

Corrected version of Thesis, date 2021.10.07

國立臺灣大學電機資訊學院電子工程學研究所



碩士論文

Graduate Institute of Electronics Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

ARM 處理器上晶格密碼學中的數論轉換多項式乘法

Number Theoretic Transform for Polynomial
Multiplication in Lattice-based Cryptography on ARM
Processors

鄭允立

Yun-Li Cheng

指導教授：黃俊郎 博士

Advisor: Jiun-Lang Huang Ph.D.

中華民國 110 年 9 月

September, 2021

國立臺灣大學碩士學位論文

口試委員會審定書



ARM 處理器上晶格密碼學中的數論轉換多項式
乘法

Number Theoretic Transform for Polynomial
Multiplication in Lattice-based Cryptography on
ARM Processors

本論文係鄭允立君（R08943155）在國立臺灣大學電子工程學研究所完成之碩士學位論文，於民國 110 年 9 月 3 日承下列考試委員審查通過及口試及格，特此證明

口試委員：_____

（指導教授）

_____	_____
_____	_____
_____	_____
_____	_____

所 長：_____





誌謝

感謝楊柏因老師、黃俊郎老師以及鄭振牟老師，幫助我度過在碩班兩年所遇到的大小事情。我要特別感謝中研院的楊柏因老師，雖然老師平時相當忙碌，但於研究上有問題時總是能迅速地回覆，並且能快速地點出我的盲點。平時會用循序漸進的方式引導我們學習新知，不知不覺中就碰到了一些能當作题目的研究方向，對我能夠順利完成論文有莫大的幫助。

求學過程中，感謝學長姊劉軒竹、魯家齊、林宜臻、鄧惇益與許瑞麟，與我們相處時毫無學長姐架子，樂於與我們分享不論是學習上或是生活上的資訊，幫助我們迅速適應碩班生活。也要感謝同學蔡明翰、賴亭融、李菁琳、趙偉捷、李育論，與他們互相協助才能在心靈上以及課業上都以良好的狀態度過。





摘要

隨著量子電腦的發展，傳統的密碼系統的安全性逐漸受到威脅，因此後量子密碼系統的重要性也相應提升。對於密碼系統來說，速度的快慢影響系統的安全性。更快的速度代表能在相同時間內使用更大的金鑰來進行加解密，所以如何有效實作一個密碼系統就顯得相當重要。而對於晶格密碼系統來說，多項式乘法的實作的重要性佔據了相當重要的地位。本篇論文將以 NTRU Prime 密碼系統在 ARM Cortex-M4 處理器上的實作為主軸，來探討多項式乘法在各種條件需求下的有效率的作法。

關鍵字：後量子密碼學、晶格密碼系統、NTRU Prime、多項式乘法、數論轉換、ARM Cortex-M4





Abstract

With the development of quantum computers, the security of traditional cryptosystems are under threat, so that it is important to have post-quantum cryptosystems. The speed of a cryptosystem affects its security in that a cryptosystem with higher speed can perform encryption or decryption with larger keys within the same amount of time, which means higher security. Therefore, it is important to have efficient implementations of cryptosystems. For lattice-based cryptography, the implementation of polynomial multiplication is quite important. We will discuss how to implement efficient polynomial multiplications by demonstrating the implementation of NTRU Prime cryptosystem on ARM Cortex-M4 processor in the paper.

Keywords: Post-quantum cryptography, Lattice-based cryptography, NTRU Prime, Polynomial multiplication, NTT, ARM Cortex-M4

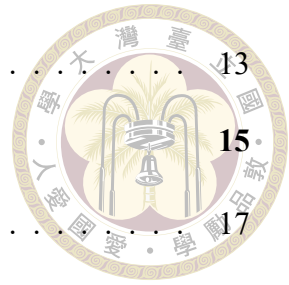




Contents

	Page
誌謝	iii
摘要	v
Abstract	vii
Contents	ix
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Related work	2
1.3 Contributions	2
Chapter 2 Background	3
2.1 NTRU Prime	3
2.1.1 Streamlined NTRU Prime	4
2.1.2 NTRU LPrime	5
2.2 Number Theoretic Transform (NTT)	6
2.2.1 Mixed-radix NTT	7
2.2.2 Rader's Algorithm	8
2.2.3 Good-Thomas Algorithm	10
2.2.4 NTTs with unfriendly modulus	12

2.2.5	Chinese Remainder Theorem	13
Chapter 3	Implementation	15
3.1	Rq_mult_small	17
3.1.1	(p, q) = (653, 4621)	17
3.1.2	(p, q) = (761, 4591)	22
3.1.3	(p, q) = (857, 5167)	24
3.1.4	(p, q) = (953, 6343)	25
3.1.5	(p, q) = (1013, 7177)	27
3.1.6	(p, q) = (1277, 7879)	28
3.2	R3_mult	30
3.3	Polymul_NxN	32
Chapter 4	Result	35
Chapter 5	Conclusion and Future Work	37
References		39
Appendix A	— ARM assembly code	41



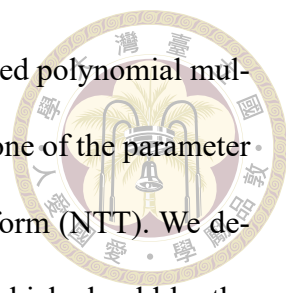


Chapter 1 Introduction

Post-quantum cryptography is the way to resist the attacks from quantum computers on classical computers. To elect new post-quantum cryptographic standards, the National Institute of Standards and Technology (NIST) began to host a competition to choose new standards of post-quantum cryptosystems. Lattice-based cryptography is an important candidate of post-quantum cryptography, NTRU Prime[2] is a lattice-based cryptosystem and a candidate of the NIST Post-Quantum Standardization. In this paper, we implemented NTRU Prime on ARM Cortex-M4 and we will discuss the detail to implement some important components of lattice-based cryptosystem through our NTRU Prime implementation.

1.1 Motivation

Since the quality of implementation of cryptosystems will affect the result of the competition of choosing post-quantum standards, it is important to have effective implementations of cryptosystems on different platforms. The pqm4[6] collects the Cortex-M4 implementations of post-quantum cryptosystems which are the latest candidates that survived the NIST Post-Quantum Cryptography Standardization. We decided to make an efficient implementation of the NTRU Prime cryptosystem on Cortex-M4 to increase the



chance of being selected. Previous work for NTRU Prime implemented polynomial multiplications with Toom-Cook algorithm and Karatsuba algorithm for one of the parameter sets while there was no implementation with number theoretic transform (NTT). We decided to implement NTRU Prime for all parameter sets with NTT which should be the most efficient algorithm for polynomial multiplication in lattice-based systems.

1.2 Related work

There was an NTRU Prime implementation on Cortex-M4 for parameter set (761, 4591) without NTT[1], we decided to implement NTRU Prime with its six parameter sets with NTT. One of our NTT methods is changing the modulus, the idea of combining NTT with Good's trick and part of the results come from [4].

1.3 Contributions

We implemented NTRU Prime on Cortex-M4 entirely for the six parameter sets. We have tried different methods to implement the important component of lattice-based cryptosystem, the polynomial multiplications. The number theoretic transform (NTT) is an efficient way to compute the polynomial multiplications, we will go through different algorithms for NTT to get efficient ways to implement NTT polynomial multiplications with different constraints. Our NTT implementations for NTRU Prime have not small improvement than the previous work without NTT. The methods we have tried should be able to satisfy the requirement of implementing polynomial multiplications with NTT in any shape of polynomial ring.



Chapter 2 Background

2.1 NTRU Prime

NTRU Prime is one of the variants of NTRU. It tweaks NTRU to use rings without some special structures that some cryptosystems with such special structures have been broken by recent attacks. There are two schemes in the NTRU Prime family, Streamlined NTRU Prime and NTRU LPRime. Streamlined NTRU Prime involves computing the inversion of polynomial in the key generation while NTRU LPRime doesn't. This makes the key generation procedure of NTRU LPRime faster than Streamlined NTRU Prime. Streamlined NTRU Prime keeps the structure of NTRU which makes the encapsulation and decapsulation of Streamlined NTRU Prime faster than NTRU LPRime.

For the convenience of introducing NTRU Prime, we abbreviate the convolution polynomial rings below.

$$R = Z[x]/(x^p - x - 1)$$

$$R_3 = (Z/3)[x]/(x^p - x - 1)$$

$$R_q = (Z/q)[x]/(x^p - x - 1)$$

small is polynomial with coefficients in $\{-1, 0, 1\}$.

weight- w means the polynomial has exactly w non-zero coefficients.

short is small and weight- w .

rounded is each coefficient normalized and divisible by 3.



2.1.1 Streamlined NTRU Prime

Streamlined NTRU Prime has parameters (p, q, w) subject to the restrictions below.

$$p, q \in \text{prime}$$

$$2p \geq 3w$$

$$q \geq 16w + 1$$

$$x^p - x - 1 \text{ is irreducible in } (\mathbb{Z}/q)[x]$$

The algorithm of Streamlined NTRU Prime is described below.

Algorithm 1 Streamlined NTRU Prime Key Generation

Generate an uniform random **small** element g that is invertible in R_3

Generate an uniform random **short** element f

Generate an uniform random **short** element ρ

$1/g \leftarrow$ inverse of g in R_3

$h \leftarrow g/3f$ in R_q

$pk \leftarrow h$

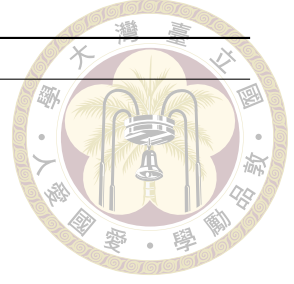
$sk \leftarrow f, 1/g$

$\underline{pk} \leftarrow \text{Encode}(pk)$

$\underline{sk} \leftarrow \text{Encode}(sk, \rho)$

return $\underline{pk}, \underline{sk}$

The details of *Encode*, *Decode*, *Round*, *HashConfirm*, and *HashSession* can be found in the document of NTRU Prime in the NIST Post-Quantum Cryptography Round 3 Submissions. We will focus on the implementation of polynomial multiplications of NTRU Prime in the paper.



Algorithm 2 Streamlined NTRU Prime Encapsulation

Input: Public key \underline{K}

Generate an uniform random **short** element r

$\underline{r} \leftarrow \text{Encode}(r)$

$h \leftarrow \text{Decode}(\underline{K})$

$hr \leftarrow h \cdot r$ in R_q

$c \leftarrow \text{Round}(hr)$

$\underline{c} \leftarrow \text{Encode}(c)$

$C \leftarrow (\underline{c}, \text{HashConfirm}(\underline{r}, \underline{K}))$

return $(C, \text{HashSession}(1, \underline{r}, C))$

Algorithm 3 Streamlined NTRU Prime Decapsulation

Input: $C = (c, \gamma)$, $sk = (f, g^{-1}, h, \rho)$

$3fc \leftarrow 3 \cdot f \cdot c$ in R_q

$e \leftarrow \text{map } 3fc \text{ to } R_3$

$ev \leftarrow e \cdot g^{-1}$ in R_3

$r' \leftarrow \text{map } ev \text{ to } R_q$

$hr' \leftarrow h \cdot r'$ in R_q

$c' \leftarrow \text{Round}(hr')$

$C' \leftarrow (c', \text{HashConfirm}(r', h))$

return $(C' == C) ? \text{HashSession}(1, r', C) : (0, \rho, C)$

2.1.2 NTRU LPRime

NTRU LPRime has parameters (p, q, w, δ, I) subject to the restrictions below.

$$p, q \in \text{prime}$$

$$2p \geq 3w$$

$$q \geq 16w + 1$$

$$x^p - x - 1 \text{ is irreducible in } (Z/q)[x]$$

The algorithms of NTRU LPRime can also be found in the document of NTRU Prime.

The main difference from Streamlined NTRU Prime is that it doesn't involve computing inverse of polynomials in the key generation while the polynomial multiplications still

appear in key generation, encapsulation and decapsulation.



2.2 Number Theoretic Transform (NTT)

Number theoretic transform (NTT) is an efficient method to compute polynomial multiplications. Since the polynomial multiplications take a great amount of time in the operations of lattice-based cryptosystems, fast and efficient implementations of polynomial multiplications are necessary. Number theoretic transform is a Fast Fourier Transform algorithm but uses integers and modulo operations instead of complex number operations. The NTT F of a vector $f \in Z_q^N$ is defined as

$$F_k = \sum_{n=0}^{N-1} f[n] \alpha^{nk}, k = 0, 1, 2, \dots, N-1$$

where α is an n th root of unity in Z_q . The difficulty of implementing NTT multiplication depends on the size N of the NTT we want to perform and whether the root of unity exists in Z_q . Usually, We don't perform NTT until the base size equals 1. It means that we don't have to find N th root of unity in Z_q but a smaller N 'th root of unity which is easier to find. If the root of unity we need can be found in Z_q , it will be easier to implement the NTT. If not, we have another approach which we find N th root unity in another ring Z_r selected by ourselves. The approach which changes the ring has an overhead of transforming coefficients between Z_q and Z_r while it may have better performance because we can choose Z_r as an "NTT-friendly" ring.

Radix-2 NTT. Radix-2 butterfly is the most well-known component of the FFT algorithm, and typical FFT multiplications are composed of several layers of radix-2 butterfly computation. However, we may not be able to find the desired n th root of unity where n

is the power of 2 in Z_q , e.g., in NTRU Prime. If so, we can apply mixed-radix NTT for implementing the NTT multiplications.



We will go through several efficient algorithms for NTT below. With these methods, we can give an acceptable or even efficient way to compute the desired NTT in any polynomial ring.

2.2.1 Mixed-radix NTT

Since the size of NTT we want to perform is subject to the n th root of unity we can find in the ring, it is not necessarily that we only choose n th root of unity where n is a power of two. In this situation, we can consider a radix- p NTT such as radix-3 or radix-5 NTT which is less efficient than the standard radix-2 NTT. However, mixed-radix NTT may give a closer NTT size to the minimum size (at least the sum of degrees of the two input polynomials + 1) for NTT multiplication, which has less redundant computation. For example, 1620 NTT[1] and 1530 NTT can be applied to NTRU Prime with the parameter set $(p, q) = (761, 4591)$, the former is composed of radix-2, radix-3 and radix-5 NTTs while the latter involves a radix-17 NTT and gives better performance. The ARM Cortex-M4 has instructions (**smuad(x)**, **smlad(x)**, **smusd(x)**, **smlsd(x)**) that can perform 2 16-bit signed multiplications with one or two 32-bit addition/subtractions in one cycle. Those instructions are suitable for implementing radix- p NTT with 16-bit coefficients. The assembly code for radix-3 NTT is described in **Algorithm 4**, the code for other radix- p NTTs used in our implementation will show in later sections.

Algorithm 4 Radix-3 butterfly for 16-bit coefficients

Input: $a_0, a_{1,2} = a_2 || a_1, \omega = \psi_3^2 || \psi_3$ where ψ_3 is 3rd root of unity, $t_0 = 0x00010001$ **Output:** reduced $a_0 = a_0 + a_1 + a_2, a_{1,2} = a_0 + a_1\psi_3^2 + a_2\psi_3 || a_0 + a_1\psi_3 + a_2\psi_3^2$

- | | |
|--|--|
| 1: smlad $t_0, a_{1,2}, t_0, a_0$ | $\triangleright t_0 = a_0 + a_1 + a_2$ |
| 2: smlad $t_1, a_{1,2}, \omega, a_0$ | $\triangleright t_1 = a_0 + a_1\psi_3 + a_2\psi_3^2$ |
| 3: smladx $t_2, a_{1,2}, \omega, a_0$ | $\triangleright t_1 = a_0 + a_1\psi_3^2 + a_2\psi_3$ |
| 4: smmulr t, t_0, q^{-1} | |
| 5: mls a_0, t, q, t_0 | $\triangleright a_0 = \text{reduced } t_0$ |
| 6: smmulr t, t_1, q^{-1} | |
| 7: mls t_1, t, q, t_1 | $\triangleright t_1 = \text{reduced } t_1$ |
| 8: smmulr t, t_2, q^{-1} | |
| 9: mls t_2, t, q, t_2 | $\triangleright t_2 = \text{reduced } t_2$ |
| 10: pkhbt $a_{1,2}, t_1, t_2, LSL \#16$ | $\triangleright a_{1,2} = t_2 t_1$ |
-

2.2.2 Rader's Algorithm

Rader's algorithm is a fast Fourier transform (FFT) algorithm that computes a prime size discrete Fourier transform (DFT) by re-expressing the DFT of size p as a cyclic convolution. After the re-expressing of the original DFT, there remain $2p-2$ additions and a positive cyclic convolution of size $p-1$ to be computed, where the cyclic convolution can be divided into two half-size cyclic convolutions. The positive part of the smaller cyclic convolutions can be divided as long as the size is an even number. Through such a method, we can perform prime size NTT in an efficient way when we need mixed-radix NTT. There is a constraint that the polynomial should be in a ring which is in the form of $F_q[x]/(x^p - 1)$. Rader's algorithm doesn't work in $F_q[x]/(x^p - \omega)$ where ω is not equal to 1. Recall the formula of NTT with $N = p$ where p is a prime number and ω is N th root of unity

$$F_k = \sum_{i=0}^{p-1} f_i \omega^{ik}, k = 0, 1, 2, \dots, p-1$$

, each of the p terms of NTT involves multiplications except the one labeled 0 (sum of coefficients). Rader's algorithm tells us how to efficiently get other $p-1$ terms by computing a cyclic convolution of size $p-1$. The steps are described below.

Compute $F_k - f_0, k = 1, 2, \dots, p - 1$. The $p - 1$ terms can be expressed as

$$F_k - f_0 = \sum_{i=1}^{p-1} f_i \omega^{ik}, k = 1, 2, \dots, p - 1$$



which can be computed by a cyclic convolution. The inputs of the cyclic convolution are the coefficients and powers of root of unity with re-expressing. The re-expressing is done by choosing a generator g of prime p and permuting these coefficients with the sequences generated by g . For example, $p = 5$ and $g = 2$, the sequence is $\{1, 2, 4, 3\}$. We permute the coefficients and powers of root of unity with the sequence one in forward and another in backward. Then we have

$$f' = \{f_1, f_2, f_4, f_3\}$$

$$\omega' = \{\omega^2, \omega^1, \omega^3, \omega^4\}$$

and we treat f' and ω' as degree-3 polynomials and compute the cyclic convolution $f' * \omega'$ in $Z_q/(x^4 - 1)$. Then we have

$$f_1 \omega^2 + f_2 \omega^4 + f_4 \omega^3 + f_3 \omega^1 = F_2 - f_0$$

$$f_1 \omega^1 + f_2 \omega^2 + f_4 \omega^4 + f_3 \omega^3 = F_1 - f_0$$

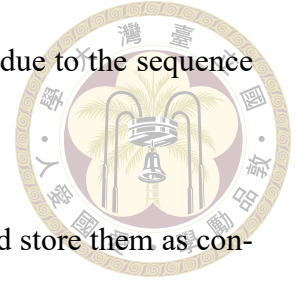
$$f_1 \omega^3 + f_2 \omega^1 + f_4 \omega^2 + f_3 \omega^4 = F_3 - f_0$$

$$f_1 \omega^4 + f_2 \omega^3 + f_4 \omega^1 + f_3 \omega^2 = F_4 - f_0$$

is the result of the convolution.

Add f_0 back to the result of the convolution and compute F_0 . After we add f_0 to each term of the convolution, we get the $p - 1$ terms of the NTT: F_1 to F_{p-1} . The term F_0 is just the sum of the p coefficients.

Then the NTT is done and needs to be stored in the correct location due to the sequence we choose.

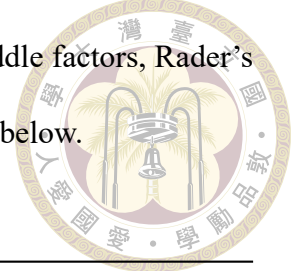


We can always precompute the powers of the root of unity ω and store them as constants to save the effort of computing them in runtime. What's more, we can combine the factor N^{-1} , which is needed when we perform inverse NTT, with the precomputed constants. By this, we can reduce the number of division operations in a size p inverse NTT layer to a proportion $2/p$ of the original number. This is achieved by just computing f_0/N and F_0/N and adding f_0/N to the temporary result from the cyclic convolution.

2.2.3 Good-Thomas Algorithm

Good-Thomas algorithm, or the prime-factor algorithm, is an FFT algorithm that re-expresses the DFT of a size $N=N_1N_2$ as a two-dimensional $N_1 \times N_2$ DFT, where N_1 and N_2 are relatively prime. The general mixed-radix NTT can also divide a DFT of size $N=N_1N_2$ into smaller transforms of size N_1 and N_2 even if N_1 and N_2 are not relatively prime, but it has the disadvantage that it also requires extra multiplications by the root of unity called twiddle factors, in addition to smaller transforms. It is usually that the size of smaller transforms are relatively prime when we perform mixed-radix NTT, so we can perform smaller transforms of different sizes which are relatively prime contiguously without extra multiplications by twiddle factors, by the Good-Thomas algorithm. The re-expressing operations can be merged into the operations of loading coefficients when performing NTT, then we will get almost no overhead for speed but larger code sizes. The re-expressing is done by assigning $x = yz$ where $y^{N_1} = z^{N_2} = 1$ and some permutations, then we will get a two-dimensional array of size $N_1 \times N_2$. Each row or column in the two-dimensional array represents a polynomial in $F[y]/(y^{N_1} - 1)$ or $F[z]/(z^{N_2} - 1)$, so we can

perform NTTs of size $N1$ or $N2$ without extra multiplications by twiddle factors, Rader's algorithm also works if the size is prime. The algorithm is described below.



Algorithm 5 Good-Thomas Algorithm

Input: polynomial $A \in F[x]/(x^{N1N2} - 1)$, $N1$ and $N2$ are relatively prime
 assign $x = yz$ where $y^{N1} = z^{N2} = 1$
 $ANS[N1][N2] \leftarrow$ a two-dimensional array by some permutations of A
return ANS

For example, we want to do a 12 NTT over the ring $(\mathbb{Z}/q)[x]/(x^{12} - 1)$, we can re-express the 12 coefficients as a 2D array with Good-Thomas algorithm, then the left 3 NTT and 4 NTT can be computed in $(\mathbb{Z}/q)[y]/(y^3 - 1)$ and $(\mathbb{Z}/q)[z]/(z^4 - 1)$ without multiplications by twiddle factors. Given a polynomial with 12 coefficients a_0, a_1, \dots, a_{11} , the reexpression is shown below.

weight	z^0	z^1	z^2	z^3
y^0	a_0	a_9	a_6	a_3
y^1	a_4	a_1	a_{10}	a_7
y^2	a_8	a_5	a_2	a_{11}

Then we can see the 12 coefficients as a polynomial

of y where the coefficients are polynomials of z

$$(a_0 + a_9z + a_6z^2 + a_3z^3) + (a_4 + a_1z + a_{10}z^2 + a_7z^3)y + (a_8 + a_5z + a_2z^2 + a_{11}z^3)y^2$$

where $y^3 = 1$, or a polynomial of z where the coefficients are polynomials of y

$$(a_0 + a_4y + a_8y^2) + (a_9 + a_1y + a_5y^2)z + (a_6 + a_{10}y + a_2y^2)z^2 + (a_3 + a_7y + a_{11}y^2)z^3$$

where $z^4 = 1$. After the permutation, we can perform both 3 NTT and 4 NTT without multiplication with twiddle factors and which NTT to perform first doesn't affect.

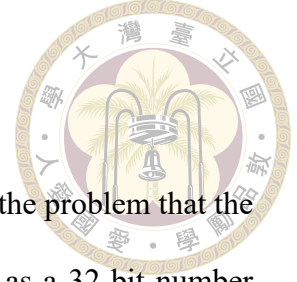
2.2.4 NTTs with unfriendly modulus



In the previous sections we introduce the Rader's algorithm and the Good-Thomas algorithm as an efficient way to perform NTTs when the modulus is not the form $2^k m + 1$, however, it is possible that there is still not an efficient way to do NTTs under the modulus. Assume that we have a polynomial multiplication of length p modulo q , we can express the polynomials with their coefficients in $(-q/2, q/2)$, then we would find that the coefficients of the result are in $(-pq^2/4, pq^2/4)$ if we temporarily ignore the modulo operation. We may want to perform a size n NTT where $n \geq 2p - 1$ and we can choose a larger prime which can cover the range of the coefficients, then we use the prime as our new modulus and it will not affect the original result. We can choose a number that is friendly to perform the size n NTT (usually involves several layers of radix-2 NTT) as the new modulus and do the NTT. In the last we modulo q then we get the correct result.

For example, the multiplication in $(\mathbb{Z}/4591)[x]/(x^{761} - x - 1)$, we let the coefficients be in the range $(-2295, 2295)$ and the lengths of polynomials to be multiplied are 761, so the possible range of the result coefficients is $(-2295^2 \cdot 761, 2295^2 \cdot 761)$ before being reduced. Then we choose a new modulus q' which is larger than $2 \cdot 2295^2 \cdot 761$ and is friendly to NTT (we can find the desired n th root of unity). While $R_{q_mult_small}$, the polynomial multiplication of NTRU Prime, multiply elements one in R_q and another in R_3 , so the possible range is $(-2295 \cdot 761, 2295 \cdot 761)$. An appropriate size $N=1536$ is suitable for implementing NTT multiplication for this case. We can choose a new modulus that has the 512th root of unity and perform incomplete NTT with only radix-2 butterfly computations. In the last, we modulo the original q and the result is what we want.

2.2.5 Chinese Remainder Theorem



When we apply the method in Section 2.2.4, we may encounter the problem that the new modulus satisfying the conditions is too large to be expressed as a 32-bit number so that it will be inconvenient to implement with such a modulus. If so, we can consider choosing a new number which is the product of two smaller prime numbers and the product is large enough. The Chinese Remainder Theorem (CRT) states a ring isomorphism where p_1 and p_2 are prime numbers

$$(Z/p_1p_2Z) \cong (Z/p_1Z) \times (Z/p_2Z)$$

which means we can perform two NTT multiplications in (Z/p_1Z) and (Z/p_2Z) respectively and get the result in (Z/p_1p_2Z) with a few computation to map the results in (Z/p_1Z) and (Z/p_2Z) back to (Z/p_1p_2Z) . Assume the minimum prime number satisfying the conditions is p' , we find two smaller prime p_1 & p_2 where $p_1 * p_2 \geq p'$ and that p_1, p_2 are both friendly to NTT. Then we perform NTTs with modulus p_1, p_2 separately. By combining with the Chinese Remainder Theorem, we can get the result which is the product of the two input polynomials without reductions modulo the real modulus. Then we reduce modulo the real modulus and get the desired answer. Through the method, we avoid dealing with numbers that are too large, but the overhead is that we need more than one NTT multiplication.





Chapter 3 Implementation

In this section, we discuss three ways for implementing the polynomial multiplications in NTRU Prime, one is the schoolbook multiplication (with Karatsuba or Toom but without NTT), another is the NTT with original modulus and the other is the NTT with changing modulus. The implementation should be constant time to avoid leaking secret information, so we use Barrett and Montgomery reductions to reduce the coefficients.

Barrett reduction. Barrett reduction uses a multiplication with an approximate number to avoid using division when we reduce numbers. Suppose we reduce a number a modulo q , we choose an approximate number m and a power of 2 (e.g. 2^{32} for here), then the Barrett reduction is described as

$$a \bmod q \equiv a - \left(\left\lfloor \frac{a \cdot m}{2^{32}} \right\rfloor \cdot q \right)$$

where $m = \lfloor \frac{2^{32}}{q} \rfloor$. The division 2^{32} and then rounding can be done by adding 2^{31} to the number and right shifting. We choose 2^{32} for the power of 2 in Barrett reduction because there is a suitable instruction **smmulr** in Cortex-M4 which multiplies 2 32-bit numbers, adds 2^{31} to the 64-bit number, and outputs the high 32-bit. The assembly code of Barrett reduction is described in **Algorithm 6**.

Montgomery modular multiplication. It is another constant time reduction algorithm that performs one multiplication and one reduction in our implementation. Since

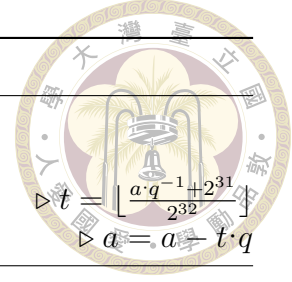
Algorithm 6 Signed Barrett reduction, $q^{-1} = \lfloor \frac{2^{32}}{q} \rfloor$

Input: a

Output: reduced a

1: **smmulr** t, a, q^{-1}

2: **mls** a, t, q, a



Cortex-M4 has only 14 available general-purpose registers, there might be some extra memory operations during NTT computations for register spilling. The Barrett reduction takes 2 registers for storing constants required for the computation, while Montgomery modular multiplication just needs one register for holding constants. However, the result of a Montgomery modular multiplication of a and b is $\frac{a \cdot b}{R} \bmod q$ instead of $a \cdot b \bmod q$ where R is a power of 2 related to the constant used in the reduction. The assembly code is described in **Algorithm 7** for 32-bit multiplication and **Algorithm 8** for 16-bit multiplication.

Algorithm 7 Montgomery modular multiplication, $R = 2^{32}, q \cdot q^{-1} \equiv 1 \bmod R$

Input: a, b

Output: $c_{high} = \frac{a \cdot b}{R} \bmod q$

1: **smull** c_{low}, c_{high}, a, b

▷ $c = a \cdot b$

2: **mul** $t, c_{low}, -q^{-1}$

3: **smlal** c_{low}, c_{high}, t, q

▷ $c \equiv a \cdot b \pmod{q}$ where $c_{low} = 0$

Algorithm 8 Montgomery modular multiplication, $R = 2^{16}, q \cdot q^{-1} \equiv 1 \bmod R$

Input: $a = a_{high} || a_{low}, b = b_{high} || b_{low}, Q = q || -q^{-1}$

Output: $c_{high} = \frac{a \cdot b}{R} \bmod q$

1: **smulbb** c, a, b

▷ $c = a_{low} \cdot b_{low}$

2: **smulbb** t, c, Q

▷ $t = c_{low} \cdot q^{-1}$

3: **smlabt** c, t, Q, c

▷ $c \equiv a \cdot b \pmod{q}$ where $c_{low} = 0$

In Algorithm 8., we choose from instructions(**smulbb**, **smulbt**, **smultb**, **smultt**) due to the position of coefficients in the register, as we store 2 16-bit coefficients in a 32-bit register.



3.1 Rq_mult_small

Rq_mult_small is the polynomial multiplication in NTRU Prime cryptosystems. The two inputs of Rq_mult_small are a polynomial in R_q and another in R_3 , and the result is a polynomial in R_q . It appears in key generation, encapsulation, and decapsulation in both Streamlined NTRU Prime and NTRU LPrime, so efficient implementations of the polynomial multiplications for different parameters are necessary. We will show three different ways to compute the polynomial multiplications for each of the six parameter sets of NTRU Prime, two of them are NTTs and the other is not. One of the NTT is changing the modulus, and the idea comes from [4]. The platform we used for the implementations is STM32F4DISCOVERY with ARM Cortex-M4 processor, we will show the CPU cycle counts for these methods as the speed evaluation. The cycle counts include the total cycle count, cycle count for two NTTs (one for polynomial in R_q and another in R_3), cycle count for one inverse NTT, and cycle count for the base multiplication. The cycle count for modulo $(x^p - x - 1)$ is merged with the cycle count for the inverse NTT. When implementing Rq_mult_small, we only care p and q . We abbreviate "polynomial multiplication" to "polymul" for later use.

3.1.1 $(p, q) = (653, 4621)$

Method 1. Schoolbook multiplication. We implemented a 704x704 polymul by Toom-4 with 176x176 polymul. The 176x176 polymul is by Toom-4 with 44x44 schoolbook polymul. Since the coefficients of the 6 NTRU Prime parameter sets can all be expressed as 16-bit numbers, we express the coefficients as 16-bit numbers and utilize the instructions (**smuad(x)**, **smlad(x)**, **smulbb**, **smlabb**, etc.) which are powerful when we multiply

and add/sub 16-bit numbers. For example, a 4x4 polymul can be implemented as described in **Algorithm 9**. The 44x44 schoolbook polymul is implemented with these instructions like **Algorithm 9** and it is combined with Toom-4 to make larger polymuls.



Algorithm 9 4x4 polymul for 16-bit coefficients

Input: $a_{0,1} = a_1 || a_0, a_{2,3} = a_3 || a_2, b_{0,1} = b_1 || b_0, b_{2,3} = b_3 || b_2$

Output: reduced $c_{0,1} = c_1 || c_0, c_{2,3} = c_3 || c_2, c_{4,5} = c_5 || c_4, c_6 = 0 || c_6$

- 1: **smuadx** $c_1, a_{0,1}, b_{0,1}$
 - 2: **smuadx** $c_3, a_{0,1}, b_{2,3}$
 - 3: **smladx** $c_3, a_{2,3}, b_{0,1}, c_3$
 - 4: **smuadx** $c_5, a_{2,3}, b_{2,3}$
 - 5: **pkhbt** $b_{0,3}, b_{0,1}, b_{2,3}$
 - 6: **pkhtb** $b_{2,1}, b_{0,1}, b_{2,3}$
 - 7: **smulbb** $c_0, a_{0,1}, b_{0,3}$
 - 8: **smuad** $c_2, a_{0,1}, b_{2,1}$
 - 9: **smlabb** $c_2, a_{2,3}, b_{0,3}, c_2$
 - 10: **smultt** $c_4, a_{0,1}, b_{0,3}$
 - 11: **smlad** $c_4, a_{2,3}, b_{2,1}, c_4$
 - 12: **smultt** $c_6, a_{2,3}, b_{0,3}$
 - 13: reduce and package as in **Algorithm 4**
-

An appropriate size of NTT multiplication for the parameter set is 1320 NTT. Since there is no 1320-th root of unity in the ring, we need to do an incomplete NTT. It means that we do not perform butterfly computations until the size of the base pointwise product is 1. It is more efficient to use incomplete NTT than complete NTT when we implemented these NTT multiplications, so the NTT multiplications implemented in this paper are all incomplete NTTs.

Method 2. NTT multiplication with original modulus. We implemented a 1320 NTT done by one radix-11 NTT using Rader's algorithm, followed by one radix-3 NTT and two radix-2 NTTs and base multiplication of size 10. The assembly code of radix-11 NTT with Rader's algorithm can be found in Appendix. The radix-3 NTT is described in **Algorithm 4** and the radix-2 NTT is described in **Algorithm 10**. When we perform NTT with small radices, we can perform more than one layer of NTT before writing the results to memory.

For example, 4 NTT or 8 NTT are composed of 2 or 3 layers of radix-2 NTT, so we can load all the coefficients we need and perform the NTT if there are enough free general-purpose registers. By this, we just need one loading operation and one storing operation for each coefficient rather than three loading operations and three storing operations for an 8 NTT(composed of 3 radix-2 NTTs).

Algorithm 10 Radix-2 butterfly(x2) for 16-bit coefficients

Input: $a = a_1 || a_0, b = b_1 || b_0, \omega = \text{root of unity in that layer}$

Output: $a + \omega b, a - \omega b$

- 1: **smulbb** t_0, ω, b
 - 2: **smulbt** t_1, ω, b
 - 3: **smmulr** t, t_0, q^{-1}
 - 4: **mls** t_0, t, q, t_0
 - 5: **smmulr** t, t_1, q^{-1}
 - 6: **mls** t_1, t, q, t_1
 - 7: **pkhbt** $t, t_0, t_1, LSL \#16$ $\triangleright \text{reduced } b\omega$
 - 8: **ssub16** b, a, t $\triangleright a - \omega b$
 - 9: **sadd16** a, a, t $\triangleright a + \omega b$
-

Another way is to use a larger prime that is friendly to NTT. The prime should have the ability to cover the range $(-2310 \cdot 1 \cdot 653, 2310 \cdot 1 \cdot 653)$ and the root of unity for several layers of radix-2 NTT. We perform NTT with this new modulus, then we modulo 4621 last and get the correct results. The new prime can't be expressed as a 16-bit number, so we need to use 32-bit instructions for this method while **Method 2.** utilize the 16-bit instructions of Cortex-M4.

Method 3. NTT multiplication with new modulus. We selected a new prime $q' = 6984193$ and implemented a 1536 NTT done by 384 NTT(1 radix-3 NTT and 7 radix-2 NTTs) with Good-Thomas algorithm and base multiplication of size 4. The q' is the same as the one in Section 3.1.2, since they use the same size of NTT multiplication in our implementation. In this method, we have 32-bit numbers and we only use radix-2 and radix-3 NTT since we can choose any satisfying modulus as we wish. The radix-2 NTT

and radix-3 NTT for 32-bit numbers are shown in **Algorithm 11** and **Algorithm 12**[5] and the 4x4 basemul is shown in **Algorithm 13**. Note that here We use Montgomery modular multiplication which returns numbers in the form $\frac{a}{R} \bmod q$, however, we can replace the ω (i.e. n th root of unity) by ωR to get the correct answer without extra multiplications. Notice that we use Montgomery modular multiplication in **Algorithm 13**, we replace ω with ωR for those terms needed multiplying with ω and then Montgomery reduce c_0 c_3 so the answers are actually in the form $\frac{a}{R} \bmod q$. The factor $\frac{1}{R}$ can be canceled when we perform inverse NTT, i.e. we multiply coefficients with $\frac{R}{N}$ instead of $\frac{1}{N}$.

Algorithm 11 Radix-2 butterfly for 32-bit coefficients

Input: $a, b, \omega = \text{root of unity in that layer}$

Output: $a + \omega b, a - \omega b$

- | | |
|--|--|
| 1: smull $c_{low}, c_{high}, b, \omega$ | |
| 2: mul t, c_{low}, q^{-1} | |
| 3: smlal c_{low}, c_{high}, t, q | $\triangleright c_{high} = \text{reduced } \omega b$ |
| 4: sub b, a, c_{high} | $\triangleright a - \omega b$ |
| 5: add a, a, c_{high} | $\triangleright a + \omega b$ |
-

Algorithm 12 Radix-3 butterfly for 32-bit coefficients

Input: $a_0, a_1, a_2, \omega, \omega^2$

Output:

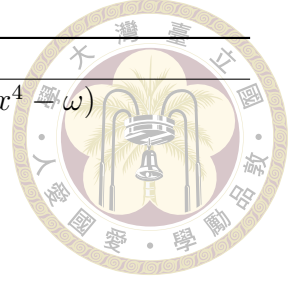
$$c_0 = a_0 + a_1 + a_2$$

$$c_1 = a_0 + a_1\omega + a_2\omega^2$$

$$c_2 = a_0 + a_1\omega^2 + a_2\omega$$

- 1: **add** c_0, a_1, a_2
 - 2: **smull** t_0, c_1, a_1, ω
 - 3: **smlal** t_0, c_1, a_2, ω^2
 - 4: **mul** t, t_0, q^{-1}
 - 5: **smlal** t_0, c_1, t, q
 - 6: **sub** c_2, a_0, c_0
 - 7: **sub** c_2, c_2, c_1
 - 8: **add** c_1, c_1, a_0
 - 9: **add** c_0, c_0, a_0
-

The flows of Method 2. and Method 3. are shown below.



Algorithm 13 4x4 basemul 32-bit coefficients

Input: $a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3, \omega$ where the 4x4 basemul modulo $(x^4 - \omega)$

Output: reduced

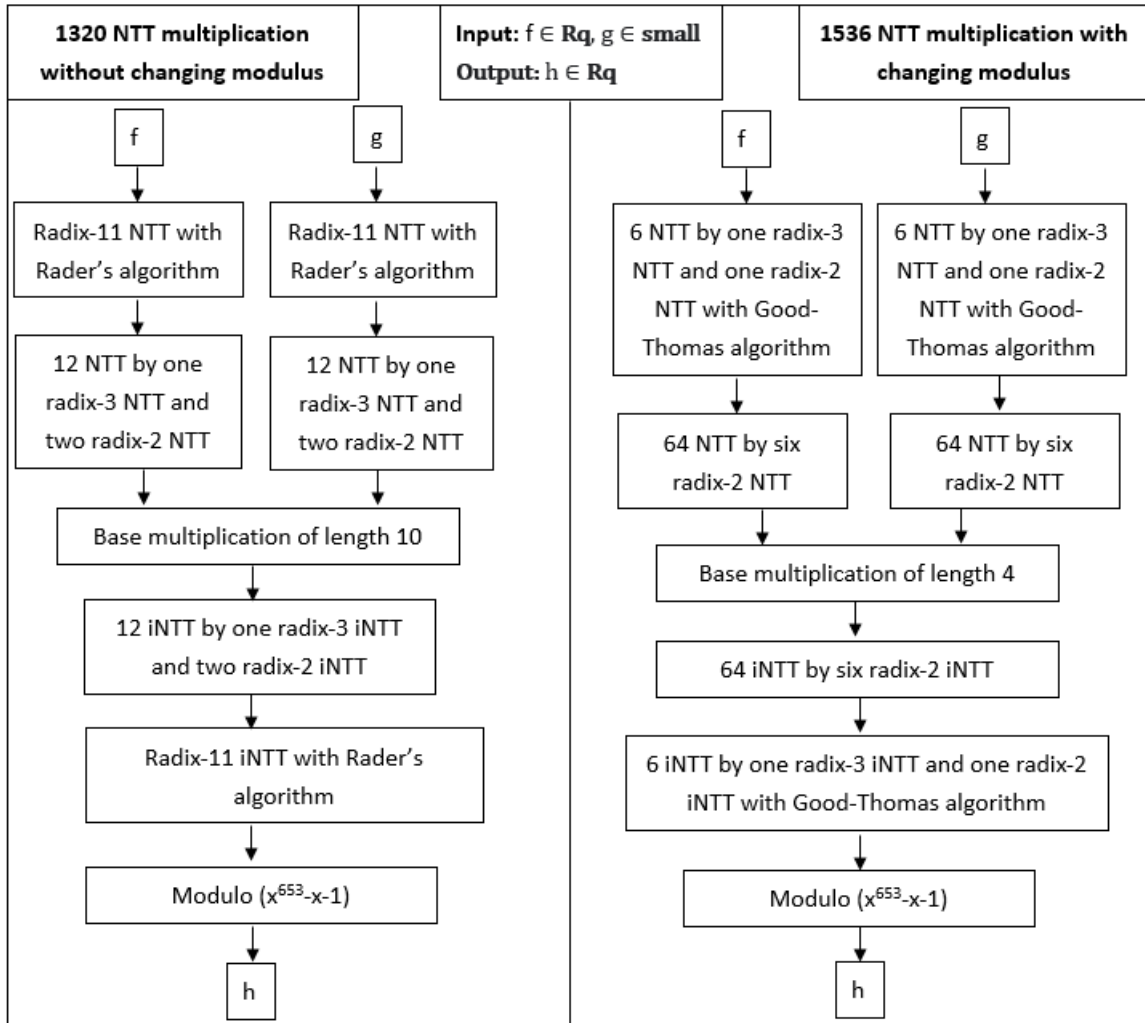
$$c_0 = a_0b_0 + (a_1b_3 + a_2b_2 + a_3b_1)\omega$$

$$c_1 = a_0b_1 + a_1b_0 + (a_2b_3 + a_3b_2)\omega$$

$$c_2 = a_0b_2 + a_1b_1 + a_2b_0 + (a_3b_3)\omega$$

$$c_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$$

- 1: **smul** c_{low}, c_0, a_1, b_3
- 2: **smlal** c_{low}, c_0, a_2, b_2
- 3: **smlal** c_{low}, c_0, a_3, b_1 $\triangleright a_1b_3 + a_2b_2 + a_3b_1$
- 4: **mul** t, c_{low}, q^{-1}
- 5: **smlal** c_{low}, c_0, t, q $\triangleright (a_1b_3 + a_2b_2 + a_3b_1)\omega$
- 6: **smul** $c_{low}, c_0, c_0, \omega$
- 7: **smlal** c_{low}, c_0, a_0, b_0
- 8: **mul** t, c_{low}, q^{-1}
- 9: **smlal** c_{low}, c_0, t, q $\triangleright c_0 = \text{reduced } a_0b_0 + (a_1b_3 + a_2b_2 + a_3b_1)\omega$
- 10: compute c_1, c_2, c_3 like we compute c_0



The cycle counts for the three methods are shown below.

parameters	implementation	cycle counts			
		total	two NTTs	inverse NTT	basemul
(653, 4621)	Method 1.	208,592	x	x	x
	Method 2.	120,137	56,779	43,115	22,321
	Method 3.[5]	152,368	79,026	49,173	24,251

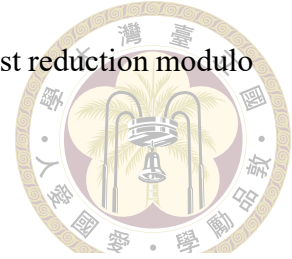
3.1.2 $(p, q) = (761, 4591)$

Method 1. Schoolbook multiplication. We implemented a 768x768 polymul by toom-4 with 192x192 polymul. The 192x192 polymul is by toom-4 with 48x48 schoolbook polymul.

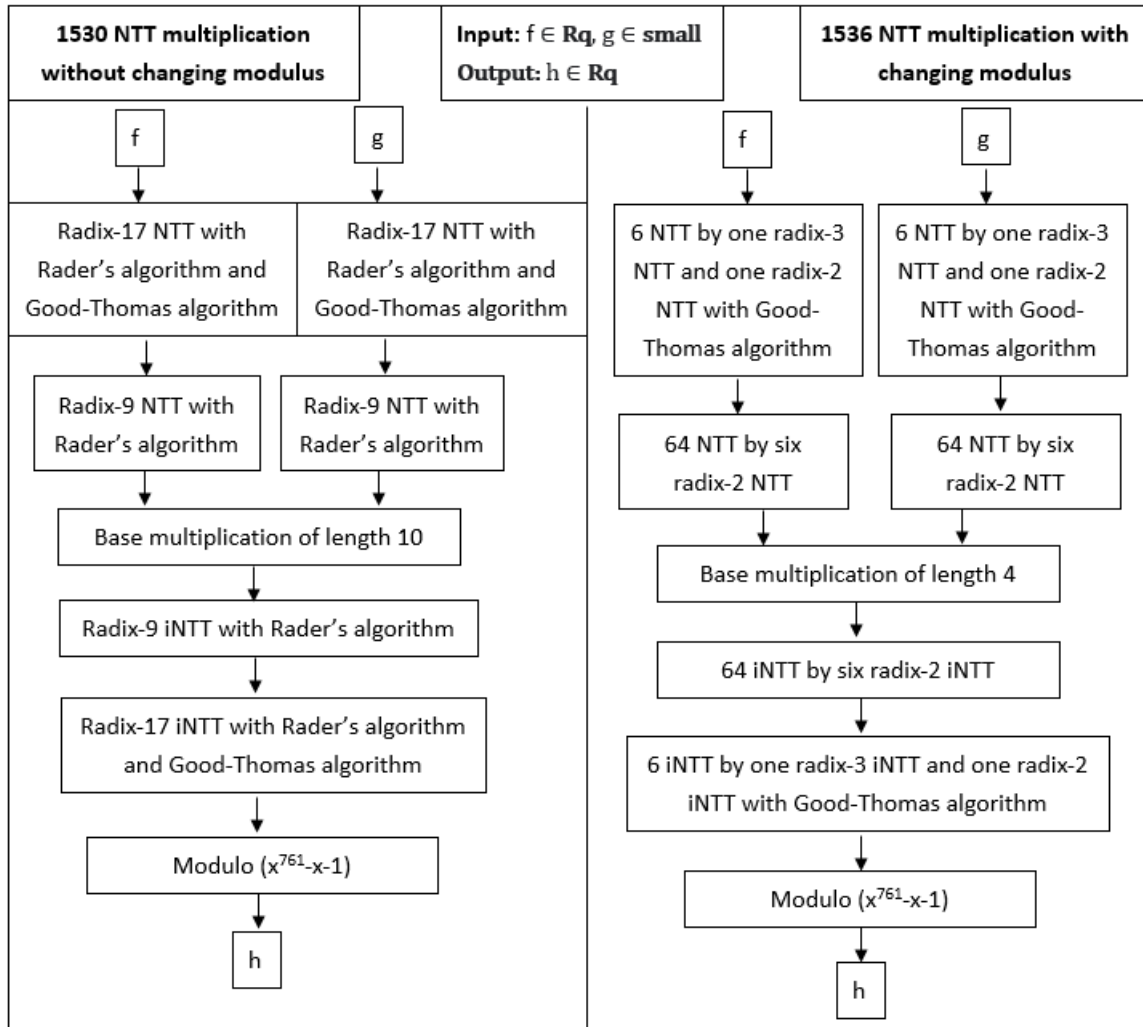
Method 2. NTT multiplication with original modulus. We implemented a 1530 NTT done by one radix-17 NTT using Rader's algorithm with Good-Thomas algorithm (or Good's trick), followed by one radix-9 NTT using Rader's algorithm and base multiplication of size 10. The idea of such application of Good's trick is already known in [5]. Rader's algorithm can also be applied to NTT with radix p^i where p is prime. The assembly code of radix-17 NTT and radix-9 NTT with Rader's algorithm can be found in Appendix. The 16x16 cyclic convolution in the radix-17 NTT can be split into smaller cyclic convolutions through CRT. The positive side of convolutions can be split into smaller pieces until the size equals 1, but we stop when the size equals 2 to utilize **smuad(x)**, **smlad(x)** to have less usage of instructions.

Method 3. NTT multiplication with new modulus. We selected a new prime and implemented a 1536 NTT done by 512 NTT with Good-Thomas algorithm and base multipli-

cation of size 3. It is the same as the one in Section 3.1.1 except the last reduction modulo 4591.



The flows of Method 2. and Method 3. are shown below.



The cycle counts for the three methods are shown below.

parameters	implementation	cycle counts			
		total	two NTTs	inverse NTT	basemul
(761, 4591)	Method 1.[1]	243,320	x	x	x
	Method 2.	142,255	72,756	46,409	25,870
	Method 3.[5]	153,719	79,300	50,254	24,254

3.1.3 $(p, q) = (857, 5167)$

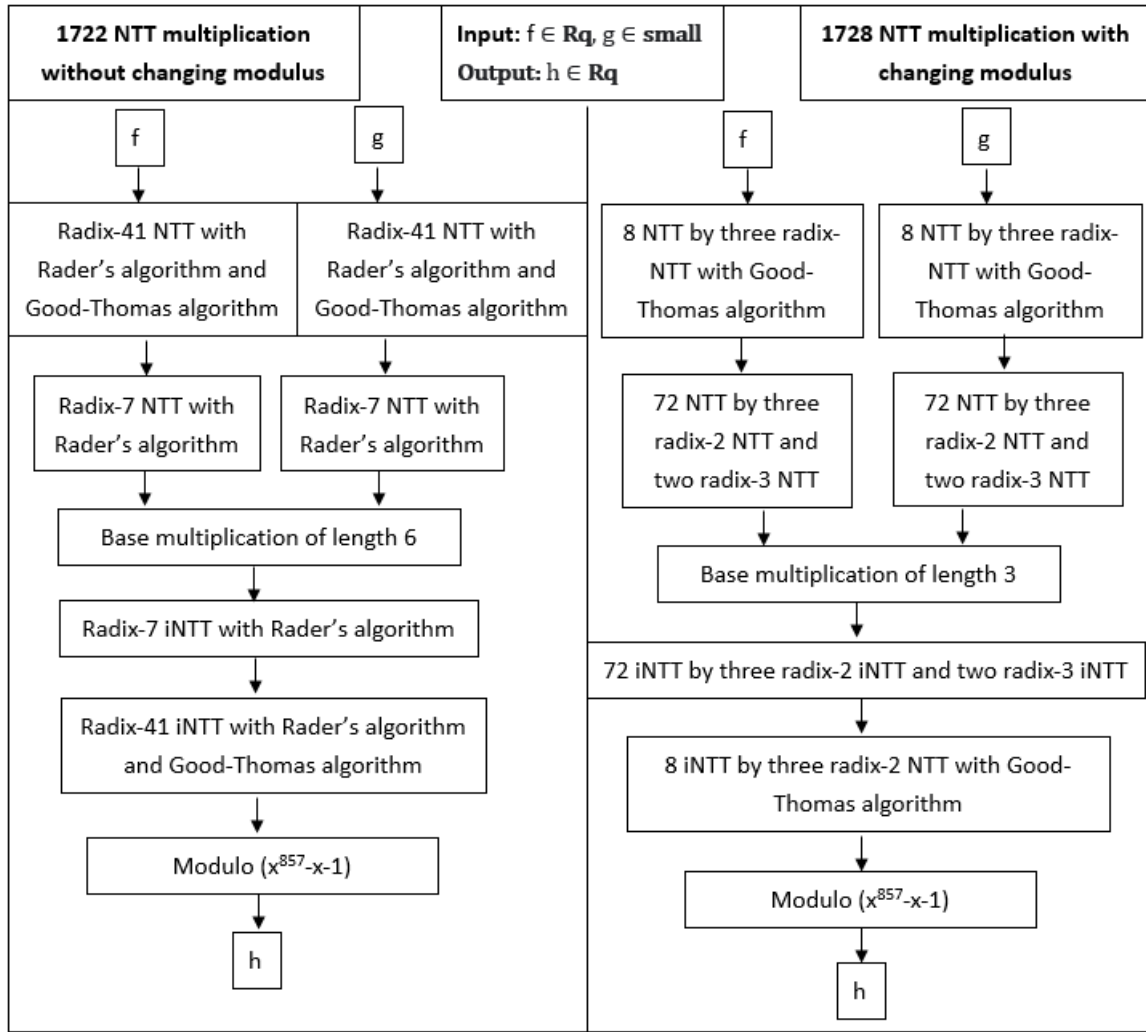


Method 1. Schoolbook multiplication. We implemented a 896x896 polymul by toom-4 with 224x224 polymul. The 224x224 polymul is by toom-4 with 56x56 schoolbook polymul.

Method 2. NTT multiplication with original modulus. We implemented a 1722 NTT done by one radix-41 NTT using Rader's algorithm with Good-Thomas algorithm, followed by one radix-7 NTT using Rader's algorithm and base multiplication of size 6. The cyclic convolution in radix-41 NTT can be split into smaller convolutions as in others with Rader's algorithm. However, there is a negative cyclic convolution in $(Z/q)[x]/(x^{20} + 1)$. We map $(Z/q)[x]/(x^{20} + 1)$ to $(Z/q)[y]/(y^{10} + 1)/(y - x^2)$, lifting to $(Z/q)[y]/(y^{10} + 1)$, and then perform the negative cyclic convolution with Karatsuba algorithm. So the negative cyclic convolution with length 20 become three negative cyclic convolutions with length 10. This not only cost less multiplication but also less register spilling because there are only 14 available general-purpose registers. The assembly codes of radix-41 NTT and radix-7 NTT with Rader's algorithm are in the Appendix.

Method 3. NTT multiplication with new modulus. We selected a new prime $q' = 4430593$ and implemented a 1728 NTT done by 576 NTT(6 layers radix-2 NTT and 2 layers radix-3 NTT) with Good-Thomas algorithm and base multiplication of size 3. The usage of instructions to compute 3x3 basemul is like in **Algorithm 13**.

The flows of Method 2. and Method 3. are shown below.



The cycle counts for the three methods are shown below.

parameters	implementation	cycle counts			
		total	two NTTs	inverse NTT	basemul
(857, 5167)	Method 1.	308,264	x	x	x
	Method 2.	203,135	110,703	69,750	22,762
	Method 3.[5]	183,430	100,236	64,562	18,632

3.1.4 $(p, q) = (953, 6343)$

Method 1. Schoolbook multiplication. We implemented a 960x960 polymul by toom-4 with 240x240 polymul. The 240x240 polymul is by toom-4 with 60x60 schoolbook

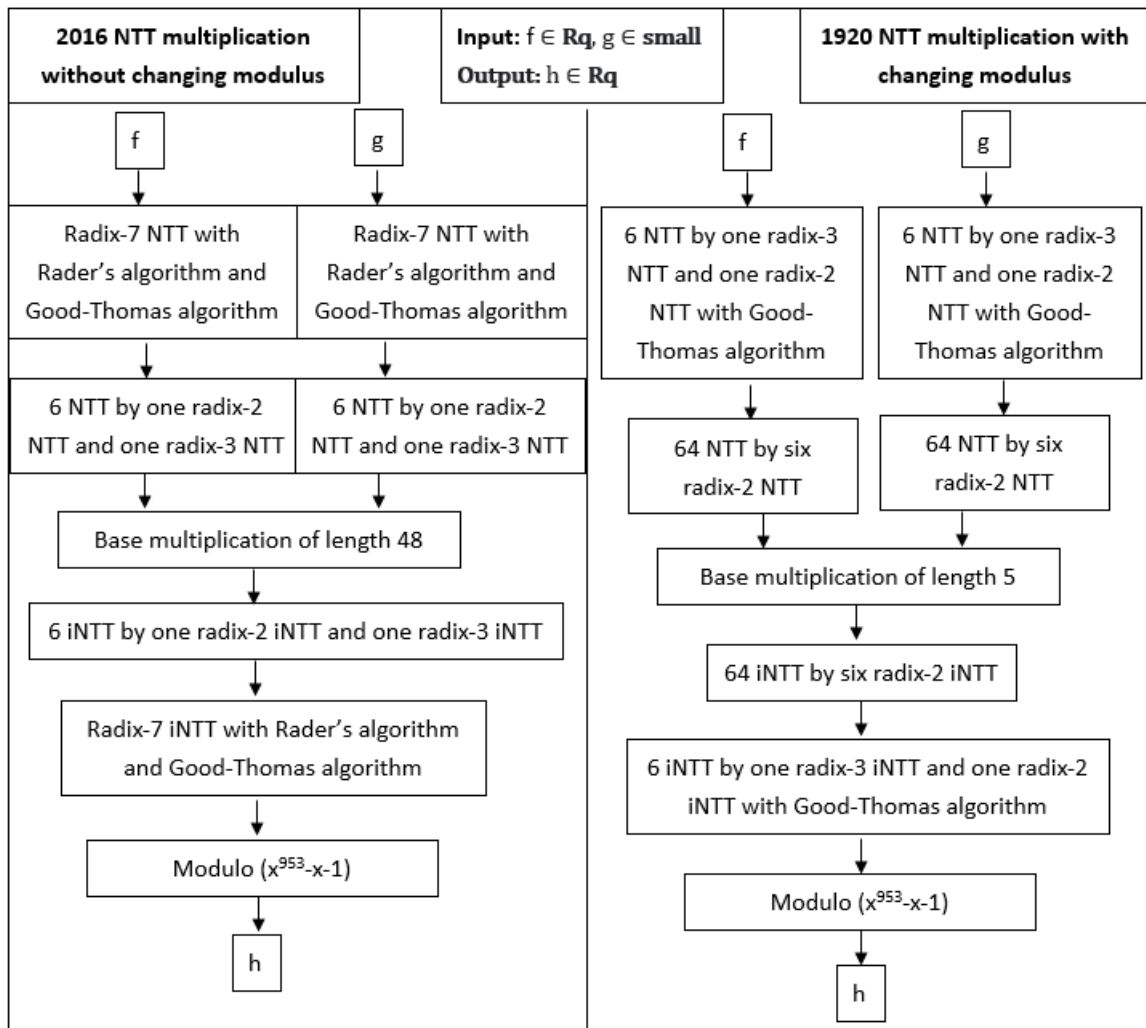
polymul.



Method 2. NTT multiplication with original modulus. We implemented a 2016 NTT done by one radix-7 NTT using Rader's algorithm with Good-Thomas algorithm, followed by one radix-3 NTT and one radix-2 NTT and base multiplication of size 48. Since $6342 = 151 \cdot 42$, the other choice is a radix-151 NTT but it is unrealistic. So we used the factor 42 to do the NTT if we want to use the original modulus 6343.

Method 3. NTT multiplication with new modulus. We selected a new prime $q' = 6045313$ and implemented a 1920 NTT done by 384 NTT(7 layers radix-2 NTT and 1 layer radix-3 NTT) with Good-Thomas algorithm and base multiplication of size 5.

The flows of Method 2. and Method 3. are shown below.



The cycle counts for the three methods are shown below.

parameters	implementation	cycle counts			
		total	two NTTs	inverse NTT	basemul
(953, 6343)	Method 1.	342,118	x	x	x
	Method 2.	265,307	88,641	52,195	127,963
	Method 3.[5]	185,790	91,279	66,011	27,310

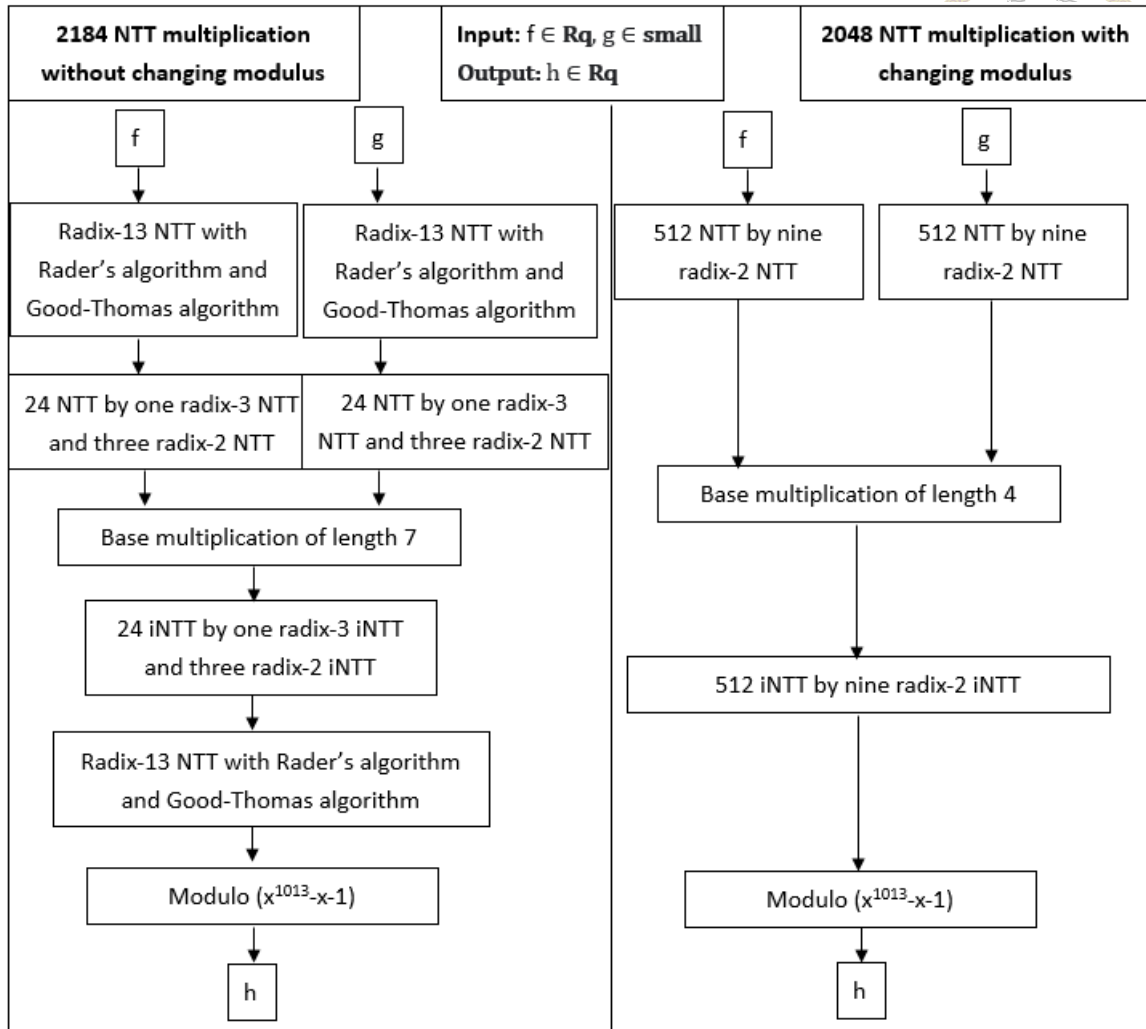
3.1.5 $(p, q) = (1013, 7177)$

Method 1. Schoolbook multiplication. We implemented a 1024x1024 polymul by toom-4 with 256x256 polymul. The 256x256 polymul is by toom-4 with 64x64 schoolbook polymul.

Method 2. NTT multiplication with original modulus. We implemented a 2184 NTT done by one radix-13 NTT using Rader's algorithm with Good-Thomas algorithm, followed by one radix-3 NTT and three radix-2 NTTs and base multiplication of size 7.

Method 3. NTT multiplication with new modulus. We selected a new prime $q' = 7272449$ and implemented a 2048 NTT done by 512 NTT(9 layers radix-2 NTT) and base multiplication of size 4.

The flows of Method 2. and Method 3. are shown below.



The cycle counts for the three methods are shown below.

parameters	implementation	cycle counts			
		total	two NTTs	inverse NTT	basemul
(1013, 7177)	Method 1.	378,697	x	x	x
	Method 2.	269,802	147,656	87,620	34,611
	Method 3.[5]	225,484	113,054	80,382	32,048

3.1.6 $(p, q) = (1277, 7879)$

Method 1. Schoolbook multiplication. We implemented a 1280x1280 polymul by toom-4 with 320x320 polymul. The 320x320 polymul is by toom-4 with 80x80 schoolbook

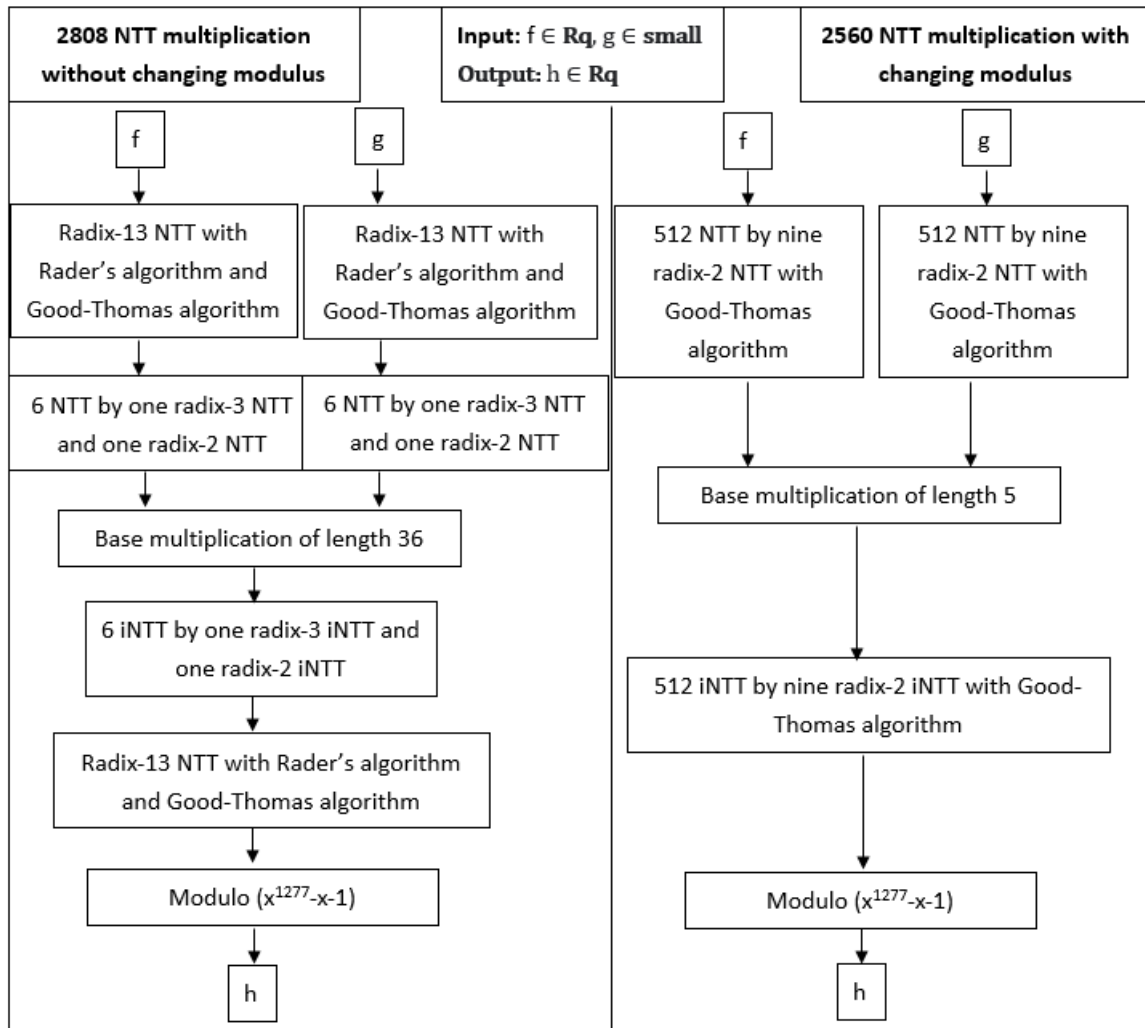
polymul.



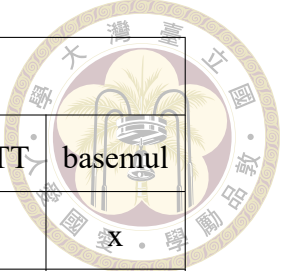
Method 2. NTT multiplication with original modulus. We implemented a 2808 NTT done by one radix-13 NTT using Rader's algorithm with Good-Thomas algorithm, followed by one radix-3 NTT and one radix-2 NTT and base multiplication of size 36.

Method 3. NTT multiplication with new modulus. We selected a new prime $q' = 10063873$ and implemented a 2560 NTT done by 512 NTT(9 layers radix-2 NTT) with Good-Thomas algorithm and base multiplication of size 5.

The flows of Method 2. and Method 3. are shown below.



The cycle counts for the three methods are shown below.

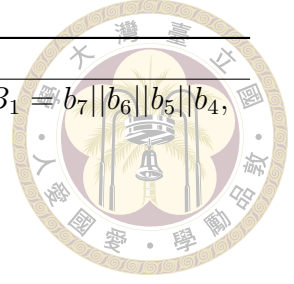


parameters	implementation	cycle counts			
		total	two NTTs	inverse NTT	basemul
(1277, 7879)	Method 1.	510,824	x	x	x
	Method 2.	357,731	134,724	80,891	142,203
	Method 3.[5]	284,015	145,994	101,622	36,399

3.2 R3_mult

There is another polynomial multiplication, `R3_mult`, in the decapsulation of Streamlined NTRU Prime, which multiplies two polynomials in R_3 . Since the polynomials in R_3 can be expressed by coefficients in $[-1, 1]$, the range of the coefficients of the resulting polynomial are in $(-p, p)$ before reduction. We can apply the same way as method 3. in Section 3.1 to implement `R3_mult`. For example, we can choose a new prime 7681 which can cover the range of the resulting polynomial before modulo 3 to compute a 1536 NTT for the parameter (761, 4591). However, Cortex-M4 has suitable instructions for implementing schoolbook multiplication for `R3_mult`, so the NTT method is slower than schoolbook multiplication here. The **`umull`**, **`umlal`** instructions multiply two unsigned 32-bit values to produce a 64-bit value, and the latter accumulates this with a 64-bit value. In this function, we express the input coefficients in $[0, 2]$ and each coefficient is stored as an unsigned 8-bit value in a byte. Then we can perform a 4x4 polymul with one **`umull`**, or one **`umlal`** if we want to accumulate it with another 8 coefficients. The 8x8 polymul example assembly code is described in **Algorithm 14**.

We implemented `R3_mult` with the Karatsuba algorithm and the results are shown below.



Algorithm 14 8x8 polymul for R_3

Input: $A_0 = a_3||a_2||a_1||a_0, A_1 = a_7||a_6||a_5||a_4, B_0 = b_3||b_2||b_1||b_0, B_1 = b_7||b_6||b_5||b_4,$

Output: reduced $(A_0 + A_1x^4) \cdot (B_0 + B_1x^4)$

$r=0x03030303$

- 1: **umull** t_0, t_1, A_0, B_0
 - 2: **umull** t_2, t_3, A_1, B_1
 - 3: **umlal** t_1, t_2, A_0, B_1
 - 4: **umlal** t_1, t_2, A_1, B_0
 - 5: **and** $t, t_0, 0x1c1c1c1c$
 - 6: **and** t_0, t_0, r
 - 7: **add** $t_0, t_0, t, LSR \#2$
 - 8: **and** $t, t_0, 0x1c1c1c1c$
 - 9: **and** t_0, t_0, r
 - 10: **add** $t_0, t_0, t, LSR \#2$
 - 11: **usub8** t, t_0, r
 - 12: **sel** t_0, t, t_0
 - 13: reduce t_1 to t_3 as steps 5 to 12
-

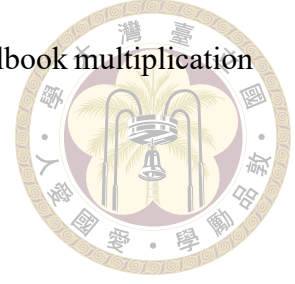
▷ reduced $t_0 = ANS_{0-3}$

parameters	cycle counts	polymul size	# layer Karatsuba
(653, 4621)	67,116	704x704	2
(761, 4591)	75,704	768x768	2
(857, 5167)	99,191	896x896	2
(953, 6343)	111,333	960x960	2
(1013, 7177)	113,585	1024x1024	2
(1277, 7879)	175,325	1280x1280	3

The results of NTTs with modulus 7681 are shown below.

size of NTT	cycle counts			
	total	NTT	inverse NTT	basemul
1536	130,081	31,012	43,094	21,583
2048	176,028	41,140	57,206	32,079
2560	224,012	51,268	71,318	44,882

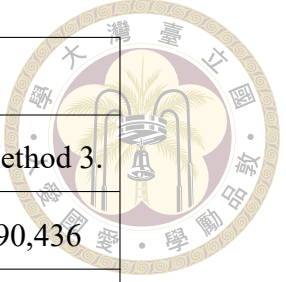
The results show that the NTT methods are not efficient as schoolbook multiplication when we implement R3_mult.



3.3 Polymul_NxN

In the key generation of Streamlined NTRU Prime, it involves computing the inverse of polynomials in both R_q and R_3 . If we use an algorithm called `jumpdivstepx`[3], we would need several polynomial multiplications with different lengths, such as 128x128, 256x256, 256x512, and so on. We can adjust the size of NTTs used in `Rq_mult_small` easily to fit the polynomial multiplications. For example, the 1320NTT in Section 3.1.1 can shrink to 264NTT to fit 128x128 polynomial multiplication, by adjusting the loop counter and the address where we load and store coefficients. Notice that the two input polynomials of `Polymul_NxN` here are both in R_q , so method 3. in Section 3.1 can not be directly applied here. If we want to use the method in Section 3.1, the new modulus will be very large and maybe can't be expressed as a 32-bit number. Therefore, we tried the approach with the Chinese Remainder Theorem mentioned in Section 2.5, and then we need to compute two NTT multiplications for one `polymul` function while the result told us it is not faster.

The cycle counts for Method 1.(schoolbook), Method 2.(NTT with original modulus), and Method 3.(NTT with changing modulus) are shown below. The result is derived from [7].



function	parameters	cycle counts		
		Method 1.	Method 2.	Method 3.
polymul_256x256	(653, 4621)	44,977	49,337	90,436
	(761, 4591)	44,977	53,461	90,436
	(857, 5167)	44,977	64,866	90,436

function	parameters	cycle counts		
		Method 1.	Method 2.	Method 3.
polymul_512x512	(857, 5167)	131,347	129,545	216,378
	(953, 6343)	131,347	119,420	216,378
	(1013, 7177)	131,347	128,224	216,378

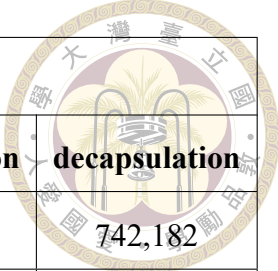




Chapter 4 Result

We choose the fastest version of implementations on Cortex-M4 we have done for each function. The comparison of previous work and our results are shown below.

scheme	cycle count		
	key generation	encapsulation	decapsulation
sntrup653	6,650,815	629,265	486,847
sntrup761	7,949,876	682,008	534,547
sntrup857	10,257,849	852,057	685,000
sntrup953	12,744,797	941,634	741,212
sntrup1013	13,983,083	1,030,507	833,363
sntrup1277	22,853,961	1,320,360	1,066,752
ntrulpr653	683,301	991,309	1,067,752
ntrulpr761	726,140	1,112,316	1,194,398
ntrulpr857	928,609	1,330,026	1,452,109
ntrulpr953	1,016,173	1,437,653	1,555,985
ntrulpr1013	1,111,710	1,587,218	1,738,997
ntrulpr1277	1,433,318	2,023,573	2,212,008



scheme	cycle count		
	key generation	encapsulation	decapsulation
sntrup761(previous work)[1]	10,901,785	789,442	742,182
ntrupr761(previous work)[1]	823,655	1,309,214	1,491,900



Chapter 5 Conclusion and Future Work

Intuitively, polynomial multiplications with NTT algorithms have better performance when the input sizes grow up, however, which NTT algorithm to use has a great influence. The results of our implementations give the guide for the choices of NTTs for different input sizes when using Cortex-M4. The mixed-radix NTT approaches may have better performance than the classical NTTs with only several layers of radix-2 NTTs with the Good-Thomas algorithm if the parameters are suitable. However, if we can choose radix-2 NTTs, they are still the better choices when the input sizes grow up. The mixed-radix NTT method and the method of NTT with changing modulus are both efficient, so we can implement both to figure out which is better under the environment. If the NTT method with changing modulus needs to be converted to two NTTs due to large modulus, the performance would not be better than schoolbook multiplication.

Our implementations can be executed on other ARM processors if they cover the instruction set of ARM Cortex-M4, e.g., ARM Cortex-A7. The performance may have further improvement if utilizing ARM NEON technology for future versions of NTRU Prime on other ARM processors.





References

- [1] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wälde, et al. Polynomial multiplication in ntru prime. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 217–238, 2021.
- [2] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. Ntru prime: reducing attack surface at low cost. In International Conference on Selected Areas in Cryptography, pages 235–260. Springer, 2017.
- [3] D. J. Bernstein and B.-Y. Yang. Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 340–398, 2019.
- [4] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang. Ntt multiplication for ntt-unfriendly rings. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 159–188, 2021.
- [5] V. Hwang. Personal communication from Vincent Hwang. 2021.
- [6] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. 2019.

- [7] C.-L. Li. Implementation of polynomial modular inversion in lattice based cryptography on arm. 2021.





Appendix A — ARM assembly code

We will demonstrate how we implement Rader's algorithm in ARM assembly through a radix-5 NTT with Rader's algorithm, we split it into several parts. The radix-5 NTT is enough and suitable to explain how Rader's algorithm works, because the code of larger radix will be too large to show here.

Step 1. Load and CRT. We first load coefficients with a special sequence and split the cyclic convolution into smaller pieces by CRT. The CRT part is optional due to the size of convolution.

Step 2. Compute the cyclic convolutions. Compute the cyclic convolutions with different sizes.

Step 3. Inverse CRT and output the result. Perform inverse CRT if CRT is used in

Step 1. Output the result with a special sequence.



Algorithm 15 Radix-5 NTT with Rader's algorithm: loading coefficients and CRT

Input: f : pointer to input polynomial

Output: $fpad$: coefficients after loading with a special sequence and CRT We choose a generator g of 5, e.g. $g = 2$, and the sequence is

$$(1, 2, 4, 3)$$

Assume the step of index of coefficients to load is d , $r0 = fpad$, $r1 = f$.

The **ldr** instruction load 32-bit while we store coefficients as 16-bit numbers, so we actually perform 2 radix-5 NTT in an iteration.

1: ldr $r2, [r1, \#1 \cdot d \cdot 2]$	▷ a_0 of convolution mod $(x^4 - 1)$
2: ldr $r3, [r1, \#2 \cdot d \cdot 2]$	▷ a_1 of convolution mod $(x^4 - 1)$
3: ldr $r4, [r1, \#4 \cdot d \cdot 2]$	▷ a_2 of convolution mod $(x^4 - 1)$
4: ldr $r5, [r1, \#3 \cdot d \cdot 2]$	▷ a_3 of convolution mod $(x^4 - 1)$
5: sadd16 $r6, r2, r4$	▷ a_0 of convolution mod $(x^2 - 1)$
6: sadd16 $r7, r3, r5$	▷ a_1 of convolution mod $(x^2 - 1)$
7: ssub16 $r8, r2, r4$	▷ a_0 of convolution mod $(x^2 + 1)$
8: ssub16 $r9, r3, r5$	▷ a_1 of convolution mod $(x^2 + 1)$
9: sadd16 $r10, r6, r7$	▷ sum of coefficients
10: pkhbt $r2, r6, r7, LSL \#16$	▷ $a_0 a_1$ of convolution mod $(x^2 - 1)$ -(1)
11: pkhbt $r3, r7, r6, ASR \#16$	▷ $a_0 a_1$ of convolution mod $(x^2 - 1)$ -(2)
12: pkhbt $r4, r8, r9, LSL \#16$	▷ $a_0 a_1$ of convolution mod $(x^2 + 1)$ -(1)
13: pkhbt $r5, r9, r8, ASR \#16$	▷ $a_0 a_1$ of convolution mod $(x^2 + 1)$ -(2)
14: str $r2, [r0, \#1 \cdot d \cdot 2]$	
15: str $r3, [r0, \#2 \cdot d \cdot 2]$	
16: str $r4, [r0, \#3 \cdot d \cdot 2]$	
17: str $r5, [r0, \#4 \cdot d \cdot 2]$	
18: str $r10, [r0], \#4$	



Algorithm 16 Radix-5 NTT with Rader's algorithm: convolutions

Input: $r0 = fpad$: pointer to the coefficients after **Algorithm 14**, ω is 4th root of unity

Output: $fpad$: the result after computing the convolutions

Backward sequence for root of unity

$$(2, 1, 3, 4)$$

So the another multiplicand of the convolution is

$$(\omega^2, \omega^1, \omega^3, \omega^4)$$

CRT applied

$$(\omega^2 + \omega^3, \omega^1 + \omega^4, \omega^2 - \omega^3, \omega^1 - \omega^4)$$

Assume the ω s are already loaded to $r1 = (\omega^1 + \omega^4) || (\omega^2 + \omega^3)$, $r2 = (\omega^1 - \omega^4) || (\omega^2 - \omega^3)$

1: ldr $r3, [r0, \#1 \cdot d \cdot 2]$	19: smusd $r7, r2, r5$
2: ldr $r4, [r0, \#2 \cdot d \cdot 2]$	20: smuadx $r8, r2, r5$
3: ldr $r5, [r0, \#3 \cdot d \cdot 2]$	21: smmulr $t, r7, q^{-1}$
4: ldr $r6, [r0, \#4 \cdot d \cdot 2]$	22: mls $r7, t, q, r7$
5: smuad $r7, r1, r3$	23: smmulr $t, r8, q^{-1}$
6: smuadx $r8, r1, r3$	24: mls $r8, t, q, r8$
7: smmulr $t, r7, q^{-1}$	25: pkhbt $r5, r7, r8, LSL \#16 \triangleright$ computed convolution mod $x^2 + 1-(1)$
8: mls $r7, t, q, r7$	26: smusd $r7, r2, r6$
9: smmulr $t, r8, q^{-1}$	27: smuadx $r8, r2, r6$
10: mls $r8, t, q, r8$	28: smmulr $t, r7, q^{-1}$
11: pkhbt $r3, r7, r8, LSL \#16 \triangleright$ computed convolution mod $x^2 - 1-(1)$	29: mls $r7, t, q, r7$
12: smuad $r7, r1, r4$	30: smmulr $t, r8, q^{-1}$
13: smuadx $r8, r1, r4$	31: mls $r8, t, q, r8$
14: smmulr $t, r7, q^{-1}$	32: pkhbt $r6, r7, r8, LSL \#16 \triangleright$ computed convolution mod $x^2 + 1-(2)$
15: mls $r7, t, q, r7$	33: str $r3, [r0, \#1 \cdot d \cdot 2]$
16: smmulr $t, r8, q^{-1}$	34: str $r4, [r0, \#2 \cdot d \cdot 2]$
17: mls $r8, t, q, r8$	35: str $r5, [r0, \#3 \cdot d \cdot 2]$
18: pkhbt $r4, r7, r8, LSL \#16 \triangleright$ computed convolution mod $x^2 - 1-(2)$	36: str $r6, [r0, \#4 \cdot d \cdot 2]$



Algorithm 17 Radix-5 NTT with Rader's algorithm: inverse CRT and output

Input: $r0 = fpad$: pointer to the coefficients after **Algorithm 15**, $r1 = f$:input polynomial

Output: $fpad$: the result of radix-5 NTT

```
1: ldr r2, [r1], #4
2: ldr r3, [r0, #1·d·2]
3: ldr r4, [r0, #2·d·2]
4: ldr r5, [r0, #3·d·2]
5: ldr r6, [r0, #4·d·2]
6: ldr r11, [r0]
7: sadd16 r7, r3, r5
8: ssub16 r8, r3, r5
9: sadd16 r9, r4, r6
10: ssub16 r10, r4, r6
11: pkhbt r3, r7, r9, LSL #16
12: pkhtb r4, r9, r7, ASR #16
13: pkhbt r5, r8, r10, LSL #16
14: pkhtb r6, r10, r8, ASR #16
15: sadd16 r11, r11, r2
16: sadd16 r3, r3, r2
17: sadd16 r4, r4, r2
18: sadd16 r5, r5, r2
19: sadd16 r6, r6, r2
20: str r3, [r0, #2·d·2]
21: str r4, [r0, #1·d·2]
22: str r5, [r0, #3·d·2]
23: str r6, [r0, #4·d·2]
24: str r11, [r0], #4
```

▷ ans_0 of convolution mod $(x^4 - 1)$
▷ ans_1 of convolution mod $(x^4 - 1)$
▷ ans_2 of convolution mod $(x^4 - 1)$
▷ ans_3 of convolution mod $(x^4 - 1)$

▷ store to $r0 + 2·d·2$
▷ store to $r0 + 1·d·2$
▷ store to $r0 + 3·d·2$
▷ store to $r0 + 4·d·2$