the unrolled multipliers by [IKPC22].

Code. See `https://github.com/vincentvbh/multi-params-ntt_NTRU_NTRUPrime`.

Related work. The incomplete transformation of Good–Thomas FFT can already be deduced from [Goo58]. [FP07] was aware of the use of incomplete Good–Thomas FFT for vectorization, but it is unclear if their program Spiral picked incomplete Good–Thomas FFT as the best strategy of code generation. [ACC+21] and [Che21] implemented polynomial multiplications for NTRU Prime on Cortex-M4. [ACC+21] also explained how to permute implicitly for Good–Thomas FFT by the proposed dedicated 3-layer radix-2 butterflies. [IKPC20] proposed the use of TMVP for Saber on Cortex-M4. Shortly after, [CHK+21] implemented NTT-based polynomial multiplications for LAC, NTRU, and Saber on Cortex-M4 and Skylake with AVX2. Finally, [IKPC22] applied TMVP to NTRU on Cortex-M4.

Structure of this paper. This paper is structured as follows: Section 2 is the background. Section 3 introduces our NTT improvements. Section 4 describes our implementations of NTT-based multiplications. Section 5 gives performance numbers.

2 Preliminaries

Section 2.1 introduces the target polynomial multiplications in NTRU and NTRU Prime. Section 2.2 introduces the Cortex-M4 and the implementations of modular reductions and multiplications. Section 2.3 explains NTTs. We then review various kinds of FFTs including Cooley–Tukey (Section 2.4), Good–Thomas (Section 2.5), and vector–radix FFT (Section 2.6). Finally, Section 2.7 explains how to apply NTTs to NTT-unfriendly rings.

2.1 Polynomial Multiplications in NTRU and NTRU Prime

The NTRU submission [CDH+20] comprises parameter sets for two similar schemes NTRU-HPS and NTRU-HRSS, both operating on the polynomial rings $\mathbb{Z}_3[x]/\langle\Phi_n\rangle$, $\mathbb{Z}_q[x]/\langle\Phi_n\rangle$, and $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$. Here q is a power of 2, n is prime, and Φ_n is the polynomial $\sum_{i=0}^{n-1} x^i$. The NTRU Prime submission [BBC+20] consists of two families of schemes NTRU LPRime and Streamlined NTRU Prime. Both operate in the polynomial rings $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ and $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ for various p and q, primes such that $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ is a finite field. We focus on polynomial multiplications in the rings $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ of NTRU and $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ of NTRU Prime each with one operand ternary (coefficients in $\{-1, 0, 1\}$). See parameters in Tables 1–2.

While NTRU-HRSS requires no sampling of polynomials with fixed numbers of $\{-1, 0, 1\}$ coefficients, NTRU-HPS, NTRU LPRime, and Streamlined NTRU Prime call a sorting network subroutine for the sampling. Furthermore, inverting polynomials are required in the key generations of NTRU-HPS, NTRUHRSS, and Streamlined NTRU Prime. We refer to the specifications [CDH+20, BBC+20] for more details.

Table 1: NTRU parameter sets. Starred parameters are covered in this paper.

	NTRU-HPS				NTRU-HRSS	
	ntruhps2048509	ntruhps2048677*	ntruhps4096821*	ntruhps40961229	ntruhrss701*	ntruhrss1373
n	509	677	821	1229	701	1373
q	2048	2048	4096	4096	8192	16384

Table 2: NTRU Prime parameter sets. Starred parameters are covered in this paper.

NTRU LPRime	ntrulpr653*	ntrulpr761*	ntrulpr857*	ntrulpr953	ntrulpr1013	ntrulpr1277
Streamlined NTRU Prime	sntrup653*	sntrup761*	sntrup857*	sntrup953	sntrup1013	sntrup1277
p	653	761	857	953	1013	1277
q	4621	4591	5167	6343	7177	7879

2.2 Cortex-M4

As selected by NIST for evaluating PQC candidates on micro-controllers, the ARM Cortex-M4 is our target platform for implementing PQC schemes. The Cortex-M4 implements the Armv7E-M architecture. Some of the most relevant features are as follows:

General-purpose registers: There are 16 core registers, named `r0`–`r15`. Except for the stack pointer (`r13`) and the program counter (`r15`), all other core registers can be treated as general-purpose registers.

Floating-point registers: There are 32 single-precision floating-point registers that can also be used as a low-latency cache [ACC+21, CHK+21, ACC+22, AHKS22].

Single cycle: Most instructions take 1 cycle each, including multiplications `mul`, `mla`, and `mls`, long multiplications `{u,s}{mul, mla}l`, and signed most-significant-word multiplications `sm{mul, mla, mls}{, r}`. An exception is that a string of l single-load instructions (`ldr{, h, sh, b, sb}`) takes $l + 1$ cycles. For more details on load/store timings, see [ARM10].

Barrel shifters: Shifts and rotates (`asr`, `lsl`, `lsr`, and `ror`), come at no extra cost when used as the "flexible second operand" of a standard data-processing instruction.

We first describe the multiplication instructions. `mul` multiplies two 32-bit values and places the lower 32-bit result to the first (destination) register. `mla` accumulates the 32-bit result to an accumulator, and `mls` subtract the 32-bit result from the accumulator. The accumulators are the last arguments named to `mla` and `mls`. `smull` multiplies two 32-bit values and places the 64-bit result in two destination registers. The first-named register holds the lower 32-bit result, and the second-named register holds the upper 32-bit result. `smlal` accumulates the 64-bit result to the destination registers. `umull` and `umlal` are their unsigned counterparts. `smmul` returns the upper 32-bit of a 64-bit product of two 32-bit values. The suffix `r` indicates that the 64-bit product is first rounded to the upper 32-bit.

Modular reductions and multiplications are the most critical parts of NTT-based polynomial multiplications. We implement 32-bit Barrett reduction [Bar86], 32-bit Montgomery reduction, and 32-bit Montgomery multiplication [Mon85, Sei18]. Throughout this paper, we assume $\mathtt{R} = 2^{32}$ and let $\bmod^{\pm}$ be the signed modular reduction. 32-bit Barrett reduction maps a value a to $a - \left\lfloor \frac{a \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil q$ as an approximation of $a - \left\lfloor \frac{a}{q} \right\rceil q = a \bmod^{\pm} q$. Algorithm 1 is an illustration. Montgomery multiplication, denoted as $\mathtt{montgomery}(a, b)$, computes

$$\frac{ab + q \cdot \left(ab \cdot \left(-q^{-1} \bmod^{\pm} \mathtt{R}\right) \bmod^{\pm} \mathtt{R}\right)}{\mathtt{R}} \equiv ab\mathtt{R}^{-1} \bmod^{\pm} q$$

from (a, b). To see why it is a reduction, we observe that in terms of absolute values, $|\mathtt{montgomery}(a, b)| \le \frac{|ab|}{\mathtt{R}} + \frac{q}{2}$. If $|a| \le \frac{\mathtt{R}}{2}$ and $|b| \le \frac{q}{2}$, we have $|\mathtt{montgomery}(a, b)| \le \frac{q}{4} + \frac{q}{2} = \frac{3q}{4}$. Concretely, we denote `mMul_des_32(l, h, a, b, t)` as the 32-bit Montgomery multiplication with input registers $\mathtt{a}, \mathtt{b}$ and output register `h` as shown in Algorithm 2.

Algorithm 1 Barrett reduction

```
Input: a = $a$
Output: a = $a - \left\lfloor \frac{a \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil \cdot q$
1: smmulr t a, $\left\lfloor \frac{\mathtt{R}}{q} \right\rceil$
2: mls a t, $q$, a
```

Algorithm 2 `mMul_des_32`

```
Input: (a, b) = $(a, b)$
Output: h $\equiv ab\mathtt{R}^{-1} \bmod^{\pm} q$
1: smull l, h, a, b
2: mul t, l, $-q^{-1} \bmod^{\pm} \mathtt{R}$
3: smlal l, h, t, $q$
```

2.3 Number–Theoretic Transforms

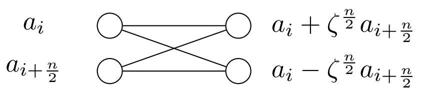
In this paper, we assume readers are already familiar with the language of tensor product of associative algebras over the same commutative ring. Nevertheless, we go through some important concepts in algebra. A homomorphism from an algebraic structure to another one is a structure-preserving map. We call it an isomorphism if there is a one-to-one correspondence between the domain and the range. The importance of isomorphisms is the underlying computational costs for converting expensive computations into cheap computations without losing any algebraic properties. Let R be a commutative ring and $f(x)$ a polynomial with coefficients in R. The polynomial ring $R[x]/\langle f(x) \rangle$ is an associative algebra over R. If $f(x)$ takes the form $x^n - 1$ for some $n \in \mathbb{N}$, $R[x]/\langle x^n - 1\rangle$ is a group algebra since it can be constructed naturally by taking elements in the cyclic group $(\mathbb{Z}_n, +, 0)$ as the basis. Polynomial multiplications in $R[x]/\langle x^n - 1\rangle$ are called (cyclic) convolutions. If $f(x) = x^n - \psi$ where $\psi \neq 1 \in R$, polynomial multiplications in $R[x]/\langle x^n - \psi \rangle$ are called weighted convolutions [CF94]. Let $R[x^{(0)}]/\left\langle f_0(x^{(0)})\right\rangle$ and $R[x^{(1)}]/\left\langle f_1(x^{(1)})\right\rangle$ be two polynomial rings. Their tensor product $R[x^{(0)}]/\left\langle f_0(x^{(0)})\right\rangle \otimes R[x^{(1)}]/\left\langle f_1(x^{(1)})\right\rangle$ is also an associative algebra over R. We also write $R[x^{(0)}]/\left\langle f_0(x^{(0)})\right\rangle \otimes R[x^{(1)}]/\left\langle f_1(x^{(1)})\right\rangle$ as $\frac{R[x^{(0)}, x^{(1)}]}{\left\langle f_0(x^{(0)}), f_1(x^{(1)})\right\rangle}$. We refer to [Jac12, Sec. 3.9] and [Bou89, Chap. III, Sec. 4.1] for a more formal treatment. At a high level our implementations convert the multiplication in $R[x]/\langle x^n - 1\rangle$ into the multiplication in a tensor product of associative algebras.

We proceed with the definitions of the number–theoretic transforms (NTTs). If ζ^m is invertible in R, m is coprime to the characteristic of R, and there is a principal m-th root of unity ω_m, we can define the size-m NTT, $\mathtt{NTT}_{R[x]/\langle x^n - \zeta^m \rangle : \omega_m}$ [Pol71, Für09, HvdH21]. If $R = \mathbb{Z}_q$ with the prime factorization $q = p_0^{l_0} \cdots p_{d-1}^{l_{d-1}}$, it can be shown that the above condition is equivalent to the existence of n fulfilling $n | \mathbf{0}(q) := \gcd(p_0 - 1, \dots, p_{d-1} - 1)$ [AB74]. $\mathtt{NTT}_{R[x]/\langle x^n - \zeta^m \rangle : \omega_m}$ transforms the ring $R[x]/\langle x^n - \zeta^m\rangle$ into the product ring $\prod_{i=0}^{m-1} R[x]/\left\langle x^{\frac{n}{m}} - \zeta\omega_m^i \right\rangle$ by first re-writing a polynomial $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i x^i$ as $\boldsymbol{a}'(x') = \sum_{i=0}^{m-1} \left(\sum_{j=0}^{\frac{n}{m}-1} a_{i\frac{n}{m}+j} x^j \right) x'^i$ and sending $\boldsymbol{a}'(x')$ to the m-tuple $\left(\boldsymbol{a}'(\zeta), \dots, \boldsymbol{a}'(\zeta\omega_m^{m-1})\right)$. If

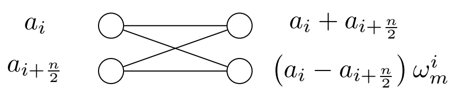
$m = n$, we call it complete NTT, and if $m \neq n$, we call it incomplete NTT. If $\zeta^m = 1$, we call it cyclic NTT and write it as $\texttt{NTT}_{R[x]:n:\omega_m}$. When the context is clear, we simply say NTT. Furthermore, let us denote by ω_n a principal n-th root of unity. If ω_n exists, there are $\phi(n)$ choices of ω_n sharing the same algebraic properties where ϕ is the Euler's totient function. For an $m|n$, we usually fix an ω_n and define $\omega_m \coloneqq \omega_n^{\frac{n}{m}l}$ where l is coprime to m.

2.4 Cooley–Tukey and Gentleman–Sande FFTs

Cooley–Tukey [CT65] and Gentleman–Sande [GS66] FFTs are popular approaches for computing size-m NTTs for highly composite m. For a factorization of $m = m_0 m_1$, Cooley–Tukey FFT computes the isomorphism $R[x]/\langle x^n - \zeta^m\rangle \cong \prod_{i=0}^{m-1} R[x]/\langle x^{\frac{n}{m}} - \zeta\omega_m^i\rangle$ via applying size-m_0 NTTs with $\omega_{m_0} \coloneqq \omega_m^{m_1}$ and size-m_1 NTTs with $\omega_{m_1} \coloneqq \omega_m^{m_0}$ as follows: $R[x]/\langle x^n - \zeta^m\rangle \cong \prod_{i_0=0}^{v_0-1} R[x]\Big/\Big\langle x^{\frac{n}{m_0}} - \zeta^{m_1}\omega_{m_0}^{i_0}\Big\rangle \cong \prod_{i_0=0}^{m_0-1}\prod_{i_1=0}^{m_1-1} R[x]/\langle x - \zeta\omega_m^{i_0}\omega_{m_1}^{i_1}\rangle$. Since $\omega_m^{i_0}\omega_{m_1}^{i_1} = \omega_m^{i_0+i_1m_0}$, the result is only different from $\texttt{NTT}_{R[x]/\langle x^n-\zeta^m\rangle:\omega_m}$ by a permutation. If $m = r^k$ for some r and k, we can apply a k-level split where each level consists of several size-r NTTs. It can be easily shown that there is a radix-r reversal between the result of CT FFT and the straightforward computation. Note that the first level maps a polynomial $\boldsymbol{a}(x)$ to $\left(\boldsymbol{a}'(\zeta^{m_1}\omega_{m_0}^{i_0})\right)_{i_0}$ by evaluating $x^{\frac{n}{m_0}}$ at several $\zeta^{m_1}\omega_{m_0}^{i_0}$'s. Such evaluations are also called Cooley–Tukey butterflies. Figure 1a is an illustration for $(\boldsymbol{a}')_0$ (up) and $(\boldsymbol{a}')_1$ (down) where $m_0 = 2$ (and $w_2 = -1$), and Figure 2a is an illustration for $(\boldsymbol{a}')_0$ (up), $(\boldsymbol{a}')_1$ (middle), and $(\boldsymbol{a}')_2$ (down) where $m_0 = 3$.

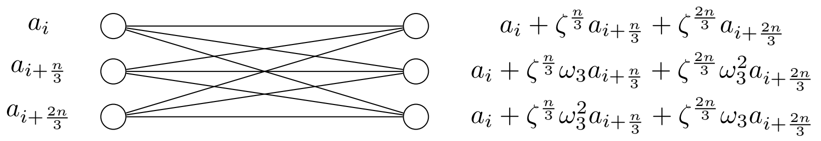

a_i
$a_{i+\frac{n}{2}}$
$a_i + \zeta^{\frac{n}{2}} a_{i+\frac{n}{2}}$
$a_i - \zeta^{\frac{n}{2}} a_{i+\frac{n}{2}}$

(a) Cooley–Tukey butterflies.

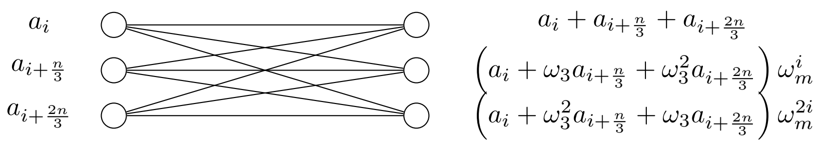

a_i
$a_{i+\frac{n}{2}}$
$a_i + a_{i+\frac{n}{2}}$
$\left(a_i - a_{i+\frac{n}{2}}\right)\omega_m^i$

(b) Gentleman–Sande butterflies.

Figure 1: Radix-2 butterflies.


a_i
$a_{i+\frac{n}{3}}$
$a_{i+\frac{2n}{3}}$
$a_i + \zeta^{\frac{n}{3}} a_{i+\frac{n}{3}} + \zeta^{\frac{2n}{3}} a_{i+\frac{2n}{3}}$
$a_i + \zeta^{\frac{n}{3}} \omega_3 a_{i+\frac{n}{3}} + \zeta^{\frac{2n}{3}} \omega_3^2 a_{i+\frac{2n}{3}}$
$a_i + \zeta^{\frac{n}{3}} \omega_3^2 a_{i+\frac{n}{3}} + \zeta^{\frac{2n}{3}} \omega_3 a_{i+\frac{2n}{3}}$

(a) Cooley–Tukey butterflies.


a_i
$a_{i+\frac{n}{3}}$
$a_{i+\frac{2n}{3}}$
$a_i + a_{i+\frac{n}{3}} + a_{i+\frac{2n}{3}}$
$\left(a_i + \omega_3 a_{i+\frac{n}{3}} + \omega_3^2 a_{i+\frac{2n}{3}}\right)\omega_m^i$
$\left(a_i + \omega_3^2 a_{i+\frac{n}{3}} + \omega_3 a_{i+\frac{2n}{3}}\right)\omega_m^{2i}$

(b) Gentleman–Sande butterflies.

Figure 2: Radix-3 butterflies.

Gentleman–Sande FFT transforms $R[x]/\langle x^n - \zeta^m\rangle$ to $R[x,y]/\langle x^{\frac{n}{m}} - \zeta y, y^m - 1\rangle$ by introducing the equivalence $x^{\frac{n}{m}} \sim \zeta y$ and regarding $R[x]/\langle x^{\frac{n}{m}} - \zeta y\rangle$ as the coefficient ring. For $\zeta^m = 1$, the computation $R[x]/\langle x^n - 1\rangle \cong \prod_{i_0=0}^{m_0-1} R[x]\Big/\Big\langle x^{\frac{n}{m_0}} - \omega_{m_0}^{i_0}\Big\rangle \cong \prod_{i_0=0}^{m_0-1} R[x,y]\Big/\Big\langle y^{\frac{n}{m_0}} - 1\Big\rangle$ is called Gentleman–Sande butterfly. Moreover, the introduction of $x^{\frac{n}{m_0}} \sim \omega_m^{i_0} y$ requires multiplications by powers of $\omega_m^{i_0}$. It is usually written as the map $x^{\frac{n}{m_0}} \mapsto \omega_m^{i_0} y$ and is also called “twisting". Figures 1b and 2b are illustrations for $m_0 = 2$ and $m_0 = 3$. In this paper, we retain the language of equivalence relations for clear explanations of optimizations, but one must keep in mind that we still need multiplications.

If there are no $r, d_0, d_1 \in \mathbb{N}$ satisfying $m_0 = r^{d_0}$ and $m_1 = r^{d_1}$, we call the FFT “mixed–radix”.

2.5 Good–Thomas FFT

Let q_0 and q_1 be two coprime integers. Good–Thomas FFT transforms $\texttt{NTT}_{R[x]:q_0q_1:\omega_{q_0q_1}}$ into $\texttt{NTT}_{R[x^{(0)}]:q_0:\omega_{q_0}} \otimes \texttt{NTT}_{R[x^{(1)}]:q_1:\omega_{q_1}}$ where $\omega_{q_0} := \omega_{q_0q_1}^{e_0}$, $\omega_{q_1} := \omega_{q_0q_1}^{e_1}$, and e_0 and e_1 are orthogonal idempotent elements ($e_{i_0}e_{i_1} = \delta_{i_0,i_1}e_{i_0}$ where δ is the Kronecker delta) realizing $a \equiv e_0(a \bmod q_0) + e_1(a \bmod q_1) \pmod{q_0q_1}$ [Goo58, Section 12][Tho63]. Since cyclic NTTs require fewer multiplications than acyclic ones, this is more favorable than the mixed-radix size-q_0q_1 CT and GS FFT. [Ber01] and [ACC$^+$21] stated the transformation as turning a group algebra into a tensor product of group algebras by introducing $x \sim x^{(0)}x^{(1)}$. However, this statement is actually weaker than the originally proposed formulation by [Goo58]. We describe a statement for convolutions implied by the work of [FP07, Paragraph Vectorized FFT, Section 3].

Let $n = vq_0q_1$ with $q_0 \perp q_1$. Applying Good–Thomas FFT to convolution is to introduce the equivalence $x^v \sim x^{(0)}x^{(1)}$ turning the group algebra $R[x]/\langle x^n - 1\rangle$ into the tensor product $\bar{R}[x^{(0)}]\Big/\Big\langle \big(x^{(0)}\big)^{q_0} - 1\Big\rangle \otimes \bar{R}[x^{(1)}]\Big/\Big\langle \big(x^{(1)}\big)^{q_1} - 1\Big\rangle$ where $\bar{R} := R[x]/\big\langle x^v - x^{(0)}x^{(1)}\big\rangle$. [FP07] was aware of the transformation, but it is unclear if their program Spiral generates the transformation. Moreover, they overlooked two important implementation aspects of multi-dimensional NTTs: (i) vector–radix FFT in the next section, and (ii) code size while permuting with Good–Thomas FFT as shown in Section 3.2.

In the literature, there are at least two maps transforming $\texttt{NTT}_{R[x]:q_0q_1:\omega_{q_0q_1}}$ into the tensor product $\texttt{NTT}_{R[x^{(0)}]:q_0:\omega_{q_0}} \otimes \texttt{NTT}_{R[x^{(1)}]:q_1:\omega_{q_1}}$. One, the “CRT map”, is described above; the other is the “Ruritanian map” [Goo71] introducing the equivalence $x^{vl} \sim x^{(0)}x^{(1)}$ where $l = (q_0 + q_1)^{-1} \bmod q_0q_1$. Since $q_0 + q_1$ is coprime to q_0q_1, $(q_0 + q_1)^{-1}$ is a generator of $(\mathbb{Z}_{q_0q_1}, +, 0)$, and the map $1 \mapsto (q_0 + q_1)^{-1}$ is an automorphism of $(\mathbb{Z}_{q_0q_1}, +, 0)$. This implies an isomorphic way for constructing the group algebra from $\mathbb{Z}_{q_0q_1}$ and defining $\texttt{NTT}_{R[x]:q_0q_1:\omega_{q_0q_1}}$. Since there are $\phi(q_0q_1)$ generators of $(\mathbb{Z}_{q_0q_1}, +, 0)$, there are $\phi(q_0q_1)$ many automorphisms and exactly the same number of maps for transforming a 1-dimensional NTT into a 2-dimensional NTT. The smallest example is $(q_0, q_1) = (2, 3)$ where $\phi(q_0q_1) = 2$. This explains why only two maps can be given in general.

Table 3: Permutation of Good–Thomas FFT for size-6 convolutions.

CRT mapping							Ruritanian mapping						
i	0	1	2	3	4	5	i	0	1	2	3	4	5
i_0	0	1	0	1	0	1	i_0	0	1	0	1	0	1
i_1	0	1	2	0	1	2	i_1	0	2	1	0	2	1

2.6 Vector–Radix FFT

Vector–radix FFT is a multi-dimensional generalization of FFT introduced by [HMCS77]. The core idea is that for a d-dimensional $\bigotimes_{i=0}^{d-1} \texttt{NTT}_{R[x^{(i)}]/\left\langle \left(x^{(i)}\right)^{n_i} - \zeta^{m_i}\right\rangle:\omega_{m_i}}$, the computations of each dimension are independent of each other and we can interleave them freely as long as the order of operations in the same dimension is preserved. Since each dimension can be written as a sequence of additions and multiplications, we can interleave the computations such that strings of multiplications are adjacent to each other. If we look closely into the strings of multiplications from several dimensions, we find that a lot of entries are multiplied by more than one twiddle factor. If we precompute the twiddle factors multiplied to the same entry, then we save multiplications.

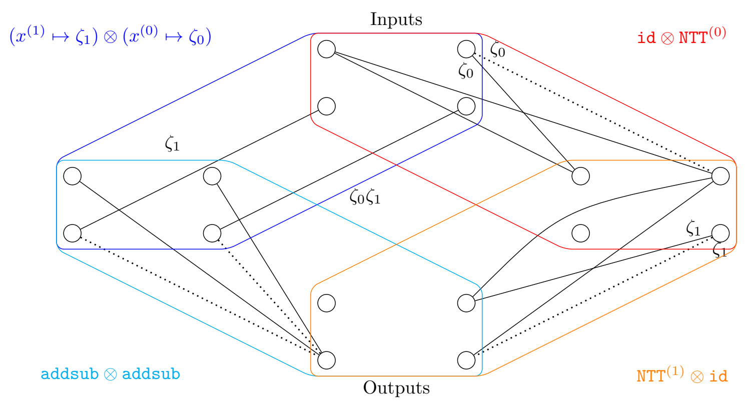

$(x^{(1)} \mapsto \zeta_1) \otimes (x^{(0)} \mapsto \zeta_0)$
Inputs
$\texttt{id} \otimes \texttt{NTT}^{(0)}$
ζ_0
ζ_0
ζ_1
$\zeta_0\zeta_1$
ζ_1
ζ_1
$\texttt{addsub} \otimes \texttt{addsub}$
Outputs
$\texttt{NTT}^{(1)} \otimes \texttt{id}$

Figure 3: Radix-$(2, 2)$ butterfly and 2-layer radix-2 butterfly comparison. A group of four circles represents four coefficients. Each circulated region applies the tensor product of isomorphisms with the same color to a group of four. The left four are scaled versions of the inputs for the radix-$(2, 2)$ butterfly, and the right four are the intermediate results of 1-dimensional radix-2 butterflies. Real lines send the scaled results and dotted lines send the negated and scaled ones. Several lines are omitted for clarity. For the details, please refer to the definitions of the symbols.

We illustrate the idea of [HMCS77] with a 2-dimensional NTT with dimension 2×2. Let $\texttt{NTT}^{(0)} := \texttt{NTT}_{R[x^{(0)}]/\left\langle \left(x^{(0)}\right)^2 - \zeta_0^2 \right\rangle : -1}$, $\texttt{NTT}^{(1)} := \texttt{NTT}_{R[x^{(1)}]/\left\langle \left(x^{(1)}\right)^2 - \zeta_1^2 \right\rangle : -1}$, $\texttt{id}$ be the identity map, and $\texttt{addsub} := (a, b) \mapsto (a + b, a - b)$. Applying 1-dimension NTTs is to compute with $\left(\texttt{NTT}^{(1)} \otimes \texttt{id}\right) \circ \left(\texttt{id} \otimes \texttt{NTT}^{(0)}\right)$ as follows.

$$\begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \overset{\texttt{id}\otimes\texttt{NTT}^{(0)}}{\mapsto} \begin{pmatrix} a_{0,0} + \zeta_0 a_{0,1} & a_{0,0} - \zeta_0 a_{0,1} \\ a_{1,0} + \zeta_0 a_{1,1} & a_{1,0} - \zeta_0 a_{1,1} \end{pmatrix}$$

$$\overset{\texttt{NTT}^{(1)}\otimes\texttt{id}}{\mapsto} \begin{pmatrix} a_{0,0} + \zeta_0 a_{0,1} + \zeta_1 a_{1,0} + \zeta_0\zeta_1 a_{1,1} & a_{0,0} - \zeta_0 a_{0,1} + \zeta_1 a_{1,0} - \zeta_0\zeta_1 a_{1,1} \\ a_{0,0} + \zeta_0 a_{0,1} - \zeta_1 a_{1,0} - \zeta_0\zeta_1 a_{1,1} & a_{0,0} - \zeta_0 a_{0,1} - \zeta_1 a_{1,0} + \zeta_0\zeta_1 a_{1,1} \end{pmatrix}.$$

Vector–radix FFT first decomposes $\texttt{NTT}^{(0)}$ into $\texttt{addsub} \circ (x^{(0)} \mapsto \zeta_0)$ and $\texttt{NTT}^{(1)}$ into $\texttt{addsub} \circ (x^{(1)} \mapsto \zeta_1)$. If we group the multiplications together, we have

$$\left(\texttt{NTT}^{(1)} \otimes \texttt{id}\right) \circ \left(\texttt{id} \otimes \texttt{NTT}^{(0)}\right) = (\texttt{addsub} \otimes \texttt{addsub}) \circ \left((x^{(1)} \mapsto \zeta_1) \otimes (x^{(0)} \mapsto \zeta_0)\right)$$

where $(x^{(1)} \mapsto \zeta_1) \otimes (x^{(0)} \mapsto \zeta_0)$ can be written as $\left(x^{(0)}\right)^{i_0} \left(x^{(1)}\right)^{i_1} \mapsto \zeta_0^{i_0}\zeta_1^{i_1}$. Obviously, $\left(x^{(0)}\right)^{i_0} \left(x^{(1)}\right)^{i_1} \mapsto \zeta_0^{i_0}\zeta_1^{i_1}$ requires 3 multiplications whereas $\left((x^{(1)} \mapsto \zeta_1) \otimes \texttt{id}\right) \circ \left(\texttt{id} \otimes (x^{(0)} \mapsto \zeta_0)\right)$ requires 4 multiplications. Figure 3 is an illustration.

Note that the core idea of vector–radix FFT is that multiplications from different dimensions can be merged. This holds if we compute both dimensions with GS FFT, and in general, even if we compute CT in one dimension and GS in the other.

2.7 NTT Multiplications for NTT-unfriendly Rings

A sequence of ring operations can be computed in a larger ring if the results of each step can be uniquely identified in the larger ring. The approach for lifting a coefficient ring to a larger one is known as "NTT multiplications for NTT-unfriendly rings" [CHK+21, FSS20]. In a broader sense, we use the same terminology for the lifting a polynomial modulus to a larger polynomial modulus in this paper.

3 Number–Theoretic Transforms

This section introduces some theoretical aspects on implementing NTTs. Section 3.1 introduces an optimization applicable to non-radix-2 butterflies. Section 3.2 illustrates a potential code size issue about Good–Thomas FFT. Section 3.3 demonstrates a principle for balancing between code size and performance.

3.1 Improving Non-Radix-2 Butterflies

We first discuss an interesting observation regarding non-radix-2 butterflies. For simplicity, we compare radix-2 and radix-3 CT butterflies, but our observation can be generalized to arbitrary radices. In this section, we assume ψ is an invertible element.

3.1.1 Radix-3 Butterflies

Algorithm 3 Radix-3 butterfly [CHK+21].

```
Input: (c0, c1, c2) = $(c_0, c_1, c_2)$
Output:
    c0' = $\hat{c0} := c_0 + \psi c_1 + \psi^2 c_2$
    c1' = $\hat{c1} := c_0 + \psi\omega_3 c_1 + \psi^2\omega_3^2 c_2$
    c2' = $\hat{c2} := c_0 + \psi\omega_3^2 c_1 + \psi^2\omega_3^4 c_2$
1: smull t1, c0', c1, $\psi$
2: smlal t1, c0', c2, $\psi^2$
3: mul t0, t1, $q'$
4: smlal t1, c0', t0, $q$                ▷ c0' = $\psi c_1 + \psi^2 c_2$
5:                                       ▷ If $\psi = 1$, we compute c0' = $c_1 + c_2$ with add instead
6: smull t1, c1', c1, $\psi\omega_3$
7: smlal t1, c1', c2, $\psi^2\omega_3^2$
8: mul t0, t1, $q'$
9: smlal t1, c1', t0, $q$                ▷ c1' = $\psi\omega_3 c_1 + \psi^2\omega_3^2 c_2$
10: smull t1, c2', c1, $\psi\omega_3^2$
11: smlal t1, c2', c2, $\psi^2\omega_3^4$
12: mul t0, t1, $q'$
13: smlal t1, c2', t0, $q$               ▷ c2' = $\psi\omega_3^2 c_1 + \psi^2\omega_3^4 c_2$
14: add.w c2', c2', c0                   ▷ c2' = $\hat{c}_2$
15: add c1', c0                          ▷ c1' = $\hat{c}_1$
16: add c0', c0                          ▷ c0' = $\hat{c}_0$
```

Recall that radix-2 CT butterfly maps (c_0, c_1, ψ) to $(c_0 + \psi c_1, c_0 - \psi c_1)$ and radix-3 CT butterfly maps (c_0, c_1, c_2, ψ) to $(\hat{c}_0, \hat{c}_1, \hat{c}_2) = (c_0 + \psi c_1 + \psi^2 c_2, c_0 + \psi\omega_3 c_1 + \psi^2\omega_3^2 c_2, c_0 + \psi\omega_3^2 c_1 + \psi^2\omega_3^4 c_2)$. 32-bit radix-3 butterflies were implemented by [CHK+21] and used in the polynomial multiplication for `ntruhps4096821`. [CHK+21] computed $(\hat{c}_0, \hat{c}_1, \hat{c}_2)$ from (c_0, c_1, ψ) via Algorithm 3, which requires 15 cycles if $\psi \neq 1$ and 12 cycles if $\psi = 1$.

We extend the existence of subtraction in radix-2 to radix-3 with the observations $\omega_3^2 = -(1 + \omega_3)$ and $\omega_3 = -(1 + \omega_3^2)$. We first compute $\psi c_1 + \psi^2 c_2$ and $\psi\omega_3 c_1 + \psi^2\omega_3^2 c_2$. For computing $\hat{c}_2$, we compute $(\psi c_1 + \psi^2 c_2) + (\psi\omega_3 c_1 + \psi^2\omega_3^2 c_2)$ with *one addition* and subtract the result from c_0. Now we have $c_0 - \big((\psi c_1 + \psi^2 c_2) + (\psi\omega_3 c_1 + \psi^2\omega_3^2 c_2)\big) = \hat{c}_2$ as desired. Finally, we add c_0 to $\psi c_1 + \psi^2 c_2$ and $\psi\omega_3 c_1 + \psi^2\omega_3^2 c_2$ to derive $\hat{c}_0$ and $\hat{c}_1$. In total, 12 cycles are required. If $\psi = 1$, then 9 cycles are required. Algorithm 4 is an illustration. Comparing to Algorithm 3, 3 cycles are saved for each radix-3 butterfly.

Algorithm 4 Improved radix-3 butterfly.

```
Input: (c0, c1, c2) = $(c_0, c_1, c_2)$
Output: (c0, c1, c2) = $(\hat{c_0}, \hat{c_1}, \hat{c_2})$
1: smull t1, t0, c1, $\psi$
2: smlal t1, t0, c2, $\psi^2$
3: mul t2, t1, $q'$
4: smlal t1, t0, t2, $q$                    ▷ t0 = $\psi c_1 + \psi^2 c_2$
5:                                          ▷ If $\psi = 1$, we compute t0 = $c_1 + c_2$ with add instead
6: smull t1, c1, c1, $\psi\omega_3$
7: smlal t1, c1, c2, $\psi^2\omega_3^2$
8: mul t2, t1, $q'$
9: smlal t1, c1, t2, $q$                    ▷ c1 = $\psi\omega_3 c_1 + \psi^2\omega_3^2 c_2$
10: add c2, c1, t0
11: rsb c2, c2, c0                          ▷ c2 = $\hat{c_2}$
12: add c1, c0                              ▷ c1 = $\hat{c_1}$
13: add c0, t0                              ▷ c0 = $\hat{c_0}$
```

3.1.2 Generalization to Arbitrary Radices

We now generalize the idea to other radices. For computing $\boldsymbol{c}(\psi\omega_r^i)_{0\le i<r}$ from $\boldsymbol{c}(x) = \sum_{k=0}^{r-1} c_k x^k$, we pick a j indicating which multiplications to be replaced. If $\psi = 1$, we pick $j > 0$. Since $rc_0 = \sum_{i=0}^{r-1} \boldsymbol{c}(\psi\omega_r^i)$ by the definition of ω_r, we have

$$\boldsymbol{c}(\psi\omega_r^j) = rc_0 - \sum_{i=0, i\neq j}^{r-1} \boldsymbol{c}(\psi\omega_r^i) = c_0 - \sum_{i=0, i\neq j}^{r-1} \left(\boldsymbol{c}(\psi\omega_r^i) - c_0\right).$$

This implies that after computing $\boldsymbol{c}(\psi\omega_r^i) - c_0 = \sum_{k=1}^{r-1} c_k\psi^k\omega_r^{ik}$ for all $i \neq j$, $\boldsymbol{c}(\psi\omega_r^j)$ can be computed with $r-2$ additions and one subtraction. Therefore, our idea replaces $r-1$ multiplications with $r-2$ additions. Finally, we subtract $\sum_{i=0,i\neq j}^{r-1} \left(\boldsymbol{c}(\psi\omega_r^i) - c_0\right)$ from c_0.

One should be aware that for a cyclic size-r NTT, there are many approaches other than the näive radix-r butterfly if r is odd. Since Good–Thomas FFT applies whenever r has more than one prime factor, we may assume that $r = p^k$ is a prime power. Winograd's FFT exploits the multiplicative structure of the unit group of $\mathbb{Z}_{p^k}$ to transform the size-p^k NTT into a size-$p^{k-1}(p-1)$ convolution [Win78]. For $k > 1$, since p is odd and $p \perp p-1$, we apply Good–Thomas FFT to transform the size-$p^{k-1}(p-1)$ convolution into a multi-dimensional convolution. Therefore, we may restrict to the case $k = 1$ [Rad68]. If $p-1$ has more than one prime factor, we can also apply Good–Thomas FFT. Since $p-1$ is even, we may assume $p-1 = 2^h$ for some h. It is well known that if 2^h+1 is a prime, then $h = 2^t$ for some t. Therefore, we only need to focus on Fermat primes $2^{2^t}+1$ of which four are known: $3, 5, 17$, and 65537. We already discuss the case $r = 3$. In the next section, we demonstrate the benefit of our ideas for $r = 5$ using 32-bit arithmetic on the Cortex-M4.

3.1.3 The Case of $r = 5$

We first count the cycles spent on the näive approach with 32-bit arithmetic. We first compute $\boldsymbol{c}(1) = c_0 + c_1 + c_2 + c_3 + c_4$ with four additions. For $i = 1, \ldots, 4$, we compute $\boldsymbol{c}(\omega_5^i) - c_0 = c_1\omega_5^i + c_2\omega_5^{2i} + c_3\omega_5^{3i} + c_4\omega_5^{4i}$ with one `smull`, three `smlal`s, and one 32-bit Montgomery reduction. Finally, we add c_0 and derive $\boldsymbol{c}(\omega_5^j)$. In total, we need $4 + (4+2+1)\cdot 4 = 32$ cycles.

We pick $j = 4$ for our improvement. While computing $\boldsymbol{c}(1)$, we first derive $\boldsymbol{c}(1) - c_0 = c_1 + c_2 + c_3 + c_4$ and proceed with the computation for $\boldsymbol{c}(\omega_5) - c_0$, $\boldsymbol{c}(\omega_5^2) - c_0$, and $\boldsymbol{c}(\omega_5^3) - c_0$. Then, we compute $\boldsymbol{c}(\omega_5^4) = c_0 - \sum_{i=0}^{3} \left(\boldsymbol{c}(\omega_5^i) - c_0\right)$ with three additions and one subtraction. Finally, we add c_0 to the first four values. In total, we need $3 + (4 + 2) \cdot 3 + 4 + 4 = 29$ cycles.

We outline Rader's approach as follows. We first observe that $(1, 2, 4, 3) = (2^0, 2^1, 2^2, 2^3)$ in $\mathbb{Z}_5$. This tells us the following

$$\pi \circ \begin{pmatrix} \boldsymbol{c}(\omega_5) - c_0 \\ \boldsymbol{c}(\omega_5^2) - c_0 \\ \boldsymbol{c}(\omega_5^3) - c_0 \\ \boldsymbol{c}(\omega_5^4) - c_0 \end{pmatrix} = \begin{pmatrix} c_{2^0}(\omega_5^{-1})^{2^0} + c_{2^1}(\omega_5^{-1})^{2^3} + c_{2^2}(\omega_5^{-1})^{2^2} + c_{2^3}(\omega_5^{-1})^{2^1} \\ c_{2^0}(\omega_5^{-1})^{2^1} + c_{2^1}(\omega_5^{-1})^{2^0} + c_{2^2}(\omega_5^{-1})^{2^3} + c_{2^3}(\omega_5^{-1})^{2^2} \\ c_{2^0}(\omega_5^{-1})^{2^2} + c_{2^1}(\omega_5^{-1})^{2^1} + c_{2^2}(\omega_5^{-1})^{2^2} + c_{2^3}(\omega_5^{-1})^{2^3} \\ c_{2^0}(\omega_5^{-1})^{2^3} + c_{2^1}(\omega_5^{-1})^{2^2} + c_{2^2}(\omega_5^{-1})^{2^1} + c_{2^3}(\omega_5^{-1})^{2^0} \end{pmatrix}$$

where $\pi = (34)$ is a permutation of four elements. We now compute the cyclic convolution of $(c_{2^0}, c_{2^1}, c_{2^3}, c_{2^4})$ and $\left((\omega_5^{-1})^{2^0}, (\omega_5^{-1})^{2^1}, (\omega_5^{-1})^{2^2}, (\omega_5^{-1})^{2^3}\right)$ with one layer of Cooley–Tukey FFT. Since the butterflies defined on the twiddle factors can be precomputed, the size-4 convolution takes $4 + (2 + 2) \cdot 4 + 4 = 24$ cycles. Finally, we add the values with c_0. Although computing a size-4 convolution with one layer of Cooley–Tukey butterflies requires division by 2, we can multiply the twiddle factors by 2^{-1} prior to the computation. Another observation is that after applying one layer of radix-2 CT butterflies, we already have $c_1 + c_3$ and $c_2 + c_4$ at hand. Therefore, we only need additional 2 cycles for $\boldsymbol{c}(1) = c_0 + c_1 + c_2 + c_3 + c_4$. In summary, we need $24 + 4 + 2 = 30$ cycles for the Rader approach.

All in all, in the case of $r = 5$, our improved näive butterfly outperforms the original näive butterfly and the Rader approach. For $r = 17$, Rader's approach is probably better because one can apply more layers of Cooley–Tukey butterflies.

3.2 Code Size Consideration of Good–Thomas FFT

In this section, we point out a potential issue of Good–Thomas FFT if we permute the coefficients on-the-fly. We illustrate the issue for transforming a size-$2^{k_0}3^{k_1}$ 1-dimensional FFT into a 2-dimensional FFT of dimensions $2^{k_0} \times 3^{k_1}$ with $3^{k_1} < 2^{k_0-1}$. Furthermore, we also assume that the upper half of the input polynomial are all zeros, and compute with dedicated radix-$(2, 3)$ butterflies.

Since 2^{k_0} and 3^{k_1} are coprime, there are 3^{k_1-1} different loops, where each loop calls 2^{k_0-1} dedicated radix-$(2, 3)$ butterflies. Note that the period of the patterns of dedicated radix-$(2, 3)$ butterflies is 3^{k_1}. We can partition the 2^{k_0-1} calls into $\left\lceil \frac{2^{k_0-1}}{3^{k_1}} \right\rceil$ sequences of dedicated radix-$(2, 3)$ butterflies. The first $\left\lfloor \frac{2^{k_0-1}}{3^{k_1}} \right\rfloor$ sequences share the same structure with 3^{k_1} dedicated radix-$(2, 3)$ butterflies and the last sequence consists of $2^{k_0-1} \bmod 3^{k_1}$ dedicated radix-$(2, 3)$ butterflies. We follow the pattern of `good` and `ungood` permutation subroutines in the AVX2 implementations of NTRU Prime for looping. The loop body consists of 3^{k_1} calls of dedicated radix-$(2, 3)$ butterflies, and the conditional test for looping is placed right after the $((2^{k_0-1} \bmod 3^{k_1}) - 1)$-th call (we start at 0). There are 3^{k_1-1} loops where each loop consists of 3^{k_1} calls of dedicated radix-$(2, 3)$ butterflies. We want 3^{2k_1-1} as small as possible for controlling the code size.

Take the parameter set `ntruhps4096821` for NTRU as an example. [CHK+21] computed size-1728 convolutions with mixed-radix CT FFT. They implemented size-576 CT FFT and computed 3×3 schoolbooks. As described in 2.5, Good–Thomas FFT is more favorable than Cooley–Tukey FFT because cyclic NTTs are cheaper than acyclic ones. `ntrulpr761/sntrp761` [ACC+21] and `ntruhps2048677/ntruhrss701` [CHK+21], using Good–Thomas FFT, suggested transforming the size-1728 convolution into a 2-dimensional convolution with dimensions 27×64. For computing the 2-dimensional FFT, applying dedicated radix-$(2, 3)$ butterflies will result in a blowup of code size since it is dominated by $3^{2\cdot 3-1} = 243$ dedicated radix-$(2, 3)$ butterflies, including memory operations. We cannot control the code size and permute the coefficients on the fly as in [ACC+21, CHK+21].

3.3 Combining Cooley–Tukey, Good–Thomas, and Vector–Radix FFTs

We describe how to compute size-$q_0\tilde{q}q_1v$ convolutions where q_0 is a power of 2, $q_1 < \frac{q_0}{2}$ is a power of 3, and $\tilde{q}\perp 3$. Furthermore, we require that at most one of $\tilde{q}$ and v is greater than 1. Here v measures the degree of incompleteness of the coprime factorization of the Good–Thomas FFT, and $\tilde{q}$ the incompleteness of the Cooley–Tukey FFT. We will fix $\tilde{q}$ and v at the end.

Let $\bar{R} = \frac{R[x]}{\langle x^v - x' \rangle}$. We compute the result of $\mathtt{NTT}_{\bar{R}[x']:q_0q_1:\omega_{q_0q_1}}$ as follows. We first convert $\mathtt{NTT}_{\bar{R}[x']:q_0q_1:\omega_{q_0q_1}}$ into $\mathtt{NTT}_{\bar{R}[x^{(0)}]:q_0:\omega_{q_0}} \otimes \mathtt{NTT}_{\bar{R}[x^{(1)}]:q_1:\omega_{q_1}}$ by introducing the equivalence $x' \sim x^{(0)}x^{(1)}$. This can be realized by permuting the coefficients. For computing $\mathtt{NTT}_{\bar{R}[x^{(0)}]:q_0:\omega_{q_0}} \otimes \mathtt{NTT}_{\bar{R}[x^{(1)}]:q_1:\omega_{q_1}}$, we look at $\min(\log_2 q_0, \log_3 q_1)$ for determining the number of layers of radix-$(2, 3)$. We then replace the 0-th layer of radix-$(2, 3)$ butterflies and the permutation $x' \sim x^{(0)}x^{(1)}$ with dedicated radix-$(2, 3)$ butterflies according to the assumption that the upper half of the coefficients are all zeros. After $\min(\log_2 q_0, \log_3 q_1)$ layers of radix-$(2, 3)$ butterflies, we compute the remaining layers of radix-2 butterflies.

We now explain how to pick v for controlling code size. We have to keep in mind that the code size of the initial layer is determined by the period and the number of distinct loops for calling dedicated radix-$(2, 3)$ butterflies. From the previous section, we must have code for $\frac{q_1^2}{3}$ dedicated radix-$(2, 3)$ butterflies. For Cortex-M4, $q_1 = 3, 9$ are reasonable numbers since there are only $\frac{3^2}{3} = 9$ or $\frac{9^2}{3} = 27$ dedicated radix-$(2, 3)$ butterflies to be programmed. For the size-1440, size-1536, and size-1728 convolutions, we choose $(\tilde{q}, v) = (5, 1), (4, 1)$, and $(1, 3)$ since $1440 = 32 \cdot 5 \cdot 9 \cdot 1$, $1536 = 128 \cdot 4 \cdot 3 \cdot 1$, and $1728 = 64 \cdot 1 \cdot 9 \cdot 3$.

Note that for some platforms, code size might not be a consideration. Very often, such platforms are powerful enough to support vector instructions. One should choose v as a multiple of the number of elements contained in a vector. Then, the entire computation, including dedicated radix-$(2, 3)$ butterflies for on-the-fly permutations, requires no permutation instructions and additional memory operations. We believe this will be useful for platforms implementing Neon, MVE, AVX2, AVX512, SSE, and SSE2.

4 Implementations

In this section we go through our implementations for NTT-based polynomial multiplications with each supporting at least one parameter for NTRU and one parameter for NTRU Prime with little modifications. We distinguish two words: level and layer. We use the word “layer” for transformations in terms of mathematics and “level” for computations between a load and a store to the same memory address. All the cycle counts refer to the Cortex-M4 cycles. This section is structured as follows: Section 4.1 introduces dedicated radix-$(2, 3)$ butterflies. Section 4.2 explains our implementations of size-1440, size-1536, and size-1728 convolutions. Section 4.3 introduces our choices of convolutions for NTT-based polynomial multiplications in NTRU and NTRU Prime.

4.1 Dedicated Vector–Radix Butterflies

We first introduce how to implement radix-$(2, 3)$ butterflies while permuting the coefficients with Good–Thomas FFT. For simplicity, we illustrate the idea for $R[x]/\langle x^6 - 1 \rangle$. Let e_0 and e_1 be idempotent elements in $\mathbb{Z}_6$ realizing $i = (e_0(i \bmod 2) + e_1(i \bmod 3)) \bmod 6$.

Given coefficients $(c_0, \dots, c_5)$, we define c_{i_0,i_1} as $c_{e_0 i_0 + e_1 i_1}$. It is clear that introducing the equivalence $x \sim x^{(0)} x^{(1)}$ converts the polynomial $\sum_{i=0}^{5} c_i x^i$ into the polynomial $\sum_{i_0=0}^{2} \sum_{i_1=0}^{1} c_{i_0,i_1} \left(x^{(0)}\right)^{i_0} \left(x^{(1)}\right)^{i_1}$. If $c_3 = c_4 = c_5 = 0$, then we have $c_{1,0} = c_{0,1} = c_{1,2} = 0$. We define the characteristic vector of (c_{i_0,i_1}) as $(\llbracket (e_0 i_0 + e_1 i_1) \bmod 6 \geq 3 \rrbracket)$ where $\llbracket\rrbracket$ is the Iverson bracket. If the higher half of the input is all zeros, then there are six different characteristic vectors if we permute with Good–Thomas FFT by a combinatorial argument. We implement all of them in our NTTs. We illustrate (Alg. 5) our implementation for $(\llbracket (e_0 i_0 + e_1 i_1) \bmod 6 \geq 3 \rrbracket) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. We call it `_6_ntt_101`, since the characteristic vector is determined by $(c_{0,0}, c_{0,1}, c_{0,2})$ because $x \sim x^{(0)} x^{(1)}$. In short, we need 17 cycles.

For the other dedicated radix-$(2,3)$ butterflies[2], we need 12, 17, 12, 17, and 18 cycles respectively. For our size-1536 convolution that will be shown in Section 4.2, there are in total $\frac{1536}{6} = 256$ radix-$(2,3)$ butterflies. Approximately, $\frac{1}{3}$ of them are 12 cycles, and $\frac{2}{3}$ of them are 17 cycles, We omit the details and refer readers to the code. If we agree $15\frac{1}{3}$ cycles are required on average, then we save $\frac{9 \cdot 2 + 2 \cdot 3 - 15\frac{1}{3}}{6} \approx 1.44$ cycles for each entry.

Algorithm 5 `_6_ntt_101`.

Input:

$$\begin{pmatrix} \mathtt{c0} & \mathtt{c2} & \mathtt{c4} \\ \mathtt{c1} & \mathtt{c3} & \mathtt{c5} \end{pmatrix} = \begin{pmatrix} c_{0,0} & 0 & c_{0,2} \\ 0 & c_{1,1} & 0 \end{pmatrix}$$

Output:

$$\begin{pmatrix} \mathtt{c0} & \mathtt{c2} & \mathtt{c4} \\ \mathtt{c1} & \mathtt{c3} & \mathtt{c5} \end{pmatrix} = \begin{pmatrix} \hat{c}_{0,0} & \hat{c}_{0,1} & \hat{c}_{0,2} \\ \hat{c}_{1,0} & \hat{c}_{1,1} & \hat{c}_{1,2} \end{pmatrix} = \left(\mathtt{NTT}_{R[x^{(0)}]:2:\omega_2} \otimes \mathtt{NTT}_{R[x^{(1)}]:3:\omega_3}\right) \begin{pmatrix} c_{0,0} & 0 & c_{0,2} \\ 0 & c_{1,1} & 0 \end{pmatrix}$$

```
1: mMul_des_32 t, c5, c4, $\omega_3$, c1        ▷ c5 = $\omega_3 c_{0,2}$
2: add c2, c4, c5                               ▷ c2 = $-\omega_3^2 c_{0,2}$
3: sub c2, c0, c2                               ▷ c2 = $c_{0,0} + \omega_3^2 c_{0,2}$
4: add c5, c5, c0                               ▷ c5 = $c_{0,0} + \omega_3 c_{0,2}$
5: add c0, c0, c4                               ▷ c0 = $c_{0,0} + c_{0,2}$
6: mMul_des_32 c1, t, c3, $\omega_3$, c4        ▷ t = $\omega_3 c_{1,1}$
7: add c4, c3, t                                ▷ c4 = $-\omega_3^2 c_{1,1}$
8: add c0, c0, c3                               ▷ c0 = $\hat{c}_{0,0}$
9: sub c1, c0, c3, lsl #1                       ▷ c1 = $\hat{c}_{1,0}$
10: add c2, c2, t                               ▷ c2 = $\hat{c}_{0,1}$
11: sub c3, c2, t, lsl #1                       ▷ c3 = $\hat{c}_{1,1}$
12: add c5, c5, c4                              ▷ c5 = $\hat{c}_{1,2}$
13: sub c4, c5, c4, lsl #1                      ▷ c4 = $\hat{c}_{0,2}$
```

4.2 Implementing Convolutions

In this section, we describe in detail our chosen transformations. We illustrate the details of our size-1728 convolution because it is the most complicated one and because it includes all the ideas. We also denote each $\cong$ as an isomorphism corresponding to a level of computation. Table 4 summarizes the transformations for convolutions, and Table 5 summarizes the implementations of transformations.

[2]named `_6_ntt_011`, `_6_ntt_110`, `_6_ntt_100`, `_6_ntt_010`, and `_6_ntt_001`, respectively

Table 4: Summary of transformations.

Convolution	Transformation		
	This work	[ACC+21]	[CHK+21]
Size-1440	$\mathtt{NTT}_{R[x^{(0)}]:5:\omega_5} \otimes \mathtt{NTT}_{R[x^{(1)}]:160:\omega_{32}}$	-	-
Size-1536	$\mathtt{NTT}_{R[x^{(0)}]:3:\omega_3} \otimes \mathtt{NTT}_{R[x^{(1)}]:512:\omega_{128}}$	$\mathtt{NTT}_{R[x^{(0)}]:512:\omega_{512}} \otimes I_3$	$\mathtt{NTT}_{R[x^{(0)}]:512:\omega_{512}} \otimes I_3$
Size-1728	$\left(\mathtt{NTT}_{\bar{R}[x^{(0)}]:9:\omega_9} \otimes \mathtt{NTT}_{\bar{R}[x^{(1)}]:64:\omega_{64}}\right) \otimes I_3$	-	$\mathtt{NTT}_{R[x]:1728:\omega_{576}}$

Table 5: Summary of implementations.

Transformation	NTT	base multiplication
$\mathtt{NTT}_{R[x^{(0)}]:5:\omega_5} \otimes \mathtt{NTT}_{R[x^{(1)}]:160:\omega_{32}}$	$2\times$ radix-$(3,2)$ + 3-layer radix-2	5×5 schoolbook
$\mathtt{NTT}_{R[x^{(0)}]:3:\omega_3} \otimes \mathtt{NTT}_{R[x^{(1)}]:512:\omega_{128}}$	radix-$(3,2)$ + $2 \times$ 3-layer radix-2	4×4 schoolbook
$\left(\mathtt{NTT}_{\bar{R}[x^{(0)}]:9:\omega_9} \otimes \mathtt{NTT}_{\bar{R}[x^{(1)}]:64:\omega_{64}}\right) \otimes I_3$	$2\times$ radix-$(3,2)$ + 4-layer radix-2	3×3 schoolbook

Size-1728 convolution. First of all, we introduce the equivalence $x^3 \sim x^{(0)}x^{(1)}$ to perform an incomplete permutation for Good–Thomas FFT. We now regard $\bar{R} = \frac{\mathbb{Z}_{q'}[x]}{\left\langle x^3 - x^{(0)}x^{(1)} \right\rangle}$ as the coefficient ring. Since $\frac{1728}{3} = 9 \cdot 64$, we perform a 2-dimensional FFT defined over the ring $\frac{\bar{R}[x^{(0)},x^{(1)}]}{\left\langle \left(x^{(0)}\right)^9 - 1, \left(x^{(1)}\right)^{64} - 1 \right\rangle}$ with vector-radix FFT. Our vector-radix FFT is built upon the tensor product of the size-9 CT FFT on $\bar{R}[x^{(0)}] \Big/ \left\langle \left(x^{(0)}\right)^9 - 1 \right\rangle$ and the size-64 CT FFT on $\bar{R}[x^{(1)}] \Big/ \left\langle \left(x^{(1)}\right)^{64} - 1 \right\rangle$. We apply one level of dedicated radix-$(2,3)$ butterflies, one level of radix-$(2,3)$ butterflies, and one level of 4-layer radix-2 butterflies. For applying radix-$(2,3)$ butterflies, we merge the multiplications of twiddles from different dimensions into the improved radix-3 butterflies. The main reason for introducing the equivalence $x^3 \sim x^{(0)}x^{(1)}$ is to permute the coefficients without a blow of code size as explained in Section 3.2. If we instead introduce $x \sim x^{(0)}x^{(1)}$, then there is no hope to permute on-the-fly and compute dedicated radix-$(2,3)$ butterflies at the same time with compact code size. The entire computation computes takes the following route from $\frac{\mathbb{Z}_{q'}[x]}{\left\langle x^{1728} - 1 \right\rangle}$

$$
\begin{aligned}
&\overset{\text{radix-(3, 2)}}{\cong} && \prod_{i_0=0}^{2} \prod_{i_1=0}^{1} \bar{R}[x^{(0)}, x^{(1)}] \Big/ \left\langle \left(x^{(0)}\right)^3 - \omega_9^{3i_0}, \left(x^{(1)}\right)^{32} - \omega_{64}^{32\mathtt{rev}_{(2)}(i_1)} \right\rangle \\
&\overset{\text{radix-(3, 2)}}{\cong} && \prod_{i_{0,0},i_{0,1}=0}^{2} \prod_{i_1=0}^{3} \frac{\bar{R}[x^{(0)},x^{(1)},y^{(0)}]}{\left\langle x^{(0)} - \omega_9^{i_{0,0}} y^{(0)}, y^{(0)} - \omega_9^{3i_{0,1}}, \left(x^{(1)}\right)^{16} - \omega_{64}^{16\mathtt{rev}_{(2:2)}(i_1)} \right\rangle} \\
&= && \prod_{i_0=0}^{8} \prod_{i_1=0}^{3} \left(\bar{R}[x^{(0)}, x^{(1)}] \Big/ \left\langle x^{(0)} - \omega_9^{\mathtt{rev}_{(3:2)}(i_0)}, \left(x^{(1)}\right)^{16} - \omega_{64}^{16\mathtt{rev}_{(2:2)}(i_1)} \right\rangle \right) \\
&\overset{\text{4-layer radix-2}}{\cong} && \prod_{i_0=0}^{8} \prod_{i_1=0}^{63} \left(\bar{R}[x^{(0)}, x^{(1)}] \Big/ \left\langle x^{(0)} - \omega_9^{\mathtt{rev}_{(3:2)}(i_0)}, x^{(1)} - \omega_{64}^{\mathtt{rev}_{(2;6)}(i_1)} \right\rangle \right) \\
&= && \prod_{i_0=0}^{8} \prod_{i_1=0}^{63} \left(\mathbb{Z}_{q'}[x] \Big/ \left\langle x^3 - \omega_9^{\mathtt{rev}_{(3:2)}(i_0)} \omega_{64}^{\mathtt{rev}_{(2:6)}(i_1)} \right\rangle \right).
\end{aligned}
$$

Comparison to [CHK+21]. We compare our implementation to the size-1728 convolution by [CHK+21]. There are two differences: (i) the number of distinct twiddle factors for NTTs, and (ii) the approach for computing the result of a size-576 NTT. For (i), they required $9 + 31 + 63 \cdot 8 = 544$ distinct twiddle factors in their NTTs. We require $9 + 9 \cdot 1 + 30 = 48$ distinct twiddle factors where the $9 \cdot 1$ are the twiddles $\omega_9^{i_0}\omega_{64}^{16}$ used in the vector–radix FFT. This implies fewer memory operations. For (ii), we compute the result of 1-*dimensional* size-576 NTT with 2-*dimensional* FFT where [CHK+21] computes the result of 1-*dimensional* size-576 NTT with 1-*dimensional* FFT. There are two benefits: the 2-dimensional FFT requires fewer multiplications than the 1-dimensional FFT regardless of whether half of the inputs are zeros, and (ii) dedicated radix-$(2,3)$ butterflies save approximately 1.44 cycles for each entry while dedicated 3-layer radix-2 butterflies save only 1 cycle for each entry.

Size-1440 convolution. Let $R = \mathbb{Z}_{q'}$. We introduce the equivalence $x \sim x^{(0)}x^{(1)}$, $\left(x^{(0)}\right)^9 \sim 1$, and $\left(x^{(1)}\right)^{160} \sim 1$, and compute the 2-dimensional transformation $\texttt{NTT}_{R[x^{(0)}]:9:\omega_9} \otimes \texttt{NTT}_{R[x^{(1)}]:160:\omega_{32}}$ with one level of dedicated radix-$(2,3)$ butterflies, one level of radix-$(2,3)$ butterflies, and one level of 3-layer radix-2 butterflies.

Size-1536 convolution. Let $R = \mathbb{Z}_{q'}$. For computing a size-1536 convolution, [ACC+21] and [CHK+21] compute with Good–Thomas FFT by introducing the equivalence $x \sim x^{(0)}x^{(1)}$, $\left(x^{(0)}\right)^3 \sim 1$, and $\left(x^{(1)}\right)^{512} \sim 1$. They then compute three size-512 NTTs and 512 3×3 convolutions. We introduce the same equivalence but proceed differently. We compute the 2-dimensional transformation $\texttt{NTT}_{R[x^{(0)}]:3:\omega_3} \otimes \texttt{NTT}_{R[x^{(1)}]:512:\omega_{128}}$ with one level of dedicated radix-$(2,3)$ butterflies and two levels of 3-layer radix-2 butterflies.

Comparison to [ACC+21]. We now explain why dedicated radix-$(2,3)$ butterflies are more favorable than the Good–Thomas FFT with dedicated 3-layer radix-2 butterflies by [ACC+21]. For simplicity, we compare the computation of $R[x]/\langle x^{24} - 1\rangle$ since $24 = 8 \cdot 3 = 6 \cdot 4$ where the upper half of the coefficients are all zeros. The computation of [ACC+21] can be simplified as applying dedicated 3-layer radix-2 butterflies followed by 3×3 convolutions and the inverses of 3-layer radix-2 butterflies. After carefully counting the arithmetic cycles of implementations by [ACC+21], approximately 31 cycles are required for dedicated 3-layer radix-2 butterflies, 39 cycles are required for inverses 3-layer radix-2 butterflies, and 15 cycles are required for a 3×3 convolution. Therefore, $31 \cdot 3 \cdot 2 + 15 \cdot 8 + 39 \cdot 3 = 432$ cycles are required for a size-24 convolution.

We first assume that a dedicated radix-$(2,3)$ butterflies take $15\frac{1}{3}$ cycles on average (this is not true for a size-24 convolution, but it is true for our size-1536 and size-1728 convolutions), a 4×4 schoolbook takes 30 cycles, and a radix-$(2,3)$ butterfly takes $9 \cdot 2 + 2 \cdot 3 = 24$ cycles. Then we only need $15\frac{1}{3} \cdot 4 \cdot 2 + 30 \cdot 6 + 24 \cdot 4 \approx 399$ cycles, only 92% of the approach by [ACC+21]. The actual saving is larger since after computing dedicated radix-$(2,3)$ butterflies, we reach layer 1 of the dimension of radix-2 while applying dedicated 3-layer radix-2 butterflies by [ACC+21] ends up the layer 3 of the dimension of radix-2. This implies that they need more Montgomery multiplications for the follow up 2 levels of 3-layer radix-2 butterflies.

4.3 Multi-Parameter Support

As alluded to earlier, each of our NTT-based convolutions supports the polynomial multiplications of more than one parameter of NTRU and NTRU Prime. We compute the result in $\mathbb{Z}[x]$ with a chosen NTT-based convolution, and call the specific routine `final_map` for reducing the result to the target polynomial ring. Among our polynomial multiplications, this is the only difference for comparable parameter sets. Furthermore, our NTT-based convolutions for parameters with larger polynomial degrees apply to polynomial multiplications of the smaller parameter sets. Table 6 summarizes the applicability of our NTT-based convolutions. We discuss some possible scenarios demonstrating the benefits in reducing engineering effort.

If more than one comparable parameter is selected by NIST or other institutions (cf. OpenSSH). The first scenario is when more than one comparable parameter sets are selected by multiple institutions. The state-of-the-art polynomial multiplications for NTRU [IKPC22] apply only to the polynomial rings selected by NTRU. Adapting their multipliers incurs two significant performance penalties: (i) the arithmetic in NTRU is in $\mathbb{Z}_{2^k}$ while we need $\mathbb{Z}_q$ for a prime q in NTRU Prime; and (ii) the polynomial moduli $x^p - x - 1$ in NTRU Prime are incompatible to the structure of Toeplitz matrices without doubling the sizes of convolutions. (i) implies many modular reductions in $\mathbb{Z}_q$ for a prime q, and (ii) implies one has to double the sizes of target convolutions. Next, the state-of-the-art polynomial multiplications for NTRU Prime [Che21] rely on the special structure of coefficient rings for choosing the sizes of convolutions. They are therefore not applicable to NTRU. On the other hand, for comparable parameters in NTRU and NTRU Prime, we only need to replace the `final_map` reducing to the target polynomial rings while the other parts, including `NTT`, `NTT_small`, `basemul`, and `iNTT`, remain the same.

Table 6: Summary of the applicability of NTT-based convolutions. Starred checkmarks are implemented in this paper.

	NTRU (n, q)			NTRU Prime (p, q)		
Conv.	$(677, 2048)$	$(701, 8192)$	$(821, 4096)$	$(653, 4621)$	$(761, 4591)$	$(857, 5167)$
Size-1440	✓*	✓*	-	✓*	-	-
Size-1536	✓*	✓*	-	✓*	✓*	-
Size-1728	✓	✓	✓*	✓	✓	✓*

If more than one parameter of a single scheme are selected for the Cortex-M4. Suppose we want to deploy multiple parameters of a scheme X on the Cortex-M4 where X is NTRU or NTRU Prime. The state-of-the-art polynomial multiplications for NTRU Prime on Cortex-M4 require one to provide a different source code for the multipliers. On the other hand, each of our convolutions supports polynomial multiplications up to a certain size. In the extreme case, our size-1728 convolution suffices for all parameters of X with a modified `final_map`. For the state-of-the-art polynomial multiplications for NTRU on Cortex-M4, the Toeplitz-matrix-based approach supports smaller parameters by padding some zeros. However, our more compact implementations are already faster than their unrolled multipliers as we will see in the next section.

5 Results

This section reports the performance numbers of our implementations. We will first describe our benchmark environment in Section 5.1. We will then go through the implementations of convolutions in Section 5.2 and illustrate their impact on NTRU and NTRU Prime in Section 5.3. All of our implementations target “big by small” polynomial multiplications in NTRU and NTRU Prime.

5.1 Benchmark Environment

We target the `STM32F407-DISCOVERY` board featuring an `STM32F407VG` Cortex-M4 microcontroller with 196 kB of SRAM and 1 MB of flash. Our benchmarking setup is based on `pqm4` [KRSS]. We clock at 24 MHz for benchmarking entire schemes. For individual functions, we clock at 24 MHz for consistent setup in the literature. Furthermore, we include cycle counts at 168 MHz to demonstrate the impact of code size. Although our code size optimization is specific to our board, our programs are designed with flexible loop-unrolling by simply adjusting the numbers.

5.2 Performance of Polynomial Multiplications

We compare our implementations to existing works. Table 7 compares our work to [ACC$^+$21, CHK$^+$21, Che21, IKPC22]. We also show the detailed numbers of our implementations in Table 8.

Table 7: Benchmarks of polynomial multiplications. Numbers are rounded to the nearest thousands. We benchmark our implementations at both 24 MHz and 168 MHz. The first number is benchmarked at 24 MHz and the second one is benchmarked at 168 MHz. Implementations in the literature are all reported at 24 MHz by the authors.

NTRU				
(n, q)	Convolution	This work	[CHK+21]	[IKPC22]
$(677, 2048)$	Size-677	−/−	−/−	144k/−
	Size-1440	140k/143k	−/−	−/−
	Size-1536	147k/149k	156k/−	−/−
$(701, 8192)$	Size-701	−/−	−/−	144k/−
	Size-1440	141k/143k	−/−	−/−
	Size-1536	148k/150k	156k/−	−/−
$(821, 4096)$	Size-821	−/−	−/−	193k/−
	Size-1728	178k/182k	199k/−	−/−

NTRU Prime				
(p, q)	Convolution	This work	[ACC+21]	[Che21][1]
$(653, 4621)$	Size-1320	−/−	−/−	120k/−
	Size-1440	142k/147k	−/−	−/−
$(761, 4591)$	Size-1530	−/−	152k/−	142k/−
	Size-1536	151k/153k	159k/−	−/−
	Size-1620	−/−	185k/−	−/−
$(857, 5167)$	Size-1722	−/−	−/−	203k/−
	Size-1728	182k/186k	−/−	−/−

[1] Method 2, NTT-based polynomial multiplications without changing coefficient rings.

5.2.1 Polynomial Multiplications in NTRU

For `ntruhps2048677`, our size-1440 convolution outperforms the size-677 convolution from [IKPC22] by 2.8% and the size-1536 convolution from [CHK+21] by 10.3%. Our size-1536 convolution is slower than [IKPC22] but still faster than [CHK+21]. For `ntruhrss701`, we have similar results. For `ntruhps4096821`, our size-1728 convolution outperforms the size-821 convolution from [IKPC22] by 7.8% and the size-1728 convolution from [CHK+21] by 10.6%. Notice that when benchmarking at full speed (168 MHz), we only pay 1.3%–3.5% additional cycles. After carefully examining the implementations by [IKPC22], their implementations are fully unrolled. Although we believe that their implementations can be made much more compact with some care, our implementations at 168 MHz are already faster than their implementations at 24 MHz. This implies for practical deployment, users have a much wider range of frequency to fit the implementations into their use without sacrificing performance. Additionally, no algebraic properties are exploited in our implementations, while [IKPC22] only applies to (weighted) convolutions. Therefore, with little modifications, our implementations support polynomial multiplications in NTRU Prime as shown in the next section.

5.2.2 Polynomial Multiplications in NTRU Prime

We compare our polynomial multiplications to [ACC+21] and [Che21, Method 2]. Notice that the implementations by [Che21] made use of the special structures of the coefficient rings. For `ntrulpr857/sntrup857`, our size-1728 convolution outperforms the size-1722 convolution from [Che21] by 10.3%. For `ntrulpr761/sntrup761`, our size-1536 convolution outperforms all the implementations by [ACC+21], but it is slower than the size-1530 convolution by [Che21]. For `ntrulpr653/sntrup653`, our size-1440 convolution is slower than the size-1320 convolution by [Che21].

Finally, we emphasize again that most of the effort is spent on exploring suitable sizes of convolutions supporting a wide range of polynomial multiplications. Since we make no assumptions on the algebraic structure, with few modifications, our implementations support NTRU and NTRU Prime while balancing between performance and code size, which is much more practical for deployment and extension.

5.2.3 Detailed Numbers for Polynomial Multiplications

We go through the detailed numbers of our polynomial multiplications. Each of our convolutions supports at least one parameter set of NTRU and one parameter set of NTRU Prime. Our size-1440 convolution supports `ntruhps2048677`, `ntruhrss701`, and `ntrulpr653/sntrup653`. The implementations are exactly the same except for the `final_-maps`, which are tuned with the reduction to target polynomial rings. Our size-1536 convolution supports `ntruhps2048677`, `ntruhrss701`, and `ntrulpr761/sntrup761` with only differences in `final_maps`. Our size-1728 convolution supports `ntruhps4096821`, and `ntrulpr857/sntrup857` with only differences in `final_maps`. Table 8 is the summary.

Table 8: Performance of `polymul`s in NTRU and NTRU Prime on Cortex-M4.

NTRU							
(n, q)	Size	`polymul`	`NTT`	`NTT_small`	`basemul`	`iNTT`	`final_map`
$(677, 2048)$	1440	140 444	34 102	33 241	27 690	36 756	8 835
		143 016	34 963	34 093	27 825	37 214	9 208
$(677, 2048)$	1536	147 126	37 485	36 573	23 322	41 437	8 489
		149 174	38 076	37 139	23 506	42 001	8 717
$(701, 8192)$	1440	140 577	34 102	33 241	27 690	36 756	8 968
		143 239	34 957	34 087	27 819	37 208	9 431
$(701, 8192)$	1536	147 670	37 485	36 573	23 322	41 437	9 033
		149 771	38 076	37 139	23 506	42 001	9 314
$(821, 4096)$	1728	181 534	48 629	47 627	21 848	53 098	10 512
		186 197	49 480	48 507	22 349	55 569	10 564
NTRU Prime							
(p, q)	Size	`polymul`	`NTT`	`NTT_small`	`basemul`	`iNTT`	`final_map`
$(653, 4621)$	1440	142 244	34 104	33 244	27 690	36 756	10 629
		146 665	34 992	34 095	27 813	37 214	12 823
$(761, 4591)$	1536	151 374	37 487	36 573	23 322	41 435	12 739
		153 299	38 069	37 138	23 510	42 001	12 861
$(857, 5167)$	1728	184 714	48 629	47 623	21 848	53 099	13 695
		189 523	49 483	48 499	22 336	55 720	13 743

5.3 Schemes

Table 9 summarizes our work about NTRU on Cortex-M4 and Table 10 is about NTRU Prime on Cortex-M4. We also optimize the `crypto_sort`.

5.3.1 NTRU Performance

Aside from our NTT-based multiplications, we collect various optimizations applicable to NTRU, including the fast constant-time GCD for NTRU implemented by [Li21], the `crypto_sort` in NTRU Prime, and the TMVP for NTRU by [IKPC22]. The overall performance is summarized in Table 9.

We first compare the encapsulations. For `ntruhps2048677` and `ntruhps4096821`, we outperform [IKPC22] by 35.7% − 36.3%. The majority of the improvement comes from the more optimized `crypto_sort`. A fair comparison is `ntruhrss701`, where the only difference is one big by small polynomial multiplication. We outperform [IKPC22] by 2.2% for `ntruhrss701`.

For decapsulations, the differences between our implementations and the TMVP by [IKPC22] are one big by small polynomial multiplication and one polynomial multiplication in $\mathbb{Z}_3$. We outperform [IKPC22] by 0.7% − 1.5%.

Our NTT-based multiplications have a limited impact on key generation. Key generations are dominated by computing inverses in $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$. The inverses are first computed in $\mathbb{Z}_2[x]/\langle x^n - 1\rangle$ and then lifted to $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$. [Li21] implemented the inverses in $\mathbb{Z}_2[x]/\langle x^n - 1\rangle$ with the fast constant-time GCD by [BY19]. [IKPC22] applied their improved polynomial multiplications to lifting $\mathbb{Z}_2[x]/\langle x^n - 1\rangle$ to $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$. We simply integrate their work, and plug in the more improved `crypto_sort`. The majority of the improvement comes from [Li21]. For the rest, the improvement mainly comes from the more improved `crypto_sort` and [IKPC22] for lifting to $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$.

Table 9: NTRU cycle counts of the fastest approaches in this work.

	`ntruhps2048677`			`ntruhrss701`			`ntruhps4096821`		
	K	**E**	**D**	**K**	**E**	**D**	**K**	**E**	**D**
[CHK+21]	143 725k	821k	818k	153 403k	377k	871k	207 495k	1 027k	1 030k
[IKPC22]	142 378k	816k	729k	153 479k	369k	787k	212 377k	1 026k	914k
[Li21][1]	4 625k	820k	812k	4 233k	376k	868k	6 116k	1 027k	1 031k
This work	3 912k	525k	718k	3 822k	361k	778k	5 217k	654k	908k

[1] Integrated into `pqm4` in commit `2691b4915b76db8b765ba89e4e09adc6b999763f`.

5.3.2 NTRU Prime Performance

We apply our NTT-based multiplications to all the big by small polynomial multiplications in NTRU Prime. We replace AES with secret-dependent input by the fixslicing AES in [AP21]. This increases the overall performance numbers of NTRU LPRime. Furthermore, we improve the `crypto_sort`. Table 10 summarizes the overall performance numbers.

We first compare NTRU LPRime. [ACC+21] reported performance numbers with an AES implementation with secret-dependent table lookup. At the same time, [AP21] proposed fixslicing AES. The timings increase drastically by changing to fixslicing AES for secret-dependent operations. Therefore, our `ntrulpr761` is slower than the fastest approach by [ACC+21] even though our polynomial multiplication is comparable to [ACC+21]. A fair comparison is comparing against [Che21]. For `ntrulpr653` and `ntrulpr761`, since our polynomial multiplications are slower than [Che21], the overall performance is expected to be slower. However, during the encapsulation, [Che21] computed two multiplications by a polynomial with two polynomial multiplications. We instead cache the NTT of one of the operands and reuse it later. This explains why our encapsulations are faster while key generations and decapsulations are slower than [Che21].

Next, we compare Streamlined NTRU Prime. In Streamlined NTRU Prime, we need `crypto_sort` in key generations and encapsulations. Since we optimize the `crypto_sort`, our key generations are faster than [Che21] even though our polynomial multiplications are slower than [Che21]. Our decapsulations are slower than [Che21] since the only difference is two generic-by-ternary polynomial multiplications.

Finally, we present the performance numbers of `ntrulpr857` and `sntrup857`.

Table 10: NTRU Prime cycle counts for the fastest approaches in this work.

	ntrulpr653			ntrulpr761			ntrulpr857		
	K	**E**	**D**	**K**	**E**	**D**	**K**	**E**	**D**
[ACC$^+$21][2]	-	-	-	731k	1 102k	1 200k	-	-	-
[Che21][3]	678k	1 158k	1 233k	727k	1 312k	1 394k	-	-	-
This work	669k	1 131k	1 231k	710k	1 266k	1 365k	886k	1 465k	1 596k
	sntrup653			sntrup761			sntrup857		
	K	**E**	**D**	**K**	**E**	**D**	**K**	**E**	**D**
[ACC$^+$21][2]	-	-	-	10 778k	694k	572k	-	-	-
[Che21][3]	6 715k	632k	487k	7 951k	684k	538k	-	-	-
This work	6 623k	621k	527k	7 937k	666k	563k	10 192k	812k	685k

[2] uses secret-dependent table lookup AES, see `https://github.com/mupq/pqm4/pull/173` for details.
[3] [Che21, Method 2] is integrated into `pqm4` in commit `844e7cafdb5df8416abef3c03b49edb810b7e396`.

Acknowledgments

This research is indebted to Taiwan Ministry of Sci. and Tech. grants 110-2221-E-001-008-MY3 and 111-2119-M-001-004, Academia Sinica Investigator Award AS-IA-109-M01, and the Cybersecurity Center of Excellence Project at Nat'l Applied Research Labs, Taiwan.

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.

[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`.

[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`.

[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`.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Dann Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. 2022. To appear at ACNS 2022, available as `https://eprint.iacr.org/2022/112`.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like Ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425, 2021. `https://tches.iacr.org/index.php/TCHES/article/view/8739`.

[ARM10] ARM. *Cortex-M4 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0439/b/.

[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/.

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

[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.

[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.

[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.

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

[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.

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

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

[Che21] Yun-Li Cheng. Number Theoretic Transform for Polynomial Multiplication in Lattice-based Cryptography on ARM Processors. Master's thesis, 2021. https://github.com/dean3154/ntrup_m4.

[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.

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

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8683.

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

[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.

[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.

[Goo71] I. J. Good. The relationship between two fast fourier transforms. *IEEE Transactions on Computers*, 100(3):310–317, 1971.

[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.

[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.

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time O (n log n). *Annals of Mathematics*, 193(2):563–617, 2021.

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

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

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

[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.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Li21] Ching-Lin Li. Implementation of Polynomial Modular Inversion in Lattice-based cryptography on ARM. Master's thesis, 2021. https://github.com/trista5658321/polyinv-m4.

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

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

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

[Pol71] John M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of computation*, 25(114):365–374, 1971.

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

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

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

[Win78] Shmuel Winograd. On Computing the Discrete Fourier Transform. *Mathematics of computation*, 32(141):175–199, 1978.