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

Erdem Alkim¹, Vincent Hwang²,³ and Bo-Yin Yang³

¹ Dokuz Eylul University, Izmir, Turkey
erdemalkim@gmail.com
² National Taiwan University, Taipei, Taiwan
³ Academia Sinica, Taipei, Taiwan
vincentvh7@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 \( \mathbb{Z}[x]/\langle \Phi_n \rangle \), \( \mathbb{Z}[x]/\langle \Phi_n \rangle \), and \( \mathbb{Z}[x]/\langle x^n - 1 \rangle \). Here \( q \) is a power of 2, \( n \) is prime, and \( \Phi_n \) 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 \( \mathbb{Z}[x]/\langle x^p - x - 1 \rangle \) and \( \mathbb{Z}[x]/\langle x^q - x - 1 \rangle \) for various \( p \) and \( q \), primes such that \( \mathbb{Z}[x]/\langle x^p - x - 1 \rangle \) is a finite field. We focus on polynomial multiplications in the rings \( \mathbb{Z}[x]/\langle x^n - 1 \rangle \) of NTRU and \( \mathbb{Z}[x]/\langle x^p - x - 1 \rangle \) of NTRU Prime each with one operand ternary \( \{\{-1, 0, 1\}\} \) (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.

<table>
<thead>
<tr>
<th>NTRU-HPS</th>
<th>NTRU-HRSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 509</td>
<td>n = 677</td>
</tr>
<tr>
<td>q = 2048</td>
<td>q = 2048</td>
</tr>
<tr>
<td>n = 821</td>
<td>q = 4096</td>
</tr>
<tr>
<td>n = 1229</td>
<td>q = 4096</td>
</tr>
<tr>
<td>n = 701</td>
<td>q = 8192</td>
</tr>
<tr>
<td>n = 1373</td>
<td>q = 16384</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>NTRU LPrime</th>
<th>Streamlined NTRU Prime</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 653</td>
<td>p = 4621</td>
</tr>
<tr>
<td>n = 761</td>
<td>q = 4591</td>
</tr>
<tr>
<td>n = 857</td>
<td>q = 5167</td>
</tr>
<tr>
<td>n = 953</td>
<td>q = 6343</td>
</tr>
<tr>
<td>n = 1013</td>
<td>q = 7177</td>
</tr>
<tr>
<td>n = 1277</td>
<td>q = 7879</td>
</tr>
</tbody>
</table>

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 $\text{mul}$, $\text{mla}$, and $\text{mls}$, long multiplications $\{\text{mul}, \text{mla}\}$, and signed most-significant-word multiplications $\text{smla}\{\text{mul}, \text{mla}, \text{mls}\}$, and signed most-significant-word multiplications $\text{smla}\{\text{mul}, \text{mla}, \text{mls}\}$, $\text{smla}$. An exception is that a string of $l$ single-load instructions ($\text{ldr}\{, \text{h}, \text{sh}, \text{b}, \text{sb}\}$) takes $l + 1$ cycles. For more details on load/store timings, see [ARM10].

Barrel shifters: Shifts and rotates ($\text{asr}$, $\text{lsl}$, $\text{lsr}$, and $\text{ror}$), come at no extra cost when used as the “flexible second operand” of a standard data-processing instruction.

We first describe the multiplication instructions. $\text{mul}$ multiplies two 32-bit values and places the lower 32-bit result to the first (destination) register. $\text{mla}$ accumulates the 32-bit result to an accumulator, and $\text{mls}$ subtract the 32-bit result from the accumulator. The accumulators are the last arguments named to $\text{mla}$ and $\text{mls}$. $\text{smla}$ 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. $\text{smla}$ accumulates the 64-bit result to the destination registers. $\text{umull}$ and $\text{umlal}$ are their unsigned counterparts. $\text{smul}$ 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 \( R = 2^{32} \) and let \( \mod \frac{q}{R} \) be the signed modular reduction. 32-bit Barrett reduction maps a value \( a \) to \( a - \left\lfloor \frac{a}{q} \right\rfloor q \) as an approximation of \( a - \left\lfloor \frac{a}{q} \right\rfloor q = a \mod \frac{q}{R} \). Algorithm 1 is an illustration. Montgomery multiplication, denoted as \( \text{montgomery}(a, b) \), computes
\[
\frac{ab + q \cdot (ab \cdot (-q^{-1} \mod \frac{q}{R}) \mod \frac{q}{R})}{R} \equiv abr^{-1} \mod \frac{q}{R}
\]
from \( (a, b) \). To see why it is a reduction, we observe that in terms of absolute values, \( |\text{montgomery}(a, b)| \leq |ab| + \frac{q}{2} \). If \( |a| \leq \frac{q}{2} \) and \( |b| \leq \frac{q}{2} \), we have \( |\text{montgomery}(a, b)| \leq \frac{q}{2} + \frac{q}{2} \). Concretely, we denote \( \text{mul} \_{\text{des}} \_{32}(1, b, a, b, t) \) as the 32-bit Montgomery multiplication with input registers \( a, b \) and output register \( h \) as shown in Algorithm 2.

### Algorithm 1 Barrett reduction

**Input:** \( a = a \)  
**Output:** \( a = a - \left\lfloor \frac{a}{q} \right\rfloor q \)

1. `smlul r t, a, \frac{\frac{q}{q}}{q}`  
2. `mlt a t, q, a`

### Algorithm 2 `mul_des_32`  

**Input:** \( (a, b) = (a, b) \)  
**Output:** \( h \equiv abR^{-1} \mod \frac{q}{R} \)

1. `smlal l, h, a, b`
2. `mul t, l, -q^{-1} \mod 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 \( R[x]/\langle f_0(x^0) \rangle \) and \( R[x]/\langle f_1(x^1) \rangle \) be two polynomial rings. Their tensor product \( R[x]/\langle f_0(x^0) \rangle \otimes R[x]/\langle f_1(x^1) \rangle \) is also an associative algebra over \( R \). We also write \( R[x]/\langle f_0(x^0) \rangle \otimes R[x]/\langle f_1(x^1) \rangle \) as \( R[x]/\langle f_0(x^0), f_1(x^1) \rangle \). We refer to [Jac12, Sec. 3.9] and [Bon89, Chap. III, Sec. 4.1] for 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 \( \zeta^n \) is invertible in \( R \), \( n \) is coprime to the characteristic of \( R \), and there is a principal \( m \)-th root of unity \( \omega_m \), we can define the size-\( m \) NTT, \( \text{NTT} \ R[x]/\langle x^n - \zeta^m \rangle \omega_m \ [Pol71, \text{Fur99, HvH21]}. \) If \( R = \mathbb{Z}/q \) with the prime factorization \( q = p_0^{a_0} \cdots p_{d-1}^{a_{d-1}} \), it can be shown that the above conditions are equivalent to the existence of \( n \) fulfilling \( n0(q) \equiv \gcd(p_0 - 1, \ldots, p_d - 1) \) [AB74]. \( \text{NTT} \ R[x]/\langle x^n - \zeta^m \rangle \omega_m \) transforms the ring \( R[x]/\langle x^n - \zeta^m \rangle \) into the product ring \( \prod_{i=0}^{m-1} \ R[x]/\langle x^{\frac{n}{m}} - \zeta^i \omega_m \rangle \) by first re-writing a polynomial \( a(x) = \sum_{i=0}^{n-1} a_i x^i \) as \( a'(x') = \sum_{i=0}^{\frac{n-1}{m}} \left( \sum_{j=0}^{\frac{n-1}{m}} a_{i+jm} x^j \right) x'^i \) and sending \( a'(x') \) to the \( m \)-tuple \( (a'(\zeta), \ldots, a'(\zeta^{n-1})) \). If
m = n, we call it complete NTT, and if m ≠ n, we call it incomplete NTT. If ℂ m = 1, we call it cyclic NTT and write it as NTT R[x, n, ω 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 φ(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 ω m := ω n 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 = m₀m₁, Cooley–Tukey FFT computes the isomorphism \( R[x]/(x^n - ℂ^m) \cong \prod_{i=0}^{m-1} R[x]/(x^m - ℂ^m) \) via applying size-m₀ NTTs with \( ω_{m₀} := ω_{m₀}^0 \) and size-m₁ NTTs with \( ω_{m₁} := ω_{m₀}^m \) as follows:

\[
R[x]/(x^n - ℂ^m) \cong \prod_{i=0}^{m₀ - 1} R[x]/(x^{m₁} - ℂ^{m₀}) \cong \prod_{i=0}^{m₁ - 1} R[x]/(x - ℂ^{m₀})
\]

Since \( ω_{m₀}^0 ω_{m₁}^{m₀} = ω_{m₀}^{m₁} \), the result is only different from NTT \( R[x]/(x^n - ℂ^m) \) by a permutation. If m = rᵏ 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 \( a(x) \) to \( \{a(\zeta^m \omega_{m₀}^0)x\}_{y₀} \) by evaluating \( x^{m₀} \) at several ℂ m₀. Such evaluations are also called Cooley–Tukey butterflies. Figure 1a is an illustration for \( (a')_0 \) (up) and \( (a')_1 \) (down) where \( m₀ = 2 \) (and \( w₂ = -1 \)), and Figure 2a is an illustration for \( (a')_0 \) (up), \( (a')_1 \) (middle), and \( (a')_2 \) (down) where \( m₀ = 3 \).

![Cooley–Tukey butterflies](image1.png)

![Gentleman–Sande butterflies](image2.png)

Figure 1: Radix-2 butterflies.

Figure 2: Radix-3 butterflies.

Gentleman–Sande FFT transforms \( R[x]/(x^n - ℂ^m) \) to \( R[x, y]/(x^{m₀} - ℂ y, y^{m₁} - 1) \) by introducing the equivalence \( x^{m₀} \sim ℂ y \) and regarding \( R[x]/(x^{m₀} - ℂ y) \) as the coefficient ring. For ℂ m = 1, the computation \( R[x]/(x^n - 1) \cong \prod_{i=0}^{m₀ - 1} R[x]/(x^{m₀}) \cong \prod_{i=0}^{m₁ - 1} R[x]/(y^{m₁}) \) is called Gentleman–Sande butterfly. Moreover, the introduction of \( x^{m₀} ∼ ℂ y \) requires multiplications by powers of \( ω_{m₀}^m \). It is usually written as the
map $x \mapsto y$ 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 $\text{NTT}_{R[x]:q_0 q_1; \omega_{q_0 q_1}}$ into $\text{NTT}_{R[x^{(0)}]:q_0; \omega_{q_0}} \otimes \text{NTT}_{R[x^{(1)}]:q_1; \omega_{q_1}}$ where $\omega_{q_0} := \omega_{q_0 q_1}^0, \omega_{q_1} := \omega_{q_0 q_1}^1$, and $e_0$ and $e_1$ are orthogonal idempotent elements ($e_0 e_1 = \delta_{e_0, e_1}$ where $\delta$ is the Kronecker delta) realizing $a \equiv e_0 (a \mod q_0) + e_1 (a \mod q_1)$ (mod $q_0 q_1$) [Goo58, Section 12][Tho63]. Since cyclic NTTs require fewer multiplications than acyclic ones, this is more favorable than the mixed-radix size-$q_0 q_1$ 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 $n = v q_0 q_1$ 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 $R[x]/(x^v - 1)$ into the tensor product $\hat{R}[x^{(0)}]/(x^{(0)} - 1) \otimes \hat{R}[x^{(1)}]/(x^{(1)} - 1)$ where $\hat{R} := R[x]/(x^v - x^{(0)} x^{(1)})$. [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 $\text{NTT}_{R[x]:q_0 q_1; \omega_{q_0 q_1}}$ into the tensor product $\text{NTT}_{R[x^{(0)}]:q_0; \omega_{q_0}} \otimes \text{NTT}_{R[x^{(1)}]:q_1; \omega_{q_1}}$. 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 $l = (q_0 + q_1)^{-1} \mod q_0 q_1$. Since $q_0 + q_1$ is coprime to $q_0 q_1$, $(q_0 + q_1)^{-1}$ is a generator of $(\mathbb{Z}_{q_0 q_1}, +, 0)$, and the map $1 \mapsto (q_0 + q_1)^{-1}$ is an automorphism of $(\mathbb{Z}_{q_0 q_1}, +, 0)$. This implies an isomorphic way for constructing the group algebra from $\mathbb{Z}_{q_0 q_1}$ and defining NTT $\text{NTT}_{R[x]:q_0 q_1; \omega_{q_0 q_1}}$. Since there are $\phi(q_0 q_1)$ generators of $(\mathbb{Z}_{q_0 q_1}, +, 0)$, there are $\phi(q_0 q_1)$ 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 $\phi(q_0 q_1) = 2$. 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 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| $i$ | 0 | 1 | 0 | 1 | 0 | 1 |
| $i_0$ | 0 | 1 | 0 | 1 | 0 | 1 |
| $i_1$ | 0 | 1 | 2 | 0 | 1 | 2 |

### 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 $\bigotimes_{i=1}^{d-1} \text{NTT}_{R[x^{(i)}]/(x^{(i)} \cdot \omega_{m_i})}$, 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.

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 [HMC77] with a 2-dimensional NTT with dimension $2 \times 2$. Let $\text{NTT}^{(0)} := \text{NTT}_{R[x^{(0)}]/\langle (x^{(0)})^2-\zeta_0^2 \rangle;: -1}$, $\text{NTT}^{(1)} := \text{NTT}_{R[x^{(1)}]/\langle (x^{(1)})^2-\zeta_1^2 \rangle;: -1}$, $\text{id}$ be the identity map, and $\text{addsub} := (a, b) \mapsto (a + b, a - b)$. Applying 1-dimension NTTs is to compute with $(\text{NTT}^{(1)} \otimes \text{id}) \circ (\text{id} \otimes \text{NTT}^{(0)})$ as follows.

\[
\begin{pmatrix}
0_{0,0} & 0_{0,1} \\
0_{1,0} & 0_{1,1}
\end{pmatrix}
\xmapsto{\text{id} \otimes \text{NTT}^{(0)}}
\begin{pmatrix}
0_{0,0} + \zeta_0 a_{0,1} & 0_{0,0} - \zeta_0 a_{0,1} \\
0_{1,0} + \zeta_0 a_{1,1} & 0_{1,0} - \zeta_0 a_{1,1}
\end{pmatrix}
\xmapsto{\text{NTT}^{(1)} \otimes \text{id}}
\begin{pmatrix}
0_{0,0} + \zeta_0 a_{0,1} + \zeta_0 \zeta_1 a_{0,1} + \zeta_0 \zeta_1 a_{1,1} & 0_{0,0} - \zeta_0 a_{0,1} + \zeta_1 a_{1,0} - \zeta_0 \zeta_1 a_{1,1} \\
0_{0,0} + \zeta_0 a_{0,1} - \zeta_1 a_{1,0} - \zeta_0 \zeta_1 a_{1,1} & 0_{0,0} - \zeta_0 a_{0,1} - \zeta_1 a_{1,0} + \zeta_0 \zeta_1 a_{1,1}
\end{pmatrix}.
\]

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

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

where $(x^{(1)} \mapsto \zeta_1) \otimes (x^{(0)} \mapsto \zeta_0)$ can be written as $(x^{(0)})^{i_0} (x^{(1)})^{i_1} \mapsto \zeta_0^{i_0} \zeta_1^{i_1}$. Obviously, $(x^{(0)})^{i_0} (x^{(1)})^{i_1} \mapsto \zeta_0^{i_0} \zeta_1^{i_1}$ requires 3 multiplications whereas $(x^{(1)} \mapsto \zeta_1) \otimes \text{id} \circ (\text{id} \otimes (x^{(0)} \mapsto \zeta_0))$ 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, Algorithm 3 we compare radix-2 and radix-3 CT butterflies, but our observation can be generalized to arbitrary radices. In this section, we assume $\psi$ is an invertible element.

3.1.1 Radix-3 Butterflies

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

**Input:** $(c_0, c_1, c_2) = (c_0, c_1, c_2)$

**Output:**

\[ c' = c := c_0 + \psi c_1 + \psi^2 c_2 \]
\[ c'_1 = c_1 := c_0 + \psi \omega_3 c_1 + \psi^2 \omega_3^2 c_2 \]
\[ c'_2 = c_2 := c_0 + \psi \omega_3^2 c_1 + \psi^2 \omega_3^3 c_2 \]

1. `smul t1, c0', c1, ψ`
2. `smul t1, c0', c2, ψ^2`
3. `mul t0, t1, q`
4. `smul t1, c0', t0, q`
5. `smul t1, c1', c1, ψω^3_`
6. `mul t0, t1, q'
7. `smul t1, c1', c2, ψ^2ω^3_`
8. `smul t1, c2', c1, ψ^2ω^3_`
9. `mul t0, t1, q'
10. `smul t1, c2', c2, ψ^2ω^3_`
11. `mul t0, t1, q'
12. `smul t1, c2', t0, q`
13. `add w c2', c2', c0`
14. `add c1', c0`
15. `add c0', c0`
16. `add c0', c0`

Recall that radix-2 CT butterfly maps $(c_0, c_1, \psi)$ to $(c_0 + \psi c_1, c_0 - \psi c_1)$ and radix-3 CT butterfly maps $(c_0, c_1, c_2, \psi)$ to $(\hat{c}_0, \hat{c}_1, \hat{c}_2) = (c_0 + \psi c_1 + \psi^2 c_2, c_0 - \psi c_1 + \psi^2 c_2, c_0 + \psi \omega_3 c_1 + \psi^2 \omega_3^2 c_2)$. 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 $(c_0, c_1, \psi)$ 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 $c_0 - ((\psi c_1 + \psi^2 c_2) + (\psi \omega_3 c_1 + \psi^2 \omega_3^2 c_2)) = \hat{c}_2$ 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:** \((c_0, c_1, c_2) = (c_0, c_1, c_2)\)

**Output:** \((c_0, c_1, c_2) = (c_0, c_1, c_2)\)

1. \texttt{smull} \(t_1, t_0, c_1, \psi\) \\
2. \texttt{smalal} \(t_1, t_0, c_2, \psi^2\) \\
3. \texttt{mul} \(t_2, t_1, \psi^q\) \\
4. \texttt{smalal} \(t_1, t_0, t_2, q\) \\
5. \texttt{smull} \(t_1, c_1, c_1, \psi\omega_3\) \\
6. \texttt{small} \(t_1, c_1, c_2, \psi^2\omega_3^2\) \\
7. \texttt{mul} \(t_2, t_1, \psi^q\) \\
8. \texttt{smalal} \(t_1, c_1, t_2, q\) \\
9. \texttt{add} \(c_2, c_1, t_0\) \(\triangleright t_0 = \psi c_1 + \psi^2 c_2\) \\
10. \texttt{resb} \(c_2, c_2, c_0\) \(\triangleright c_2 = \hat{c}_2\) \\
11. \texttt{add} \(c_1, c_0\) \(\triangleright c_1 = \hat{c}_1\) \\
12. \texttt{add} \(c_0, t_0\) \(\triangleright c_0 = \hat{c}_0\)

### 3.1.2 Generalization to Arbitrary Radices

We now generalize the idea to other radices. For computing \((c_0, c_1, c_2)\) from \(c(x) = \sum_{k=0}^{r-1} c_k x^k\), we pick a \(j\) indicating which multiplications to be replaced. If \(\psi = 1\), we pick \(j > 0\). Since \(rc_0 = \sum_{i=0}^{r-1} (c(\psi^i))\) by the definition of \(\omega_r\), we have

\[
c(\psi^i) = rc_0 - \sum_{i=0,i \neq j}^{r-1} c(\psi^i) = c_0 - \sum_{i=0,i \neq j}^{r-1} (c(\psi^i) - c_0)
\]

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

One should be aware that for a cyclic size-\(r\) NTT, there are many approaches other than the naïve 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 \(\mathbb{Z}_{p^h}\) 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 five are known: 3, 5, 17, 257, 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 naïve approach with 32-bit arithmetic. We first compute \(c(1) = c_0 + c_1 + c_2 + c_3 + c_4\) with four additions. For \(i = 1, \ldots, 4\), we compute \(c(\omega_5^i) - c_0 = c_1 \omega_5^i + c_2 \omega_5^{2i} + c_3 \omega_5^{3i} + c_4 \omega_5^{4i}\) with one \texttt{smull}, three \texttt{smalals}, and one 32-bit Montgomery reduction. Finally, we add \(c_0\) and derive \(c(\omega_5^i)\). In total, we need \(4 + (4 + 2 + 1)\cdot 4 = 32\) cycles.
We pick $j = 4$ for our improvement. While computing $c(1)$, we first derive $c(1) - c_0 = c_1 + c_2 + c_3 + c_4$ and proceed with the computation for $c(\omega_5) - c_0$, $c(\omega_5^2) - c_0$, and $c(\omega_5^3) - c_0$. Then, we compute $c(\omega_5^4) = c_0 - \sum_{i=0}^{3} (c(\omega_5^i) - c_0)$ 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

\[
\pi \circ \begin{pmatrix}
    c(\omega_5) - c_0 \\
    c(\omega_5^2) - c_0 \\
    c(\omega_5^3) - c_0 \\
    c(\omega_5^4) - c_0 \\
\end{pmatrix} = \begin{pmatrix}
    c_2(\omega_5^{-1})^{2^0} + c_2(\omega_5^{-1})^{2^3} + c_2(\omega_5^{-1})^{2^2} + c_2(\omega_5^{-1})^{2^1} \\
    c_2(\omega_5^{-1})^{2^2} + c_2(\omega_5^{-1})^{2^3} + c_2(\omega_5^{-1})^{2^0} + c_2(\omega_5^{-1})^{2^1} \\
    c_2(\omega_5^{-1})^{2^1} + c_2(\omega_5^{-1})^{2^3} + c_2(\omega_5^{-1})^{2^2} + c_2(\omega_5^{-1})^{2^0} \\
    c_2(\omega_5^{-1})^{2^0} + c_2(\omega_5^{-1})^{2^2} + c_2(\omega_5^{-1})^{2^3} + c_2(\omega_5^{-1})^{2^1}
\end{pmatrix}
\]

where $\pi = (34)$ is a permutation of four elements. We now compute the cyclic convolution of $(c_{2^0}, c_{2^1}, c_{2^2}, c_{2^3})$ and $(\omega_5^{-1})^{2^0}, (\omega_5^{-1})^{2^1}, (\omega_5^{-1})^{2^2}, (\omega_5^{-1})^{2^3}$ 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 $c(1) = c_0 + c_1 + c_2 + c_3 + c_4$. In summary, we need $24 + 4 + 2 = 30$ cycles for the Rader approach.

All in all, in the case of $r = 5$, our improved naïve butterfly outperforms the original naïve 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-$2^{2k_0}3^{k_1}$ 1-dimensional FFT into a 2-dimensional FFT of dimensions $2^{2k_0} \times 3^{k_1}$ with $3^{k_1} < 2^{2k_0}$. Furthermore, we also assume that the upper half of the input polynomial are all zeros, and compute with dedicated radix-(2, 3) butterflies.

Since $2^{2k_0}$ and $3^{k_1}$ are coprime, there are $3^{k_1-1}$ different loops, where each loop calls $2^{2k_0-1}$ 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 $2^{2k_0-1}$ calls into $\left\lceil \frac{2^{2k_0-1}}{3^{k_1}} \right\rceil$ sequences of dedicated radix-(2, 3) butterflies. The first $\left\lfloor \frac{2^{2k_0-1}}{3^{k_1}} \right\rfloor$ sequences share the same structure with $3^{k_1}$ dedicated radix-(2, 3) butterflies and the last sequence consists of $2^{2k_0-1} \mod 3^{k_1}$ 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 $((2^{2k_0-1} \mod 3^{k_1}) - 1)$-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 $3 \times 3$ 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 $ntruhps2048677/ntruhras701$ [CHK+21], using Good–Thomas FFT, suggested transforming the size-1728 convolution into a 2-dimensional
convolution with dimensions $27 \times 64$. 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 $3^2 \cdot 3^2 - 1 = 243$ 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-$q_0\tilde{q} v$ convolutions where $q_0$ is a power of 2, $q_1 < \frac{q_0}{2}$ is a power of 3, and $\tilde{q} \leq 3$. 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 $\tilde{R} = \frac{R[x]}{(x-2^3x)}$. We compute the result of $\text{NTT}_{\tilde{R}[x^v]:q_0\tilde{q} v,\omega_{q_0} q_1}$ as follows. We first convert $\text{NTT}_{\tilde{R}[x^v]:q_0\tilde{q} v,\omega_{q_0} q_1}$ into $\text{NTT}_{\tilde{R}[x^{v_0}]: q_0\omega_{q_1} q_0, \text{NTT}_{\tilde{R}[x^{v_1}]: q_1\omega_{q_1}}$ by introducing the equivalence $x' \sim x^{v_0} x^{v_1}$. This can be realized by permuting the coefficients. For computing $\text{NTT}_{\tilde{R}[x^{v_0}]: q_0\omega_{q_1} q_0} \otimes \text{NTT}_{\tilde{R}[x^{v_1}]: q_1\omega_{q_1}}$, 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 dedicated radix-(2,3) butterflies and the permutation $x' \sim x^{v_0} x^{v_1}$ 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 $\frac{q_0 q_1}{2}$ dedicated radix-(2,3) butterflies. For Cortex-M4, $q_1 = 3, 9$ are reasonable numbers since there are only $\frac{3^2}{2} = 3$ or $\frac{9^2}{2} = 27$ 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 $i = (e_0 (i \mod 2) + e_1 (i \mod 3)) \mod 6$. 

Given coefficients \((c_0, \ldots, c_5)\), we define \(c_{i_0,i_1}\) as \(c_{e_{i_0}i_0 + e_{i_1}i_1}\). 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 \(\sum_{i_0=0}^{2} \sum_{i_1=0}^{1} c_{i_0,i_1} (x^{(0)})^{i_0} (x^{(1)})^{i_1}\). 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 \(\left(\left(\left(c_{0,0} + c_{1,1}\right) \bmod 6 \geq 3\right)\right)\) where \(\bmod\) 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 \(\sum x\).

We define the characteristic vector of \(\left(\left(\left(c_{0,0} + c_{1,1}\right) \bmod 6 \geq 3\right)\right)\). 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, \(\frac{1}{2}\) 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 \(15\frac{1}{2}\) cycles are required on average, then we save \(\frac{9.2 + 2.3 - 15.2}{6} \approx 1.44\) cycles for each entry.

### Algorithm 5 \_6\_ntt\_101.

**Input:**

\[
\begin{pmatrix}
    c_0 & c_2 & c_4 \\
    c_1 & c_3 & c_5
\end{pmatrix} = \begin{pmatrix}
    c_{0,0} & 0 & c_{0,2} \\
    0 & c_{1,1} & 0
\end{pmatrix}
\]

**Output:**

\[
\begin{pmatrix}
    c_0 & c_2 & c_4 \\
    c_1 & c_3 & c_5
\end{pmatrix} = \begin{pmatrix}
    \hat{c}_{0,0} & \hat{c}_{0,1} & \hat{c}_{0,2} \\
    \hat{c}_{1,0} & \hat{c}_{1,1} & \hat{c}_{1,2}
\end{pmatrix} = \left(\text{NTT}_{R[x^{(0)}]; 2 \omega^2} \otimes \text{NTT}_{R[x^{(1)}]; 3 \omega^3}\right) \begin{pmatrix}
    c_{0,0} & 0 & c_{0,2} \\
    0 & c_{1,1} & 0
\end{pmatrix}
\]

1: \text{mMul\_des\_32 t, c5, c4, }\omega_3, c1
2: \text{add c2, c4, c5}
3: \text{sub c2, c0, c2}
4: \text{add c5, c5, c0}
5: \text{add c0, c0, c4}
6: \text{mMul\_des\_32 c1, t, c3, }\omega_3, c4
7: \text{add c4, c3, t}
8: \text{add c0, c0, c3}
9: \text{sub c1, c0, c3, lsl #1}
10: \text{add c2, c2, t}
11: \text{sub c3, c2, t, lsl #1}
12: \text{add c5, c5, c4}
13: \text{sub c4, c5, c4, lsl #1}

\[\text{c5 = } \omega_3 c_{0,2}\]
\[\text{c2 = } -\omega_2^2 c_{0,2}\]
\[\text{c5 = } c_{0,0} + \omega_3 c_{0,2}\]
\[\text{c5 = } c_{0,0} + \omega_2 c_{0,2}\]
\[\text{t = } \omega_3 c_{1,1}\]
\[\text{c4 = } -\omega_3^2 c_{1,1}\]
\[\text{c0 = } \hat{c}_{0,0}\]
\[\text{c1 = } \hat{c}_{1,0}\]
\[\text{c2 = } \hat{c}_{0,1}\]
\[\text{c3 = } \hat{c}_{1,1}\]
\[\text{c5 = } \hat{c}_{1,2}\]
\[\text{c4 = } \hat{c}_{0,2}\]

### 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.

<table>
<thead>
<tr>
<th>Convolution</th>
<th>This work</th>
<th>[ACC+21]</th>
<th>[CHK+21]</th>
</tr>
</thead>
</table>

Table 5: Summary of implementations.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>NTT</th>
<th>base multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NTT[ Państwu] ![5:160ω163ω122] ⊗ NTT[ Państwu] ![5:163ω122]) ⊗ I₃</td>
<td>2 × (radix-(3, 2) + 4-layer radix-2)</td>
<td>3 × 3 schoolbook</td>
</tr>
</tbody>
</table>

**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 $\hat{R} = \frac{Z_q[x]}{(x^3 - x^{(0)}x^{(1)1})}$ as the coefficient ring. Since $\frac{1728}{3} = 9 \cdot 64$, we perform a 2-dimensional FFT defined over the ring $\langle \frac{x^{(0)}}{3}, x^{(1)} \rangle$ with vector-radix FFT. Our vector-radix FFT is built upon the tensor product of the size-9 CT FFT on $\hat{R}[x^{(0)}] / \langle (x^{(0)})^9 - 1 \rangle$ and the size-64 CT FFT on $\hat{R}[x^{(1)}] / \langle (x^{(1)})^{64} - 1 \rangle$. 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 takes the following route from $\frac{Z_q[x]}{(x^3 - x^{(0)}x^{(1)})}$.

$$\begin{align*}
\text{radix-(3, 2)} &\cong \Pi_{i=0}^{2} \Pi_{i=0}^{1} \hat{R}[x^{(0)}, x^{(1)}] / \langle (x^{(0)})^3 - \omega_9^{2i}, (x^{(1)})^3 - \omega_9^{2i} \rangle \times \omega_9^{32\text{rev}_{(3)}(i)} \rangle \\
\text{radix-(3, 2)} &\cong \Pi_{i=0}^{2} \Pi_{i=0}^{1} \left( \hat{R}[x^{(0)}, x^{(1)}] / \langle (x^{(0)})^3 - \omega_9^{2i}, (x^{(1)})^3 - \omega_9^{2i} \rangle \times \omega_9^{32\text{rev}_{(3)}(i)} \rangle \right) \\
&= \Pi_{i=0}^{8} \Pi_{i=0}^{1} 64 \hat{R}[x^{(0)}, x^{(1)}] / \langle x^{(0)} - \omega_9^{\text{rev}_{(3)}(i)}, (x^{(1)})^16 - \omega_9^{16\text{rev}_{(3)}(i)} \rangle \rangle \\
&= \Pi_{i=0}^{8} \Pi_{i=0}^{1} \left( \hat{R}[x^{(0)}, x^{(1)}] / \langle x^{(0)} - \omega_9^{\text{rev}_{(3)}(i)}, (x^{(1)})^16 - \omega_9^{16\text{rev}_{(3)}(i)} \rangle \rangle \right) \\
\text{4-layer radix-2} &\cong \Pi_{i=0}^{8} \Pi_{i=0}^{1} \left( \hat{R}[x^{(0)}, x^{(1)}] / \langle x^{(0)} - \omega_9^{\text{rev}_{(3)}(i)}, (x^{(1)})^16 - \omega_9^{16\text{rev}_{(3)}(i)} \rangle \rangle \right) \\
\text{4-layer radix-2} &\cong \Pi_{i=0}^{8} \Pi_{i=0}^{1} \left( \hat{R}[x^{(0)}, x^{(1)}] / \langle x^{(0)} - \omega_9^{\text{rev}_{(3)}(i)}, (x^{(1)})^16 - \omega_9^{16\text{rev}_{(3)}(i)} \rangle \rangle \right).
\end{align*}$$

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 $9 + 31 + 63 \cdot 2 = 544$ distinct twiddle factors in their NTTs. We require $9 + 9 \cdot 1 + 30 = 48$ distinct twiddle factors where the 9 ones are the twiddles $\omega_9^{i}, \omega_9^{16i}$ 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 \( x \sim x^{(0)} x^{(1)} \), \( (x^{(0)})^9 \sim 1 \), and \((x^{(1)})^{160} \sim 1 \), and compute the 2-dimensional transformation \( \text{NTT}_{R[x^{(0)}]}; q \omega_{q} \otimes \text{NTT}_{R[x^{(1)}]; 160 \omega_{q}^{32}} \) 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 \( x \sim x^{(0)} x^{(1)} \), \( (x^{(0)})^3 \sim 1 \), and \((x^{(1)})^{512} \sim 1 \). They then compute three size-512 NTTs and 512 \( 3 \times 3 \) convolutions. We introduce the same equivalence but proceed differently. We compute the 2-dimensional transformation \( \text{NTT}_{R[x^{(0)}]; 3 \omega_{q} \otimes \text{NTT}_{R[x^{(1)}]; 512 \omega_{q}^{32}} } \) 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 \( 3 \times 3 \) 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 \( 3 \times 3 \) 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 \( \frac{1}{2} \) 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 \( 4 \times 4 \) schoolbook takes 30 cycles, and a radix-(2, 3) butterfly takes \( 9 \cdot 2 + 2 \cdot 3 = 24 \) cycles. Then we only need \( 15 \frac{1}{2} \cdot 4 \cdot 2 + 30 \cdot 6 + 24 \cdot 4 \approx 399 \) 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 \texttt{NTT}, \texttt{NTT_small}, \texttt{basemul}, and \texttt{iNTT}, remain the same.

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

<table>
<thead>
<tr>
<th>Conv.</th>
<th>NTRU ((n,q))</th>
<th>NTRU Prime ((p,q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size-1440</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Size-1536</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Size-1728</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>(n, q)</th>
<th>Convolution</th>
<th>This work</th>
<th>CHK’21</th>
<th>IKPC’22</th>
</tr>
</thead>
<tbody>
<tr>
<td>(677, 2048)</td>
<td>Size-677</td>
<td>–/–</td>
<td>–/–</td>
<td>144k/–</td>
</tr>
<tr>
<td></td>
<td>Size-1440</td>
<td>140k/143k</td>
<td>–/–</td>
<td>–/–</td>
</tr>
<tr>
<td></td>
<td>Size-1536</td>
<td>147k/149k</td>
<td>156k/–</td>
<td>–/–</td>
</tr>
<tr>
<td>(701, 8192)</td>
<td>Size-701</td>
<td>–/–</td>
<td>–/–</td>
<td>144k/–</td>
</tr>
<tr>
<td></td>
<td>Size-1440</td>
<td>141k/143k</td>
<td>–/–</td>
<td>–/–</td>
</tr>
<tr>
<td></td>
<td>Size-1536</td>
<td>148k/150k</td>
<td>156k/–</td>
<td>–/–</td>
</tr>
<tr>
<td>(821, 4096)</td>
<td>Size-821</td>
<td>–/–</td>
<td>–/–</td>
<td>193k/–</td>
</tr>
<tr>
<td></td>
<td>Size-1728</td>
<td>178k/182k</td>
<td>199k/–</td>
<td>–/–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(p, q)</th>
<th>Convolution</th>
<th>This work</th>
<th>ACC’21</th>
<th>Che21</th>
</tr>
</thead>
<tbody>
<tr>
<td>(653, 4621)</td>
<td>Size-1320</td>
<td>–/–</td>
<td>–/–</td>
<td>120k/–</td>
</tr>
<tr>
<td></td>
<td>Size-1440</td>
<td>142k/147k</td>
<td>–/–</td>
<td>–/–</td>
</tr>
<tr>
<td>(761, 4591)</td>
<td>Size-1530</td>
<td>–/–</td>
<td>152k/–</td>
<td>142k/–</td>
</tr>
<tr>
<td></td>
<td>Size-1536</td>
<td>151k/153k</td>
<td>159k/–</td>
<td>–/–</td>
</tr>
<tr>
<td></td>
<td>Size-1620</td>
<td>–/–</td>
<td>185k/–</td>
<td>–/–</td>
</tr>
<tr>
<td>(857, 5167)</td>
<td>Size-1722</td>
<td>–/–</td>
<td>–/–</td>
<td>203k/–</td>
</tr>
<tr>
<td></td>
<td>Size-1728</td>
<td>182k/186k</td>
<td>–/–</td>
<td>–/–</td>
</tr>
</tbody>
</table>

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 [IKPC’22] by 2.8% and the size-1536 convolution from [CHK’21] by 10.3%. Our size-1536 convolution is slower than [IKPC’22] but still faster than [CHK’21]. For ntruhrss701, we have similar results. For ntruhps4096821, our size-1728 convolution outperforms the size-821 convolution from [IKPC’22] 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 [IKPC’22], 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 [IKPC’22] 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
A Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4 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_map, 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_map. Our size-1728 convolution supports ntruhps4096821, and ntrulpr857/sntrup857 with only differences in final_map. Table 8 is the summary.

Table 8: Performance of polymuls in NTRU and NTRU Prime on Cortex-M4.

<table>
<thead>
<tr>
<th>(n, q)</th>
<th>Size</th>
<th>polymul</th>
<th>NTT</th>
<th>NTT_small</th>
<th>basemul</th>
<th>iNTT</th>
<th>final_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTRU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(677, 2048)</td>
<td>1440</td>
<td>140444</td>
<td>34102</td>
<td>33241</td>
<td>27690</td>
<td>36756</td>
<td>8935</td>
</tr>
<tr>
<td></td>
<td></td>
<td>143016</td>
<td>34963</td>
<td>34093</td>
<td>27825</td>
<td>37214</td>
<td>9208</td>
</tr>
<tr>
<td>(677, 2048)</td>
<td>1536</td>
<td>147126</td>
<td>37485</td>
<td>36573</td>
<td>23322</td>
<td>41437</td>
<td>8489</td>
</tr>
<tr>
<td></td>
<td></td>
<td>149174</td>
<td>38076</td>
<td>37139</td>
<td>23506</td>
<td>42001</td>
<td>8717</td>
</tr>
<tr>
<td>(701, 8192)</td>
<td>1440</td>
<td>140577</td>
<td>34102</td>
<td>33241</td>
<td>27690</td>
<td>36756</td>
<td>8968</td>
</tr>
<tr>
<td></td>
<td></td>
<td>143239</td>
<td>34957</td>
<td>34087</td>
<td>27819</td>
<td>37208</td>
<td>9431</td>
</tr>
<tr>
<td>(701, 8192)</td>
<td>1536</td>
<td>147670</td>
<td>37485</td>
<td>36573</td>
<td>23322</td>
<td>41437</td>
<td>9033</td>
</tr>
<tr>
<td></td>
<td></td>
<td>149771</td>
<td>38076</td>
<td>37139</td>
<td>23506</td>
<td>42001</td>
<td>9314</td>
</tr>
<tr>
<td>(821, 4096)</td>
<td>1728</td>
<td>181534</td>
<td>48629</td>
<td>47627</td>
<td>21848</td>
<td>53098</td>
<td>10512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>186197</td>
<td>49480</td>
<td>48507</td>
<td>22349</td>
<td>55569</td>
<td>10564</td>
</tr>
<tr>
<td>NTRU Prime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(653, 4621)</td>
<td>1440</td>
<td>142244</td>
<td>34104</td>
<td>33244</td>
<td>27690</td>
<td>36756</td>
<td>10629</td>
</tr>
<tr>
<td></td>
<td></td>
<td>146665</td>
<td>34992</td>
<td>34095</td>
<td>27813</td>
<td>37214</td>
<td>12823</td>
</tr>
<tr>
<td>(761, 4591)</td>
<td>1536</td>
<td>151374</td>
<td>37487</td>
<td>36573</td>
<td>23322</td>
<td>41435</td>
<td>12379</td>
</tr>
<tr>
<td></td>
<td></td>
<td>153399</td>
<td>38069</td>
<td>37138</td>
<td>23510</td>
<td>42001</td>
<td>12861</td>
</tr>
<tr>
<td>(857, 5167)</td>
<td>1728</td>
<td>184714</td>
<td>48629</td>
<td>47623</td>
<td>21848</td>
<td>53099</td>
<td>13995</td>
</tr>
<tr>
<td></td>
<td></td>
<td>189523</td>
<td>49483</td>
<td>48499</td>
<td>22336</td>
<td>55720</td>
<td>13743</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>ntruhps2048677</th>
<th>ntruhrss701</th>
<th>ntruhps4096821</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>[CHK +21]</td>
<td>143725k</td>
<td>821k</td>
<td>818k</td>
</tr>
<tr>
<td>[IKPC22]</td>
<td>142378k</td>
<td>816k</td>
<td>729k</td>
</tr>
<tr>
<td>[Li21]¹</td>
<td>4625k</td>
<td>820k</td>
<td>812k</td>
</tr>
<tr>
<td>This work</td>
<td>3912k</td>
<td>525k</td>
<td>718k</td>
</tr>
</tbody>
</table>

¹ Integrated into pqm4 in commit 2691b4915b76db8b765ba89e4e09ad5c6399763f.

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. [ACC⁰²¹] 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 [ACC⁰²¹] even though our polynomial multiplication is comparable to [ACC⁰²¹]. 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.

<table>
<thead>
<tr>
<th></th>
<th>ntrulp653</th>
<th>ntrulp761</th>
<th>ntrulp857</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K E D</td>
<td>K E D</td>
<td>K E D</td>
</tr>
<tr>
<td>ACC\textsuperscript{+}21\textsuperscript{a}</td>
<td>- - -</td>
<td>731k 1102k 1200k</td>
<td>- - -</td>
</tr>
<tr>
<td>Che21\textsuperscript{b}</td>
<td>678k 1158k 1233k</td>
<td>727k 1312k 1394k</td>
<td>- - -</td>
</tr>
<tr>
<td>This work</td>
<td>699k 1131k 1231k</td>
<td>710k 1266k 1365k</td>
<td>886k 1465k 1596k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ntrup653</th>
<th>ntrup761</th>
<th>ntrup857</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K E D</td>
<td>K E D</td>
<td>K E D</td>
</tr>
<tr>
<td>ACC\textsuperscript{+}21\textsuperscript{a}</td>
<td>- - -</td>
<td>10 778k 694k 572k</td>
<td>- - -</td>
</tr>
<tr>
<td>Che21\textsuperscript{b}</td>
<td>6715k 632k 487k</td>
<td>7951k 684k 538k</td>
<td>- - -</td>
</tr>
<tr>
<td>This work</td>
<td>6623k 621k 527k</td>
<td>7937k 666k 563k</td>
<td>10 192k 812k 685k</td>
</tr>
</tbody>
</table>

\textsuperscript{a} uses secret-dependent table lookup AES, see https://github.com/mupq/pqm4/pull/173 for details.

\textsuperscript{b} [Che21, Method 2] is integrated into pqm4 in commit 844e7cafd5b5f8416aebf3c03b49ed8b10b7e396.

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


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


