國立臺灣大學電機資訊學院資訊工程學系
碩士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science

National Taiwan University
Master Thesis

使用Armv7-M、Armv7E-M、Armv8-A 實作數論變換之案例研究

Case Studies on Implementing Number-Theoretic Transforms with Armv7-M, Armv7E-M, and Armv8-A

黃柏文
Vincent Bert Hwang

指導教授：李彥寰 博士
Advisor: Dr. Yen-Huan Li

中華民國 111 年 6 月
June, 2022

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Acknowledgements

First and foremost, I would like to thank my mother for her understanding and trust on my decisions. Lastly, I also want to thank Dr. Yen-Huan Li, my advisor, for giving me a chance attending the master's program.

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

國立臺灣大學
敦品勵學 · 愛國愛人

摘要

這篇論文收集在碩士研究中所發表或是投稿的相關論文。我們專注於數論變換在 Cortex-M3、Cortex-M4、Cortex-A72 上的組合語言實作。在 Cortex-M3 上，我們用 Armv7-M 來實作；在 Cortex-M4 上，我們用 Armv7E-M 來實作；在 Cortex-A72 上，我們用 Armv8.0-A 來實作。爲了呈現優化的效果，我們將數論變換的實作應用於 Dilithium、Kyber、NTRU、NTRU Prime 和 Saber 這些晶格密碼系統中。

我們在 Cortex-M3 上實作了 Saber；在 Cortex-M4 上實作了 NTRU、NTRU Prime 和 Saber；在 Cortex-A72 上實作了 Dilithium、Kyber 和 Saber。我們針對速度、記憶體和程式碼大小來優化 Saber 在 Cortex-M3 和 Cortex-M4 上的實作、針對速度和程式碼大小來優化 NTRU 和 NTRU Prime 在 Cortex-M4 上的實作、針對速度來優化 Dilithium、Kyber 和 Saber 在 Cortex-A72 上的實作。

關鍵詞： 晶格密碼系統、數論變換、快速傅立葉變換、Arm 組合語言、Cortex-M3、Cortex-M4、Cortex-A72

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

國立臺灣大學
敦品勵學・愛國愛人

Abstract

This thesis collects selected works published or submitted during the study of the master's program. We focus on the assembly realizations of number–theoretic transforms (NTTs) on Cortex-M3, Cortex-M4, and Cortex-A72. In particular, we implement the NTTs in Armv7-M for Cortex-M3, Armv7E-M for Cortex-M4, and Armv8.0-A for Cortex-A72. To demonstrate the optimizations, we select the lattice-based cryptosystems Dilithium, Kyber, NTRU, NTRU Prime, and Saber for evaluation.

We apply our optimizations to Saber on Cortex-M3; NTRU, NTRU Prime, and Saber on Cortex-M4; and Dilithium, Kyber, and Saber on Cortex-A72. We optimize Saber on Cortex-M3 and Cortex-M4 for speed, memory usage, and code size; NTRU and NTRU Prime on Cortex-M4 for speed and code size; and Dilithium, Kyber, and Saber on Cortex-A72 for speed.

Keywords: lattice-based cryptosystem, number–theoretic transform, fast Fourier transform, Arm assembly, Cortex-M3, Cortex-M4, Cortex-A72

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Contents

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

List of Figures

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

List of Tables

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

List of Algorithms

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

List of Listings

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Chapter 1

Introduction

Since the late 1920s, the rapid development of the theory of computation has led to many great theorems and conjectures. Common strategies for approaching computational problems are often based on abstract computational models. In practice, as computational problems must be modeled as transformations between configurations, several concerns occur.

1. Which hardware is desirable for representing configurations?
2. How well can we control the target hardware?

While this thesis is not trying to address both concerns, the second concern is an essential factor of the methodology adopted in this thesis. There is often a specification on how implementors can communicate with the hardware. Modern approaches to the communication often consist of the following:

1. A human-accessible interface supporting the production of sequences of strings as realizations of target algorithms.
2. A predefined procedure with some knowledge of target platforms transforming the sequences of strings into sequences of strings honoring the specifications.

The first step is often realized with general-purpose programming languages such as C and Haskell, and the sequences of strings are called the source code. The second step then refers to a compiler compiling source code into sequences of platform-dependent instructions. During the compilation, a compiler can apply various kinds of optimizations. Modern compilers usually produce reasonably fast sequences of platform-dependent instructions so humans can focus on the high-level realizations of target algorithms.

However, there are now at least two factors in the evaluation of the algorithms:

1. How well the algorithms are realized with the chosen general-purpose programming language.
2. How well the realizations are optimized into sequences of platform-dependent instructions by the chosen compiler.

Instead of separating the two factors for scientific reasoning, this thesis chooses a different approach by providing data points according to the following principle: How

doi:10.6342/NTU202200738

well the algorithms are realized with sequences of platform-dependent instructions. In other words, this thesis aims to find pairings between algorithms and sequences of platform-dependent instructions for evaluating the performance of target algorithms with minimal intervention from compiler optimizations.

1.1 Selected Subjects and Platforms

As realizing algorithms with sequences of platform-dependent instructions requires great effort, this study is valuable only if we choose the subjects and platforms wisely.

The implementations of cryptosystems are the chosen subjects of this thesis since they are used for maintaining the confidentiality, authenticity, and integrity of messages. There are at least two types of cryptosystems: secret-key cryptosystems and public-key cryptosystems. Secret-key cryptosystems offer security based on shared secrets, and public-key cryptosystems achieve this with the secret keys of key pairs. A key pair consists of a public key and a secret key. The secret key is kept secret, and the public key is shared with others through possibly insecure channels. Secret-key and public-key encryptions achieve confidentiality, and message-authentication codes and digital signatures achieve authenticity and integrity. For encryptions, since public-key encryptions often transmit much more data than secret-key encryptions, the security through possibly insecure channels is achieved by combining secret-key and public-key encryptions. A key-encapsulation mechanism (KEM) allows both parties to agree on a secret with public-key encryption. The agreed secret is then used as a shared secret for secret-key encryption.

Widely deployed public-key cryptosystems are usually based on RSA and ECC. In 1994, Shor introduced algorithms for factorizing integers and solving discrete logarithms in polynomial time with quantum computation [Sho94]. The former breaks RSA and the latter breaks ECC once large-scale quantum computers are available. Since then, researchers have been developing cryptosystems without known weaknesses from quantum computers. This line of research is known as Post-Quantum Cryptography (PQC). At PQCrypto 2016, the National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization program. In July 2020, seven finalists and eight alternate candidates were announced and categorized as follows.

- Lattice-based cryptosystems.
  - Finalists: Dilithium [ABD+20a], Falcon [PFH+20], Kyber [ABD+20b], NTRU [CDH+20], and Saber [DKRV20].
  - Alternates: FrodoKEM [NAB+20] and NTRU Prime [BBC+20].
- Multivariate cryptosystems.
  - Finalist: Rainbow [DCP+20].
  - Alternate: GeMSS [CFMR+20].
- Hash-based cryptosystems.
  - Finalists: None.

doi:10.6342/NTU202200738

  - Alternate: SPHINCS^{+} [HBD+20].

- Code-based cryptosystems.

  - Finalist: Classic McEliece [ABC+20].
  - Alternates: BIKE [ABB+20] and HQC [MAB+20].

- Supersingular elliptic curve isogeny cryptosystems.

  - Finalists: None.
  - Alternate: SIKE [JAC+20].

NIST planned to announce the algorithms for standardization at the end of March 2022 and is still undergoing some legal and procedural steps for the announcement[1].

We focus on the lattice-based cryptosystems Dilithium, Kyber, NTRU, NTRU Prime, and Saber. In particular, we are interested in the dominating operations – polynomial multiplications and their derived operations.

Two ends of platforms are chosen for realizations. For low-end processors, the ARM Cortex-M3 and Cortex-M4 are chosen. For high-end processors, the ARM Cortex-A72 is chosen. Cortex-M3 implements the Armv7-M instruction set architecture (ISA), Cortex-M4 implements the Armv7-M with Digital Signal Processing (DSP) extension (also called Armv7E-M), and Cortex-A72 implements the Armv8.0-A instruction set architecture.

NIST chose Cortex-M4 as one of the platforms for evaluating submitted cryptosystems. Cortex-M4 is one of the Central Processing Units (CPUs) in the Cortex-M series aiming for microcontrollers and embedded systems, which are widely deployed in consumer devices[2]. The most prominent features of Cortex-M4 are the one-cycle long multiplications and the Simple-Instruction-Multiple-Data (SIMD) technology provided by the DSP extension. Long multiplications are used in 32-bit NTTs and the DSP extension is used in 16-bit NTTs.

Cortex-M3 is also one of the CPUs in the Cortex-M series. The study of Cortex-M3 is to evaluate the practical impact of the absence of the prominent features of Cortex-M4. In particular,

- The DSP extension is not supported by Cortex-M3.

- The long multiplications are early-terminating on Cortex-M3 and can not be used for computing secret data.

The absence of the DSP extension impacts the 16-bit approach, and the non-applicability of long multiplications to confidential data impacts the 32-bit approach.

The Cortex-A series aim for application uses and are widely deployed in mobile devices. Platforms with Cortex-A cores often implement the ARM “big.LITTLE” distinguishing feature. This thesis chooses the “big” core Cortex-A72 for realization. Cortex-A72 implements the Armv8.0-A where Neon technology is supported. Neon is a SIMD technology providing various kinds of permutations, additions, subtractions, multiplications, and shift operations with a wide variety of data types.

[1] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/fvnhyQ25jUg/m/izNIg5BABwAJ

[2] https://www.arm.com/products/silicon-ip-cpu

doi:10.6342/NTU202200738

Cortex-A72 has an in-order frontend with a 3-way decoder and an out-of-order backend with eight pipelines. There are two pipelines `F0` and `F1` for SIMD support among the eight pipelines. Both `F0` and `F1` are capable of permutations, additions, and subtractions. However, only `F0` is capable of multiplications, and only `F1` is capable of shift operations. For polynomial multiplications, since the most extensively used instructions are additions, subtractions, and multiplications, realizations will be `F0`-bounded. A challenging task is to reduce the chances of dispatching additions and subtractions to `F0` to reduce the workload of `F0`.

1.2 Contributions

This thesis is based on two published works at TCHES 2022 and one submitted work. All the works were either published or conducted during the study of the master's program from September 2021 to June 2022.

1.2.1 Cortex-M3 and Cortex-M4

For Cortex-M3 and Cortex-M4 implementations, one published work and one submited work are selected.

> Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. Submitted in April 2022.

The first work targets the so-called "big by small" polynomial multiplications in NTRU and NTRU Prime. We propose dedicated radix-$(2, 3)$ butterflies supporting on-the-fly permutations for combinations of Good–Thomas FFT and vector–radix FFT. Comparing to the previous works by [ACC$^+$21] and [CHK$^+$21], the dedicated radix-$(2, 3)$ butterflies show that computations derived from radix-3 butterflies are worth implementing. To control the code size for the on-the-fly permutations, we present the Good–Thomas FFT as an isomorphism from a group algebra to a tensor product of associative algebras. In [ACC$^+$21] and [Ber01], Good–Thomas FFT was stated in a weaker form. We also improve all the non-radix-2 butterflies by extending the existence of subtraction in radix-2 butterflies to non-radix-2 butterflies. For a radix-r butterfly with $r \neq 2$, we replace $r - 1$ multiplications and one addition with $r - 2$ additions and one subtraction.

Finally, since no algebraic assumptions on the polynomial rings are made, with slight modifications, each of the convolutions in this work support at least one parameter of NTRU and one parameter of NTRU Prime. In particular, our size-1440 convolutions support `ntruhps2048677`, `ntruhrss701`, `ntrulpr653`, and `sntrup653`; our size-1536 convolutions support `ntruhps2048677`, `ntruhrss701`, `ntrulpr761`, and `sntrup761`; and our size-1728 convolutions support `ntruhps4096821`, `ntrulpr857`, and `sntrup857`.

Below are the contributions covered by this thesis:

- The proposed implementations, covered in Section 8.5.1.
- The proposed improvement of the näive computations of NTTs, covered in Sections 4.3.5 and 7.2.6.

doi:10.6342/NTU202200738

- The proposed dedicated radix-$(2, 3)$ butterflies, covered in Section 7.2.7.
- The proposed code size optimization of Good–Thomas FFT, covered in Section 8.2.3.

Additionally, this thesis includes the implementations for `ntrulpr1013`, `sntrup1013`, `ntrulpr1277`, and `sntrup1277`.

> Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127-151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292.
> Reference [ACC+22]. Full version available at https://eprint.iacr.org/2021/995.

The second work considers the implementations of Saber on Cortex-M3 and Cortex-M4. We focus on the dominating operations: the matrix-to-vector and vector-to-vector multiplications. Our implementations provide a flexible time-memory tradeoffs by (i) carefully analyzing the structure of matrix-to-vector and vector-to-vector multiplications, and (ii) proposing multi-moduli NTTs. Our speed-optimized implementations benefit from the structure of matrix-to-vector and vector-to-vector multiplications. Our stack-optimized implementations demonstrate how to mitigate the growth of memory usage due to the increases of precisions without a huge sacrifice of performance. We show that NTT-based multiplications are the most suitable approaches for Saber on Cortex-M3 and Cortex-M4. In particular, our stack-optimized implementations outperform all existing non-NTT-based multiplications, whether they are optimized solely for speed or memory usage.

On Cortex-M4, our 32-bit NTTs rely on the one-cycle long multiplications and our 16-bit NTTs rely on the DSP extension. Results show that the 32-bit NTT approach is faster than the 16-bit NTT appraoch. On Cortex-M3, things become quite different. Our 16-bit NTT approach is faster than the 32-bit NTT approach. Our results on Cortex-M3 demonstrate the impact of the absence or the non-applicability of the prominent features of Cortex-M4.

Finally, we improve the cyclic radix-2 NTTs on Cortex-M4. We propose the CT–GS butterflies as a disproof for the commonly believed optimality. CT–GS butterflies achieve a smaller amount of modular reductions by moving some twiddle factors to the earlier layers.

Below are the contributions covered by this thesis:

- The proposed implementations on Cortex-M4, covered in Section 8.5.2.
- The proposed implementations with 16-bit NTTs on Cortex-M3, covered in Section 8.6.1.
- The proposed thorough analysis of time-memory tradeoffs for Saber, covered in Section 8.3.3.
- The proposed stack optimization on Cortex-M4, covered in Section 8.4.
- The proposed CT–GS butterflies, covered in Section 7.2.5.

Note that the 32-bit NTT approach on Cortex-M3 is not covered by this thesis. The 32-bit NTT on Cortex-M3 is implemented by Amin Abdulrahman.

doi:10.6342/NTU202200738

1.2.2 Cortex-A72

Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221-224, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. Reference [BHK+22]. Full version available at https://eprint.iacr.org/2021/986.

The third work is about vectorizing NTTs with the Neon technology. We implement the NTTs in Dilithium, Kyber, and Saber on Cortex-A72. We introduce the signed Barrett multiplication and establish its correspondence to Montgomery multiplication. In the full version, we clarify that the unsigned Barrett multiplicaiton was already known in the literature [Har14]. We also introduce the "asymmetric multiplciation", an optimization when the results of incomplete NTTs are cached. The "asymmetric multiplication" does not rely on any algebraic assumptions of the coefficient rings. Our approach is closely related to the implementation of Ed25519 by [BDL+12]. [BDL+12] multiplied two field elements in $\mathbb{Z}_{2^{255}-19}$ essentialy as in $R[x]/\langle x^5 - 19\rangle$. The "asymmetric multiplication" furthers the idea to the matrix-to-vector multiplications and demonstrates the savings of multiplications. Finally, we introduce a scheduling policy for NTTs based on the asymmetric design of the SIMD pipelines on Cortex-A72.

Below are contributions covered by this thesis:

- The proposed implementations, covered in Section 8.7.
- The proposed "asymmetric multiplication", covered in Section 8.3.2.
- The proposed scheduling policy, covered in Section 7.2.4.

1.3 Other Works

This section lists other co-authored works that are not covered by this thesis. The summaries are added by myself.

1.3.1 Other Works During My Master's Program

- Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient Multiplication of Somewhat Small Integers using Number-Theoretic Transforms. To appear at *International Workshop on Security 2022.*
  - Summary: We challenge the belief of applying FFT-based integer multiplications only for integers of several tens of thousands of bits. We apply our NTT-based integer multiplications to the encryption and decryption of RSA-4096 on Cortex-M3 and Cortex-M55. Our results show that the critical point of NTT-based integer multiplications is 2048 bits.
  - Available at https://eprint.iacr.org/2022/439.
- Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Dann Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. To appear at *Applied Cryptography and Network Security 2022.*

doi:10.6342/NTU202200738

- Summary: This paper improves the NTT-based multiplications in Dilithium and Kyber on Cortex-M4. Our contributions are threefold: We first apply various known techniques of NTTs on Cortex-M4 to Dilithium and Kyber. Secondly, we introduce a faster 16-bit signed Barrett reduction for 16-bit NTTs. Thirdly, we apply the NTT-based multiplications defined over small moduli to the signature generation of Dilithium. We apply the 16-bit NTT for `dilithium3`, and the Fermat number transforms with modulus 257 to `dilithium2` and `dilithium5`.
- Reference [AHKS22].
- Available at https://eprint.iacr.org/2022/112.

1.3.2 Other Related Works Prior to My Master's Program

- Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159-188, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8791.
  - Summary: This paper demonstrates the superiority of NTT-based polynomial multiplications for Saber and NTRU on Cortex-M4 and Skylake.
  - Talk: https://www.youtube.com/watch?v=a9_-jhD2ZG0.
  - Slide: https://iacr.org/submit/files/slides/2021/ches/ches2021/30796/slides.pdf.
  - Reference [CHK+21].
  - Full version available at https://eprint.iacr.org/2020/1397.
- Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217-238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733.
  - Summary: This paper proposes mixed–radix FFTs and Good–Thomas FFT for NTRU Prime on Cortex-M4. For our mixed–radix FFTs, we propose two different approaches. The first one is based on the smoothness of the size of the NTT, and the second one is based on the Rader's FFT for handling NTTs of prime sizes.
  - Talk: https://youtube.com/watch?v=F95gXPfXrBA.
  - Slide: https://iacr.org/submit/files/slides/2021/ches/ches2021/30766/slides.pdf.
  - Reference [ACC+21].
  - Full version available at https://eprint.iacr.org/2020/1216.

1.4 Organization of this Thesis

This thesis consists of selected works and notes during the study. Contents are organized constructively. It is expected that one already knows or can easily pick up either the mathematical background or the knowledge of the platforms. However, it is not expected that one knows both the mathematical background and the platforms. Therefore, there are four chapters about the foundations, and they are for referential purposes. Chapters 3 and 4 go through the mathematical background, and Chapters 5 and 6 describe the target platforms. In Chapters 7 and 8, we explain in detail how to translate the mathematics in Chapters 3 and 4 into platform-dependent instructions recognized by our platforms. Finally, Chapter 9 presents the performance numbers.

1.5 Source Code Management

As a thesis concerning implementations, around 2% of the source code is about the writing of the thesis. Around 91.7% are assembly programs and 6.2% are C programs. Source code management is, therefore, a challenging task. The essential components in this study are the macros realizing various kinds of building blocks used in the NTTs. Macros scattering over the selected works are modified to maintain consistency. The toolkit developed during the study is also summarized (documentation under construction). For achieving maximum support with minimum requirements, the toolkit consists of high-order functions with generic typings in C. All the implementations and the toolkit are available at https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A.

doi:10.6342/NTU202200738

Chapter 2

Schemes

This chapter introduces the schemes targeted by this thesis. Only the parts relevant to this thesis are included and the rest is omitted. No contribution is claimed in this chapter.

Section 2.1 introduces the digital signature Dilithium. Next, four key encapsulation mechanisms are introduced. Section 2.2 introduces Kyber, Section 2.3 introduces NTRU, Section 2.4 introduces NTRU Prime, and Section 2.5 introduces Saber.

2.1 Dilithium

Dilithium is a lattice-based digital signature submitted to NIST's PQC Standardization [NIS]. The hardness of Dilithium is based on Module Small Integer Solutions (M-SIS) and Module Learning with Errors (M-LWE) problems. The ring in Dilithium is $\mathbb{Z}_{8380417}[x]/\langle x^{256}+1\rangle$. We omit the specification and refer to [ABD$^+$20a] for more details. We focus on the NTT-related operations, including the `NTT`and `NTT`$^{-1}$ defined in the specification, the multiplication of a $k \times \ell$ matrix by an $\ell \times 1$ vector, and the 1×1 base multiplication of two vectors. The parameters k and ℓ vary from security levels and are summarized in Table 2.1.

Table 2.1: Dilithium parameter sets.

Parameter set	k	ℓ
`dilithium2`	4	4
`dilithium3`	6	5
`dilithium5`	8	7

2.2 Kyber

Kyber is a lattice-based key encapsulation mechanism (KEM) submitted to NIST's PQC Standardization [NIS]. The hardness of Kyber is based on the M-LWE problem. The ring in Kyber is $\mathbb{Z}_{3329}[x]/\langle x^{256}+1\rangle$. We omit the specification and refer to [ABD$^+$20b] for more details. We focus on the NTT-related operations, including the `NTT`and `NTT`$^{-1}$ defined in the specification, the multiplication of an $\ell \times \ell$

doi:10.6342/NTU202200738

matrix by an $\ell \times 1$ vector, the inner product of two $\ell \times 1$ vectors, and the 2×2 base multiplication of two vectors. The parameter ℓ varies from security levels and is summarized in Table 2.2.

Table 2.2: Kyber parameter sets.

Parameter set	ℓ
kyber512	2
kyber768	3
kyber1024	4

2.3 NTRU

NTRU is a lattice-based KEM submitted to NIST's PQC Standardization [NIS]. The hardness of NTRU is based on the NTRU problem. The rings in NTRU are $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$, $\mathbb{Z}_3[x]/\langle x^{n-1} + x^{n-2} + \cdots + 1\rangle$, and $\mathbb{Z}_q[x]/\langle x^{n-1} + x^{n-2} + \cdots + 1\rangle$ where q is a power of two and n is a prime. We omit the specification and refer to [CDH+20] for more details. We focus on the so-called "big by small" polynomial multiplication. A polynomial multiplication is called "big by small" if the coefficients of one of the multiplicands are bounded by a small predefined constant. In NTRU, one of the multiplicands of a "big by small" polynomial multiplication consists of coefficients in $[-1, 1]$. The parameters n and q vary from security levels and are summarized in Table 2.3.

Table 2.3: NTRU parameter sets.

Parameter set	q	n
ntruhps2048509	2048	509
ntruhps2048677	2048	677
ntruhrss701	8192	701
ntruhps4096821	4096	821
ntruhps40961229	4096	1229
ntruhrss1373	8192	1373

2.4 NTRU Prime

NTRU Prime is a set of lattice-based KEMs submitted to NIST's PQC Standardization [NIS]. There are two KEMs in NTRU Prime: Streamlined NTRU Prime and NTRU LPRime. Streamlined NTRU Prime is based on the NTRU problem and NTRU LPRime is based on the Ring Learning with Errors (R-LWE) problem. Comparing to NTRU, the polynomial ring is a finite field $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ with prime-degree large Galois group and inert-modulus [BBC+20]. We omit the specification and refer to [BBC+20] for more details. We focus on the "big by small" polynomial multiplication. In NTRU Prime, one of the multiplicands of a "big by

doi:10.6342/NTU202200738

small” polynomial multiplication consists of coefficients in $[-1, 1]$. The parameters p and q vary from security levels and are summarized in Table 2.3.

Table 2.4: NTRU Prime parameter sets.

Parameter set	p	q
`ntrulpr653/sntrup653`	653	4621
`ntrulpr761/sntrup761`	761	4591
`ntrulpr857/sntrup857`	857	5167
`ntrulpr953/sntrup953`	953	6343
`ntrulpr1013/sntrup1013`	1013	7177
`ntrulpr1277/sntrup1277`	1277	7879

2.5 Saber

Saber is a lattice-based key encapsulation mechanism (KEM) submitted to NIST’s PQC Standardization [NIS]. The hardness of Saber is based on the Module Learning with Rounding (M-LWR) problem. The ring in Saber is $\mathbb{Z}_{8192}[x]/\langle x^{256} + 1\rangle$. We omit the specification and refer to [DKRV20] for more details. We focus on the `MatrixVectorMul` and `InnerProd` in Saber. `MatrixVectorMul` multiplies an $l \times l$ matrix by an $l \times 1$ vector and `InnerProd` computes the inner product of two $l \times 1$ vectors. In Saber, one of the multiplicands consists of coefficients in $[-\frac{\mu}{2}, \frac{\mu}{2}]$. The parameters l and μ vary from security levels and are summarized in Table 2.5.

Table 2.5: Saber parameter sets.

Parameter set	l	μ
`lightsaber`	2	10
`saber`	3	8
`firesaber`	4	6

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Chapter 3

Mathematical Background

This chapter introduces the mathematical background, including sets in Section 3.1, basic algebras in Section 3.2, and abstract algebra in Section 3.3. All the contents are for referential purposes. Readers familiar with algebras can skip the entire chapter. No contribution is claimed in this chapter.

3.1 Sets

3.1.1 Sets, Subsets, Power Sets, Cartesian Products

We choose Zermelo–Fraenkel set theory with the axiom of choice for this thesis. Also, we use set-builder notation when applying axiom schema of specification. We first review some basic constructions of sets – subsets, the power set and product of sets. Subsets and the power set will be used for characterizing the behavior of a map, which will be introduced in the next section. The product of sets is the foundation for understanding various algorithmic efficiency and implementation tradeoffs of maps.

Definition 1 (Subset)**.** Let S be a set. A set S' is called a subset of S, denoted as $S' \subseteq S$, if:

$$\forall s' \in S', s \in S.$$

Definition 2 (Power set)**.** Let S be a set. The power set of S, denoted as $\mathcal{P}(S)$, is defined as:

$$\mathcal{P}(S) \coloneqq \{S' | S' \subseteq S\}.$$

Definition 3 (Cartesian product)**.** Let S_0 and S_1 be sets. The Cartesian product of S_0 and S_1, denoted as $S_0 \times S_1$, is the set defined as:

$$S_0 \times S_1 \coloneqq \{(s_0, s_1) | s_0 \in S_0, s_1 \in S_1\}.$$

For sets S_0, S_1, and S_2, we adopt the convention that the product is associative in the sense that $(S_0 \times S_1) \times S_2 = S_0 \times (S_1 \times S_2)$. In general, given a positive integer n, we recognize the n-fold Cartesian product as the following.

Definition 4 (n-fold Cartesian product)**.** Let n be a positive integer, and $S_0, \ldots, S_{n-1}$ be sets. The n-fold Cartesian product of $S_0, \ldots, S_{n-1}$, denoted as $S_0 \times \cdots \times S_{n-1}$ or $\prod_{i=0}^{n-1} S_i$, is the set defined as:

$$\prod_{i=0}^{n-1} S_i \coloneqq \{(s_0, \ldots, s_{n-1}) | s_0 \in S_0, \ldots, s_{n-1} \in S_{n-1}\}.$$

doi:10.6342/NTU202200738

Since there is no confusion, we also say the Cartesian product for the n-fold Cartesian product. Additionally, for the case $S_0 = \cdots = S_{n-1} = S$, we denote the Cartesian product as S^n.

3.1.2 Maps, Compositions, Cartesian Products

We now review the concept of maps. In particular, the composition of maps and Cartesian products of maps. Compositions of maps are used to illustrate optimized implementations of maps in a step-by-step fashion, Cartesian product of maps is the basic construct for applying several maps to the corresponding Cartesian product of sets in parallel.

Definition 5 (Map)**.** Let S and X be sets. We call Φ a map from S to X, denoted as $\Phi : S \to X$, if Φ is a pairing of elements in S and X assigning every s in S a $\Phi(s)$ in X.

Definition 6 (Image of a map)**.** Let $\Phi : S \to X$ be a map. The image of Φ, denoted as $\Phi(S)$, is the subset of X defined as:

$$\Phi(S) \coloneqq \{x | \exists s \in S, x = \Phi(s)\}.$$

For a subset of a set, we naturally introduce the restriction of a map as follows.

Definition 7 (Restriction of a map)**.** Let $\Phi : S \to X$ be a map and S' be a subset of S. The restriction of Φ on S', denoted as $\Phi_{|S'}$, is the map from S' to X defined by:

$$\forall s' \in S', \Phi_{|S'}(s') = \Phi(s').$$

Next, we investigate some correspondences that can be established by a map – injectivity, surjectivity, and bijectivity.

Definition 8 (Injective map)**.** Let $\Phi : S \to X$ be a map. Φ is called injective if:

$$\forall s, s' \in S, (\Phi(s) = \Phi(s')) \longrightarrow (s = s').$$

Definition 9 (Surjective map)**.** Let $\Phi : S \to X$ be a map. Φ is called surjective if:

$$X = \Phi(S).$$

Definition 10 (Bijective map)**.** Let $\Phi : S \to X$ be a map. Φ is called bijective if it is injective and surjective.

If a map is injective then we call it an injective map. Similarly, we call it a surjective map or a bijective map if the required correspondences are established. Among the correspondences, bijectivity is the core of our context. In particular, composing several bijective maps gives rise to a bijective map implying no correspondences are lost while applying the maps. The composition of maps is defined as follows.

Definition 11 (Composition of maps)**.** Let $\Phi_0 : S \to X$ and $\Phi_1 : X \to H$ be maps. The composition of maps Φ_1 and Φ_0, denoted as $\Phi_1 \circ \Phi_0$, is the map from S to H defined by:

$$\forall s \in S_0, (\Phi_1 \circ \Phi_0)(s) = \Phi_1(\Phi_0(s)).$$

doi:10.6342/NTU202200738

To formally characterize the losslessness of correspondences, we illustrate with the concept of identity maps and the two-side inverse.

Definition 12 (Identity map)**.** The identity map id_S of S is the map from S to itself defined by:

$$\forall s \in S, \mathrm{id}_S(s) = s.$$

Definition 13 (Inverse of a bijective map)**.** Let $\Phi : S \to X$ be a bijective map. The (two-side) inverse of Φ, denoted as Φ^{-1}, is the map from X to S defined by:

$$\begin{aligned} &\Phi^{-1} \circ \Phi = \mathrm{id}_S. \quad \wedge \\ &\Phi \circ \Phi^{-1} = \mathrm{id}_T. \end{aligned}$$

Next, we define the Cartesian product of maps, which aligns with the Cartesian product of sets.

Definition 14 (Cartesian product of maps)**.** Let $\Phi_0 : S_0 \to T_0$ and $\Phi_1 : S_1 \to T_1$ be maps. The Cartesian product of Φ_0 and Φ_1, denoted as $\Phi_0 \times \Phi_1$, is the map from $S_0 \times S_1$ to $T_0 \times T_1$ defined by:

$$\forall (s_0, s_1) \in S_0 \times S_1, (\Phi_0 \times \Phi_1)(s_0, s_1) = (\Phi_0(s_0), \Phi_1(s_1)).$$

For convenience, we abbreviate $\prod_{i=0}^{n-1} \Phi$ as Φ^n, which also aligns with mapping S^n to T^n.

3.1.3 Equivalence Relations, Partitions, Quotient sets

Before going through how a map can be factorized, we review the concept of equivalence relations and partitions, and most importantly, the quotient set in illustrating the factorization of maps.

First we introduce relations and equivalence relations.

Definition 15 (Relation)**.** Let S be a set and n be a positive integer. An n-ary relation R on S is defined as any subset of S^n. In other words, the only condition for R being an n-ary relation is:

$$R \subseteq S^n.$$

Conventionally, we say R is a binary relation for $n = 2$.

Definition 16 (Equivalence relation)**.** Let E be a binary relation on S. We call E an equivalence relation on S if:

$$\begin{aligned} &\forall s \in S, (s, s) \in E. && \wedge \\ &\forall (s_0, s_1) \in E, (s_1, s_0) \in E. && \wedge \\ &\forall (s_0, s_1), (s_1, s_2) \in E, (s_0, s_2) \in E. && \end{aligned}$$

Definition 17 (Partition)**.** Let $\mathcal{P}(S)$ be the power set of S. We call $\pi \subseteq \mathcal{P}(S)$ a partition of S if:

$$\begin{aligned} &\bigcup_{S' \in \pi} S' \supseteq S. && \wedge \\ &\forall S_0, S_1 \in \pi, S_0 \neq S_1 \longrightarrow S_0 \cap S_1 = \emptyset. && \end{aligned}$$

doi:10.6342/NTU202200738

Essentially, equivalence relation and partition are dual to each other. Rigorously speaking, any equivalence relation gives raise to a partition and any partition gives raise to an equivalence relation. Furthermore, successive derivation from the derived one gives the deriving one.

We first show how an equivalence relation induces a partition. Given an equivalence relation, we define equivalence classes and the quotient set as follows.

Definition 18 (Equivalence class)**.** Let E be an equivalence relation on S and s be an element of S. The equivalence class $\bar{s}$ is defined as:

$$\bar{s} := \{s' | (s, s') \in E\} .$$

Definition 19 (Quotient set)**.** Let E be an equivalence relation on S. The quotient set S/E is the partition of S defined as:

$$S/E := \{\bar{s} | s \in S\} .$$

On the other hand, for a given partition, we can define an equivalence relation as follows.

Definition 20 (Partition-induced equivalence relation)**.** Let π be a partition of S. Then the π-induced equivalence relation E_π is defined as follows:

$$E_\pi := \{(s, s') | \bar{s} = \bar{s'}\} .$$

Clearly, for a partition π of S, $\pi = S/E_\pi$. Conversely, for an equivalence relation E on S, $E = E_{S/E}$. Now we see why they are just dual to each other.

3.1.4 Factorization of Maps

Finally, we review how a map can be factorized. For a map $\Phi : S \to X$, it induces an equivalence relation, a quotient set, and a natural map as follows. After identifying the natural map induced by Φ, we can then factorize Φ as a composition of an injective map $\bar{\Phi}$ and the natural map. Furthermore, if Φ is surjective, then $\bar{\Phi}$ is bijective. In this thesis, we study various kinds of $\bar{\Phi}$ defined on rings and in general, on associative algebras.

Definition 21 (Map-induced equivalence relation)**.** Let $\Phi : S \to X$ be a map. The map-induced equivalence relation E_Φ on S is defined as:

$$\{(s, s') | \Phi(s) = \Phi(s')\} .$$

Definition 22 (Map-induced quotient set)**.** Let $\Phi : S \to X$ be a map and E_Φ be the equivalence relation induced by Φ. Then we also say the quotient set S/E_Φ is induced by Φ.

Definition 23 (Map-induced natural map)**.** Let $\Phi : S \to X$ be a map inducing the equivalence relation E_Φ and the quotient set S/E_Φ. The natural map v_Φ induced by Φ is the surjective map from S to S/E_Φ defined by:

$$\forall s \in S, s \in v_\Phi(s).$$

doi:10.6342/NTU202200738

Theorem 1 (Factoring a map [Jac12])**.** [Jac12, Section 0.3] Let $\Phi : S \to X$ be a map inducing the equivalence relation E_Φ, the quotient set S/E_Φ, and the natural map v_Φ. Then there is an injective map $\bar{\Phi}$ from S/E_Φ to X satisfying:

$$\Phi = \bar{\Phi} \circ v_\Phi.$$

Furthermore, if Φ is surjective then $\bar{\Phi}$ is bijective.

Proof. Trivial. □

3.2 Basic Algebra

Monoids, Groups, Submonoids, Subgroups, Cartesian Product

Definition 24 (Monoid)**.** A monoid is a triple $(M, \cdot, 1)$ where

i M is a set.

ii $\cdot$ is an associative map from M^2 to M in the sense that

$$\forall a, b, c \in M, (a \cdot (b \cdot c)) = ((a \cdot b) \cdot c)\,.$$

iii $1 \in M$ is an element satisfying

$$\forall a \in M, (1 \cdot a) = a = (a \cdot 1).$$

When there is no confusion, we simply write ab for $a \cdot b$.

Definition 25 (Group)**.** A group is a triple $(G, \cdot, 1)$ where

i $(G, \cdot, 1)$ is a monoid.

ii $\forall g \in G \exists g^{-1} \in G, gg^{-1} = 1 = g^{-1}g.$

If for any elements a and b of G, $ab = ba$, then we use the notation $+$ for $\cdot$ instead and call G an abelian group. When the context is clear, we simply say M is a monoid and G is a group.

Definition 26 (Submonoid)**.** Let $(M, \cdot_M, 1_M)$ be a monoid. A monoid $(M', \cdot_{M'}, 1_{M'})$ is called a submonoid of $(M, \cdot_M, 1_M)$ if:

$$\begin{aligned} &M' \subseteq M. && \wedge \\ &\cdot_{M'} = (\cdot_M)_{|M'}. && \wedge \\ &1'_M = 1_M. \end{aligned}$$

Definition 27 (Subgroup)**.** Let G be a group with a submonoid G'. If G' is also a group, then it is also called a subgroup of G)

Conventionally, we write $M' \leq M$ and $G' \leq G$ for submonoid and subgroup relations.

doi:10.6342/NTU202200738

Definition 28 (Cartesian product of monoids)**.** Let $(M_0, \cdot_{M_0}, 1_{M_0})$ and $(M_1, \cdot_{M_1}, 1_{M_1})$ be monoids. The Cartesian product of monoids $(M_0, \cdot_{M_0}, 1_{M_0})$ and $(M_1, \cdot_{M_1}, 1_{M_1})$, denoted as $(M_0 \times M_1, \cdot_{M_0 \times M_1}, 1_{M_0 \times M_1})$ or simply $M_0 \times M_1$, is the monoid defined as:

$$\begin{aligned} &\cdot_{M_0 \times M_1} = (\cdot_{M_0} \times \cdot_{M_1}) \circ T_{2,2}. && \wedge \\ &1_{M_0 \times M_1} = (1_{M_0}, 1_{M_1}). \end{aligned}$$

Definition 29 (Cartesian product of groups)**.** Let G_0 and G_1 be groups. The Cartesian product of groups G_0 and G_1, denoted as $G_0 \times G_1$, is the group defined by the product monoid $G_0 \times G_1$.

3.2.1 Monoid and Group Homomorphisms

Definition 30 (Monoid homomorphism)**.** Let M_0 and M_1 be monoids and $\Phi : M_0 \to M_1$ be a map. Then Φ is called a monoid homomorphism from M_0 to M_1 if:

$$\begin{aligned} &\Phi(1_{M_0}) = 1_{M_1}. && \wedge \\ &\forall a, b \in M_0, \Phi(ab) = \Phi(a)\Phi(b). \end{aligned}$$

Definition 31 (Group homomorphism)**.** Let G_0 and G_1 be groups and $\Phi : G_0 \to G_1$ be a map. Then Φ is called a group homomorphism from G_0 to G_1 if it is a monoid homomorphism from G_0 to G_1.

Definition 32 (Monoid congruence)**.** Let M be a monoid and E be a congruence on M. Then E is called a monoid congruence on M if:

$$\forall (a, a'), (b, b') \in E, (ab, a'b') \in E.$$

Definition 33 (Group congruence)**.** Let G be a group and E be a congruence on G. Then E is called a group congruence on G if it is a monoid congruence on G.

3.2.2 Rings, Cartesian Product

Definition 34 (Ring)**.** A ring is a tuple $(R, +, \cdot, 0, 1)$ where

1. $(R, +, 0)$ is an abelian group.
2. $(R, \cdot, 1)$ is a monoid.
3. $\forall a, b, c \in R, \begin{cases} a(b + c) = ab + ac, \\ (b + c)a = ba + ca. \end{cases}$

Definition 35 (Idempotent element)**.** Let $(R, +, \cdot, 0, 1)$ be a ring and e an element in R. We call e an idempotent element if:

$$e^2 = e.$$

Corollary 1. Let $(R, +, \cdot, 0, 1)$ be a ring. Then 0 and 1 are idempotent elements.

Proof. Trivial. □

doi:10.6342/NTU202200738

Definition 36 (Commutative ring)**.** Let $(R, +, \cdot, 0, 1)$ be a ring. We call R a commutative ring if:

$$\forall a, b \in R, ab = ba.$$

For the remainder of this thesis, we use 'ring' for a commutative ring.

Definition 37 (Characteristic)**.** Let $(R, +, \cdot, 0, 1)$ be a ring. The characteristic of R, denoted as $\mathrm{char}(R)$, is the smallest number $n \in \mathbb{N}$ satisfying:

$$\underbrace{1 + 1 + \cdots + 1}_{n} = 0.$$

If there is no such n, then $\mathrm{char}(R)$ is defined as $0 \in \mathbb{Z}$.

Definition 38 (Cartesian product of rings)**.** Let R_0 and R_1 be rings. The product ring $R_0 \times R_1$ is the ring defined as:

$$(R_0 \times R_1, +_{R_0 \times R_1}, \cdot_{R_0 \times R_1}, 0_{R_0 \times R_1}, 1_{R_0 \times R_1}).$$

3.2.3 Ring Homomorphisms

Definition 39 (Ring homomorphism)**.** Let R_0 and R_1 be rings and $\Phi : R_0 \to R_1$ be a map. Φ is called a ring homomorphism if it is a group homomorphism from $(R_0, +_{R_0}, 0_{R_0})$ to $(R_1, +_{R_1}, 0_{R_1})$ and a monoid homomorphism from $(R_0, \cdot_{R_0}, 1_{R_0})$ to $(R_1, \cdot_{R_1}, 1_{R_1})$.

Definition 40 (Monomorphism)**.** Let $\Phi : R_0 \to R_1$ be a ring homomorphism. Φ is called a monomorphism if it is injective.

Definition 41 (Epimorphism)**.** Let $\Phi : R_0 \to R_1$ be a ring homomorphism. Φ is called an epimorphism if it is surjective.

Definition 42 (Isomorphism)**.** Let $\Phi : R_0 \to R_1$ be a ring homomorphism. Φ is called an isomorphism if it is bijective. Furthermore, we denote $R_0 \cong R_1$ if there is an isomorphism from R_0 to R_1.

As a side note, *monomorphism*, *epimorphisms*, and *isomorphisms* are also used in the context of category theory where ring homomorphisms are called *morphisms between two objects in the category of rings.*

3.2.4 Ring Congruences, Ideals

Definition 43 (Ring congruence)**.** Let R be a ring and E be an equivalence relation on R. Then E is called a ring congruence if it is a group congruence on $(R, +, 0)$ and a monoid congruence on $(R, \cdot, 1)$.

Definition 44 (Ideal)**.** Let R be a ring and I be a subgroup of $(R, +, 0)$. We call I an ideal if:

$$\forall a \in R \forall b \in I, (ab \in I) \wedge (ba \in I).$$

For convenience, for a ring element $z \in R$, the ideal $\{rz | r \in R\}$ is denoted as $\langle z \rangle$.

doi:10.6342/NTU202200738

Corollary 2. Let I be an ideal of R. Then I induces a ring congruence E_I on R defined as:

$$\{(a, a')|a - a' \in I\}.$$

Proof. Trivial. □

Conventionally, we denote r mod I for the equivalence class of r under E_I.

Corollary 3. Let E be a ring congruence on R. Then E induces an ideal I_E of R defined as:

$$\{a|(0, a) \in E\}.$$

Proof. Trivial. □

3.2.5 Quotient Rings

Definition 45 (Quotient ring)**.** Let R be a ring, I be an ideal of R, and E_I be the induced congruence. The quotient ring R/I is the partition R/E_I accompanied with the ring operations $(+_{R/I}, \cdot_{R/I})$ defined as:

$$\begin{aligned} \forall r_0, r_1 \in R, (r_0 \bmod I) +_{R/I} (r_1 \bmod I) &\coloneqq (r_0 + r_1) \bmod I. \\ (r_0 \bmod I) \cdot_{R/I} (r_1 \bmod I) &\coloneqq (r_0 r_1) \bmod I. \end{aligned}$$

and $(0_{R/I}, 1_{R/I})$ defined as:

$$\begin{aligned} 0_{R/I} &\coloneqq 0 \bmod I. \\ 1_{R/I} &\coloneqq 1 \bmod I. \end{aligned}$$

3.2.6 Kernel, Factorization of Ring Homomorphisms

Definition 46 (Kernel of a homomorphism)**.** Let $\Phi : R_0 \to R_1$ be a ring homomorphism. The kernel of Φ, denoted as $\ker(\Phi)$, is the ideal of R_0 defined as:

$$\{a|\Phi(a) = 0\}.$$

Theorem 2 ([Jac12])**.** [Jac12, Fundamental theorem of homomorphisms of rings, Section 2.7] Let $\Phi : R_0 \to R_1$ be a ring homomorphism and v_Φ be the natural map from R_0 to $R/\ker(\Phi)$. Then there is a ring monomorphism $\bar{\Phi}$ from $R_0/\ker(\Phi)$ to R_1 satisfying:

$$\Phi = \bar{\Phi} \circ v_\Phi.$$

Furthermore, if Φ is an epimorphism then $\bar{\Phi}$ is an isomorphism.

Proof. Trivial. □

3.2.7 Examples

Example 1 (Integer ring)**.** The set of integers $\mathbb{Z}$ can be regarded as a ring where $(+, \cdot, 0, 1)$ are all the ones defined on integers.

Example 2 (Quotient of the integer ring)**.** Every integer m gives raise to the ideal $m\mathbb{Z} \coloneqq \{mz|z \in \mathbb{Z}\}$ of $\mathbb{Z}$ and the quotient ring $\mathbb{Z}_m \coloneqq \mathbb{Z}/m\mathbb{Z}$.

doi:10.6342/NTU202200738

Definition 47 (Polynomial ring)**.** Let R be a ring and x be an indeterminate. The polynomial ring $R[x]$ is the ring where each element is a sequence (r_i) of elements in R such that only finitely many of r_i's are non-zero. The ring operations $(+, \cdot)$ are defined as:

$$\begin{aligned} (a_i) + (b_i) &\coloneqq (a_i + b_i). \\ (a_i) \cdot (b_i) &\coloneqq \left(\sum_{j=0}^{i} a_j b_{i-j} \right). \end{aligned}$$

with $0 \coloneqq (0, 0, \dots)$ and $1 \coloneqq (1, 0, \dots)$.

Example 3. Let $R[x]$ be a polynomial ring, n be a positive integer, and ξ be an element in R. Then R^n can be regarded as the ring $R[x]/\langle x^n - \xi \rangle$ where the ring operations are:

$$\begin{aligned} (a_i) + (b_i) &\coloneqq (a_i + b_i). \\ (a_i) \cdot_{R[x]/\langle x^n - \xi \rangle} (b_i) &\coloneqq \left(\left(\sum_{j=0}^{i} a_j b_{i-j} \right) + \xi \left(\sum_{j=i+1}^{n-1} a_j b_{n+i-j} \right) \right). \end{aligned}$$

3.3 Abstract Algebra

This section is about the definitions of modules, associative algebras, and tensor products. Starting from this section, we use the word “algebra” for an associative algebra over a (commutative) ring. The goal of this section is to fix the foundations of multi-dimensional FFT as algebra isomorphisms between algebras. This level of abstraction is intentional to explain various implementation considerations. As we are asking very little for the well-definedness, we have more flexibility on optimizing the target polynomial multiplications.

3.3.1 Modules, Tensor Product of Modules

A module is a pairing of an abelian group and a scalar multiplication by ring elements. Personally, I don't like to say modules are generalizations of vector spaces (although this is true by definition), because this requires one to have a good understanding of vector spaces and I'm not such a person. We will go through the definition straightforwardly without explaining what vector spaces are.

Definition 48 (Left module)**.** Let R be a (possibly non-commutative) ring. A left R-module ${}_R M$ is a tuple

$$((M, +_M, 0_M), (R, +_R, \cdot_R, 0_R, 1_R), \cdot_{_R M})$$

where:

- $(M, +_M, 0_M)$ is an abelian group.
- $(R, +_R, \cdot_R, 0_R, 1_R)$ is a ring.
- $\cdot_{_R M}$ is a map from $R \times M$ to M (scalar multiplication) satisfying:

doi:10.6342/NTU202200738

- Distributivity of scalar multiplication:

$$\forall r \in R \forall x, y \in M, r \cdot_{RM} (x +_M y) = (r \cdot_{RM} x) +_M (r \cdot_{RM} y).$$

- Preservation of identity:

$$1_R \cdot_{RM} x = x.$$

- Compatibility with ring operations:

$$\forall r, s \in R \forall x \in M, \begin{cases} (r \cdot_R s) \cdot_{RM} x = r \cdot_{RM} (s \cdot_{RM} x). \\ (r +_R s) \cdot_{RM} x = (r \cdot_{RM} x) +_M (s \cdot_{RM} x). \end{cases}$$

Definition 49 (Right module)**.** Let R be a (possibly non-commutative) ring. A right R-module M_R is a tuple

$$((M, +_M, 0_M), (R, +_R, \cdot_R, 0_R, 1_R), \cdot_{M_R})$$

where the only differences with left module are that $\cdot_{M_R}$ is a map from $M \times R$ to M and the defining conditions are about right multiplications.

Conventionally, we say M is a left R-module or right R-module when the conditions are satisfied.

Definition 50 (Opposite ring)**.** Let R be a (possibly non-commutative) ring. The opposite ring R^{op} is the ring R except the multiplication is replaced with $\cdot_{R^{op}}$ defined as:

$$\forall r, s \in R^{op}, r \cdot_{R^{op}} s := s \cdot_R r.$$

Corollary 4 ([Jac12])**.** [Jac12, Section 3.2] Let R be a (possibly non-commutative) ring. A left (right) R-module can be identified as a right (left) R^{op}-module.

Proof. Trivial. □

If R is a commutative ring, then left R-modules can be identified as right R-modules. Therefore, we simply say R-modules. We only discuss R-modules for the rest of this thesis.

Definition 51 (Module homomorphism)**.** Let M and N be R-modules and $\Phi : M \to N$ be a map. Φ is called a module homomorphism if the following is satisfied:

$$\forall r_0, r_1 \in R, \forall m_0, m_1 \in M, \Phi(r_0 \cdot_{RM} m_0 + r_1 \cdot_{RM} m_1) = r_0 \cdot_{RN} \Phi(m_0) + r_1 \cdot_{RN} \Phi(m_1).$$

For a module homomorphism, we call it monomorphism, epimorphism, or isomorphism if it is injective, surjective, or bijective.

Next, we review how tensor products of modules are defined. We define the tensor product of two R-modules M and N as the quotient of the free space $F(M \times N)$ by the relation realizing an R-balanced product.

doi:10.6342/NTU202200738

Definition 52 (Tensor product of modules)**.** Let M and N be R-modules. The tensor product $M \otimes N$ is the R-module defined as:

$$F(M \times N)/\sim$$

where $F(M \times N)$ is the free space consisting of all elements of the form:

$$\sum_{i=0}^{k-1} r_k(m_k, n_k)$$

for some $k \in \mathbb{N}, r_k \in R, m_k \in M, n_k \in N$, and $\sim$ is the equivalence defined by:

- $\forall m_0, m_1 \in M \forall n_0, n_1 \in N$,

$$(m_0 + m_1, n_0 + n_1) \sim (m_0, n_0) + (m_0, n_1) + (m_1, n_0) + (m_1, n_1).$$

- $\forall r \in R \forall m \in M \forall n \in N$,

$$(rm, n) \sim (m, rn).$$

For $m \in M$ and $n \in N$, the corresponding equivalence class in $M \otimes N$ is denoted as $m \otimes n$.

Definition 53 (Tensor product of module homomorphisms)**.** Let $f_0 : M_0 \to N_0$ and $f_1 : M_1 \to N_1$ be module homomorphisms. The tensor product $f_0 \otimes f_1$ is defined as:

$$f_0 \otimes f_1 : \begin{cases} M_0 \otimes M_1 & \to N_0 \otimes N_1, \\ m_0 \otimes m_1 & \mapsto f_0(m_0) \otimes f_1(m_1). \end{cases}$$

3.3.2 Associative Algebras, Tensor Product of Associative Algebras

We next review the definitions of associative algebras and their tensor products. We will shortly identify a polynomial ring as an associative algebra over its coefficient ring.

Definition 54 (Associative algebra)**.** Let R and $\mathcal{R}$ be rings. An associative algebra is given as turning $\mathcal{R}$ into an R-module where scalar multiplication $\cdot_{R\mathcal{R}}$ is defined by:

$$\forall r \in R \forall \boldsymbol{r}_0, \boldsymbol{r}_1 \in \mathcal{R}, (r \cdot_{R\mathcal{R}} \boldsymbol{r}_0)\boldsymbol{r}_1 = r \cdot_{R\mathcal{R}} (\boldsymbol{r}_0\boldsymbol{r}_1) = \boldsymbol{r}_0(r \cdot_{R\mathcal{R}} \boldsymbol{r}_1).$$

Furthermore, we call $\mathcal{R}$ an R-algebra in this case.

Example 4 (Polynomial rings)**.** Let R be ring and $R[x]/\langle f(x) \rangle$ be a polynomial ring. Then $R[x]/\langle f(x) \rangle$ is an associative algebra over R.

Definition 55 (Algebra homomorphisms)**.** Let R be a ring, $\mathcal{A}_0$ and $\mathcal{A}_1$ be R-algebras and $\Phi : \mathcal{A}_0 \to \mathcal{A}_1$ be a map. Φ is called an algebra homomorphism if the following is satisfied:

$$\forall r \in R, \forall a_0, a_1 \in \mathcal{A}_0, \Phi(r \cdot_{R\mathcal{A}_0} a_0 + a_1) = r \cdot_{R\mathcal{A}_1} \Phi(a_0) + \Phi(a_1) \wedge \\ \Phi(a_0 a_1) = \Phi(a_0)\Phi(a_1).$$

doi:10.6342/NTU202200738

Similarly, we call Φ an monomorphism, epimorphism, or isomorphism if it is injective, surjective, or bijective.

Definition 56 (Tensor product of associative algebras)**.** Let R be a ring and $\mathcal{A}_0$ and $\mathcal{A}_1$ be R-algebras. The tensor product $\mathcal{A}_0 \otimes \mathcal{A}_1$ is the R-algebra defined by adjoining the ring multiplication $\cdot_{\mathcal{A}_0 \otimes \mathcal{A}_1}$ to the tensor product $\mathcal{A}_0 \otimes \mathcal{A}_1$ as R-modules. The ring multiplication $\cdot_{\mathcal{A}_0 \otimes \mathcal{A}_1}$ is defined as $\forall \boldsymbol{a}_{0,0} \otimes \boldsymbol{a}_{1,0}, \boldsymbol{a}_{0,1} \otimes \boldsymbol{a}_{1,1} \in \mathcal{A}_0 \otimes \mathcal{A}_1$,

$$\left(\boldsymbol{a}_{0,0} \otimes \boldsymbol{a}_{1,0}\right) \cdot_{\mathcal{A}_0 \otimes \mathcal{A}_1} \left(\boldsymbol{a}_{0,1} \otimes \boldsymbol{a}_{1,1}\right) \coloneqq \left(\boldsymbol{a}_{0,0} \cdot_{\mathcal{A}_0} \boldsymbol{a}_{0,1}\right) \otimes \left(\boldsymbol{a}_{1,0} \cdot_{\mathcal{A}_1} \boldsymbol{a}_{1,1}\right).$$

Furthermore, $\mathcal{A}_0 \otimes \mathcal{A}_1$ is an R-algebra.

Example 5 (Tensor product of polynomial rings)**.** Let R be a ring, and $\frac{R[x^{(0)}]}{\left\langle f_0(x^{(0)}) \right\rangle}$ and $\frac{R[x^{(1)}]}{\left\langle f_1(x^{(1)}) \right\rangle}$ be polynomial rings with the coefficient ring R. As in Example 4, both $\frac{R[x^{(0)}]}{\left\langle f_0(x^{(0)}) \right\rangle}$ and $\frac{R[x^{(1)}]}{\left\langle f_1(x^{(1)}) \right\rangle}$ are R-algebras. Therefore, we naturally have the tensor product $\frac{R[x^{(0)}]}{\left\langle f_0(x^{(0)}) \right\rangle} \otimes \frac{R[x^{(1)}]}{\left\langle f_1(x^{(1)}) \right\rangle}$ as an R-algebra. We also write it as

$$\frac{R[x^{(0)}, x^{(1)}]}{\left\langle f_0(x^{(0)}), f_1(x^{(1)}) \right\rangle}.$$

Definition 57 (Tensor product of associative algebra homomorphisms)**.** Let R be a ring, $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{B}_0$, and $\mathcal{B}_1$ be R-algebras, and $\Phi_0 : \mathcal{A}_0 \to \mathcal{B}_0$ and $\Phi_1 : \mathcal{A}_1 \to \mathcal{B}_1$ be algebra homomorphisms. The tensor product $\Phi_0 \otimes \Phi_1$ is defined as:

$$\Phi_0 \otimes \Phi_1 : \begin{cases} \mathcal{A}_0 \otimes \mathcal{A}_1 \to & \mathcal{B}_0 \otimes \mathcal{B}_1, \\ a_0 \otimes a_1 \mapsto & \Phi_0(a_0) \otimes \Phi_1(a_1). \end{cases}$$

doi:10.6342/NTU202200738

Chapter 4

Number-Theoretic Transforms

This chapter introduces number-theoretic transforms and fast Fourier transforms. Most of the material in this chapter is already known in the literature. I do not claim for contribution except for Section 4.3.5. Section 4.3.5 is a contribution of the submitted work:

- Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. Submitted in April 2022.

4.1 Convolutions

4.1.1 Convolutions

Theorem 3. Let n be a positive integer, η be an additive group automorphism on $\mathbb{Z}_n$. Then η induces an automorphism on $R[x]/\langle x^n - 1\rangle$ by sending x to $x^{\eta(1)}$. Furthermore, there are $\phi(n)$ many of such ηs.

Proof. It suffices to show that $\eta(\boldsymbol{a}(x)\boldsymbol{b}(x)) = \eta(\boldsymbol{a}(x))\eta(\boldsymbol{b}(x))$ as follows (we assume $b_{j'} = b_{j'+n}$ if $j' < 0$):

$$\begin{aligned}
\eta(\boldsymbol{a}(x)\boldsymbol{b}(x)) &= \eta\left(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_j b_{i-j} x^i\right) \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_j b_{i-j} x^{\eta(i)} \\
&= \sum_{\eta^{-1}(i)=0}^{n-1}\ \sum_{\eta^{-1}(j)=0}^{n-1} a_{\eta^{-1}(j)} b_{\eta^{-1}(i-j)} x^i \\
&= \left(\sum_{\eta^{-1}(i)=0}^{n-1} a_{\eta^{-1}(i)} x^i\right)\left(\sum_{\eta^{-1}(i)=0}^{n-1} b_{\eta^{-1}(i)} x^i\right) \\
&= \left(\sum_{i=0}^{n-1} a_i x^{\eta(i)}\right)\left(\sum_{i=0}^{n-1} b_i x^{\eta(i)}\right) \\
&= \eta(\boldsymbol{a}(x))\eta(\boldsymbol{b}(x))
\end{aligned}$$

doi:10.6342/NTU202200738

Since the additive group of $\mathbb{Z}_n$ is cyclic, any automorphisms can be characterized as mapping 1 to a generator. Clearly, the generators are numbers coprime to n and there are $\phi(n)$ many of them. □

4.1.2 Weighted Convolutions

Definition 58 ([CF94]). Let n be a positive integer and $\psi \in R$. The multiplication in the ring $R[x]/\langle x^n - \psi \rangle$ is also called a weighted convolution.

4.1.3 Toeplitz Matrices

A weighted convolution can be regarded as applying a square Toeplitz matrix to a vector. An $m \times n$ matrix A is called a Toeplitz matrix if $A_{i,j} = A_{i-h,j-h}$ for all $h \leq \min(i, j)$. In otherwords, A can be represented by the $(m + n - 1)$-tuple

$$(A_{m-1,0}, \dots, A_{0,0}, \dots, A_{0,n-1}).$$

We call $(A_{m-1,0}, \dots, A_{0,0}, \dots, A_{0,n-1})$ the array representation of A.

Let $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i x^i$ and $\boldsymbol{b}(x) = \sum_{i=0}^{n-1} b_i x^i$ be polynomials in the ring $R[x]/\langle x^n - \psi \rangle$. The computation of $\boldsymbol{c}(x) = \sum_{i=0}^{n-1} c_i x^i = \boldsymbol{a}(x)\boldsymbol{b}(x)$ can be written as

$$\sum_{i=0}^{n-1} \left(\left(\sum_{j=0}^{i} a_j b_{i-j} \right) + \psi \left(\sum_{j=i+1}^{n-1} a_j b_{n+i-j} \right) \right) x^i,$$

or equivalently,

$$\sum_{i=0}^{n-1} \left(\left(\sum_{j=0}^{i} a_j b_{i-j} \right) + \left(\sum_{j=i+1}^{n-1} a_j (\psi b_{n+i-j}) \right) \right) x^i.$$

We can also write $\boldsymbol{c}(x)$ as the column vector

$$\begin{pmatrix} c_0 \\ \vdots \\ c_{n-1} \end{pmatrix} = B \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

where B is the Toeplitz matrix with the array representation

$$(b_{n-1}, \dots, b_0, \psi b_{n-1}, \dots, \psi b_1).$$

Recent works by [IKPC20, IKPC22] showed that Toeplitz matrices admit practical implementations for NTRU and Saber on Cortex-M4. The idea is that for a constant k, an application of a Toeplitz matrix can be decomposed into applications of smaller Toeplitz matrices by applying a k-way formula with linear time overhead. We do not elaborate the ideas in this thesis and refer to [IKPC20, IKPC22] for more details. In this thesis, we will see an optimization based on the array representations of low-dimensional Toeplitz matrices for $\psi \neq \pm 1$ in Section 8.3.2.

doi:10.6342/NTU202200738

4.2 The Chinese Remainder Theorem for Rings

This section introduces two equivalent formulations of the Chinese remainder theorem (CRT) for rings. Factoring a ring via coprime ideals is the most commonly seen characterization. But we'll find that idempotent elements provide a more penetrable explanation for number-theoretic transforms, which is the core of this thesis.

Theorem 4 (Coprime ideals formulation for the CRT, [Bou89, Proposition 9, Section 8.11, Chapter I])**.** *Let R be a ring and $I_0, \dots, I_{d-1}$ be ideals satisfying $I_i + I_j = R$ for $i \neq j$, and define $I := \bigcap_{i=0}^{d-1} I_i$. Then there is an isomorphism Φ as follows:*

$$\Phi : \begin{cases} R/I & \to \prod_{i=0}^{d-1} R/I_i \ , \\ r \bmod I & \mapsto (r \bmod I_0, \dots, r \bmod I_{d-1}) \, . \end{cases}$$

Proof. Trivial. □

Theorem 5 (Idempotent elements formulation for the CRT, [Bou89, Proposition 10, Section 8.11, Chapter I])**.** *Let R be a ring and $e_0, \dots, e_{d-1}$ be orthogonal idempotent elements in R satisfying:*

$$\forall i, j, e_i e_j = \delta_{i,j} e_i \quad \wedge$$
$$\sum_{i=0}^{d-1} e_i = 1,$$

and let $I_i := \{(1 - e_i) r | r \in R\}$ be ideals of R. Then, $I := \bigcap_{i=0}^{d-1} I_i = \{0\}$ and there is an isomorphism Ψ as follows:

$$\Psi : \begin{cases} \prod_{i=0}^{d-1} R/I_i & \to R, \\ (a_0, \dots, a_{d-1}) & \mapsto \sum_{i=0}^{d-1} e_i a_i. \end{cases}$$

Proof. Trivial. □

Since $\left(\sum_{j=0}^{d-1} e_j a_j\right) \bmod I_i = e_i a_i \bmod I_i = a_i$, we have $\Psi = \Phi^{-1}$ if $I = 0$. Even if $I \neq 0$, we can still derive an isomorphism between R/I and $\prod_{i=0}^{d-1} R/I_i$ with idempotent elements as follows. First we observe that I/I is the zero of the quotient ring R/I and $\bigcap_{j=0}^{d-1} I_j/I = I/I$. Next, for all $j = 0, \dots, d-1$, we have

$$\frac{R/I}{I_j/I} \cong R/I_j \ .$$

Therefore, we can choose orthogonal idempotent elements $e_0, \dots, e_{d-1}$ from R/I realizing

$$\prod_{i=0}^{d-1} \frac{R/I}{I_i/I} \cong R/I \ ,$$

and derive $R/I \cong \prod_{i=0}^{d-1} \frac{R/I}{I_i/I} \cong \prod_{i=0}^{d-1} R/I_i$.

doi:10.6342/NTU202200738

4.3 Number-Theoretic Transforms

Number-theoretic transforms (NTTs) have great value in both theory and practice. In theory, NTTs are building blocks for improving large integer multiplications in $O\left(n \log n \log \log n\right)$ by [SS71], in $O\left(n \log n 2^{O(\log^* n)}\right)$ by [Für09], and recently in $O\left(n \log n\right)$ by [HvdH21], attacking the conjectured $\Omega\left(n \log n\right)$ lower bound by [SS71].

In this section, we explain how NTTs can be defined for a polynomial ring and characterize NTTs as CRT with the aid of idempotent elements. Furthermore, we investigate how polynomial multiplications can be implemented with NTTs.

Table 4.1: List of notations for NTTs.

Rings	
R	The target ring.
$\bar{\mathcal{R}}$	Base ring.
$\mathcal{R}$	Polynomial ring.
$\mathcal{R}^{(i)}$	Polynomial rings that are also treated as coefficient rings.
$\mathcal{R}_i$	Rings satisfying $\prod_i \mathcal{R}_i \cong \mathcal{R}$.
$\mathcal{R}_j^{(i)}$	Rings satisfying $\prod_j \mathcal{R}_j^{(i)} \cong \mathcal{R}^{(i)}$.
Ring elements	
ζ	An invertible element.
ω_n	A principal n-th root of unity.
ω	A principal n-th root of unity when the context is clear.
$[n]_q$	The q-analog of n.
Number-theoretic transforms	
NTT	Number-theoretic transform.
NTT	A specific number-theoretic transform.
NTT^{-1}	The inverse of a specific number-theoretic transform.
$\mathsf{NTT}_A \times \mathsf{NTT}_B$	The Cartesian product of NTT_A and NTT_B.

4.3.1 Principal n-th Root of Unity

We first established some facts that are crucial for defining NTTs.

Definition 59 (q-analog)**.** Let n be a positive integer and q be a symbol. The q-analog of n, denoted as $[n]_q$, is defined as:

$$[n]_q \coloneqq \sum_{i=0}^{n-1} q^i.$$

The q-analog of the positive integer n is essentially the symbolic generalization of n. We start by writing $n = \underbrace{1 + 1 + \cdots + 1}_{n}$ and replace each symbol "1" with another symbol q^i. Then, $q^0 + q^1 + \cdots + q^{n-1} = \sum_{i=0}^{n-1} q^i$ defines $[n]_q$. The construction itself

doi:10.6342/NTU202200738

seems purely artificial, but it gives a nice interpretation of the characteristic of a ring. If n is coprime to the characteristic of a ring and ω is a non-zero element, then the identity 1 can be written as

$$\begin{aligned} 1 = & \ (\underbrace{1+1+\cdots+1}_{n})^{-1}(\underbrace{1+1+\cdots+1}_{n}) \\ = & \ (\underbrace{1+1+\cdots+1}_{n})^{-1}\left((\omega^0)^0+(\omega^0)^1+\cdots+(\omega^0)^{n-1}\right). \end{aligned}$$

If $\omega^n = 1$, then we call ω an n-th root of unity. We investigate the case where ω is endowed with a special kind of orthogonality and call ω a principal n-th root of unity defined as follows.

Definition 60 (Principal n-th root of unity [Für09, HvdH21])**.** Let R be a ring and n be a positive integer. A root of unity $\omega \in R$ is called a principal n-th root of unity if:

$$\forall i = 0, 1, \ldots, n-1, [n]_{\omega^i} = [n]_1 \cdot \delta_{0,i}.$$

Corollary 5. Let R be a ring, ω be principal n-th root of unity in R, and x be an indeterminate. Then ω is a principal n-th root of unity in $R[x]$.

Proof. Trivial. □

4.3.2 The Number-Theoretic Transform

We now define the number-theoretic transform (NTT) for polynomial rings. Essentially, the NTT is just a special case of CRT for polynomial rings where the conditions are satisfied by the invertibility of the size n of the NTT and the orthogonality of the chosen principal n-th root of unity. Furthermore, we choose to define NTT as a certain kind of discrete weighted transform from [CF94].

Definition 61 (Number-theoretic transform, [CF94])**.** Let R be a ring, n be a positive integer coprime to $\mathrm{char}(R)$, ζ be an invertible element in R, and ω be a principal n-th root of unity in R. The number-theoretic transform $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ is the isomorphism from $R[x]/\langle x^n-\zeta^n\rangle$ to $\prod_{i=0}^{n-1} R[x]/\langle x-\zeta\omega^i\rangle$ defined as:

$$\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega} : \begin{cases} R[x]/\langle x^n-\zeta^n\rangle & \to \prod_{i=0}^{n-1} R[x]/\left\langle x-\zeta\omega^i\right\rangle, \\ \boldsymbol{a}(x) & \mapsto \left(\boldsymbol{a}(\zeta\omega^i)\right). \end{cases}$$

Furthermore, we can find the $(\boldsymbol{r}_i) := \left(\frac{1}{n}[n]_{\zeta^{-1}\omega^{-i}x}\right) \in \left(R[x]/\langle x^n-\zeta^n\rangle\right)^n$ to identify the inverse $\mathsf{NTT}^{-1}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ as follows:

$$\mathsf{NTT}^{-1}_{R[x]/\langle x^n-\zeta^n\rangle:\omega} : \begin{cases} \prod_{i=0}^{n-1} R[x]/\left\langle x-\zeta\omega^i\right\rangle & \to R[x]/\langle x^n-\zeta^n\rangle, \\ (a_i) & \mapsto \sum_{i=0}^{n-1} \boldsymbol{r}_i a_i. \end{cases}$$

To see that $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ and $\mathsf{NTT}^{-1}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ are isomorphisms realizing CRT, we have to verify the following two conditions:

doi:10.6342/NTU202200738

1. For all i, j, $\boldsymbol{r}_i \boldsymbol{r}_j = \delta_{i,j} \boldsymbol{r}_i$.
2. The sum of all $\boldsymbol{r}_i$'s gives $\sum_{i=0}^{n-1} \boldsymbol{r}_i = 1$.

First, we verify $\boldsymbol{r}_i$'s are orthogonal idempotent elements. We derive the k-th coefficient of $\boldsymbol{r}_i \boldsymbol{r}_j$ as follows:

$$\begin{aligned}
[x^k]\boldsymbol{r}_i\boldsymbol{r}_j &= \frac{\left(\sum_{h=0}^{k}(\zeta^{-1}\omega^{-i})^h(\zeta^{-1}\omega^{-j})^{k-h}\right) + \zeta^n\left(\sum_{h=k+1}^{n-1}(\zeta^{-1}\omega^{-i})^h(\zeta^{-1}\omega^{-j})^{n+k-h}\right)}{n^2} \\
&= \frac{1}{n^2}\sum_{h=0}^{n-1}(\zeta^{-1}\omega^{-i})^h(\zeta^{-1}\omega^{-j})^{k-h} \\
&= \frac{1}{n^2}(\zeta^{-1}\omega^{-i})^k\left(\sum_{h=0}^{n-1}(\omega^{i-j})^{k-h}\right) \\
&= \frac{1}{n}(\zeta^{-1}\omega^{-i})^k\delta_{i,j} \\
&= \delta_{i,j}\boldsymbol{r}_i.
\end{aligned}$$

Finally, we also see that

$$\sum_{i=0}^{n-1}\boldsymbol{r}_i = \sum_{i=0}^{n-1}\frac{1}{n}\sum_{j=0}^{n-1}\left(\zeta^{-j}\omega^{-ij}\right)x^j = \sum_{j=0}^{n-1}\zeta^{-j}\frac{1}{n}\left(\sum_{i=0}^{n-1}\omega^{-ij}\right)x^j = \sum_{j=0}^{n-1}\zeta^{-j}\frac{1}{n}[n]_{\omega^{-j}}x^j = 1.$$

It is now clear that $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ and $\mathsf{NTT}^{-1}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ are the isomorphisms realizing the CRT for polynomial rings.

Corollary 6. $\mathsf{NTT}^{-1}_{R[x]/\langle x^n-\zeta^n\rangle:\omega} = \frac{1}{n}\mathsf{NTT}_{R[x]/\langle x^n-\zeta^{-n}\rangle:\omega^{-1}}$ as maps from R^n to itself. Therefore, any algorithms applicable to $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ apply to $\mathsf{NTT}^{-1}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$.

Proof. Trivial. □

Definition 62 (Cyclic NTT)**.** If $\zeta^n = 1$, then $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ is called a cyclic NTT.

Definition 63 (Negacyclic NTT)**.** If $\zeta^n = -1$, then $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$ is called a negacyclic NTT.

4.3.3 Realizing $\cdot_{R[x]/\langle x^n-\zeta^n\rangle}$ via $\mathsf{NTT}_{R[x]/\langle x^n-\zeta^n\rangle:\omega}$

Let $\mathcal{R} = R[x]/\langle x^n - \zeta^n\rangle$ and for $i = 0, 1, \ldots, n-1$, $\mathcal{R}_i = R[x]/\langle x - \zeta\omega^i\rangle$. Since $\mathsf{NTT}_{\mathcal{R}:\omega}$ is an isomorphism between $\mathcal{R}$ and $\prod_{i=0}^{n-1}\mathcal{R}_i$, we have

$$\mathsf{NTT}_{\mathcal{R}:\omega}\left(\boldsymbol{a}(x)\cdot_{\mathcal{R}}\boldsymbol{b}(x)\right) = \mathsf{NTT}_{\mathcal{R}:\omega}(\boldsymbol{a}(x))\cdot_{\prod_{i=0}^{n-1}\mathcal{R}_i}\mathsf{NTT}_{\mathcal{R}:\omega}(\boldsymbol{b}(x)).$$

And since $\mathsf{NTT}_{\mathcal{R}:\omega}$ is invertible,

$$\boldsymbol{a}(x)\cdot_{\mathcal{R}}\boldsymbol{b}(x) = \mathsf{NTT}^{-1}_{\mathcal{R}:\omega}\left(\mathsf{NTT}_{\mathcal{R}:\omega}(\boldsymbol{a}(x))\cdot_{\prod_{i=0}^{n-1}\mathcal{R}_i}\mathsf{NTT}_{\mathcal{R}:\omega}(\boldsymbol{b}(x))\right).$$

Implementing $\boldsymbol{a}(x)\cdot_{\mathcal{R}}\boldsymbol{b}(x)$ with $\mathsf{NTT}_{\mathcal{R}:\omega}$ and $\mathsf{NTT}^{-1}_{\mathcal{R}:\omega}$ is the core idea of this thesis. We will go through several strategies on how NTTs can be implemented efficiently.

doi:10.6342/NTU202200738

4.3.4 Incomplete NTTs

Let m and n be positive integers and $R[x]/\langle x^{mn}-\xi\rangle$ be a polynomial ring. In the previous sections, we see that the existence of a principal mn-th root of unity ω_{mn} and an mn-th root of ξ are required for defining the NTT. But what if one of the requirements is not met? If we introduce the equivalence $x^m \sim x^{(0)}$ and re-write the ring $R[x]/\langle x^{mn}-\xi\rangle$ as

$$\frac{\left(\frac{R[x]}{\left\langle x^m - x^{(0)}\right\rangle}\right)[x^{(0)}]}{\left\langle (x^{(0)})^n - \xi\right\rangle},$$

then we can define NTTs by treating $R[x]/\left\langle x^m - x^{(0)}\right\rangle$ as the coefficient ring. The requirement is then about the existence of an n-th root of ξ and a principal n-th root of unity. Suppose the requirements are met where $\zeta = \xi^{\frac{1}{n}}$ and ω_n a principal n-th root of unity in R, then $\mathsf{NTT}_{\bar{\mathcal{R}}[x^{(0)}]/\left\langle (x^{(0)})^n-\zeta^n\right\rangle:\omega_n}$ is defined where $\bar{\mathcal{R}} = R[x]/\left\langle x^m - x^{(0)}\right\rangle$. Note that a principal n-th root of unity in R is also a principal n-th root of unity in $R[x]$ and $\bar{\mathcal{R}}$.

For polynomials $\boldsymbol{a}(x) = \sum_{i=0}^{mn-1} a_i x^i$ and $\boldsymbol{b}(x) = \sum_{i=0}^{mn-1} b_i x^i$ in $R[x]/\langle x^{mn}-\xi\rangle$. We rewrite them as follows:

$$\begin{cases} \boldsymbol{a}(x) = \boldsymbol{a}^{(0)}(x^{(0)}) & \coloneqq \sum_{i=0}^{n-1}\left(\sum_{j=0}^{m-1} a_{im+j}x^j\right)(x^{(0)})^i, \\ \boldsymbol{b}(x) = \boldsymbol{b}^{(0)}(x^{(0)}) & \coloneqq \sum_{i=0}^{n-1}\left(\sum_{j=0}^{m-1} b_{im+j}x^j\right)(x^{(0)})^i. \end{cases}$$

Clearly, $\boldsymbol{a}(x)$ and $\boldsymbol{b}(x)$ can now be regarded as polynomials $\boldsymbol{a}^{(0)}(x^{(0)})$ and $\boldsymbol{b}^{(0)}(x^{(0)})$ in $\bar{\mathcal{R}}[x^{(0)}]/\left\langle (x^{(0)})^n - \xi\right\rangle$. For defining an NTT, let

- $\forall i = 0, \ldots, n-1, \mathcal{R}_i \coloneqq R[x]/\langle x^m - \zeta\omega_n^i\rangle$,
- $\mathcal{R}^{(0)} \coloneqq \bar{\mathcal{R}}[x^{(0)}]/\left\langle \left(x^{(0)}\right)^n - \zeta^n\right\rangle$, and
- $\forall i = 0, \ldots, n-1, \mathcal{R}_i^{(0)} \coloneqq \bar{\mathcal{R}}[x^{(0)}]/\left\langle x^{(0)} - \zeta\omega_n^i\right\rangle$.

We now define $\mathsf{NTT}_{\mathcal{R}:\omega_n}(\boldsymbol{a}(x))$ as $\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_n}(\boldsymbol{a}^{(0)}(x^{(0)}))$. It is clear that $\mathsf{NTT}_{\mathcal{R}:\omega_n}(\boldsymbol{a}(x))$ is an isomorphism from $\mathcal{R}$ to $\prod_{i=0}^{n-1}\mathcal{R}_i$ since:

$$\begin{aligned} \mathsf{NTT}_{\mathcal{R}:\omega_n}\left(\boldsymbol{a}(x)\cdot_{\mathcal{R}}\boldsymbol{b}(x)\right) = {} & \mathsf{NTT}_{\mathcal{R}:\omega_n}\left(\boldsymbol{c}(x)\right) \\ = {} & \mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_n}\left(\boldsymbol{c}^{(0)}(x^{(0)})\right) \\ = {} & \mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_n}\left(\boldsymbol{a}^{(0)}(x^{(0)})\cdot_{\mathcal{R}^{(0)}}\boldsymbol{b}^{(0)}(x^{(0)})\right) \\ = {} & \mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_n}(\boldsymbol{a}^{(0)}(x^{(0)}))\cdot_{\prod_{i=0}^{n-1}\mathcal{R}_i^{(0)}}\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_n}(\boldsymbol{b}^{(0)}(x^{(0)})) \\ = {} & \mathsf{NTT}_{\mathcal{R}:\omega_n}(\boldsymbol{a}(x))\cdot_{\prod_{i=0}^{n-1}\mathcal{R}_i}\mathsf{NTT}_{\mathcal{R}:\omega_n}(\boldsymbol{b}(x)) \end{aligned}$$

where $\cdot_{\prod_{i=0}^{n-1}\mathcal{R}_i^{(0)}}$ is the ring multiplication in $\prod_{i=0}^{n-1}\mathcal{R}_i^{(0)}$ and it is the same as $\cdot_{\prod_{i=0}^{n-1}\mathcal{R}_i}$, the ring multiplication in $\prod_{i=0}^{n-1}\mathcal{R}_i$.

4.3.5 Näive Computation of NTTs

We first discuss some näive approaches for computing NTTs before going through several fast algorithms with much lower asymptotic run time.

doi:10.6342/NTU202200738

Lemma 1. Let n be a positive integer, R be a ring, $\zeta \in R$, ω be a principal n-th root of unity in R, and $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i x^i$, then $\sum_{j=0}^{n-1} \boldsymbol{a}(\zeta\omega^j) = na_0$.

Proof.

$$\begin{aligned} \sum_{j=0}^{n-1} \boldsymbol{a}(\zeta\omega^j) &= \sum_{j=0}^{n-1}\sum_{i=0}^{n-1} a_i \zeta^i \omega^{ij} \\ &= \sum_{i=0}^{n-1} a_i \zeta^i \sum_{j=0}^{n-1} \omega^{ij} \\ &= \sum_{i=0}^{n-1} a_i \zeta^i n \delta_{i,0} \\ &= na_0. \end{aligned}$$

□

Theorem 6. Let n be a positive integer, R be a ring, ω be a principal n-th root of unity in R, and ζ be an element in R. For a polynomial $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i x^i$, we can compute the NTT transforming $\boldsymbol{a}(x)$ into $(\boldsymbol{a}(\zeta\omega^i))$ with $(n-1)^2$ multiplications and $n(n-1)$ additions/subtractions. Furthermore, if $\zeta = 1$, then the number of multiplications is reduced to $(n-1)(n-2)$.

Proof. For $i = 0, \ldots, n-2$, we first compute $\boldsymbol{a}(\zeta\omega^i) - a_0 = \sum_{j=1}^{n-1} a_j \zeta^j \omega^{ij}$ with in total $(n-1)^2$ multiplications and $(n-1)(n-2)$ additions. Then, we compute $\boldsymbol{a}(\zeta\omega^{n-1})$ as

$$\boldsymbol{a}(\zeta\omega^{n-1}) = na_0 - \sum_{j=0}^{n-2} \boldsymbol{a}(\zeta\omega^j) = a_0 - \sum_{i=0}^{n-2} \big(\boldsymbol{a}(\zeta\omega^i) - a_0\big)$$

with $n-1$ additions/subtractions. Finally, for $i = 0, \ldots, n-2$, we compute $\boldsymbol{a}(\zeta\omega^i) = (\boldsymbol{a}(\zeta\omega^i) - a_0) + a_0$ with in total $(n-1)$ additions.

In summary, we only require $(n-1)^2$ multiplications and $n(n-1)$ additions/-subtractions. Furthermore, if $\zeta = 1$, then we don't need any multiplications for $\boldsymbol{a}(\zeta) - a_0$ and hence $n-1$ multiplications are saved. □

4.4 Fast Fourier Transforms

In this section, we investigate the applicability of various fast Fourier transforms and their interactions with convolutions in polynomial rings of the form $R[x]/\langle x^{\mathcal{Q}} - \zeta^{\mathcal{Q}} \rangle$ where $\mathcal{Q} \in \mathbb{N}$.

doi:10.6342/NTU202200738

Table 4.2: List of notations for FFTs.

$(q_0, \dots, q_{l-1})$	An l-radix number system.
$\mathcal{Q}$	A product of integers $\prod_{i=0}^{l-1} q_i$.
$\omega_{\mathcal{Q}}$	A principal $\mathcal{Q}$-th root of unity.
$\mathcal{R}$	Target polynomial ring $\frac{R[x]}{\langle x^{\mathcal{Q}} - \zeta^{\mathcal{Q}} \rangle}$.
$\bar{\mathcal{R}}_i$	Product ring $\prod_{i=0}^{l-1} \frac{R[x]}{\left\langle x - \zeta \omega_{\mathcal{Q}}^{i} \right\rangle}$.
$\mathcal{Q} = q_0 q_1$	
$\mathcal{R}_{i_0}$	Polynomial ring $\frac{R[x]}{\left\langle x^{q_1} - \left(\zeta \omega_{\mathcal{Q}}^{i_0}\right)^{q_1} \right\rangle}$.
$\mathcal{R}_{i_0, i_1}$	Polynomial ring $\frac{R[x]}{\left\langle x - \zeta \omega_{\mathcal{Q}}^{i_0 + i_1 q_0} \right\rangle}$.
$\mathcal{Q} = q_0 q_1$ and $\zeta = 1$	
$\mathcal{R}'_{i_0}$	Polynomial ring $\frac{R[y]}{\left\langle x - \omega_{\mathcal{Q}}^{i_0} y, y^{q_1} - 1 \right\rangle}$.
$\mathcal{R}'_{i_0, i_1}$	Polynomial ring $\frac{R[y]}{\left\langle x - \omega_{\mathcal{Q}}^{i_0} y, y - \omega_{\mathcal{Q}}^{i_1 q_0} \right\rangle}$.
Coprime q_ks and $\zeta = 1$	
$(e_0, \dots, e_{l-1})$	Idempotents with $m \equiv \sum_{k=0}^{l-1} e_k \, (m \bmod q_k) \pmod{\mathcal{Q}}$.
$\mathcal{R}^{(k)}$	Polynomial ring $\frac{R[x^{(k)}]}{\left\langle \left(x^{(k)}\right)^{q_k} - 1 \right\rangle}$.
$\bar{\mathcal{R}}^{(k)}_{i_k}$	Polynomial ring $\frac{R[x^{(k)}]}{\left\langle x^{(k)} - \omega_{\mathcal{Q}}^{e_k i_k} \right\rangle}$.
$\mathcal{Q} = q_0 q_1$, $q_0 \perp q_1$, $h_0 \vert q_0$, $h_1 \vert q_1$, and $\zeta = 1$	
$\mathcal{R}^{(k)}_{i_k}$	Polynomial ring $\frac{R[x^{(k)}]}{\left\langle \left(x^{(k)}\right)^{\frac{q_k}{h_k}} - \omega_{\mathcal{Q}}^{e_k \frac{q_k i_k}{h_k}} \right\rangle}$.
$\mathcal{R}'^{(k)}_{i_k}$	Polynomial ring $\frac{R[x^{(k)}]}{\left\langle x^{(k)} - \omega_{\mathcal{Q}}^{e_k i_k} y^{(k)}, \left(y^{(k)}\right)^{\frac{q_k}{h_k}} - 1 \right\rangle}$.
$\mathcal{R}'^{(k)}_{i_k, j_k}$	Polynomial ring $\frac{R[x^{(k)}]}{\left\langle x^{(k)} - \omega_{\mathcal{Q}}^{e_k i_k} y^{(k)}, y^{(k)} - \omega_{\mathcal{Q}}^{e_k j_k h_k} \right\rangle}$.

4.4.1 General Digit Reversal

We first introduce the the general digit reversal.

Definition 64 (l-radix number system, [Knu14, Section 4.1])**.** Let $(q_0, \dots, q_{l-1})$ be a tuple of positive integers, then we call the tuple an l-radix number system for the set of integers $\left\{0, \dots, \prod_{i=0}^{l-1} q_i - 1\right\}$. In particular, we can uniquely represent a number m in $\left\{0, \dots, \prod_{i=0}^{l-1} q_i - 1\right\}$ as

$$m_{(q_0, \dots, q_{l-1})} = (m_0, \dots m_{l-1})$$

where $\sum_{i=0}^{l-1} m_i \prod_{j=i+1}^{l-1} q_j = m$ and $0 \le m_i \le q_i - 1$.

We denote $(q_k : d_k;)$ for $(\underbrace{q_k, \dots, q_k}_{d_k})$ and simply $(q_k;)$ if $d_k = 1$.

doi:10.6342/NTU202200738

Clearly, the reversal $(q_{l-1}, \ldots, q_0)$ is also an l-radix number system for the same set of integers. This observation is very helpful in this thesis, in particular, we are interested in the transformations between them.

Definition 65 (General digit reversal)**.** Let $(q_0, \ldots, q_{l-1})$ be an l-radix number system. The general digit reversal $\mathrm{rev}_{(q_0,\ldots,q_{l-1})}$ is the bijection defined as:

$$\mathrm{rev}_{(q_0,\ldots,q_{l-1})} : \begin{cases} \left\{0, \ldots, \prod_{i=0}^{l-1} q_i - 1\right\} & \to \left\{0, \ldots, \prod_{i=0}^{l-1} q_i - 1\right\}, \\ \sum_{i=0}^{l-1} m_i \prod_{j=i+1}^{l-1} q_j & \mapsto \sum_{i=0}^{l-1} m_i \prod_{j=0}^{i-1} q_j. \end{cases}$$

Obviously, the inverse of $\mathrm{rev}_{(q_0,\ldots,q_{l-1})}$ is $\mathrm{rev}_{(q_{l-1},\ldots,q_0)}$. Furthermore, for all $m \in \left\{0, \ldots, \prod_{i=0}^{l-1} q_i - 1\right\}$, $m_{(q_0,\ldots,q_{l-1})}$ is the reversal of $(\mathrm{rev}_{(q_0,\ldots,q_{l-1})}(m))_{(q_{l-1},\ldots,q_0)}$. We can expand further to have the following characterization.

Proposition 1. Let $(q_0, \ldots, q_{l-1})$ be an l-radix number system. Then, $\forall k = 1, \ldots, l-1$,

$$\begin{aligned} \mathrm{rev}_{(q_0,\ldots,q_{l-1})} = & \left(m \mapsto \mathrm{rev}_{(q_k,\ldots,q_{l-1})}\left(\left\lfloor \frac{m}{\prod_{j=0}^{k-1} q_j} \right\rfloor \right) + \left(m \bmod \left(\prod_{j=0}^{k-1} q_j \right) \right) \right) \\ & \circ \, \mathrm{rev}_{\left(q_0,\ldots,q_{k-1},\prod_{j=k}^{l-1} q_j\right)} \end{aligned}$$

Proof. Trivial. □

4.4.2 Cooley–Tukey FFT

For an l-radix number system $(q_0, \ldots, q_{l-1})$, Cooley–Tukey FFT splits the computation of a size-$\prod_{i=0}^{l-1} q_{l-1}$ NTT into l layers of computation where layer i consists of several size-q_i NTTs.

Definition 66 (Cooley–Tukey FFT, [CT65])**.** Let $\mathcal{Q} = q_0 q_1$ be an invertible element in the ring R, $\omega_{\mathcal{Q}}$ be a principal $\mathcal{Q}$-th root of unity in R, $\omega_{q_0} := \omega_{\mathcal{Q}}^{q_1}$, and $\omega_{q_1} := \omega_{\mathcal{Q}}^{q_0}$. The Cooley–Tukey FFT with the 2-radix number system (r_0, r_1) is defined as splitting $\mathcal{R}$ as follows:

$$\begin{aligned} \mathcal{R} &\cong \prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0} \\ &\cong \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}_{i_0,i_1} \end{aligned}$$

Furthermore, we write the transformation as

$$\left(\prod_{i_0=0}^{q_0-1} \mathsf{NTT}_{\mathcal{R}_{i_0}:\omega_{q_1}} \right) \circ \mathsf{NTT}_{\mathcal{R}:\omega_{r_0}}.$$

doi:10.6342/NTU202200738

Recall that computing with näive $\mathsf{NTT}_{\mathcal{R}:\omega_{\mathcal{Q}}}$ gives

$$\mathcal{R} \cong \prod_{i=0}^{\mathcal{Q}-1} \bar{\mathcal{R}}_i = \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \frac{R[x]}{\left\langle x - \zeta \omega_{\mathcal{Q}}^{i_1+i_0 q_1} \right\rangle},$$

we find that the result of Cooley–Tukey FFT is equivalent to applying $\mathrm{rev}_{(q_0,q_1)}$ to the result of the näive computation. Figure 4.1 is an illustration.

Figure 4.1: Core idea of Cooley–Tukey FFT.

$$\begin{tikzcd}
\mathcal{R} \arrow[r, "\mathsf{NTT}_{\mathcal{R}:\omega_{q_0}}"] \arrow[d, "\mathsf{NTT}_{\mathcal{R}:\omega_{\mathcal{Q}}}"'] & \prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0} \arrow[d, "\prod_{i_0=0}^{q_0-1} \mathsf{NTT}_{\mathcal{R}_{i_0}:\omega_{q_1}}"] \\
\prod_{i=0}^{\mathcal{Q}-1} \bar{\mathcal{R}}_i \arrow[r, "\mathrm{rev}_{(q_0,q_1)}"'] & \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}_{i_0,i_1}
\end{tikzcd}$$

The reversal property can be generalized to Cooley–Tukey FFTs with l-radix number systems. We justify this inductively as follows. For an l-radix number system $(q_0, \ldots, q_{l-1})$, we perform a radix-q_0 split giving the product ring $\prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0}$ as usual. For each $\mathcal{R}_{i_0}$, we apply the Cooley–Tukey FFT with the $(l-1)$-radix number system $(q_1, \ldots, q_{l-1})$. Now we look at the permutations. The first radix-q_0 split corresponds to the reversal $\mathrm{rev}_{\left(q_0, \prod_{j=1}^{l-1} q_j\right)}$ and each of the follow up Cooley–Tukey FFT corresponds to applying $\mathrm{rev}_{(q_1,\ldots,q_{l-1})}$ to the upper $l-1$ digits. By Proposition 1, the overall permutation is $\mathrm{rev}_{(q_0,\ldots,q_{l-1})}$.

We now turn to the cost of computation and show that Cooley–Tukey is indeed a "fast" Fourier transform. Following Theorem 6, the näive computation for a size-$\mathcal{Q}$ NTT requires $O\left(\mathcal{Q}^2\right)$ ring operations. But if we apply Cooley–Tukey FFT, we see that $\mathsf{NTT}_{\mathcal{R}:\omega_{q_0}}$ requires $O\left(q_0 \mathcal{Q}\right)$ ring operations and $\prod_{i_0=0}^{q_0-1} \mathsf{NTT}_{\mathcal{R}_{i_0}:\omega_{q_1}}$ requires $O\left(q_1 \mathcal{Q}\right)$ ring operations. In summary, the number of ring operations is reduced from $O\left(\mathcal{Q}^2\right)$ to $O\left(\mathcal{Q}(q_0 + q_1)\right)$.

If $q_1 = q_0^{l-1}$ is a power of q_0, we can apply the idea recursively and reduce the number of ring operations to $O\left(l q_0^{l+1}\right)$. If the size $\mathcal{Q}$ of the transformation is a power of a small constant, then Cooley–Tukey FFT reduces the complexity to $O\left(\mathcal{Q} \lg \mathcal{Q}\right)$ ring operations. In Section 4.4.4, we will see how to reduce the complexity to $O\left(\mathcal{Q} \lg \mathcal{Q}\right)$ even if $\mathcal{Q}$ is not a power of a small constant.

4.4.3 Gentleman–Sande FFT

Gentleman–Sande FFT transforms a non-cyclic NTT into a cyclic NTT by multiplying the coefficients by powers of $\omega_{\mathcal{Q}}$ [GS66]. Assuming $\zeta = 1$, we start with the ring $\mathcal{R}$ and apply $\mathsf{NTT}_{\mathcal{R}:\omega_{q_0}}$ giving the product ring $\prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0}$. Then, for each $i_0 = 0, \ldots q_0 - 1$, we introduce the equivalence $x \sim \omega_{\mathcal{Q}}^{i_0} y$ by multiplying the coefficients with powers of $\omega_{\mathcal{Q}}^{i_0}$. In particular, for an element $\boldsymbol{a}_{i_0}(x) = \sum_{i_1}^{q_1-1} a_{(i_0,i_1)} x^{i_1} \in \mathcal{R}_{i_0}$, introducing the equivalence $x \sim \omega_{\mathcal{Q}}^{i_0} y$ means writing $\boldsymbol{a}_{i_0}(x)$ as $\boldsymbol{a}_{i_0}(\omega_{\mathcal{Q}}^{i_0} y) = \sum_{i_1}^{q_1-1} a_{(i_0,i_1)} \omega_{\mathcal{Q}}^{i_0} y^{i_1}$ and to regard it as a polynomial in y, we have to compute $a_{(i_0,i_1)} \omega_{\mathcal{Q}}^{i_0}$. The following is an example of the computation.

doi:10.6342/NTU202200738

$$\begin{aligned}
\mathcal{R} &\cong \prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0} \\
&\overset{\eta}{\cong} \prod_{i_0=0}^{q_0-1} \mathcal{R}'_{i_0} \\
&\cong \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}'_{i_0,i_1} \\
&= \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}_{i_0,i_1}
\end{aligned}$$

where $\eta = \prod_{i_0=0}^{q_0-1}(x \mapsto \omega_{\mathcal{Q}}^{i_0} y)$. Obviously, the asymptotic behavior of Gentleman–Sande FFT is the same as Cooley–Tukey FFT.

4.4.4 Good–Thomas FFT

Let $\mathcal{Q} = \prod_{k=0}^{l-1} q_k$ be a coprime factorization. Good–Thomas FFT transforms a size-$\mathcal{Q}$ cyclic NTT into a tensor product of cyclic NTTs of sizes $q_0, \dots, q_{l-1}$.

Theorem 7 (Good–Thomas FFT, [Goo58])**.** Let $\mathcal{Q} = \prod_{k=0}^{l-1} q_k$ be a coprime factorization of $\mathcal{Q}$, $e_0, \dots, e_{l-1}$ be the idempotent elements realizing

$$\forall m \in \mathbb{Z}_{\mathcal{Q}}, m \equiv \sum_{k=0}^{l-1} e_k (m \bmod q_k) \pmod{\mathcal{Q}}.$$

We define η as follows:

$$\eta : \begin{cases} \mathbb{Z}_{\mathcal{Q}} & \to \prod_{k=0}^{l-1} \mathbb{Z}_{q_k}, \\ m & \mapsto (m \bmod q_k). \end{cases}$$

Then, η induces a bijection $\pi_\eta : R^{\mathcal{Q}} \to R^{\mathcal{Q}}$ by sending x to $\prod_{k=0}^{l-1} x^{(k)}$ which further induces the following isomorphism:

$$\frac{R[x]}{\langle x^{\mathcal{Q}} - 1 \rangle} \cong \bigotimes_{k=0}^{l-1} \frac{R[x^{(k)}]}{\langle (x^{(k)})^{q_k} - 1 \rangle}.$$

Furthermore, if we choose $\omega_{q_k} := \omega_{\mathcal{Q}}^{e_k}$, we have

$$\mathsf{NTT}_{\frac{R[x]}{\langle x^{\mathcal{Q}}-1 \rangle}:\omega_{\mathcal{Q}}} = \pi_\eta^{-1} \circ \bigotimes_{k=0}^{l-1} \mathsf{NTT}_{\frac{R[x^{(i)}]}{\langle (x^{(k)})^{q_k}-1 \rangle}:\omega_{q_k}} \circ \pi_\eta.$$

Proof. Trivial. □

Figure 4.2 is an illustration of the isomorphisms.

doi:10.6342/NTU202200738

Figure 4.2: Core idea of Good–Thomas FFT.

$$\begin{tikzcd}
\mathcal{R} \arrow[r, "\pi_\eta"] \arrow[d, "\mathsf{NTT}_{\mathcal{R}:\omega_{\mathcal{Q}}}"'] & \bigotimes_{k=0}^{l-1} \mathcal{R}^{(k)} \arrow[d, "\bigotimes_{k=0}^{l-1} \mathsf{NTT}_{\mathcal{R}^{(k)}:\omega_{q_k}}"] \\
\prod_{i=0}^{\mathcal{Q}-1} \mathcal{R}_i & \bigotimes_{k=0}^{l-1} \prod_{i_k=0}^{q_k-1} \mathcal{R}^{(k)}_{i_k} \arrow[l, "\pi_\eta^{-1}"]
\end{tikzcd}$$

One should note that $m \mapsto (m \bmod q_k)$ is not the only map resulting in a multi-dimensional convolution. In general, any additive group isomorphism from $\mathbb{Z}_{\mathcal{Q}}$ to $\prod_{k=0}^{l-1} \mathbb{Z}_{q_k}$ suffices. Fix an arbitrary additive group isomorphism Φ from $\mathbb{Z}_{\mathcal{Q}}$ to $\prod_{k=0}^{l-1} \mathbb{Z}_{q_k}$. Then, for any additive group isomorphism Ψ, $\Phi^{-1} \circ \Psi$ is an additive group automorphism on $\mathbb{Z}_{\mathcal{Q}}$ and $\Psi = \Phi \circ (\Phi^{-1} \circ \Psi)$. This implies that the number of additive group isomorphisms from $\mathbb{Z}_{\mathcal{Q}}$ to $\prod_{k=0}^{l-1} \mathbb{Z}_{q_k}$ is equal to the number automorphisms on $\mathbb{Z}_{\mathcal{Q}}$ and there are $\phi(\mathcal{Q})$ many of them by Theorem 3.

If the size $\mathcal{Q}$ of a cyclic NTT admits the prime factorization $\mathcal{Q} = \prod_{k=0}^{l-1} p_k^{d_k}$, then Good–Thomas FFT requires $O\left(\mathcal{Q} \sum_{k=0}^{l-1} \frac{\mathcal{C}\left(p_k^{d_k}\right)}{p_k^{d_k}}\right)$ ring operations where $\mathcal{C}\left(p_k^{d_k}\right)$ is the cost of the size-$p_k^{d_k}$ cyclic NTT. If p_k is a small constant, then $\mathcal{C}\left(p_k^{d_k}\right) = O\left(p_k^{d_k} \lg p_k^{d_k}\right)$. Otherwise, we assume $p_k > 2$. Since there is an element g in $\mathbb{Z}_{p_k^{d_k}}$ generating the unit group, the size-$p_k^{d_k}$ NTT can be turned into a size-$\lambda\left(p_k^{d_k}\right)$ convolution where λ is the Carmichael's λ function. For the size-$\lambda\left(p_k^{d_k}\right)$ convolution, we pad it to the size-$2^{\left\lceil \lg \lambda\left(p_k^{d_k}\right)\right\rceil + 1}$ convolution and compute with radix-2 Cooley–Tukey FFT requiring $O\left(p_k^{d_k} \lg p_k^{d_k}\right)$ ring operations. This implies $\mathcal{C}\left(p_k^{d_k}\right) = O\left(p_k^{d_k} \lg p_k^{d_k}\right)$. Therefore, we can compute the size-$\mathcal{Q}$ NTT with $O\left(\mathcal{Q} \sum_{k=0}^{l-1} \lg p_k^{d_k}\right) = O\left(\mathcal{Q} \lg \mathcal{Q}\right)$ ring operations. Such technique is known by [Win78]. Obviously, we also see that Cooley–Tukey FFT for arbitrary $\mathcal{Q}$ requires $O\left(\mathcal{Q} \lg \mathcal{Q}\right)$ ring operations by applying radix-$p_k^{d_k}$ splits.

4.4.5 Comparisons between Cooley–Tukey, Gentleman–Sande, and Good–Thomas FFTs

We shall now compare Cooley–Tukey, Gentleman–Sande, and Good–Thomas FFTs. We restrict the comparisons to the case $l = 2$ with coprime q_0 and q_1. For simplicity, we assume Theorem 6 for the NTT computation. Since it will be clear that the number of additions and subtractions are the same for these three FFTs, we only compare the numbers of multiplications. Note that for Cooley–Tukey and Gentleman–Sande FFTs, $\omega_{q_0} = \omega_{\mathcal{Q}}^{q_1}$ and $\omega_{q_1} = \omega_{\mathcal{Q}}^{q_0}$ while for Good–Thomas FFT, $\omega_{q_0} = \omega_{\mathcal{Q}}^{e_0}$ and $\omega_{q_1} = \omega_{\mathcal{Q}}^{e_1}$ where e_0 and e_1 are idempotent elements realizing

doi:10.6342/NTU202200738

$\forall m \in \mathbb{Z}_{\mathcal{Q}}, m \equiv (e_0(m \bmod q_0) + e_1(m \bmod q_1)) \pmod{\mathcal{Q}}$.

Cooley–Tukey FFT. Cooley–Tukey FFT is to apply size-q_0 and size-q_1 NTTs to the desired polynomial rings. We first apply $\mathsf{NTT}_{\mathcal{R}:\omega_{q_0}}$ and then apply $\prod_{i_0=0}^{q_0-1} \mathsf{NTT}_{\mathcal{R}_{i_0}:\omega_{q_1}}$ to the product ring. For the first step, since the NTT is cyclic, we only need $(q_0-1)(q_0-2)q_1$ multiplications. For the second step, among the q_0 size-q_1 NTTs, the first one is cyclic and the rest are non-cyclic NTTs. This implies $(q_1-1)(q_1-2)+(q_0-1)(q_1-1)^2$ are performed. In summary, the number of multiplications is

$$(q_0-1)(q_0-2)q_1+(q_1-1)(q_1-2)+(q_0-1)(q_1-1)^2 = q_0q_1(q_0+q_1)-5q_0q_1+q_0+q_1+1.$$

The following is the transformation with Cooley–Tukey FFT.

$$\begin{aligned}
\mathcal{R} &\cong \prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0} && (q_0-1)(q_0-2)q_1 \text{ multiplications} \\
&\cong \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}_{i_0,i_1} && (q_1-1)(q_1-2)+(q_0-1)(q_1-1)^2 \text{ multiplications} \\
&\cong \prod_{i=0}^{q_0q_1-1} \bar{\mathcal{R}}_i && \mathrm{rev}_{q_1,q_0}
\end{aligned}$$

Gentleman–Sande FFT. Gentleman–Sande FFT is also applying size-q_0 and size-q_1 NTTs to the polynomial rings. Different from Cooley–Tukey FFT, we always compute cyclic NTTs and if the polynomial ring is not cyclic, we multiply the coefficients with powers of the root and then apply cyclic NTTs. After applying $\mathsf{NTT}_{\mathcal{R}:\omega_{q_0}}$ to $\mathcal{R}$, we now have the product ring $\prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0}$. For $i_0 = 0, \ldots, q_0-1$, we then introduce the equivalence $x \sim \omega_{q_0q_1}^{i_0} y$ and the product ring now becomes $\prod_{i_0=0}^{q_0-1} \mathcal{R}'_{i_0}$. Since all the rings in the product ring are now cyclic in terms of y, we apply q_0 size-q_1 cyclic NTTs. In summary, the number of multiplications is

$$(q_0-1)(q_0-2)q_1+(q_0-1)(q_1-1)+q_0(q_1-1)(q_2-1),$$

the same as Cooley–Tukey FFT. The following is the transformation with Gentleman–Sande FFT.

$$\begin{aligned}
\mathcal{R} &\cong \prod_{i_0=0}^{q_0-1} \mathcal{R}_{i_0} && (q_0-1)(q_0-2)q_1 \text{ multiplications} \\
&\cong \prod_{i_0=0}^{q_0-1} \mathcal{R}'_{i_0} && (q_0-1)(q_1-1) \text{ multiplications} \\
&\cong \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}'_{i_0,i_1} && q_0(q_1-1)(q_2-1) \text{ multiplications} \\
&= \prod_{i_0=0}^{q_0-1} \prod_{i_1=0}^{q_1-1} \mathcal{R}_{i_0,i_1} && \\
&\cong \prod_{i=0}^{q_0q_1-1} \bar{\mathcal{R}}_i && \mathrm{rev}_{q_1,q_0}
\end{aligned}$$

doi:10.6342/NTU202200738

Good–Thomas FFT. Good–Thomas FFT first transforms a one-dimensional convolution into a multi-dimensional convolution by permuting the coefficients. For the ring $\mathcal{R}$, we transform it into the tensor product $\mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)}$, and apply q_1 size-q_0 cyclic NTTs to the dimension $x^{(0)}$ and q_0 size-q_1 cyclic NTTs to the dimension $x^{(1)}$. Since we compute cyclic NTTs, the number of multiplications is

$$(q_0 - 1)(q_0 - 2)q_1 + q_0(q_1 - 1)(q_1 - 2) = q_0 q_1 (q_0 + q_1) - 6 q_0 q_1 + 2q_0 + 2q_1.$$

The following is an illustration with Good–Thomas FFT.

$$\begin{aligned}
\mathcal{R} \cong\ & \mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)} && \pi_\eta \\
\cong\ & \left(\prod_{i_0=0}^{q_0-1} \mathcal{R}^{(0)}_{i_0} \right) \otimes \mathcal{R}^{(1)} && (q_0 - 1)(q_0 - 2)q_1 \text{ multiplications} \\
\cong\ & \left(\prod_{i_0=0}^{q_0-1} \mathcal{R}^{(0)}_{i_0} \right) \otimes \left(\prod_{i_1=0}^{q_1-1} \mathcal{R}^{(1)}_{i_1} \right) && q_0(q_1 - 1)(q_1 - 2) \text{ multiplications} \\
\cong\ & \prod_{i_0=0}^{q_0-1} \prod_{i=1}^{q_1-1} \bar{\mathcal{R}}_{e_0 i_0 + e_1 i_1} && \pi_\eta^{-1}
\end{aligned}$$

If we compare Good–Thomas FFT to Cooley–Tukey and Gentleman–Sande FFTs, the number of multiplications is reduced by

$$q_0 q_1 - q_0 - q_1 + 1 = (q_0 - 1)(q_1 - 1)$$

at the cost of permutations by π_η and π_η^{-1}. However, in practice, the permutations are either not needed or implemented implicitly. Therefore, we save $(q_0 - 1)(q_1 - 1)$ multiplications if we compute a size-$q_0 q_1$ cyclic NTT with Good–Thomas FFT instead.

4.4.6 Vector-Radix FFT

Vector-radix FFT is a straightforward generalization of the one-dimensional FFT introduced by [HMCS77]. We illustrate the idea for computing the convolution in $\mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)}$. Let h_0, h_1 be positive integers with $h_0 | q_0$, $h_1 | q_1$, $h_0 \ll q_0$, and $h_1 \ll q_1$.

We illustrate the idea by applying Gentleman–Sande FFT. We first compute $\mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)}$ by applying one-dimensional Gentleman–Sande FFT to each dimension. After applying the size-h_0 cyclic NTT, we split $\mathcal{R}^{(0)}$ into h_0 rings where the first ring is cyclic and the rest are not cyclic. For rings other than the first one, we twist them into cyclic rings. The product of maps is $\eta_0 \coloneqq \prod_{i_0=0}^{h_0-1} \left(x^{(0)} \mapsto \omega_{q_0}^{i_0} y^{(0)} \right)$. Then, we apply h_0 size-$\frac{q_0}{h_0}$ cyclic NTTs. For $\mathcal{R}^{(1)}$, we apply Gentleman–Sande FFT with $\eta_1 \coloneqq \prod_{i_1=0}^{h_1-1} \left(x^{(1)} \mapsto \omega_{q_1}^{i_1} y^{(1)} \right)$. The following is an illustration where $f_0 \coloneqq \prod_{i_0=0}^{h_0-1} \mathsf{NTT}_{\mathcal{R}'^{(0)}_{i_0} : \omega_{\frac{q_0}{h_0}}}$ and $f_1 \coloneqq \prod_{i_1=0}^{h_1-1} \mathsf{NTT}_{\mathcal{R}'^{(1)}_{i_1} : \omega_{\frac{q_1}{h_1}}}$.

doi:10.6342/NTU202200738

$$
\begin{aligned}
\mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)} &\overset{\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_{h_0}} \otimes \mathrm{id}}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}^{(0)}_{i_0} \right) \otimes \mathcal{R}^{(1)} \\
&\overset{\eta_0 \otimes \mathrm{id}}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0} \right) \otimes \mathcal{R}^{(1)} \\
&\overset{\mathrm{id} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_{h_1}}}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0} \right) \otimes \left(\prod_{i_1=0}^{n_1-1} \mathcal{R}^{(1)}_{i_1} \right) \\
&\overset{\mathrm{id} \otimes \eta_1}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0} \right) \otimes \left(\prod_{i_1=0}^{n_1-1} \mathcal{R}'^{(1)}_{i_1} \right) \\
&\overset{f_0 \otimes f_1}{\cong} \left(\prod_{i_0=0}^{h_0-1} \prod_{j_0=0}^{\frac{q_0}{h_0}-1} \mathcal{R}'^{(0)}_{i_0,j_0} \right) \otimes \left(\prod_{i_1=0}^{h_1-1} \prod_{j_1=0}^{\frac{q_1}{h_1}-1} \mathcal{R}'^{(1)}_{i_1,j_1} \right)
\end{aligned}
$$

Clearly, the total number of multiplications spent on $\eta_0 \otimes \mathrm{id}$ and $\mathrm{id} \otimes \eta_1$ is

$$
\begin{aligned}
&(h_0 - 1)\left(\frac{q_0}{h_0} - 1 \right) q_1 + (h_1 - 1)\left(\frac{q_1}{h_1} - 1 \right) q_0 \\
&\approx q_0 q_1 \left(2 - \frac{1}{h_0} - \frac{1}{h_1} \right).
\end{aligned}
$$

Now if we postpone the computation of $\eta_0 \otimes \mathrm{id}$ and merge it with $\mathrm{id} \otimes \eta_1$ to obtain $\eta_0 \otimes \eta_1$, the number of multiplications spent on $\eta_0 \otimes \eta_1$ is

$$
\begin{aligned}
&(h_0 - 1)(h_1 - 1)\left(\frac{q_0 q_1}{h_0 h_1} - 1 \right) + (h_0 - 1)\left(\frac{q_0}{h_0} - 1 \right) \frac{q_1}{h_1} + (h_1 - 1)\left(\frac{q_1}{h_1} - 1 \right) \frac{q_0}{h_0} \\
&\approx q_0 q_1 \left(1 - \frac{1}{h_0 h_1} \right).
\end{aligned}
$$

The following is an illustration realizing the same 2-dimensional FFT.

$$
\begin{aligned}
\mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)} &\overset{\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_{h_0}} \otimes \mathrm{id}}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}^{(0)}_{i_0} \right) \otimes \mathcal{R}^{(1)} \\
&\overset{\mathrm{id} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_{h_1}}}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}^{(0)}_{i_0} \right) \otimes \left(\prod_{i_1=0}^{h_1-1} \mathcal{R}^{(1)}_{i_1} \right) \\
&\overset{\eta_0 \otimes \eta_1}{\cong} \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0} \right) \otimes \left(\prod_{i_1=0}^{h_1-1} \mathcal{R}'^{(1)}_{i_1} \right) \\
&\overset{f_0 \otimes f_1}{\cong} \left(\prod_{i_0=0}^{h_0-1} \prod_{j_0=0}^{\frac{q_0}{h_0}-1} \mathcal{R}'^{(0)}_{i_0,j_0} \right) \otimes \left(\prod_{i_1=0}^{h_1-1} \prod_{j_1=0}^{\frac{q_1}{h_1}-1} \mathcal{R}'^{(1)}_{i_1,j_1} \right)
\end{aligned}
$$

Figure 4.3 is an overview of the transformations.

doi:10.6342/NTU202200738

Figure 4.3: Overview of vector-radix FFT.

$$\begin{tikzcd}
 & \mathcal{R}^{(0)} \otimes \mathcal{R}^{(1)} \arrow[d, "\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_{h_0}} \otimes \mathrm{id}"] \\
\left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0}\right) \otimes \mathcal{R}^{(1)} \arrow[d, "\mathrm{id} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_{h_1}}"'] & \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}^{(0)}_{i_0}\right) \otimes \mathcal{R}^{(1)} \arrow[l, "\eta_0 \otimes \mathrm{id}"'] \arrow[d, "\mathrm{id} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_{h_1}}"] \\
\left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0}\right) \otimes \left(\prod_{i_1=0}^{h_1-1} \mathcal{R}^{(1)}_{i_1}\right) \arrow[dr, "\mathrm{id} \otimes \eta_1"'] & \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}^{(0)}_{i_0}\right) \otimes \left(\prod_{i_1=0}^{h_1-1} \mathcal{R}^{(1)}_{i_1}\right) \arrow[d, "\eta_0 \otimes \eta_1"] \\
 & \left(\prod_{i_0=0}^{h_0-1} \mathcal{R}'^{(0)}_{i_0}\right) \otimes \left(\prod_{i_1=0}^{h_1-1} \mathcal{R}'^{(1)}_{i_1}\right) \arrow[d, "f_0 \otimes f_1"] \\
 & \left(\prod_{i_0=0}^{h_0-1} \prod_{j_0=0}^{\frac{q_0}{h_0}-1} \mathcal{R}'^{(0)}_{i_0,j_0}\right) \otimes \left(\prod_{i_1=0}^{h_1-1} \prod_{j_1=0}^{\frac{q_1}{h_1}-1} \mathcal{R}'^{(1)}_{i_1,j_1}\right)
\end{tikzcd}$$

In summary, we save approximately $q_0 q_1 \left(1 - \frac{1}{h_0}\right)\left(1 - \frac{1}{h_1}\right)$ multiplications. Furthermore, it is also possible to merge $\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_{h_0}} \otimes \mathrm{id}$ and $\mathrm{id} \otimes \eta_1$ into $\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_{h_0}} \otimes \eta_1$ as we will see in Chapter 8.

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Chapter 5

Instruction Set Architectures

This chapter introduces the instruction set architectures Armv7-M and Armv8-A and is simply for referential purpose. The main reason for writing a whole chapter about the material is because I don't find the current documentations satisfactory. The current documentations contain a bunch of information (more than 11,000 pages) that is not required for understanding this body of work. No contribution is claimed in this chapter.

5.1 Armv7-M

This section introduces the instruction set architecture Armv7-M.

5.1.1 General-Purpose Registers

There are 13 core registers, naming from `r0` to `r12`, and three special registers `sp`, `lr`, and `pc` [ARM21b, Section A2.3.1]. `sp` is the stack pointer, `lr` is the link register, and `pc` is the program counter. We preserve `lr` so it can be regarded as a general-purpose register. Since `sp` and `pc` can only be used with some restrictions, we do not base our implementations on them. For the other 14 registers, we treat them as general-purpose registers. Figure 5.1 is an illustration.

Figure 5.1: Core registers in Armv7-M architecture.

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15
													↑ sp	↑ lr	↑ pc

5.1.2 Thumb Instruction Sets

There are two additional instruction sets, Thumb and Thumb-2 instruction sets, for improving the code density at the cost of more limited operands. Thumb instruction set introduces instructions with 16-bit encoding and Thumb-2 instruction set extends Thumb by adding some instructions with 32-bit encoding. Thumb and Thumb-2 instruction sets maintain the compatibility with Arm instruction sets. Unless stated otherwise, we assume instructions belong to Arm instruction sets.

doi:10.6342/NTU202200738

5.1.3 Instructions

Below we give an overview of the instructions used for implementing NTTs and categorize them as follows:

- Memory operations: `ldrb`, `ldrsb`, `ldrh`, `ldrsh`, `ldr`, `ldm`, `strb`, `strh`, `str`, `stm`.
- Bitwise operations: `and`, `orr`, `eor`.
- Shift operations: `asr`, `lsr`, `lsl`, `ror`.
- Addition/Subtraction: `add`, `sub`, `rsb`, `neg`.
- Multiplications: `mul`, `mla`, `mls`, `smull`, `smlal`, `umull`, `umlal`.
- Bit-field operations: `sbfx`, `ubfx`.
- Branches, compares, conditional executions: `b`, `cmp`, `cmn`, `it`.

Instructions `and`, `eor`, `orr`, `add`, `sub`, `rsb` are standard data-processing instructions where the second operand can be optionally shifted [ARM21b, Section 4.4.1]. The optionally appended shift operations are implemented with barrel shifters.

`ldrb`, `ldrsb`, `ldrh`, `ldrsh`, `ldr`, `ldm`, `strb`, `strh`, `str`, `stm`. For loading from memory, there are `ldrb` loading a byte, `ldrsb` loading a signed byte, `ldrh` loading an unsigned halfword, `ldrsh` loading a signed halfword, `ldr` loading a word, and `ldm` loading multiple words. To store the values from registers to memory, there are `strb` storing a byte, `strh` storing a halfword, `str` storing a word, and `stm` storing multiple words [ARM21b, Sections A4.6, A4.7].

`and`, `orr`, `eor`. `and` is the bitwise logical and, `orr` is the bitwise logical or, and `eor` is the bitwise logical exclusive or [ARM21b, Section A4.4.1].

`asr`, `lsr`, `lsl`, `ror`. `asr` is the arithmetic right shift and it is the same as the right shift operator in the C language. `lsr` is the logical right shift meaning that zero is padded instead of the sign bit. `lsl` is the logical left shift and it is the same as the left shift operator in the C language. `ror` is rotating a 32-bit value. For the specified right rotation by `v`, `ror` is equivalent to `( (c >> v) & ((1 << (32 - v)) - 1) ) || (c << (32 - v))` [ARM21b, Section A4.4.2].

`add`, `sub`, `rsb`, `neg`. `add` adds the operands and `sub` subtracts the second operand from the first operand. `rsb` subtracts the first operand from the second operand. `neg` is the synonym for `rsb` where the second operand is the immediate 0 [ARM21b, Section A4.4.1].

doi:10.6342/NTU202200738

`mul`, `mla`, `mls`, `smull`, `smlal`, `umull`, `umlal`. `mul` multiplies two 32-bit operands and produces a 32-bit result. `mla` adds the result to the accumulator and `mls` subtracts the result from the accumulator. `smull` produces the 64-bit result. The lower 32 bits of the result are put into the first destination register and the upper 32 bits are put into the second destination register. `smlal` adds the 64-bit result to the two destination registers. `umull` and `umlal` are their unsigned counterparts [ARM21b, Section A4.4.3].

`sbfx`, `ubfx`. `sbfx` sign-extends any bit field to a 32-bit value and `ubfx` is the unsigned counterpart [ARM21b, Section A4.4.8].

`b`, `cmp`, `cmn`, `it`. `b` instructs the processor to branch to the target address. `cmp` compares the two sources and updates the flags. `cmn` compares the two sources with the second source negated and updates the flags. `it` makes up to four IT blocks according to the tested conditional flag. Each IT block will then executes with a specific condition. If a condition is coherent with the APSR, the result is taken. Otherwise, the result is discarded.

Algorithm 1 $[\left\lfloor \frac{q}{2} \right\rfloor, q] \mapsto [-\left\lceil \frac{q}{2} \right\rceil, 0]$.

```
Inputs: a = $c \in [\left\lfloor \frac{q}{2} \right\rfloor, q]$
Outputs: a = $c \bmod^{\pm} q \in [-\left\lceil \frac{q}{2} \right\rceil, 0]$
sub a, $q$
```

Algorithm 2 $[-q, -\left\lfloor \frac{q}{2} \right\rfloor) \mapsto [0, \left\lceil \frac{q}{2} \right\rceil)$.

```
Inputs: a = $c \in [-q, -\left\lfloor \frac{q}{2} \right\rfloor)$
Outputs: a = $c \bmod^{\pm} q \in [0, \left\lceil \frac{q}{2} \right\rceil)$
add a, $q$
```

Algorithm 3 $[-q, q] \mapsto [-\frac{q}{2}, \frac{q}{2})$ with `cmp`, `cmn`, `it`, `sub`, `add`.

```
cmp a, $\left\lfloor \frac{q}{2} \right\rfloor$
it ge
                         ▷ 16-bit encoded cmp and it in Thumb/Thumb-2.
subge a, $q$
                         ▷ 32-bit encoded sub in Armv7-M.
cmn a, $\left\lfloor \frac{q}{2} \right\rfloor$
it lt
                         ▷ 16-bit encoded cmn and it in Thumb/Thumb-2.
addlt q, $q$
                         ▷ 32-bit encoded add in Armv7-M.
```

5.1.4 Digital Signal Processing Extension

Digital Signal Processing extension, or DSP extension, provides parallel additions and subtractions, signed multiplications for specific halves of the entire registers, dual signed multiplications, signed most-significant-word multiplications, and pack instructions. Armv7-M with the DSP extension is also called Armv7E-M. We denote $a_1 || a_0$ for a 32-bit register holding two halfwords where a_0 is the bottom 16-bit and a_1 is the upper 16-bit. Although there are instructions treating registers as four 8-bit values, these instructions are not used in this thesis and we skip them.

The following instructions in the DSP extension are used for implementing NTTs.

- Parallel additions and subtractions. `sadd16`, `ssub16`, `uadd16`, `usub16`.

doi:10.6342/NTU202200738

- Halfword multiplications. smulbb, smulbt, smultb, smultt, smlabb, smlabt, smlatb, smlatt
- Dual halfword multiplications. smuad, smusd, smuadx, smusdx, smlad, smlsd, smladx, smlsdx.
- Signed most-significant-word multiplications. smmul, smmulr.
- Halfwords packing. pkhbt, pkhtb.

Table 5.1: Parallel additions and subtractions where c is the destination register.

[instruction] c, a_1||a_0, b_1||b_0			
[instruction]	c	[instruction]	c
sadd16	$(a_1 + b_1)$||$(a_0 + b_0)$	uadd16	$(a_1 + b_1)$||$(a_0 + b_0)$
ssub16	$(a_1 - b_1)$||$(a_0 - b_0)$	usub16	$(a_1 - b_1)$||$(a_0 - b_0)$

sadd16, uadd16, ssub16, usub16. sadd16 adds signed halfwords in parallel and ssub16 subtracts signed halfwords in parallel. uadd16 and usub16 are their unsigned counterparts [ARM21b, Section A4.4.7]. Table 5.1 is an illustration.

Table 5.2: Signed multiplications for specific halfwords where d is the destination register.

[instruction] d, a_1||a_0, b_1||b_0		[instruction] d, a_1||a_0, b_1||b_0, c	
[instruction]	d	[instruction]	d
smulbb	$a_0 \cdot b_0$	smlabb	$c + a_0 \cdot b_0$
smulbt	$a_0 \cdot b_1$	smlabt	$c + a_0 \cdot b_1$
smultb	$a_1 \cdot b_0$	smlatb	$c + a_1 \cdot b_0$
smultt	$a_1 \cdot b_1$	smlatt	$c + a_1 \cdot b_1$

smulbb, smulbt, smultb, smultt, smlabb, smlabt, smlatb, smlatt. For X, Y $\in$ {b, t}, smulXY multiplies the specified halfwords. If X = b, then the bottom 16-bit of the first operand is specified or otherwise the upper 16-bit is specified. And If Y = b, then the bottom 16-bit of the second operand is specified or otherwise the upper 16-bit is specified. smlabb, smlabt, smlatb, smlatt are the versions where the result is added with the accumulator [ARM21b, Section A4.4.3]. Table 5.2 is an illustration.

doi:10.6342/NTU202200738

Table 5.3: Dual signed multiplications for specific halfwords where d is the destination register.

[instruction] d, $a_1 \Vert a_0$, $b_1 \Vert b_0$		[instruction] d, $a_1 \Vert a_0$, $b_1 \Vert b_0$, c	
[instruction]	d	[instruction]	d
smuad	$a_0 \cdot b_0 + a_1 \cdot b_1$	smlad	$c + a_0 \cdot b_0 + a_1 \cdot b_1$
smusd	$a_0 \cdot b_0 - a_1 \cdot b_1$	smlsd	$c + a_0 \cdot b_0 - a_1 \cdot b_1$
smuadx	$a_0 \cdot b_1 + a_1 \cdot b_0$	smladx	$c + a_0 \cdot b_1 + a_1 \cdot b_0$
smusdx	$a_0 \cdot b_1 - a_1 \cdot b_0$	smlsdx	$c + a_0 \cdot b_1 - a_1 \cdot b_0$

smuad, smuadx, smusd, smusdx, smlad, smladx, smlsd, smlsdx. smuad performs two multiplications of halfwords and adds the products together. smusd performs two multiplications of halfwords and the second product is subtracted from the first product. smlad and smlsd are the versions where the result is added to the accumulator. If there is an x following the instruction, then the halfwords of the second operand is specified in the reversed manner [ARM21b, Section A4.4.3]. Table 5.3 is an illustration.

Table 5.4: Signed most-significant-word multiplications where c is the destination register.

[instruction] c, a, b		[instruction] c, a, b	
[instruction]	c	[instruction]	c
smmul	$\left\lfloor \frac{ab}{2^{32}} \right\rfloor$	smmulr	$\left\lfloor \frac{ab}{2^{32}} \right\rceil$

smmul, smmulr. smmul multiplies two 32-bit values and returns the most significant 32-bit of the 64-bit product. smmulr returns the most significant 32-bit where the 64-bit product is first added with 2^{31} [ARM21b, Section A4.4.3]. Table 5.4 is an illustration.

Table 5.5: Packing instructions where c is the destination register.

[instruction] c, $a_1 \Vert a_0$, $b_1 \Vert b_0$		
[instruction]	Suffix	c
pkhbt	none	$b_1 \Vert a_0$
	, lsl #16	$b_0 \Vert a_0$
pkhtb	none	$b_0 \Vert a_1$
	, asr #16	$b_1 \Vert a_1$

pkhbt, pkhtb. pkhbt packs the lower 16-bit of the first operand and the upper 16-bit of the second operand together. One can append lsl #16 to the second operand so the lower 16-bit of both are packed. pkhtb packs the upper 16-bit of the first operand and the lower 16-bit of the second operand together. One can append asr

doi:10.6342/NTU202200738

#16 to the second operand so the upper 16-bit of both are packed [ARM21b, Section A4.4.5]. Table 5.5 is an illustration.

5.1.5 FPv4-SP Extension

There are 32 single-precision floating-point registers, named from s0 to s31 [ARM21b, Section A2.5]. Figure 5.2 is an illustration. FPv4-SP only supports single-precision arithmetic and we do not base our implementation on any floating-point arithmetic. However, on Cortex-M4, since transferring data between core registers and floating-point registers is much faster than memory operations, it is more favorable for caching some immediate values with floating-point registers during computation [ACC+21]. This can be achieved by the instruction vmov. vmov is mapped to several encodings and we only mention the most frequently used in our implementation. vmov sn, rt and vmov rt, sn transfer values between a single-precision floating-point register and a core register. vmov sn, sn1, rt, rt2 and vmov rt, rt2, sn, sn1 transfer values between two consecutive single-precision floating-point registers and two core registers [ARM21b, Section A4.12].

Figure 5.2: Floating-point registers in FPv4-SP extension.

s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15

s16	s17	s18	s19	s20	s21	s22	s23	s24	s25	s26	s27	s28	s29	s30	s31

5.2 64-bit Armv8-A

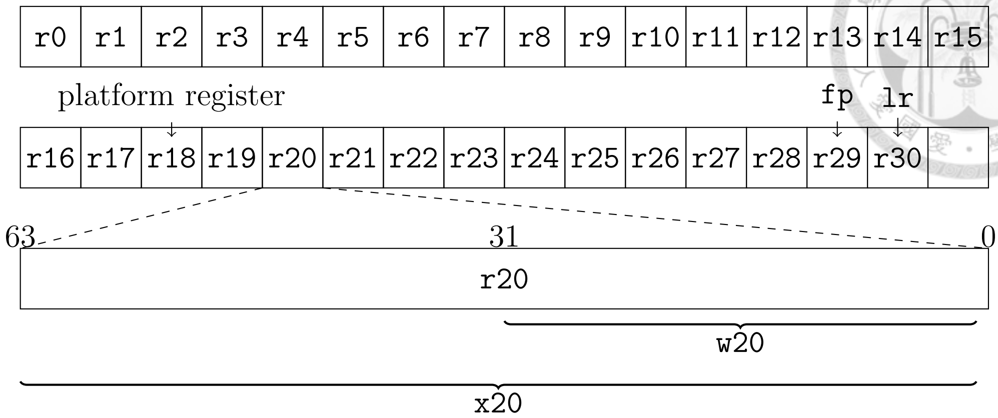
This section introduces the 64-bit instruction set architecture Armv8-A. We mainly focus on a small set of SIMD instructions.

5.2.1 General-Purpose Registers

There are 31 64-bit general-purpose registers, named from r0 to r30. Each register can be accessed as a 64-bit register xn or a 32-bit register wn [ARM21a, Section B1.2.1]. xn's and wn's are referred to as the 64-bit names and the 32-bit names of the general-purpose registers. x29 is the frame pointer fp and x30 is the link register lr. Unlike Armv7-M architecture, there are dedicated stack pointer sp and program counter pc. Furthermore, x18 is a platform register and shall not be used for platform-independent code [ARM21c, Section 6.1.1].

doi:10.6342/NTU202200738

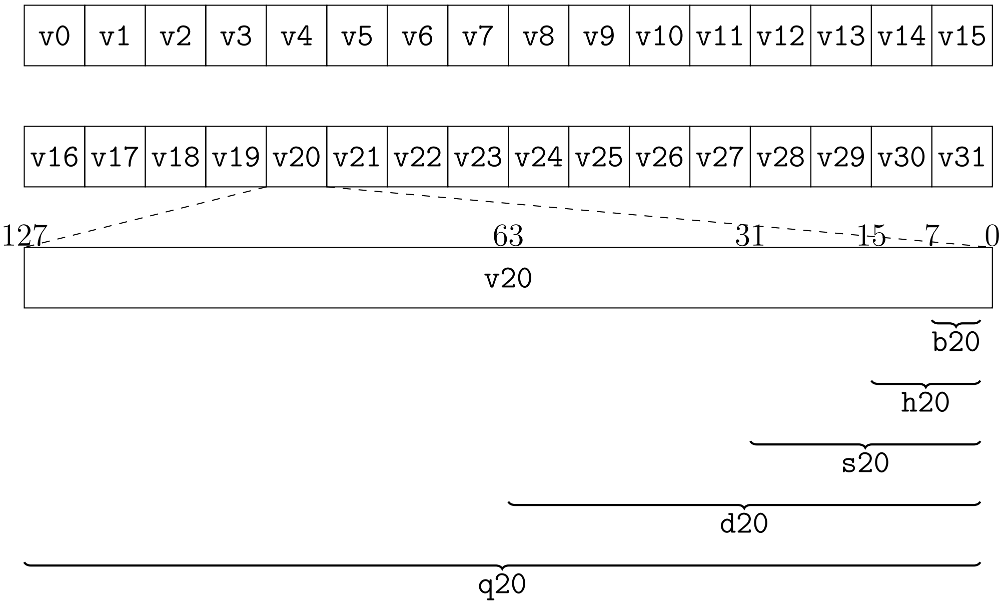
Figure 5.3: General-purpose registers in Armv8-A architecture.



5.2.2 SIMD Registers

There are 32 128-bit SIMD registers, named from `v0` to `v31`. For a SIMD register `vn`, there are several names for referring the lower $8 * 2^k$ bits where $k = 0, 1, \ldots, 4$. If $k = 0$, then the lower 8 bits form the register name `bn`. For $k = 1, 2, 3, 4$, the lower $8 * 2^k$ bits form the register names `hn`, `sn`, `dn`, and `qn`, respectively. Figure 5.4 is an illustration with `n` = `20` [ARM21a, Section B1.2.1].

Figure 5.4: SIMD&FP registers in Armv8-A architecture.



5.2.3 Instruction Format

Vector form and by-element form. In Armv8-A architecture, SIMD instructions with two input SIMD registers commonly have at least two different encodings. One is the encoding for vector form where both operands are treated as a vector of elements. The other one is the encoding for by-element form where the first operand is treated as a vector of elements and only the specified element in the second operand is used. We commonly use the by-element form of an instruction to

doi:10.6342/NTU202200738

save SIMD registers. This is applicable only if all the elements of the first operand share the same computation with the same value. If they share the same computation with different values, then we can only use the vector form of an instruction where the values are packed into the second operand.

Specifiers. For specifying the size of an element in a SIMD register, specifiers `B`, `H`, `S`, and `D` are introduced. `B` indicates that each element is an 8-bit value, while `H`, `S`, and `D` indicate that each element is a 16-bit, 32-bit, and 64-bit value, respectively. If a SIMD register is treated as a vector of elements, then we have to prepend the number of elements to `B`, `H`, `S`, and `D` [ARM21a, Section A1.4.1]. If only a specific element of a SIMD register is used, then we have to append the index `[index]` to `B`, `H`, `S`, and `D`. The indexing is the standard one where a SIMD register is regarded as an array of elements of the specified size.

Common features. Armv8-A architecture also provides several instructions operating and giving unequal size elements [ARM21a, Sections C3.5.1, F1.9.4]. They are categorized as follows:

- Suffix `l` indicates double-width results are computed where both of the operands are regarded as vectors of single-width elements.
- Suffix `w` indicates double-width results are computed where the first operand is regarded as a vector of double-width elements and the second operand is regarded as a vector of single-width elements.
- Suffix `n` indicates single-width results are computed where both of the operands are regarded as vectors of double-width elements.

If a SIMD register is regarded as a vector of single-width elements, then only a certain 64-bit is referenced. If there is no additional suffix, then the lower 64-bit is referenced. On the other hand, if there is a suffix `2`, then the upper 64-bit is referenced and the number in the specifier must be doubled. This feature is also called lane set specifiers [ARM21a, Section C3.5.1]. Table 5.6 is a summary.

Table 5.6: Lane set specifiers.

Suffix	Types of the SIMD registers	Ta	Tb
l	Ta, Tb, Tb	n(2X)	nX
l2	Ta, Tb, Tb	n(2X)	(2n)X
w	Ta, Ta, Tb	n(2X)	nX
w2	Ta, Ta, Tb	n(2X)	(2n)X
n	Tb, Ta, Ta	n(2X)	nX
n2	Tb, Ta, Ta	n(2X)	(2n)X

5.2.4 Instructions

Below we give an overview of the SIMD instructions that are used or related to the used and categorize them as follows:

doi:10.6342/NTU202200738

- Memory operations: `ldr`, `str`, `ld1`, `ld2`, `ld3`, `ld4`, `st1`, `st2`, `st3`, `st4`.
- Permutations: `ext`, `trn1`, `trn2`, `uzp1`, `uzp2`, `zip1`, `zip2`.
- Transferring operations: `dup`, `mov`.
- Comparisons: `cmeq`, `cmge`, `cmgt`, `cmhs`, `cmhi`.
- Shift operations: `shl`, `sshr`, `srshr`, `ushr`, `urshr`.
- Additions and subtractions: `add`, `sub`, `shadd`, `shsub`.
- Multiplications: `mul`, `mla`, `mls`, `smull`, `smull2`, `smlal`, `smlal2`, `smlsl`, `smlsl2`, `sqdmulh`, `sqrdmulh`, `sqrdmlah`, `sqrdmlsh`.

`ldr`, `str`, `ld1`, `ld2`, `ld3`, `ld4`, `st1`, `st2`, `st3`, `st4`. `ldr` loads one byte, one halfword, one word, one doubleword, or one quadword to a SIMD register. `ld1` loads multiple 1-element structures to one, two, three, or four consecutive SIMD registers. `ld2` loads multiple 2-element structures to two SIMD registers where the first lanes of the registers are holding the first structure and so on. `ld3` loads multiple 3-element structures to three SIMD registers with the similar interleaving as `ld2`. `ld4` is the 4-element-structure variant. `str`, `st1`, `st2`, `st3`, and `st4` are their reversals [ARM21a, Sections C3.2.9, C3.2.10]. Figures 5.5, 5.6, and 5.7 are word-wise examples of `ld2`, `st2`, `ld3`, `st3`, `ld4`, and `st4`.

Figure 5.5: `ld2` and `st2`.

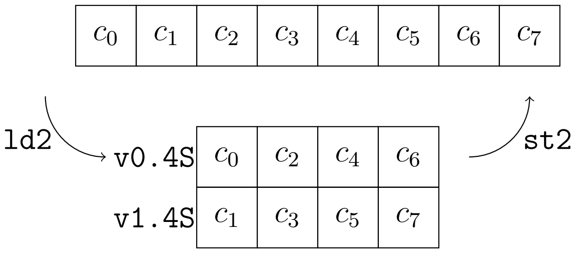
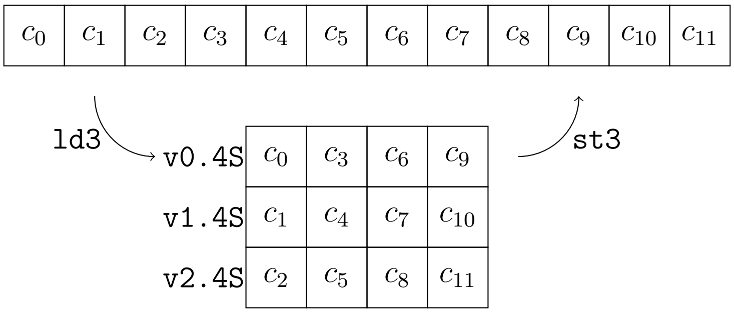
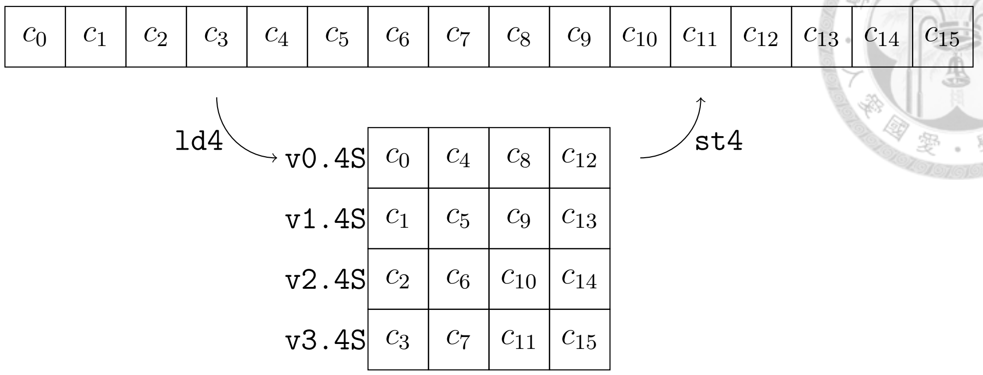

c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7
ld2
st2
v0.4S c_0 c_2 c_4 c_6
v1.4S c_1 c_3 c_5 c_7

Figure 5.6: `ld3` and `st3`.


c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11}
ld3
st3
v0.4S c_0 c_3 c_6 c_9
v1.4S c_1 c_4 c_7 c_{10}
v2.4S c_2 c_5 c_8 c_{11}

doi:10.6342/NTU202200738

Figure 5.7: `ld4` and `st4`.


c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15}

`ld4` `st4`

`v0.4S` c_0 c_4 c_8 c_{12}

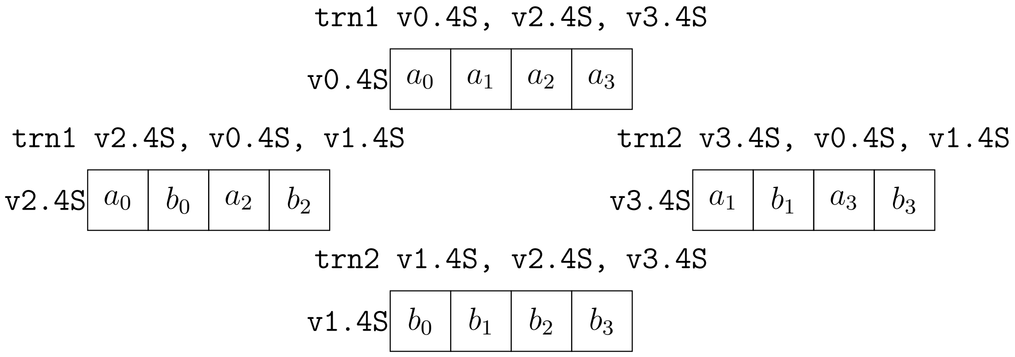
`v1.4S` c_1 c_5 c_9 c_{13}

`v2.4S` c_2 c_6 c_{10} c_{14}

`v3.4S` c_3 c_7 c_{11} c_{15}

`ext`, `trn1`, `trn2`, `uzp1`, `uzp2`, `zip1`, `zip2`. `ext` extracts contiguous blocks of vector elements from the two operands and combines them. `trn1` moves the even positions of the first operand to the even positions of the destination register and the even positions of the second operand to the odd positions of the destination register. `trn2` moves the odd positions instead. `uzp1` moves the even positions of the two operands to the first half of the destination register. `uzp2` moves the odd positions instead. `zip1` and `zip2` are the reversals of `uzp1` and `uzp2` [ARM21a, Section C3.5.19]. Figures 5.8, and 5.9 word-wise examples of `trn1`, `trn2`, `uzp1`, `uzp2`, `zip1`, and `zip2`.

Figure 5.8: `trn1` and `trn2`.


`trn1 v0.4S, v2.4S, v3.4S`

`v0.4S` a_0 a_1 a_2 a_3

`trn1 v2.4S, v0.4S, v1.4S`

`v2.4S` a_0 b_0 a_2 b_2

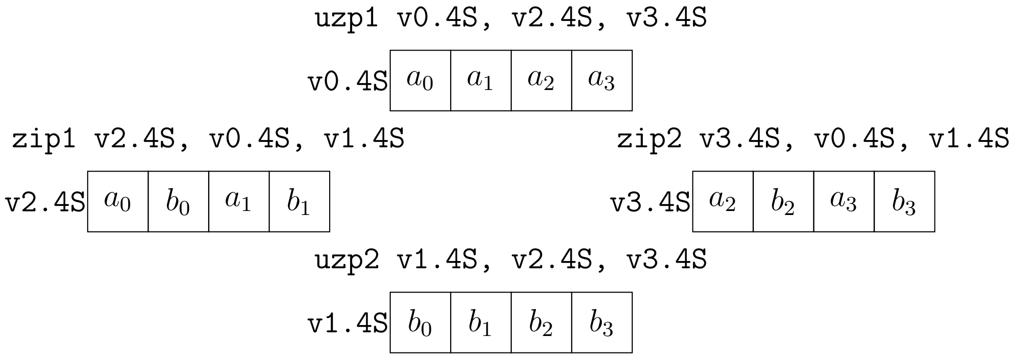
`trn2 v3.4S, v0.4S, v1.4S`

`v3.4S` a_1 b_1 a_3 b_3

`trn2 v1.4S, v2.4S, v3.4S`

`v1.4S` b_0 b_1 b_2 b_3

Figure 5.9: `zip1`, `zip2`, `uzp1`, and `uzp2`.


`uzp1 v0.4S, v2.4S, v3.4S`

`v0.4S` a_0 a_1 a_2 a_3

`zip1 v2.4S, v0.4S, v1.4S`

`v2.4S` a_0 b_0 a_1 b_1

`zip2 v3.4S, v0.4S, v1.4S`

`v3.4S` a_2 b_2 a_3 b_3

`uzp2 v1.4S, v2.4S, v3.4S`

`v1.4S` b_0 b_1 b_2 b_3

doi:10.6342/NTU202200738

Combining `trn1` and `trn2`: `trn_2x2`, `trn_4x4`, `trn_8x8`, `trn_16x16`. Combining `trn1` and `trn2` transposes SIMD registers as they are the rows of a matrix. `trn1` and `trn2` with `.2D` specifier alone transpose a 2 by 2 matrix as shown in Algorithm 4. Observing that transposing a 4 by 4 matrix M can be achieved by applying different resolutions of 2 by 2 transpositions on different combinations of registers, we can transpose M by applying `trn1` and `trn2` with `.4S` specifier following `.2D` specifier as shown in Algorithm 5. Similarly, we transpose 8 by 8 and 16 by 16 matrices as illustrated in Algorithms 6 and 7.

Algorithm 4 `trn_2x2` for doublewords (Armv8-A).

```
Inputs: ai = $(c_{2i}, c_{2i+1})$
Outputs: ai = $(c_i, c_{2+i})$
trn1 t0.2D, a0.2D, a1.2D
trn2 t1.2D, a0.2D, a1.2D
```

Algorithm 5 `trn_4x4` for words (Armv8-A).

```
Inputs: ai = $(c_{4i}, c_{4i+1} \ldots, c_{4i+3})$
Outputs: ai = $(c_i, c_{4+i}, \ldots, c_{12+i})$
(t0, t2) = trn_2x2(a0, a2)
(t1, t3) = trn_2x2(a1, a3)
trn1 a0.4S, t0.4S, t1.4S
trn2 a1.4S, t0.4S, t1.4S
trn1 a2.4S, t2.4S, t3.4S
trn2 a3.4S, t2.4S, t3.4S
```

Algorithm 6 `trn_8x8` for halfwords (Armv8-A).

```
Inputs: ai = $(c_{8i}, c_{8i+1} \ldots, c_{8i+7})$
Outputs: ai = $(c_i, c_{8+i}, \ldots, c_{56+i})$
(a0, a2, a4, a6) = trn_4x4(a0, a2, a4, a6)
(a1, a3, a5, a7) = trn_4x4(a1, a3, a5, a7)
trn1 t0.8H, a0.8H, a1.8H
trn2 t1.8H, a0.8H, a1.8H
trn1 t2.8H, a2.8H, a3.8H
trn2 t3.8H, a2.8H, a3.8H
trn1 t4.8H, a4.8H, a5.8H
trn2 t5.8H, a4.8H, a5.8H
trn1 t6.8H, a6.8H, a7.8H
trn2 t7.8H, a6.8H, a7.8H
```

doi:10.6342/NTU202200738

Algorithm 7 `trn_16x16` for bytes (Armv8-A).

```
Inputs: ai = $(c_{16i}, c_{16i+1} \dots, c_{16i+15})$
Outputs: ai = $(c_i, c_{16+i}, \dots, c_{240+i})$
(t0, t2, t4, t6, t8, t10, t12, t14) = trn_8x8(a0, a2, a4, a6, a8, a10, a12, a14)
(t1, t3, t5, t7, t9, t11, t13, t15) = trn_8x8(a1, a3, a5, a7, a9, a11, a13, a15)
trn1 a0.16B, t0.16B, t1.16B
trn2 a0.16B, t0.16B, t1.16B
trn1 a2.16B, t2.16B, t3.16B
trn2 a2.16B, t2.16B, t3.16B
trn1 a4.16B, t4.16B, t5.16B
trn2 a4.16B, t4.16B, t5.16B
trn1 a6.16B, t6.16B, t7.16B
trn2 a6.16B, t6.16B, t7.16B
trn1 a8.16B, t8.16B, t9.16B
trn2 a8.16B, t8.16B, t9.16B
trn1 a10.16B, t10.16B, t11.16B
trn2 a10.16B, t10.16B, t11.16B
trn1 a12.16B, t12.16B, t13.16B
trn2 a12.16B, t12.16B, t13.16B
trn1 a14.16B, t14.16B, t15.16B
trn2 a14.16B, t14.16B, t15.16B
```

`dup`, `ins`, `mov`. `dup` duplicates a scalar to a vector or a vector element. `ins` inserts a scalar or a vector element into a vector element. `mov` moves a scalar to a vector element or a general-purpose register [ARM21a, Section C3.5.13].

`cmeq`, `cmge`, `cmgt`, `cmhs`, `cmhi`. SIMD comparisons place -1 to the corresponding positions if the result is true and 0 otherwise. `cmeq` tests if two vector elements are equal. For each vector element of the first operand, `cmgt` tests if it is greater than the corresponding vector element of the second operand with signed representation. `cmge` is testing the greater-or-equal-than condition with signed representation. `cmgs` and `cmhi` are unsigned comparisons [ARM21a, Section C3.5.15].

`shl`, `sshr`, `srshr`, `ushr`, `urshr`. `shl` performs logical shift left on the vector elements. `sshr` performs arithmetic shift right and `srshr` is the rounding verion. `ushr` and `urshr` are their logical counterparts [ARM21a, Section C3.5.14].

`add`, `sub`, `shadd`, `uhadd` `shsub`, `uhsub`. `add` adds the corresponding vector elements and `sub` subtracts vector elements of the second operand from the vector elements of the first operand. `shadd` is equivalent to `add` followed by `sshr #1` and `shsub` is equivalent to `sub` followed by `sshr #1`. `uhadd` and `uhsub` are their unsigned counterparts [ARM21a, Section C3.5.14].

Reducing $[-q, q]$ to $(-\frac{q}{2}, \frac{q}{2})$ for odd q. Suppose q is odd. A straightforward approach for reducing vector elements from $(-q, q)$ to $(-\frac{q}{2}, \frac{q}{2})$ is realized with the sequence $($`cmgt`, `mls`, `cmgt`, `mla`$)$ as shown in Algorithm 8. Since at most one of the conditions is satisfied, we can convert one `mla` or `mls` into `sub` as shown in

doi:10.6342/NTU202200738

Algorithm 9. Additionally, we can also convert each multiplication into an `and` followed by an `add` or a `sub`. Algorithm 8 is converted into Algorithm 10 and Algorithm 9 is converted into Algorithm 11.

Algorithm 8 $[-q, q] \to (-\frac{q}{2}, \frac{q}{2})$ with two multiplications for odd q (Armv8-A).

```
Inputs: a = $c \in [-q, q]$
Outputs: a = $c \bmod^{\pm} q \in (-\frac{q}{2}, \frac{q}{2})$
cmgt t, $-\lfloor \frac{q}{2} \rfloor$, a
mls  a, t, $q$
cmgt t, a, $\lfloor \frac{q}{2} \rfloor$
mla  a, t, $q$
```

Algorithm 9 $[-q, q] \to (-\frac{q}{2}, \frac{q}{2})$ with one multiplication for odd q (Armv8-A).

```
Inputs: a = $c \in [-q, q]$
Outputs: a = $c \bmod^{\pm} q \in (-\frac{q}{2}, \frac{q}{2})$
cmgt t0, $-\lfloor \frac{q}{2} \rfloor$, a
cmgt t1, a, $\lfloor \frac{q}{2} \rfloor$
sub  t0, t0, t1
mls  a, t0, $q$
```

Algorithm 10 $[-q, q] \to (-\frac{q}{2}, \frac{q}{2})$ with two `and`'s for odd q (Armv8-A).

```
Inputs: a = $c \in [-q, q]$
Outputs: a = $c \bmod^{\pm} q \in (-\frac{q}{2}, \frac{q}{2})$
cmgt t, $-\lfloor \frac{q}{2} \rfloor$, a
and  t, t, $q$
add  a, a, t
cmgt t, a, $\lfloor \frac{q}{2} \rfloor$
and  t, t, $q$
sub  a, a, t
```

Algorithm 11 $[-q, q] \to (-\frac{q}{2}, \frac{q}{2})$ with one `and` for odd q (Armv8-A).

```
Inputs: a = $c \in [-q, q]$
Outputs: a = $c \bmod^{\pm} q \in (-\frac{q}{2}, \frac{q}{2})$
cmgt t0, $-\lfloor \frac{q}{2} \rfloor$, a
cmgt t1, a, $\lfloor \frac{q}{2} \rfloor$
sub  t0, t0, t1
and  t0, t0, $q$
add  a, a, t0
```

`mul`, `mla`, `mls`, `smull`, `smull2`, `smlal`, `smlal2`, `smlsl`, `smlsl2`, `sqdmulh`, `sqrdmulh`, `sqrdmlah`, `sqrdmlsh`. `mul` multiplies vector elements by the corresponding vector elements. `mla` adds the result to the accumulator and `mls` subtracts the result from the accumulator. `smull` computes the double-size product of the corresponding vector elements in the lower 64-bit. The 128-bit result is then stored to the destination SIMD register. `smull2` computes the double-size product from the upper 64-bit. `smlal` and `smlal2` add the result to the 128-bit accumulator and `smlsl` and `smlsl2` subtract the result from the 128-bit accumulator. `sqdmulh` computes the high product, doubles the result, and stores the final value to the destination SIMD register. `sqrdmulh` rounds the result before doubling. `sqrdmlah` adds the result to the accumulator and `sqrdmlsh` subtracts the result from the accumulator [ARM21a, Sections C3.5.14, C3.5.16]. Table 5.7 is the summary.

Table 5.7: Instructions returning high product where `d` is the destination register.

`[instruction] d, a, b`		`[instruction] d, a, b`	
`[instruction]`	`d[i]`	`[instruction]`	`d[i]`
`sqdmulh`	$\left\lfloor \frac{\texttt{2a[i]b[i]}}{\texttt{R}} \right\rfloor$	`sqrdmlah`	$\texttt{d[i]} + \left\lfloor \frac{\texttt{2a[i]b[i]}}{\texttt{R}} \right\rceil$
`sqrdmulh`	$\left\lfloor \frac{\texttt{2a[i]b[i]}}{\texttt{R}} \right\rceil$	`sqrdmlsh`	$\texttt{d[i]} - \left\lfloor \frac{\texttt{2a[i]b[i]}}{\texttt{R}} \right\rceil$

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Chapter 6

Central Processing Units

This chapter introduces the Central Processing Units used for experiments and is for referential purposes. We mainly follow the official documents. Furthermore, for memory operations, we do not discuss PC-relative operations as they are not used in this thesis. No contribution is claimed in this chapter.

6.1 Cortex-M4 Instruction Timing

Arm Cortex-M4 is a microcontroller implementing Armv7E-M architecture. It is a single-issue processor with a 3-stage pipeline. Below is the summary of the instruction timing from the Technical Reference Manual of Cortex-M4 [ARM10b].

Memory operations. A store instruction with an immediate offset is always 1 cycle. A load instruction with base update is not pipelined unless the next instruction does not read from a register. A single load instruction takes 2 cycles. If the load is followed by a non-base updating store or a load from an address that is not computed from the destination register, then they are pipelined and 1 cycle is removed. Load instructions with base update is pipelined when possible.

bitwise operations, shift operations, addition/subtraction, and bit-field operations, multiplications, and DSP extension. All bitwise operations, shift operations, addition/subtraction, and bit-field operations, multiplications, and instructions in DSP extension take 1 cycle. Cortex-M4 also implements a barrel shifter so that instructions `asr`, `lsr`, `lsl`, and `ror` take no cycles when they are appended to the `Rm` field of standard data-processing instructions.

`vmov`. `vmov` takes 1 cycle if it is transferring a floating-point register to or from a general-purpose register. `vmov` also takes 1 cycle for transferring an immediate or a floating-point register to a floating-point register. If `vmov` is transferring two consecutive floating-point registers to or from two general-purpose registers, it takes 2 cycles. Same hold for transferring two consecutive floating-point registers to or from two consecutive floating-point registers. The instruction timing is derived by experiment since the technical reference is confusing.

`it`. Instruction `it` can be folded onto a preceding 16-bit Thumb instruction and takes no cycles.

doi:10.6342/NTU202200738

6.2 Cortex-M3 Instruction Timing

Arm Cortex-M3 is a microcontroller implementing Armv7-M architecture. It is a single-issue processor with a 3-stage pipeline. Below is the summary of the instruction timing from the Technical Reference Manual of Cortex-M3 [ARM10a].

Memory operations. Almost all timings follow from Cortex-M4 except that load instructions with base update are not pipelined.

Bitwise operations, shift operations, addition/subtraction, and bit-field operations. All bitwise operations, shift operations, addition/subtraction, and bit-field operations take 1 cycle. Cortex-M4 also implements a barrel shifter.

Multiplications. `mul` takes 1 cycle while `mla` and `mls` take 2 cycles. `smull`, `smlal`, `umull`, and `umlal` are early-terminating instructions. We avoid any uses of these instructions on Cortex-M3.

6.3 Cortex-A72

6.3.1 Pipeline

Arm Cortex-A72 implements the 64-bit Armv8-A architecture. The pipeline of Cortex-A72 consists of the in-order frontend and the out-of-order backend. We mainly follow the Software Optimization Guide of Cortex-A72 [ARM15].

In-order frontend. In the in-order frontend, instructions are fetched and decoded into internal micro-operations (μops). After renaming the registers, μops are dispatched to the out-of-order backend in the oldest-to-youngest age order.

Out-of-order backend. In the out-of-order backend, μops are issued to one of the eight pipelines. Each pipeline can complete one μop in one cycle. There is one branch pipeline `B`, two integer pipelines `I0` and `I1`, one integer multi-cycle pipeline `M`, two FP/ASIMD pipelines `F0` and `F1`, one load pipeline `L`, and one store pipeline `S`.

3-way decoding. Cortex-A72 can decode up two three instructions in a single cycle. However, there are limitations on the number of each type of μops that can be dispatched simultaneously. The following are the numbers for each type in a single cycle: one μop using `B`, up to two μops using `I0`/`I1`, up to two μops using `M`, one μop using `F0`, one μop using `F1`, and up to two μops using `L`/`S`.

6.3.2 Instruction Timing

In this thesis, we only focus on the pipelines `F0` and `F1`. In particular, we restrict the discussion to the Q-form of the instructions. The following is the summary of the utilized pipelines and timings of the instructions.

doi:10.6342/NTU202200738

Logical operations, addition/subtraction, and permutations. `and`, `add`, `sub`, `uzp1`, `uzp2`, `trn1`, `trn2` can be dispatched to `F0` and `F1`. Each of them takes only $\frac{1}{2}$ cycles.

Shift operations with immediates. `sshr`, `srshr` can only be dispatched to `F1`. Each of them takes 1 cycle.

Multiplications. `sqdmulh`, `sqrdmulh`, `mul`, `smull`, `smull2`, `smlal`, and `smlal2` can only be dispatched to `F0`. Single-width multiplications `sqdmulh`, `sqrdmulh`, and `mul` take 2 cycles. Multiplications with widening `smull`, `smull2`, `smlal`, and `smlal2` take 1 cycle.

doi:10.6342/NTU202200738

國立臺灣大學 · 愛國 · 愛學 · 勵品 · 敦

doi:10.6342/NTU202200738

Chapter 7

Building Blocks

In this section, we go through the implementations of various building blocks used in NTTs. The building blocks are all implemented with macros for easier uses in assembly programming.

Implementations and ideas from the following works are contributions of this thesis with some exceptions.

- Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127-151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292. Reference [ACC+22]. Full version available at https://eprint.iacr.org/2021/995.

- Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221-224, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. Reference [BHK+22]. Full version available at https://eprint.iacr.org/2021/986.

There are contributions that I do not claim for. In [ACC+22], the 32-bit NTT-based Saber on Cortex-M3 was implemented by Amin Abdulrahman and is not a contribution of this thesis. In [BHK+22], the correspondence between Barrett reduction and Montgomery reduction, and the resulting Barrett multiplication are not contributions of this thesis. I don't not claim to be proposing the idea but integrating the idea into the implementations with my best effort. Furthermore, Sections 7.2.6 and 7.2.7 are contributions of the submitted work:

- Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. Submitted in April 2022.

7.1 Modular Reductions and Multiplications

This section introduces the implementations of modular reductions and multiplications. Given a positive integer $q \in \mathbb{N}$, we are interested in the signed values in $\mathbb{Z}_q$.

doi:10.6342/NTU202200738

In other words, the integers in $[-\frac{q}{2}, \frac{q}{2})$ are what we are looking for, and we now regard $\mathbb{Z}_q$ as the set of integers $[-\frac{q}{2}, \frac{q}{2})$. We denote $a \bmod {}^{\pm}q$ as the representative of a in $\mathbb{Z}_q$ for an integer a. Since it is expensive to compute the results in $\mathbb{Z}_q$, we usually don't require intermediate results to be in $\mathbb{Z}_q$. We only reduce the value to avoid overflow. Since division instructions commonly take input-dependent time to finish, we cannot base cryptographic implementations on divisions. Instead, we implement Barrett reduction by [Bar86], Montgomery reduction and multiplication by [Mon85], and Barrett multiplication by [BHK+22].

Throughout this section, we assume $\mathtt{R}$ is a power of 2.

7.1.1 Barrett Reduction

Following [Bar86], Barrett reduction reduces a value a by approximating $a \bmod {}^{\pm}q = a - \left\lfloor \frac{a}{q} \right\rceil q$ with $\mathsf{barrett}_{q,\mathtt{R}}(a) \coloneqq a - \left\lfloor \frac{a \cdot \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil q$ as shown in Definition 67. Since $\mathtt{R}$ and q are known constants, we can precompute $\left\lfloor \frac{\mathtt{R}}{q} \right\rceil$ to avoid division by q. Now, we only need multiplications and a division by $\mathtt{R}$. Commonly, there are input-independent instructions realizing multiplications and divisions by $\mathtt{R}$. Barrett reduction is defined as follows.

Definition 67 (Barrett reduction [Bar86])**.** Let $q, \mathtt{R}$ be positive constants known in prior. For an integer a, Barrett reduction computes

$$\mathsf{barrett}_{q,\mathtt{R}}(a) \coloneqq a - \left\lfloor \frac{a \cdot \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil \cdot q$$

as an approximation of $a - \left\lfloor \frac{a}{q} \right\rceil q$.

We denote $\mathsf{barrett}(\cdot)$ for $\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$ when the context is clear.

7.1.2 Montgomery Reduction and Multiplication

Let a and b be values we wish to multiply, and q be an odd constant. Instead of computing $ab \bmod {}^{\pm}q$, Montgomery multiplication computes $ab\mathtt{R}^{-1} \bmod {}^{\pm}q$ by applying Montgomery reduction to the full-size product ab.

To understand the internal of Montgomery reduction, we introduce a constant $q' = -q^{-1} \bmod {}^{\pm}\mathtt{R}$. Since $0 \equiv 1 + q' \cdot q \pmod{\mathtt{R}}$, for arbitrary a, $a + (a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q$ is divisible by $\mathtt{R}$ and $\frac{a+(a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}$ is an integer. Furthermore, since $a \equiv \mathtt{R} \cdot \frac{a+(a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}} \pmod{q}$, it follows that $\frac{a+(a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}} \equiv a\mathtt{R}^{-1} \pmod{q}$. We define $\mathsf{montgomery}_{q,\mathtt{R}}(a) \coloneqq \frac{a+(a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}$ and formulate Montgomery reduction as Definition 68.

Definition 68 (Montgomery reduction [Mon85])**.** Let $q, \mathtt{R}$ be positive constants known in prior with $q \perp \mathtt{R}$. For an integer a, Montgomery reduction computes

$$\mathsf{montgomery}_{q,\mathtt{R}}(a) \coloneqq \frac{a + (a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}$$

satisfying $\mathsf{montgomery}_{q,\mathtt{R}}(a) \equiv a\mathtt{R}^{-1} \pmod{q}$.

doi:10.6342/NTU202200738

To see why the computation is called a "reduction," we assume $|a| \le \frac{\mathtt{R}}{2}$ and $q < \mathtt{R}$. Then, the absolute value of $\mathsf{montgomery}_{q,\mathtt{R}}(a)$ is bounded by

$$\begin{aligned}
\left|\mathsf{montgomery}_{q,\mathtt{R}}(a)\right| &= \left|\frac{a + (a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}\right| \\
&\le \frac{|a + (a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q|}{\mathtt{R}} \\
&\le \frac{\left|a + \frac{\mathtt{R}}{2} \cdot q\right|}{\mathtt{R}} \\
&\le \frac{|a|}{\mathtt{R}} + \frac{q}{2}
\end{aligned}$$

Since $\mathsf{montgomery}_{q,\mathtt{R}}(a)$ is an integer and $\frac{|a|}{\mathtt{R}}$ is a fraction, $\left|\mathsf{montgomery}_{q,\mathtt{R}}(a)\right| \le \frac{q}{2}$ [Mon85, Sei18].

Now we turn to Montgomery multiplication. Given two values a and b, we first derive the full-size product $a \cdot b$. After applying Montgomery reduction to ab, we now have $\mathsf{montgomery}_{q,\mathtt{R}}(ab) \equiv ab\mathtt{R}^{-1} \pmod{q}$ and $\left|\mathsf{montgomery}_{q,\mathtt{R}}(ab)\right| \le \frac{|ab|}{\mathtt{R}} + \frac{q}{2}$. Montgomery multiplication, denoted as $\mathsf{montgomery}_{q,\mathtt{R}}(a, b)$, is the composition of Montgomery reduction and multiplication of a by b. We formulate Montgomery multiplication as Definition 69 and call it a multiplicative form of $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot)$.

Definition 69 (Montgomery multiplication [Mon85])**.** Let $q, \mathtt{R}$ be positive constants known in prior with $q \perp \mathtt{R}$. For integers a and b, Montgomery multiplication computes

$$\mathsf{montgomery}_{q,\mathtt{R}}(a, b) := \frac{a \cdot b + ((a \cdot b) \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}$$

such that $\mathsf{montgomery}q, \mathtt{R}(a, b) \equiv ab\mathtt{R}^{-1} \pmod{q}$ and $|\mathsf{montgomery}q, \mathtt{R}(a, b)| \le \frac{|ab|}{\mathtt{R}} + \frac{q}{2}$.

We now distinguish two cases on how to mitigate the extra $\mathtt{R}^{-1} \pmod{q}$. First of all, if b is known prior, we precompute $b\mathtt{R} \bmod {}^{\pm}q$ and apply Montgomery reduction to $a \cdot (b\mathtt{R} \bmod {}^{\pm}q)$ during the computation. The result $\mathsf{montgomery}_{q,\mathtt{R}}(a, b\mathtt{R} \bmod {}^{\pm}q)$ now satisfies $\mathsf{montgomery}_{q,\mathtt{R}}(a, b\mathtt{R} \bmod {}^{\pm}q) \equiv ab \pmod{q}$. On the other hand, if both a and b are unknown, we perform Montgomery multiplication with the constant $\mathtt{R}^2 \bmod {}^{\pm}q$ right before deriving the correct result in $\mathbb{Z}_q$.

For brevity, we denote $\mathsf{montgomery}(\cdot)$ for $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot)$ and $\mathsf{montgomery}(\cdot, \cdot)$ for $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot, \cdot)$ when the context is clear.

7.1.3 Barrett–Montgomery Correspondences

This thesis implements the "Barrett multiplication" introduced by [BHK+22]. The Barrett multiplication can be regarded as a multiplicative form extended from $\mathsf{barrett}(\cdot)$. The multiplicative form for unsigned $\mathsf{barrett}(\cdot)$ was known by Shoup, see [Har14].

Two key observations are required for deriving the extension. The first one is the relation between $\mathsf{montgomery}(\cdot)$ and $\mathsf{montgomery}(\cdot, \cdot)$, and the second one is the relation between $\mathsf{barrett}(\cdot)$ and $\mathsf{montgomery}(\cdot)$. [Mon85] introduced the relation between $\mathsf{montgomery}(\cdot)$ and $\mathsf{montgomery}(\cdot, \cdot)$. Recently, [BHK+22] established the relation between $\mathsf{barrett}(\cdot)$ and $\mathsf{montgomery}(\cdot)$. We present some ideas by [BHK+22]

doi:10.6342/NTU202200738

that are necessary for understanding the implementations. In particular, we base our implementations on Barrett multiplication with rounding, $\lfloor \rceil$, as an integer approximation. Interested readers are encouraged to look at the paper to understand various modular multiplications thoroughly.

Theorem 8 (($\lfloor \rceil$-type) Barrett–Montgomery correspondence of modular reductions [BHK+22, Proposition 1])**.** Let $\mathtt{R}$ be a power of 2 and q be an odd integer known in prior. For an integer a, we have

$$\mathsf{barrett}_{q,\mathtt{R}}(a) = \mathsf{montgomery}_{q,\mathtt{R}}(a(\mathtt{R} \bmod {}^{\pm}q)).$$

Proof. See [BHK+22, Section 3.1.1]. □

Definition 70 (($\lfloor \rceil$-type) Barrett multiplication [BHK+22])**.** Let $\mathtt{R}$ be a power of 2, q be an odd integer, and b be an integer known in prior. For an integer a, Barrett multiplication computes

$$\mathsf{barrett}_{q,\mathtt{R}}(a, b) := a \cdot b - \left\lfloor \frac{a \cdot \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil \cdot q$$

as a representative of $ab \bmod {}^{\pm}q$.

For the correctness, we apply the Barrett–Montgomery correspondence of modular multiplications described as follows.

Theorem 9 (($\lfloor \rceil$-type) Barrett–Montgomery correspondence of modular multiplications [BHK+22, Proposition 2])**.** Let $\mathtt{R}$ be a power of 2, q be an odd integer, and b be an integer known in prior. For an integer a, we have

$$\mathsf{barrett}_{q,\mathtt{R}}(a, b) = \mathsf{montgomery}_{q,\mathtt{R}}(a, b\mathtt{R} \bmod {}^{\pm}q).$$

Proof. See [BHK+22, Section 3.1.2]. □

7.1.4 Cortex-M3 Implementations

We go through the Cortex-M3 implementations of modular reductions and multiplications with Armv7-M.

$\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$**.** Barrett reduction reduces the magnitude of a by computing

$$\mathsf{barrett}_{q,\mathtt{R}}(a) = a - \left\lfloor \frac{a \cdot \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil \cdot q.$$

Since rounding to an integer is equivalent to the composition of addition by 0.5 and truncation, we turn the composition of division by $\mathtt{R}$ and rounding into the composition of addition with $0.5\mathtt{R}$, division by $\mathtt{R}$, and truncation. In particular, $\left\lfloor \frac{a \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil$ is implemented as

$$\left\lfloor \frac{a \left\lfloor \frac{\mathtt{R}}{q} \right\rceil + 0.5\mathtt{R}}{\mathtt{R}} \right\rfloor$$

doi:10.6342/NTU202200738

with instructions `mul`, `add`, and `asr`. After computing $\left\lfloor \frac{a\left\lfloor \frac{\mathtt{R}}{q}\right\rceil}{\mathtt{R}} \right\rceil$, we compute $a - \left\lfloor \frac{a\left\lfloor \frac{\mathtt{R}}{q}\right\rceil}{\mathtt{R}} \right\rceil \cdot q$ with `mls` [GKS21]. While deriving the cycles spent on $\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$, we notice that each of `mul`, `add`, and `asr` takes one cycle, and `mls` takes two cycles. Therefore, five cycles are spent on $\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$. Algorithm 12 is an illustration.

```
Algorithm 12 $\log_2 \mathtt{R}$-bit Barrett reduction (Cortex-M3) [GKS21].
Signature: barrett a, t
Inputs: $\mathtt{a} = a$
Outputs: $\mathtt{a} = a - \left\lfloor \frac{a\left\lfloor \frac{\mathtt{R}}{q}\right\rceil}{\mathtt{R}} \right\rceil q \equiv a \pmod{q}$
1: mul.w t, a, $\left\lfloor \frac{\mathtt{R}}{q} \right\rceil$        ▷ $\mathtt{t} = a\left\lfloor \frac{\mathtt{R}}{q} \right\rceil$
2: add.w t, t, $0.5\mathtt{R}$        ▷ $\mathtt{t} = a\left\lfloor \frac{\mathtt{R}}{q} \right\rceil + 0.5\mathtt{R}$
3: asr.w t, t, #$\log_2 \mathtt{R}$        ▷ $\mathtt{t} = \left\lfloor \frac{a\left\lfloor \frac{\mathtt{R}}{q}\right\rceil}{\mathtt{R}} \right\rceil$
4: mls a, t, $q$, a        ▷ $\mathtt{a} = a - \left\lfloor \frac{a\left\lfloor \frac{\mathtt{R}}{q}\right\rceil}{\mathtt{R}} \right\rceil \cdot q$
```

$\mathsf{montgomery}_{q,\mathtt{R}}(\cdot)$ and $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot,\cdot)$. Montgomery reduction computes

$$\mathsf{montgomery}_{q,\mathtt{R}}(a) = \frac{a + (a \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}$$

as a representative of $a\mathtt{R}^{-1} \bmod {}^{\pm}q$ and Montgomery multiplication computes

$$\mathsf{montgomery}_{q,\mathtt{R}}(a, b) = \frac{a \cdot b + ((a \cdot b) \cdot q' \bmod {}^{\pm}\mathtt{R}) \cdot q}{\mathtt{R}}$$

for $ab\mathtt{R}^{-1} \bmod {}^{\pm}q$ where $q' = -q^{-1} \bmod {}^{\pm}\mathtt{R}$. We fix $\mathtt{R} = 2^{16}$ for Cortex-M3 implementations, since we exclude the use of long multiplications. For $\mathsf{montgomery}_{q,2^{16}}(\cdot)$, we first multiply a by q' with `mul`. Since $\mathtt{R} = 2^{16}$, we extract $aq' \bmod {}^{\pm}2^{16}$ by sign-extending the lower 16-bit value with `sxth`. Then, we compute $a + (aq' \bmod {}^{\pm}2^{16})q$ with `mla`, and right-shift the value by 16 bits with `asr` for the division by 2^{16} [GKS21]. Except for `mla` being two cycles, other instructions take one cycle each. Therefore, five cycles are spent on $\mathsf{montgomery}_{q,2^{16}}(\cdot)$. For $\mathsf{montgomery}_{q,2^{16}}(\cdot,\cdot)$, we multiply the values with `mul` first and apply $\mathsf{montgomery}_{q,2^{16}}(\cdot,\cdot)$, spending six cycles in total. Algorithm 13 is an illustration for $\mathsf{montgomery}_{q,2^{16}}(\cdot)$, and Algorithm 14 is for $\mathsf{montgomery}_{q,2^{16}}(\cdot,\cdot)$.

```
Algorithm 13 16-bit Montgomery reduction (Cortex-M3) [GKS21].
Signature: montgomery_reduce_16 a, t
Inputs: $\mathtt{a} = a$
Outputs: $\mathtt{a} = \frac{a+(aq' \bmod {}^{\pm}2^{16})q}{2^{16}} \equiv a2^{-16} \pmod{q}$
1: mul t, a, $q'$        ▷ $\mathtt{t} = aq'$
2: sxth.w t, t        ▷ $\mathtt{t} = aq' \bmod {}^{\pm}2^{16}$
3: mla a, t, $q$, a        ▷ $\mathtt{a} = a + (aq' \bmod {}^{\pm}2^{16})q$
4: asr a, a, #16        ▷ $\mathtt{a} = \frac{a+(aq' \bmod {}^{\pm}2^{16})q}{2^{16}}$
```

doi:10.6342/NTU202200738

Algorithm 14 16-bit Montgomery multiplication (Cortex-M3) [GKS21].

```
Signature: montgomery_mul_16 a, b, t
Inputs: $(\texttt{a}, \texttt{b}) = (a, b)$
Outputs: $\texttt{a} = \frac{ab+(abq' \bmod^{\pm} 2^{16})q}{2^{16}} \equiv ab2^{-16} \pmod{q}$
  mul a, a, b          ▷ $\texttt{a} = ab$
  mul t, a, $q'$       ▷ $\texttt{t} = abq'$
  sxth.w t, t          ▷ $\texttt{t} = abq' \bmod^{\pm} 2^{16}$
  mla a, t, $q$, a     ▷ $\texttt{a} = ab + (abq' \bmod^{\pm} 2^{16})q$
  asr a, a, #16        ▷ $\texttt{a} = \frac{ab+(abq' \bmod^{\pm} 2^{16})q}{2^{16}}$
```

7.1.5 Cortex-M4 Implementations

We go through the implementations of modular reductions and multiplications on Cortex-M4. Compared to Cortex-M3, we unlock two groups of instructions. The first group is the DSP extension that implements halfwords multiplications, dual halfwords multiplications, signed most-significant-word multiplications, and pack instructions. The second group is long multiplications since they have input-independent behavior on Cortex-M4.

Instructions listed below are used for implementing modular reductions and multiplications on Cortex-M4.

1. DSP extension: `smulbb`, `smulbt`, `smultb`, `smultt`, `smlabb`, `smlabt`, `smlatb`, `smlatt`, `smmulr`, `pkhbt`, `pkhtb`.

2. Long multiplications: `smull`, `smlal`.

We distinguish two cases: $\texttt{R} = 2^{32}$ and $\texttt{R} = 2^{16}$. Both cases allow very efficient implementations with the newly unlocked instructions.

$\texttt{R} = 2^{32}$**.** For implementing

$$\mathsf{barrett}_{q,2^{32}}(a) = a - \left\lfloor \frac{a \cdot \left\lfloor \frac{2^{32}}{q} \right\rceil}{2^{32}} \right\rceil \cdot q,$$

we utilize the instruction `smmulr` which performs the map $(a, b) \mapsto \left\lfloor \frac{ab}{2^{32}} \right\rceil$ (cf. Section 5.1.4). After computing $\left\lfloor \frac{a \left\lfloor \frac{2^{32}}{q} \right\rceil}{2^{32}} \right\rceil$ with `smmulr`, we compute $a - \left\lfloor \frac{a \left\lfloor \frac{2^{32}}{q} \right\rceil}{2^{32}} \right\rceil q$ with `mls` [ACC+21]. Since both `smmulr` and `mls` take one cycle, $\mathsf{barrett}_{q,2^{32}}(\cdot)$ takes two cycles. Algorithm 15 is an illustration.

doi:10.6342/NTU202200738

Algorithm 15 32-bit Barrett reduction (Cortex-M4) [ACC+21].

```
Signature:  barrett_32 a, t
Inputs: a = $a$
Outputs: a = $a - \left\lfloor \frac{a \left\lfloor \frac{2^{32}}{q} \right\rceil}{2^{32}} \right\rceil q \equiv a \pmod{q}$
smmulr t, a, $\left\lfloor \frac{2^{32}}{q} \right\rceil$        ▷ t = $a \left\lfloor \frac{\left\lfloor \frac{2^{32}}{q} \right\rceil}{2^{32}} \right\rceil$
mls a, t, $q$, a                     ▷ a = $a - \left\lfloor \frac{a \left\lfloor \frac{2^{32}}{q} \right\rceil}{2^{32}} \right\rceil q$
```

For implementing

$$\mathsf{montgomery}_{q,2^{32}}(a,b) = \frac{ab + (abq' \bmod {}^{\pm}2^{32})q}{2^{32}}$$

with $q' = -q^{-1} \bmod {}^{\pm}2^{32}$, we first derive 2^{32}`upper` + `lower` $= ab$ with `smull`. Since `lower` $= ab \bmod {}^{\pm}2^{32}$, we apply `mul` by q' to dervie $abq' \bmod {}^{\pm}2^{32}$. Finally, we `smlal` to derive 2^{32}`upper` + `lower` $= ab + (abq' \bmod {}^{\pm}2^{32})q$. As $ab + (abq' \bmod {}^{\pm}2^{32})q$ is divisible by 2^{32}, `lower` $= 0$ and `upper` $= \frac{ab+(abq' \bmod {}^{\pm}2^{32})q}{2^{32}}$ [ACC+21]. For the timing, since each of `smull`, `mul`, and `smlal` takes one cycle, $\mathsf{montgomery}_{q,2^{32}}(\cdot,\cdot)$ takes three cycles. Algorithm 16 is an illustration, and Algorithm 17 is an adaption overwriting one of the inputs.

Algorithm 16 32-bit Montgomery multiplication (Cortex-M4) [ACC+21].

```
Signature:  montgomery_mul_des_32 lower, upper, a, b, t
Inputs: (a, b) = $(a, b)$
Outputs: upper = $\frac{ab+(abq' \bmod {}^{\pm}2^{32})q}{2^{32}} \equiv ab2^{-32} \pmod{q}$
smull lower, upper, a, b                    ▷ $2^{32}$upper + lower = $ab$
mul t, lower, $q'$                              ▷ t = $abq' \bmod {}^{\pm}2^{32}$
smlal lower, upper, t, $q$       ▷ Regard t = $abq' \bmod {}^{\pm}2^{32}$ (we use smlal)
                                 ▷ $2^{32}$upper + lower = $ab + (abq' \bmod {}^{\pm}2^{32})q$
                                 ▷ lower = 0 $\implies$ upper = $\frac{ab+(abq' \bmod {}^{\pm}2^{32})q}{2^{32}}$
```

Algorithm 17 32-bit in-place Montgomery multiplication (Cortex-M4) [ACC+21].

```
Signature:  montgomery_mul_32 a, b, t0, t1
Inputs: (a, b) = $(a, b)$
Outputs: a = $\frac{ab+(abq' \bmod {}^{\pm}2^{32})q}{2^{32}} \equiv ab2^{-32} \pmod{q}$
montgomery_mul_des_32 t0, a, a, b, t1        ▷ a = $\frac{ab+(abq' \bmod {}^{\pm}2^{32})q}{2^{32}}$
```

R $= 2^{16}$. The DSP extension provides several useful instructions for implementing modular reductions and multiplications for R $= 2^{16}$. Although the literature has illustrated the applicability of DSP extension to Barrett reduction [AHKS22], we focus on the implementations of $\mathsf{montgomery}_{q,2^{16}}(\cdot)$ and $\mathsf{montgomery}_{q,2^{16}}(\cdot,\cdot)$. The most relevant ones are multiplications of particular halfwords. By specifying which halfwords to multiply, we implicitly extract the desired halfwords.

doi:10.6342/NTU202200738

We hold an auxiliary register with the packed halfwords $q \,||\, {-q^{-1}} \bmod {}^{\pm}2^{16}$. Recall that

$$\mathsf{montgomery}_{q,2^{16}}(a) = \frac{a + (a \cdot q' \bmod {}^{\pm}2^{16}) \cdot q}{2^{16}},$$

we compute $a \cdot q'$ with `smulbb` and $a + (aq' \bmod {}^{\pm}2^{16}) \cdot q$ with `smlabt`. The result is $a + (aq' \bmod {}^{\pm}2^{16})q$. Since $a + (aq' \bmod {}^{\pm}2^{16})q$ is divisible by 2^{16}, the result can be written as $\frac{a+(aq' \bmod {}^{\pm}2^{16})q}{2^{16}} \,||\, 0$ [ABCG20]. As for $\mathsf{montgomery}_{q,2^{16}}(\cdot,\cdot)$, we first multiply with `smulXY` and then apply $\mathsf{montgomery}_{q,2^{16}}(\cdot)$ where $\mathtt{X}, \mathtt{Y} \in \{\mathtt{b}, \mathtt{t}\}$ depend on the context. $\mathsf{montgomery}_{q,2^{16}}(\cdot)$ takes two cycles, and $\mathsf{montgomery}_{q,2^{16}}(\cdot,\cdots)$ takes three cycles. Algorithm 18 is an illustration for $\mathsf{montgomery}_{q,2^{16}}(\cdot)$ and Algorithm 19 is an adaption overwriting the input.

Algorithm 18 16-bit Montgomery reduction (Cortex-M4) [ABCG20].

```
Signature:  montgomery_des_16 d, a, t
Inputs: a = $a$
Outputs: d = $\frac{a+(aq' \bmod {}^{\pm}2^{16})q}{2^{16}} \,||\, 0 \equiv a2^{-16} \,||\, 0 \pmod{q}$
smulbb t, a, $q \,||\, q'$                    ▷ t = $aq'$
smlabt d, t, $q \,||\, q'$, a                 ▷ d = $\frac{a+(aq' \bmod {}^{\pm}2^{16})q}{2^{16}} \,||\, 0$
```

Algorithm 19 16-bit in-place Montgomery reduction (Cortex-M4) [ABCG20].

```
Signature:  montgomery_16 a, t
Inputs: a = $a$
Outputs: a = $\frac{a+(a \cdot q' \bmod {}^{\pm}2^{16}) \cdot q}{2^{16}} \,||\, 0 \equiv a2^{-16} \,||\, 0 \pmod{q}$
montgomery_des_16 a, a, t
```

While it is not immediately clear what's the benefit of 16-bit modular reduction and multiplication, we implement them for packed halfwords due to the much lower amount of memory operations and the existence the parallel addition and subtraction. We will justify the uses of 16-bit modular reductions and multiplications in Section 7.2.2.

7.1.6 Cortex-A72 Implementations

For simplicity, we denote the content of a vector register as $(a_i)_i$. Compared to Cortex-M3 and Cortex-M4 implementations, much more effort is required for translating the mathematics into short sequences of instructions. The following instructions are used for implementing modular reductions and multiplications on Cortex-A72.

1. Single-width multiplications: `mul`, `mla`, `mls`, `sqdmulh`, `sqrdmulh`.
2. Long multiplications: `smull`, `smull2`, `smlal`, `smlal2`.
3. Right-shift with rounding: `srshr`.

We first recall that `mul`, `mla`, `mls`, `sqdmulh`, `sqrdmulh`, `smull`, `smull2`, `smlal`, and `smlal2` can only be dispatched to `F0`, and `srshr` can only be dispatched to `F1`. Therefore, we should interleave `srshr` with multiplication instructions whenever possible.

doi:10.6342/NTU202200738

A note about rounding. Before going through the implementations. We discuss some useful property about $\lfloor\rfloor$, $\lceil\rceil$, and $\lfloor\rceil$ for translating $\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$ and $\mathsf{barrett}_{q,\mathtt{R}}(\cdot,\cdot)$ into `sqdmulh`, `sqrdmulh`, and `srshr`.

Lemma 2 ([GKP94, Equation 3.10])**.** Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous monotonically increasing partial function satisfying

$$\forall r \in \mathbb{R}, f(r) \in \mathbb{N} \longrightarrow r \in \mathbb{N}.$$

Then we have

$$\lfloor f(r) \rfloor = \lfloor f(\lfloor r \rfloor) \rfloor$$

whenever $f(r)$ and $f(\lfloor r \rfloor)$ are defined and

$$\lceil f(r) \rceil = \lceil f(\lceil r \rceil) \rceil$$

whenever $f(r)$ and $f(\lceil r \rceil)$ are defined.

Proof. See [GKP94, Section 3.2]. □

Lemma 3. Let $n \in \mathbb{N}$ and $r \in \mathbb{R}$. Then, we have

$$\left\lfloor \frac{r}{n} \right\rfloor = \left\lfloor \frac{\lfloor r \rfloor}{n} \right\rfloor .$$

Proof. Trivial. □

Theorem 10. Let n', n be integers of two with $n'|n$. Then, for all $a \in \mathbb{N}$ we have

$$\left\lfloor \frac{a}{n} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{n'} \right\rfloor}{\frac{\mathtt{R}}{n'}} \right\rfloor .$$

Proof. $\left\lfloor \frac{a}{n} \right\rfloor = \left\lfloor \frac{\frac{a}{n'}}{\frac{n}{n'}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{n'} \right\rfloor}{\frac{n}{n'}} \right\rfloor .$ □

Barrett reduction. Let $\mathtt{w}$ be the width of the specifier and $\mathtt{R} \geq 2^{\mathtt{w}}$ be a power of two. For an element a_i of the vector $(a_i)_i$, we implement

$$\mathsf{barrett}_{q,\mathtt{R}}(a_i) = a_i - \left\lfloor \frac{a_i \cdot \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil \cdot q$$

differently from Cortex-M3 and Cortex-M4 implementations. We first write $\mathsf{barrett}_{q,\mathtt{R}}(a_i)$ as

$$a_i - \left\lfloor \frac{a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil q = a_i - \left\lfloor \frac{\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} \right\rfloor}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rceil q.$$

doi:10.6342/NTU202200738

Proof.

$$
\begin{aligned}
\left\lfloor \frac{a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil
&= \left\lfloor \frac{a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil + 0.5\mathtt{R}}{\mathtt{R}} \right\rfloor \\
&= \left\lfloor \frac{\frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} + \frac{\mathtt{R}}{2^{\mathtt{w}}}}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rfloor \\
&= \left\lfloor \frac{\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} + \frac{\mathtt{R}}{2^{\mathtt{w}}} \right\rfloor}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rfloor \\
&= \left\lfloor \frac{\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} \right\rfloor + \frac{\mathtt{R}}{2^{\mathtt{w}}}}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rfloor \\
&= \left\lfloor \frac{\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} \right\rfloor}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rceil
\end{aligned}
$$

□

In Armv8-A, `sqdmulh` returns the upper halves of the doubled products and `srshr` right-shifts the values with rounding. We compute $\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} \right\rfloor$ with `sqdmulh` and $\left\lfloor \frac{\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} \right\rfloor}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rceil$ with `srshr`. Algorithm 20 is an illustration.

Algorithm 20 Single-width Barrett reduction (Cortex-A72) [BHK+22].

```
Inputs: a = $(a_i)_i$
Outputs: a = $\left( a_i - \left\lfloor \frac{\left\lfloor \frac{2a_i \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{2^{\mathtt{w}}} \right\rfloor}{\frac{2\mathtt{R}}{2^{\mathtt{w}}}} \right\rceil q \right)_i \equiv (a_i)_i \pmod q$
sqdmulh t, a, $\left\lfloor \frac{\mathtt{R}}{q} \right\rceil$
srshr t, t, $\mathtt{R} - \mathtt{w} + 1$
mls a, t, $q$
```

Interleaving single-width Barrett reductions. If there are several instances of $\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$ with no dependencies, we should interleave `srshr` with `sqdmulh` and `mls`. Algorithm 55 illustrates how to interleave eight independent $\mathsf{barrett}_{q,\mathtt{R}}(\cdot)$s where `barrett_const` $= \left\lfloor \frac{\mathtt{R}}{q} \right\rceil$ and `shrv` $=$ `R - w + 1`.

Montgomery reduction with long arithmetic. For $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot)$, we introduce a straightforward adaptation. Assuming $\mathtt{R} = 2^{\mathtt{w}}$, we write $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot)$

doi:10.6342/NTU202200738

as

$$\mathsf{montgomery}_{q,\mathtt{R}}(a_i) = \frac{a_i + (a_i q' \bmod {}^{\pm}\mathtt{R})q}{\mathtt{R}}.$$

Since we only need the lower $\mathtt{w}$ bits of $a_i q' \bmod {}^{\pm}\mathtt{R}$, we can compute with $(a_i \bmod {}^{\pm}\mathtt{R})q' \bmod {}^{\pm}\mathtt{R}$. If a_i is a $2\log_2 \mathtt{R}$-bit value, then we apply `uzp1` for extracting $a_i \bmod {}^{\pm}\mathtt{R}$ from a_i. Next, we compute $(a_i \bmod {}^{\pm}\mathtt{R})q' \bmod {}^{\pm}\mathtt{R}$ with `mul`. Finally, we compute the $2\log_2 \mathtt{R}$-bit product $a_i + ((a_i \bmod {}^{\pm}\mathtt{R})q' \bmod {}^{\pm}\mathtt{R})q$ with `smlal` and `smlal2` and extract the upper $\mathtt{w}$-bit values with `uzp2`. Algorithm 21 is an illustration.

Algorithm 21 Double-width Montgomery reduction (Cortex-A72) [BHK+22].

Inputs: $\mathtt{l} + \mathtt{h}\mathtt{R} = (a_i)_i$

Outputs: $\mathsf{c} = \left(\frac{a_i + (a_i q' \bmod {}^{\pm}\mathtt{R})q}{\mathtt{R}}\right)_i \equiv (a_i \mathtt{R}^{-1})_i \pmod q$

```
uzp1 t, l, h                        ▷ $\mathtt{t} = (a_i)_i \bmod {}^{\pm}\mathtt{R}$
mul t, t, $q'$                      ▷ $\mathtt{t} = (a_i q')_i \bmod {}^{\pm}\mathtt{R}$
smlal l, t, $q$
smlal2 h, t, $q$
                                    ▷ $(\mathtt{l}, \mathtt{h}) = (a_i + (a_i q' \bmod {}^{\pm}\mathtt{R})q)_i$
uzp2 a, l, h                        ▷ $\mathsf{c} = \left(\frac{a_i + (a_i q' \bmod {}^{\pm}\mathtt{R})q}{\mathtt{R}}\right)_i$
```

Montgomery multiplication. For $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot,\cdot)$, we first multiply with `smull` and `smull2`, and apply $\mathsf{montgomery}_{q,\mathtt{R}}(\cdot)$ to the result. Algorithm 22 is an illustration.

Algorithm 22 Double-width Montgomery multiplication (Cortex-A72) [BHK+22].

Inputs: $(\mathsf{a}, \mathsf{b}) = (a_i, b_i)_i$

Outputs: $\mathsf{c} = \left(\frac{a_i b_i + (a_i b_i q' \bmod {}^{\pm}\mathtt{R})q}{\mathtt{R}}\right)_i \equiv (a_i b_i \mathtt{R}^{-1})_i \pmod q$

```
smull l, a, b
smull2 h, a, b
                                    ▷ $\mathtt{l} + \mathtt{h}\mathtt{R} = (a_i b_i)_i$
uzp1 t, l, h
mul t, t, $q'$
smlal l, t, $q$
smlal2 h, t, $q$
uzp2 c, l, h                        ▷ $\mathsf{c} = \left(\frac{a_i b_i + (a_i b_i q' \bmod {}^{\pm}\mathtt{R})q}{\mathtt{R}}\right)_i$
```

Barrett multiplication. Let $\mathtt{R} = 2^{\mathtt{w}}$. For $\mathsf{barrett}_{q,\mathtt{R}}(\cdot,\cdot)$, we write it as

$$\mathsf{barrett}_{q,\mathtt{R}}(a_i, b_i) = a_i \cdot b_i - \left\lfloor \frac{a_i \cdot \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil \cdot q = a_i \cdot b_i - \left\lfloor \frac{2a_i \cdot \frac{\left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{2}}{\mathtt{R}} \right\rceil \cdot q$$

where $\lfloor \rceil_{2,q}$ first rounds the real number to an integer and maps the result to the even representative in $(-q, q)$. We precompute $\frac{\left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{2}$, apply `sqrdmulh` for computing

doi:10.6342/NTU202200738

$\left\lfloor \frac{2a_i \cdot \frac{\left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{2}}{\mathtt{R}} \right\rfloor$. Following Theorem 9, since $|\mathsf{barrett}_{q,\mathtt{R}}(\cdot,\cdot)| \leq q < \mathtt{R}$, we compute $a_i b_i - \left\lfloor \frac{a_i \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{\mathtt{R}} \right\rceil q$ as

$$\left(\left(a_i b_i \bmod {}^{\pm}\mathtt{R} \right) - \left(\left\lfloor \frac{a_i \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{\mathtt{R}} \right\rceil q \bmod {}^{\pm}\mathtt{R} \right) \right) \bmod {}^{\pm}\mathtt{R}$$

Clearly, now we can compute $a_i b_i - \left\lfloor \frac{a_i \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{\mathtt{R}} \right\rceil q$ with `mul` and `mls`.

Algorithm 23 Single-width Barrett multiplication (Cortex-A72) [BHK+22].

```
Inputs: (a, b, b') = $\left( a_i, b_i, \frac{\left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{2} \right)_i$
Outputs: c = $\left( a_i b_i - \left\lfloor \frac{a_i \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{\mathtt{R}} \right\rceil q \right)_i \equiv (a_i b_i)_i \pmod{q}$
mul c, a, b              ▷ c = $(a_i b_i \bmod {}^{\pm}\mathtt{R})_i$
sqrdmulh t, a, b'        ▷ t = $\left( \left\lfloor \frac{a_i \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{\mathtt{R}} \right\rceil \right)_i$
mls c, t, $q$            ▷ c = $\left( a_i b_i - \left\lfloor \frac{a_i \left\lfloor \frac{b_i \mathtt{R}}{q} \right\rceil_{2,q}}{\mathtt{R}} \right\rceil q \right)_i$
```

7.2 Butterflies

This section presents the implementations of the butterfly operations used in NTTs. We commonly compute radix-2 Cooley–Tukey butterflies, and this section is mostly about them.

7.2.1 Butterflies for NTTs

We mainly focus on implementing Cooley–Tukey and Gentleman–Sande FFT. We first recall the definition of Cooley–Tukey FFT from Definition 66 as follows. Let R be a ring, $n = 2^k$ a power of two coprime to $\mathrm{char}(R)$, ζ an invertible element in R, and ω_n a principal n-th root of unity. We split the ring $\mathbb{Z}_q[x] \Big/ \left\langle x^{2^k} - \zeta^{2^k} \right\rangle$ as follows:

$$\begin{aligned}
\mathbb{Z}_q[x] \Big/ \left\langle x^{2^k} - \zeta^{2^k} \right\rangle &\cong \mathbb{Z}_q[x] \Big/ \left\langle x^{2^{k-1}} - \zeta^{2^{k-1}} \right\rangle \times \mathbb{Z}_q[x] \Big/ \left\langle x^{2^{k-1}} - \zeta^{2^{k-1}} \omega_n^{2^{k-1}} \right\rangle \\
&\cong \mathbb{Z}_q[x] \Big/ \left\langle x^{2^{k-2}} - \zeta^{2^{k-2}} \right\rangle \times \mathbb{Z}_q[x] \Big/ \left\langle x^{2^{k-2}} - \zeta^{2^{k-2}} \omega_n^{2^{k-1}} \right\rangle \times \\
&\quad\ \mathbb{Z}_q[x] \Big/ \left\langle x^{2^{k-2}} - \zeta^{2^{k-2}} \omega_n^{2^{k-2}} \right\rangle \times \mathbb{Z}_q[x] \Big/ \left\langle x^{2^{k-2}} - \zeta^{2^{k-2}} \omega_n^{2^{k-2}+2^{k-1}} \right\rangle \\
&\ \ \vdots \\
&\cong \prod_{i=0}^{2^k-1} \mathbb{Z}_q[x] \Big/ \left\langle x - \zeta \omega_n^{\mathrm{rev}_{(2:k)}(i)} \right\rangle
\end{aligned}$$

doi:10.6342/NTU202200738

The isomorphsim

$$\mathbb{Z}_q[x]\Big/\Big\langle x^{2^k} - \zeta^{2^k}\Big\rangle \cong \mathbb{Z}_q[x]\Big/\Big\langle x^{2^{k-1}} - \zeta^{2^{k-1}}\Big\rangle \times \mathbb{Z}_q[x]\Big/\Big\langle x^{2^{k-1}} - \zeta^{2^{k-1}}\omega_n^{2^{k-1}}\Big\rangle$$

maps a size-2^k polynomial $\boldsymbol{a}(x) = \sum_{i=0}^{2^k-1} a_i x^i$ to the two size-2^{k-1} polynomials $\sum_{i=0}^{2^{k-1}-1}(a_i + \zeta a_{i+2^{k-1}})x^i$ and $\sum_{i=0}^{2^{k-1}-1}(a_i - \zeta a_{i+2^{k-1}})x^i$. We can compute the isomorphism by applying

$$\mathrm{CT} : (a_{i_1}, a_{i_2}, \zeta) \mapsto (a_{i_1} + \zeta a_{i_2}, a_{i_1} - \zeta a_{i_2})$$

to all (a_{i_1}, a_{i_2}) satisfying $i_1 = 0, 1, \ldots, 2^{k-1}-1$ and $i_2 = i_1 + 2^{k-1}$. Such computations are called Cooley–Tukey butterflies. The inverse of a Cooley–Tukey butterfly can be written down as follows.

$$(a_{i_1}, a_{i_2}, \zeta^{-1}) \mapsto \frac{1}{2}(a_{i_1} + a_{i_2}, (a_{i_1} - a_{i_2})\zeta^{-1}).$$

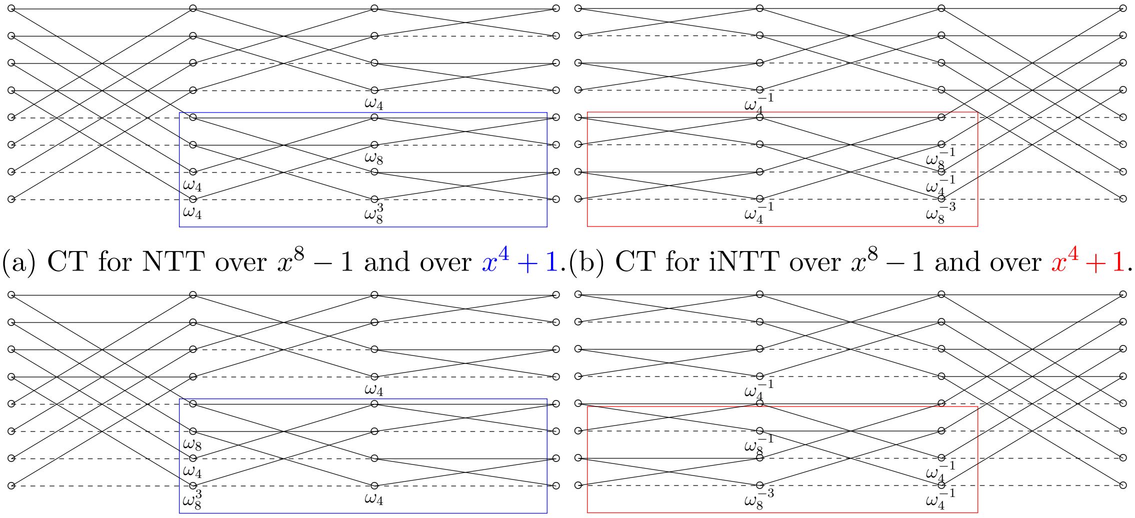
By moving the division by 2 to the end, we defined a Gentleman–Sande butterfly as

$$\mathrm{GS} : (a_{i_1}, a_{i_2}, \zeta) \mapsto (a_{i_1} + a_{i_2}, (a_{i_1} - a_{i_2})\zeta).$$

Obviously, we have

$$\mathrm{GS}(\mathrm{CT}(a, b, \zeta), \zeta^{-1}) = 2(a, b) = \mathrm{CT}(\mathrm{GS}(a, b, \zeta), \zeta^{-1}).$$

Therefore, we can invert any computations consisting of CT and GS butterflies by inverting CT butterflies with GS butterflies and GS butterflies with CT butterflies. In particular, this implies inverting the NTT implemented with CT butterflies gives a GS butterfly implementation of the inverse of the NTT. Conversely, by inverting the NTT implemented with GS butterflies, we derive a CT butterfly implementation of the inverse of NTT. Figure 7.1 is an illustration for $(k, \zeta^{2^k}) = (3, 1)$ and $(k, \zeta^{2^k}) = (2, -1)$.



(a) CT for NTT over $x^8 - 1$ and over $x^4 + 1$.(b) CT for iNTT over $x^8 - 1$ and over $x^4 + 1$.

(c) GS for NTT over $x^8 - 1$ and over $x^4 + 1$.(d) GS for iNTT over $x^8 - 1$ and over $x^4 + 1$.

Figure 7.1: CT and GS butterflies over $x^8 - 1$ and $x^4 + 1$. $\omega_n = \omega^{8/n}$ where ω is a principal 8th root of unity [ACC+22].

For simplicity, starting from this section, we use = for representatives even if it is not in $[-\frac{q}{2}, \frac{q}{2}]$ and omit the $\bmod^{\pm} q$ for the results.

doi:10.6342/NTU202200738

7.2.2 Radix-2 Butterflies on Cortex-M4

We first go through the implementation of 32-bit butterfly by [ACC+21] and the 16-bit butterfly by [ABCG20] on Cortex-M4.

32-bit. For implementing a CT butterfly with 32-bit arithmetic, we first apply 32-bit Montgomery multiplication `montgomery_mul_32` for modular multiplication. Then, we perform in-place add-sub pair with `add` and `sub` [ACC+21]. Algorithm 24 is an illustration. If the twiddle factor is 1, then we can skip the Montgomery multiplication.

Algorithm 24 32-bit CT butterfly (Cortex-M4) [ACC+21].

Inputs: $(\mathtt{a0}, \mathtt{a1}) = (a_0, a_1)$, $\mathtt{twiddle} = \zeta 2^{32} \bmod {}^{\pm}q$
Outputs: $(\mathtt{a0}, \mathtt{a1}) = (a_0 + \zeta a_1, a_0 - \zeta a_1)$

```
montgomery_mul_32 a1, twiddle, t0, t1
add.w a0, a0, a1
sub a1, a0, a1, lsl #1
```

16-bit. For 16-bit CT butterfly, we compute two 16-bit Montgomery multiplication for packed halfwords with `smulbb`, `smulbt`, `smultb`, `smultt` and pack the results together with `pkhtb` [ABCG20]. After the Montgomery multiplication, we compute the add-sub pairs for packed halfwords with `sadd16` and `ssub16` [ABCG20]. Algorithms 25 and 26 are illustrations.

Algorithm 25 16-bit Montgomery multiplications for packed halfwords (Cortex-M4) [ABCG20].

Signature: `doublemontgomery_mul_16 {t, b}, a, twiddle, t0, t1`
Inputs: $\mathtt{a} = a_1 || a_0$, $\mathtt{twiddle} = (\zeta_{\mathtt{t}} || \zeta_{\mathtt{b}}) \bmod {}^{\pm}q$
Outputs: $\mathtt{a} = \left(\zeta_{\{\mathtt{t,\ b}\}} a_1 || \zeta_{\{\mathtt{t,\ b}\}} a_0\right) 2^{-16}$

```
smulb{t, b} t0, a, twiddle
montgomery_16 t0, t1
smult{t, b} a, a, twiddle
montgomery_16 a, t1
pkhtb a, a, t0, asr #16
```

Algorithm 26 16-bit butterflies for packed halfwords (Cortex-M4) [ABCG20].

Signature: `doublebutterfly_16 {t, b}, a0, a1, twiddle, t0, t1`
Inputs: $\mathtt{a0} = a_1 || a_0$, $\mathtt{a1} = a_3 || a_2$, $\mathtt{twiddle} = (\zeta_{\mathtt{t}} || \zeta_{\mathtt{b}})\, 2^{16} \bmod {}^{\pm}q$
Outputs: $(\mathtt{a0}, \mathtt{a1}) = \left(a_0 + a_2\zeta_{\{\mathtt{t,\ b}\}} || a_1 + a_3\zeta_{\{\mathtt{t,\ b}\}}, a_0 - a_2\zeta_{\{\mathtt{t,\ b}\}} || a_1 - a_3\zeta_{\{\mathtt{t,\ b}\}}\right)$

```
smulb{t, b} t0, a1, twiddle
smult{t, b} a1, a1, twiddle
montgomery_16 t0, t1
montgomery_16 a1, t1
pkhtb t0, a1, t0, asr #16
ssub16 a1, a0, t0
sadd16 a0, a0, t0
```

doi:10.6342/NTU202200738

Multi-layer butterflies. Loading a set of coefficients involved in several isomorphisms at once greatly reduces the number of memory operations. We now show how to implement the so-called “multi-layer butterflies” for realizing the idea.

32-bit. For 32-bit arithmetic, we first look at the benefit of computing add-sub pairs while all the operands are in registers. Given a tuple $(c_0, \ldots, c_7)$, our goal is to compute $((c_0, c_2, c_4, c_6) + (c_1, c_3, c_5, c_7), (c_0, c_2, c_4, c_6) - (c_1, c_3, c_5, c_7))$. We first compute the additions with `add` and overwrite the registers holding the coefficients with even indices. Then, we subtract the coefficients of odd indices from the registers. For the subtractions, we multiply the values by 2 with `lsl #1`, which comes with no extra cost if we append it to the last operand of `sub`. Since the destination register is also one of the operands for additions, the instruction `add` is compiled into 16-bit Thumb-2 code, and we save some code size here [Shi20]. Algorithm 27 is an illustration.

Algorithm 27 Four add-sub pairs (Cortex-M3, Cortex-M4) [Shi20].

```
Signature: add_sub4 c0, c1, c2, c3, c4, c5, c6, c7
Inputs: (c0, ..., c7) = $(c_0, \ldots, c_7)$
Outputs: (c0, ..., c7) = $((c_0, c_2, c_4, c_6) + (c_1, c_3, c_5, c_7), (c_0, c_2, c_4, c_6) - (c_1, c_3, c_5, c_7))$
add c0, c1
add c2, c3
add c4, c5
add c6, c7
                                                  ▷ Thumb 2
sub c1, c0, c1, lsl #1
sub c3, c2, c3, lsl #1
sub c5, c4, c5, lsl #1
sub c7, c6, c7, lsl #1
```

We show how to implement $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_8}$ with multi-layer CT butterflies. Among the 14 general-purpose registers, one register is used as the source of the coefficients, two registers store the constants for Montgomery multiplications, and two registers are used as temporary storage for Montgomery multiplications. For the twiddle factors, we store them at floating-point registers and only transfer them to a general-purpose register with `vmov` when needed. To sum up, six registers are already occupied. Therefore, we can only load eight coefficients to registers and compute the corresponding three layers of isomorphisms without register spills [CHK+21].

At each layer of CT butterflies, we multiply four coefficients with `montgomery_mul_32` and call `add_sub4`. Furthermore, if $\zeta = 1$, we can save several `montgomery_mul_32` and `vmov`. Algorithm 39 is an illustration for

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_8}$$

and Algorithm 40 is for

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle : \omega_8}.$$

From Corollary 6, we can also use CT butterflies for implementing the inverses of NTTs. Recall that the result of CT FFT for NTT is the bit-reversal of the straightforward computation. If we want to apply CT FFT for the inverse NTT, we

doi:10.6342/NTU202200738

must compute as if the input is already bit-reversed. For computing a layer of CT FFT for the inverse NTT, we have to loop in a bit-reversal fashion. Since multilayer butterflies are essentially unrolled loops, we have to access the coefficients in the bit-reversal fashion. To realize the inverses, we also invert all the twiddle factors. Algorithm 41 implements

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$$

and Algorithm 42 implements

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}.$$

16-bit. For 16-bit arithmetic, we exploit the DSP extension. [ABCG20] implements CT butterflies for

$$\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{8}\rangle : \omega_8}$$

as shown in Algorithm 43. We extend their work to CT butterflies for

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$$

and

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}.$$

Algorithms 44 and 45 are illustrations. Notice that for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$, we perform out-of-place add-sub pairs for packed halfwords and modify the source registers for the follow-up operations. In this way, we don't need additional transfers between registers. The idea clearly applies to

$$\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle : \omega_8}.$$

We do not show the implementation because it is not implemented in this thesis.

7.2.3 Radix-2 Butterflies on Cortex-M3

Implementing butterflies on Cortex-M3 is similar to Cortex-M4. First of all, the 32-bit butterflies are also runnable on Cortex-M3, but instructions `smull` and `smlal` are early-terminating. Although the literature has demonstrated workarounds [GKS21] and their applicability to public data [ACC+22], this thesis only reports the 16-bit butterflies by [ACC+22].

Butterfly for one layer. One can easily derive 16-bit butterflies for Cortex-M3 from the 32-bit butterflies of Cortex-M4. We recall how Algorithm 14 computes $\mathsf{montgomery}_{q,2^{16}}(a, b)$. We first compute

$$ab + (abq' \bmod {}^{\pm}2^{16})q.$$

Since it is divisible by 2^{16}, we divide it by 2^{16} with `asr`. For a CT butterfly, the follow-up operation is an add-sub pair. We merge the add-sub pair with `asr` if we also consider Algorithm 27 implementing in-place add-sub pairs. For the instruction `add`, we append `asr #16` at the end, and for the `sub`, we append `asr #15` for in-place implementation. Algorithm 28 is an illustration.

doi:10.6342/NTU202200738

Algorithm 28 16-bit CT butterfly (Cortex-M3) [ACC+22].

Signature: `montgomery_mul_16_addSub c0, c1, twiddle, t`
Inputs: $(\mathtt{c0}, \mathtt{c1}) = (c_0, c_1)$, $\mathtt{twiddle} = \omega 2^{16} \bmod {}^{\pm}q$
Outputs: $(\mathtt{c0}, \mathtt{c1}) = (c_0 + \omega c_1, c_0 - \omega c_1)$

```
mul c1, c1, twiddle
mul t, c1, $q'$
sxth.w t, t
mla c1, t, $q$, c1
add c0, c0, c1, asr #16
sub c1, c0, c1, asr #15
```

Multi-layer butterflies. We do not group the Montgomery multiplications for multi-layer butterflies since we want to merge the `asr` into the add-sub pairs. Algorithm 46 implements

$$\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8 - \zeta^8 \rangle : \omega_8},$$

and Algorithm 47 implements

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8 - \zeta^{-8} \rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}.$$

This thesis also implements

$$\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8 - 1 \rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$$

with additional Montgomery reductions at the beginning for reducing the values as shown in Algorithm 48.

7.2.4 Radix-2 Butterflies on Cortex-A72

We go through the implementations of CT and GS butterflies defined on pairs of SIMD registers. We first recall that multiplication instructions can only be dispatched to `F0` while `add` and `sub` can be dispatched to `F0` and `F1`. We will first discuss how implementing CT and GS butterflies defined on a pair of SIMD registers. For implementing multi-layer butterflies, care must be taken for interleaving the instructions. We report and extend the scheduling policy by [BHK+22].

Radix-2 Butterfly. The radix-2 CT and GS butterflies are built upon the Barret multiplication. Let $(a_i, b_i)_i$ be the input vectors for a butterfly. For a CT butterfly with a single twiddle factor ω, we compute $(\omega b_i)_i$ with $\mathsf{barrett}_{q,\mathtt{R}}(\cdot, \cdot)$ followed by an add-sub pair. For a GS butterfly, we compute add-sub first and multiply $(a_i - b_i)_i$ by ω with $\mathsf{barrett}_{q,\mathtt{R}}(\cdot, \cdot)$.

Algorithm 29 Radix-2 CT butterfly (Cortex-A72) [BHK+22].

Inputs: $(\mathsf{a}, \mathsf{b}) = (a_i, b_i)_i$
Outputs: $(\mathsf{a}, \mathsf{b}) = (a_i + \omega b_i, a_i - \omega b_i)_i$

```
mul t, b, $\frac{\lfloor \frac{\omega \mathtt{R}}{q} \rceil_{2,q}}{2}$
sqrdmulh b, b, $\omega$
mls t, b, $q$                          ▷ $\mathsf{t} = (\omega b_i)_i$
sub b, a, t
add a, a, t                            ▷ $(\mathsf{a}, \mathsf{b}) = (a_i + \omega b_i, a_i - \omega b_i)_i$
```

doi:10.6342/NTU202200738

Algorithm 30 Radix-2 GS butterfly (Cortex-A72) [BHK+22].

Inputs: $(\mathtt{a}, \mathtt{b}) = (a_i, b_i)_i$

Outputs: $(\mathtt{a}, \mathtt{b}) = (a_i + b_i, (a_i - b_i)\,\omega)_i$

```
sub t, a, b
add a, a, b                              ▷ $(\mathtt{a}, \mathtt{t}) = (a_i + b_i, a_i - b_i)_i$
mul b, t, $\frac{\left\lfloor \frac{\omega \mathrm{R}}{q} \right\rceil_{2,q}}{2}$
sqrdmulh t, t, $\omega$
mls b, t, $q$                            ▷ $\mathtt{b} = ((a_i - b_i)\omega)_i$
```

Parallelizing radix-2 butterflies. Since there are 32 SIMD registers, we can compute several butterflies in parallel. Algorithm 56 computes several $\mathtt{barrett}_{q,\mathtt{R}}(\cdot, \cdot)$ and Algorithm 57 computes several add-sub pairs. Parallelized CT butterflies are realized by applying Algorithm 56 followed by Algorithm 57. For parallelized GS butterflies, we apply of Algorithm 57 first instead. In practice, although parallelized butterflies alleviate the latency issue, they incur a work-load issue. According to [ARM15], instructions are decoded in order where at most one instruction can be dispatched to `F0` and one instruction can be dispatched to `F1` in a single cycle. For Algorithm 56, all the instructions can only be dispatched to `F0`. For Algorithm 57, all the instructions can be dispatched to `F0` and `F1`. If we build an NTT based on Algorithms 56 and 57, the entire computation is `F0`-bounded.

Interleaving butterflies. To reduce the workload of `F0`, we want to find a scheduling policy dispatching `add` and `sub` to `F1` more frequently. Let $\mathrm{CT}(\mathtt{v}_{i_{0,0}}, \mathtt{v}_{i_{0,1}}) = \mathrm{CT}(a_{i_0}, b_{i_0}, \omega)_{i_0}$ and $\mathrm{CT}(\mathtt{v}_{i_{1,0}}, \mathtt{v}_{i_{1,1}}) = \mathrm{CT}(a_{i_1}, b_{i_1}, \omega)_{i_1}$ be two CT butterflies with such that $\mathtt{v}_{i_{1,1}} \neq \mathtt{v}_{i_{0,0}}$ and $\mathtt{v}_{i_{1,1}} \neq \mathtt{v}_{i_{0,1}}$. We first compute $(\omega b_{i_0})_{i_0}$. For computing the follow-up add-sub pair for $(a_{i_0}, \omega b_{i_0})_{i_0}$, we interleave the computation with $(\omega b_{i_1})_{i_1}$ as shown in Algorithm 58. For interleaving GS butterflies, we require $\mathtt{v}_{i_{0,1}} \neq \mathtt{v}_{i_{1,0}}$ and $\mathtt{v}_{i_{0,1}} \neq \mathtt{v}_{i_{1,1}}$. We first compute $(a_{i_0} + b_{i_0}, a_{i_0} - b_{i_0})_{i_0}$ and interleave the computation $((a_{i_0} - b_{i_0})\omega)_{i_0}$ with $(a_{i_1} + b_{i_1}, a_{i_1} - b_{i_1})_{i_1}$ as shown in Algorithm 59.

Multi-layer butterflies. A common approach for reducing the number of memory operations is to compute several layers of NTTs at a time with multi-layer butterflies. To avoid the dependencies when interleaving instructions, we find that for radix-2 NTTs, computing coefficients distributed over 16 SIMD registers is the most beneficial approach. This suggests that computing 4 layers at a time is possible since $2^4 = 16$.

An interesting recurrence. Before diving into the proposed policy, we first look at the following recurrence for $0 \leq i < 2^k$.

$$\mathrm{rev}^{\mathrm{rev}}_{(2:k)}(i) = \begin{cases} 2^{k-1}\,[\![2|i]\!] + \mathrm{rev}^{\mathrm{rev}}_{(2:(k-1))}\left(\left\lfloor \frac{i}{2} \right\rfloor\right) & \text{if } k \geq 0, \\ 0 & \text{if } k < 0. \end{cases}$$

We solve for $\mathrm{rev}^{\mathrm{rev}}_{(2:k)}$ and the result is $\mathrm{rev}^{\mathrm{rev}}_{(2:k)} = i \mapsto \mathrm{rev}_{(2:k)}(2^k - 1 - i)$.

doi:10.6342/NTU202200738

Proof. Prove by induction on k.

$$\begin{aligned}
\mathrm{rev}^{\mathrm{rev}}_{(2:k)}(i) &= 2^{k-1}\llbracket 2|i \rrbracket + \mathrm{rev}^{\mathrm{rev}}_{(2:(k-1))}\left(\left\lfloor \tfrac{i}{2} \right\rfloor\right) \\
&= 2^{k-1}\llbracket 2|i \rrbracket + \mathrm{rev}_{(2:(k-1))}\left(2^{k-1} - 1 - \left\lfloor \tfrac{i}{2} \right\rfloor\right) \\
&= 2^{k-1}\llbracket 2|i \rrbracket + \mathrm{rev}_{(2:(k-1))}\left(2^{k-1} - 1 - \tfrac{i-1+\llbracket 2|i \rrbracket}{2}\right) \\
&= 2^{k-1}\llbracket 2|i \rrbracket + \mathrm{rev}_{(2:(k-1))}\left(\tfrac{2^k-2-i+1-\llbracket 2|i \rrbracket}{2}\right) \\
&= 2^{k-1}\llbracket 2|i \rrbracket + \mathrm{rev}_{(2:k)}\left(2^k - 1 - i - \llbracket 2|i \rrbracket\right) \\
&= \mathrm{rev}_{(2:k)}\left(2^k - 1 - i\right)
\end{aligned}$$

□

Table 7.1 is an illustration for $k = 3$.

Table 7.1: $\mathrm{rev}^{\mathrm{rev}}_{(2:3)}$ from $\mathrm{rev}_{(2:3)}$.

	i							
	0	1	2	3	4	5	6	7
$\mathrm{rev}_{(2:3)}(i)$	0	4	2	6	1	5	3	7
$\mathrm{rev}^{\mathrm{rev}}_{(2:3)}(i)$	7	3	5	1	6	2	4	0

Policy for interleaving. Let $\mathtt{J}_{l,j} \coloneqq (i_j, i_j + 2^{k-l-1})$ be the job of applying CT butterfly to $(a_{i_j}, a_{i_j+2^{k-l-1}})$. We omit the twiddle factor parameter because it is not involved in the dependencies. First of all, for each layer $l = 0, \ldots, k-1$, we assume the tuple $(\mathtt{J}_{l,j})_j$ is sorted in lexicographic order. Table 7.2 lists all the values of $\mathtt{J}_{l,j}$ for the case $k = 4$. Next, we define the tuple of jobs at layer l as the tuple $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_j \coloneqq \left(\mathtt{J}_{l,\mathrm{rev}^{\mathrm{rev}}_{(2:(k-1))}(j)}\right)_j$. Table 7.3 lists all the values of $\mathtt{J}^{\mathrm{rev}}_{l,j}$ for $k = 4$. For interleaving the computations, we partition $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_j$ into the first half $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_<$ and the second half $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_>$. An important observation is that for $l = 0, \ldots, k-2$, the add-sub pairs of $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_<$ is independent from the $\mathsf{barrett}_{q,\mathtt{R}}(\cdot,\cdot)$s of $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_>$ and the add-sub pairs of $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_>$ is independent from the $\mathsf{barrett}_{q,\mathtt{R}}(\cdot,\cdot)$s of $\left(\mathtt{J}^{\mathrm{rev}}_{l+1,j}\right)_<$. We can then interleave $\left(\mathtt{J}^{\mathrm{rev}}_{l,j}\right)_>$ and $\left(\mathtt{J}^{\mathrm{rev}}_{l+1,j}\right)_<$ with Algorithm 58 for all $l = 0, \ldots, k-1$.

Table 7.2: $\mathtt{J}_{l,j}$ for $(l,j) \in \{0,\ldots,3\} \times \{0,\ldots,7\}$.

		j							
		0	1	2	3	4	5	6	7
l	0	$(0,8)$	$(1,9)$	$(2,10)$	$(3,11)$	$(4,12)$	$(5,13)$	$(6,14)$	$(7,15)$
	1	$(0,4)$	$(1,5)$	$(2,6)$	$(3,7)$	$(8,12)$	$(9,13)$	$(10,14)$	$(11,15)$
	2	$(0,2)$	$(1,3)$	$(4,6)$	$(5,7)$	$(8,10)$	$(9,11)$	$(12,14)$	$(13,15)$
	3	$(0,1)$	$(2,3)$	$(4,5)$	$(6,7)$	$(8,9)$	$(10,11)$	$(12,13)$	$(14,15)$

doi:10.6342/NTU202200738

Table 7.3: $\mathsf{J}^{\mathrm{rev}}_{l,j}$ for $(l, j) \in \{0, \ldots, 3\} \times \{0, \ldots, 7\}$.

		j							
		0	1	2	3	4	5	6	7
l	0	$(7, 15)$	$(3, 11)$	$(5, 13)$	$(1, 9)$	$(6, 14)$	$(2, 10)$	$(4, 12)$	$(0, 8)$
	1	$(11, 15)$	$(3, 7)$	$(9, 13)$	$(1, 5)$	$(10, 14)$	$(2, 6)$	$(8, 12)$	$(0, 4)$
	2	$(13, 15)$	$(5, 7)$	$(9, 11)$	$(1, 3)$	$(12, 14)$	$(4, 6)$	$(8, 10)$	$(0, 2)$
	3	$(14, 15)$	$(6, 7)$	$(10, 11)$	$(2, 3)$	$(12, 13)$	$(4, 5)$	$(8, 9)$	$(0, 1)$

The policy based on $\mathrm{rev}^{\mathrm{rev}}_{(2:k)}$ was only applied partially in practice as such observation was derived months after the publication. Algorithm 61 implements a level of interleaved 4-layer CT butterflies. Its reverse implements a level of interleaved 4-layer GS butterflies. Due to the characteristic of Cortex-A72, partially applying the policy is already very close to the theoretic optimal of the pipelining involving `F0` and `F1`.

Below is my own opinion: The policy is expected to reach the limit of various platforms with many pipelines when memory operations are excluded. Therefore, we should apply the policy as the first draft for platforms with many pipelines and later tune the implementations according to the pattern of memory operations. The formalization of the claim of optimality requires a deep understanding of graph entropy [CFJ+10, CFJ+13, CF13] and is left for future research.

7.2.5 CT–GS butterflies over $x^8 - 1$ on Cortex-M4

In this section, we report the CT–GS butterflies for $\mathsf{NTT}_{R[x]/\langle x^8-1\rangle:\omega_8}$ and $\mathsf{NTT}^{-1}_{R[x]/\langle x^8-1\rangle:\omega_8}$ introduced by [ACC+22]. They coined the term CT–GS butterflies since it can be derived from either CT or GS butterflies. We first illustrate the idea with CT butterflies. Let's say applying $\mathsf{NTT}_{R[x]/\langle x^8-1\rangle:\omega_8}$ to $(a_0, \ldots, a_7)$ with CT butterflies is to derive $(a'''_0, \ldots, a'''_7)$ as follows:

1. $(a_0, \ldots, a_7) \mapsto (a'_0, \ldots, a'_7)$ where

$$\begin{array}{rcl} (a'_0, \ldots, a'_3) & = & (a_0, \ldots, a_3) + (a_4, \ldots, a_7) \\ (a'_4, \ldots, a'_7) & = & (a_0, \ldots, a_3) - (a_4, \ldots, a_7) \end{array}$$

2. $(a'_0, \ldots, a'_7) \mapsto (a''_0, \ldots, a''_7)$ where

$$\begin{array}{rcr} (a''_0, a''_1) & = & (a'_0, a'_1) + (a'_2, a'_3) \\ (a''_2, a''_3) & = & (a'_0, a'_1) - (a'_2, a'_3) \\ (a''_4, a''_5) & = & (a'_4, a'_5) + \omega_4 (a'_6, a'_7) \\ (a''_6, a''_7) & = & (a'_4, a'_5) - \omega_4 (a'_6, a'_7) \end{array}$$

doi:10.6342/NTU202200738

3. $(a''_0, \ldots, a''_7) \mapsto (a'''_0, \ldots, a'''_7)$ where

$$\begin{aligned}
a'''_0 &= a''_0 + a''_1 \\
a'''_1 &= a''_0 - a''_1 \\
a'''_2 &= a''_2 + \omega_4 a''_3 \\
a'''_3 &= a''_2 - \omega_4 a''_3 \\
a'''_4 &= a''_4 + \omega_8 a''_5 \\
a'''_5 &= a''_4 - \omega_8 a''_5 \\
a'''_6 &= a''_6 + \omega_8^3 a''_7 \\
a'''_7 &= a''_6 - \omega_8^3 a''_7
\end{aligned}$$

The computation can be re-written as $(a_0, a_2, a_4, a_6) \mapsto (a''_0, \omega_4 a''_2, \omega_8 a''_4, \omega_8^3 a''_6)$ and $(a_1, a_3, a_5, a_7) \mapsto (a''_1, \omega_4 a''_3, \omega_8 a''_5, \omega_8^3 a''_7)$, followed by

$$\texttt{add\_sub4}\left((a''_0, \omega_4 a''_2, \omega_8 a''_4, \omega_8^3 a''_6), (a''_1, \omega_4 a''_3, \omega_8 a''_5, \omega_8^3 a''_7)\right)$$

where `add_sub4` is the component-wise add-sub giving a pair as result.

We present here a faster computation for $(a_1, a_3, a_5, a_7) \mapsto (a''_1, \omega_4 a''_3, \omega_8 a''_5, \omega_8^3 a''_7)$ as follows:

1. $(a_1, a_3, a_5, a_7) \mapsto (a'_1, a'_3, a'_5, a'_7)$
2. $(a'_1, a'_3) \mapsto (a''_1, a''_3)$
3. $a''_3 \mapsto \omega_4 a''_3$
4. $(a'_5, a'_7) \mapsto (\omega_8 a'_5 + \omega_8^3 a'_7, \omega_8^3 a'_5 + \omega_8 a'_7)$ where

$$\begin{aligned}
& (\omega_8 a'_5 + \omega_8^3 a'_7, \omega_8^3 a'_5 + \omega_8 a'_7) \\
= {} & (\omega_8(a'_5 + \omega_8^2 a'_7), \omega_8^3(a'_5 + \omega_8^6 a'_7)) \\
= {} & (\omega_8(a'_5 + \omega_4 a'_7), \omega_8^3(a'_5 + \omega_4^3 a'_7)) \\
= {} & (\omega_8(a'_5 + \omega_4 a'_7), \omega_8^3(a'_5 - \omega_4 a'_7)) \\
= {} & (\omega_8 a''_5, \omega_8^3 a''_7)
\end{aligned}$$

Therefore, we can compute with 2 `smull`s and 2 `smlal`s for the 64-bit value of $\omega_8 a'_5 + \omega_8^3 a'_7, \omega_8^3 a'_5 + \omega_8 a'_7$ and then reduce them to 32-bit with $\mathsf{montgomery}_{q,2^{32}}(\cdot)$. To sum up, we replace three `smull`s, three $\mathsf{montgomery}_{q,2^{32}}(\cdot)$s and one add-sub pair with two `smull`s, two `smlal`s, and two $\mathsf{montgomery}_{q,2^{32}}(\cdot)$s, saving 3 cycles.

To see how to derive the same shape of computation from GS butterflies, let's say applying $\mathsf{NTT}_{R[x]/\langle x^8-1\rangle:\omega_8}$ to $(a_0, \ldots, a_7)$ with GS butterflies is to derive $(a'''_0, \ldots, a'''_7)$ as follows:

1. $(a_0, \ldots, a_7) \mapsto (a'_0, \ldots, a'_7)$ where

$$\begin{aligned}
a'_0 &= a_0 + a_4 \\
a'_1 &= a_1 + a_5 \\
a'_2 &= a_2 + a_6 \\
a'_3 &= a_3 + a_7 \\
a'_4 &= a_0 - a_4 \\
a'_5 &= (a_1 - a_5)\omega_8 \\
a'_6 &= (a_2 - a_6)\omega_4 \\
a'_7 &= (a_3 - a_7)\omega_8^3
\end{aligned}$$

doi:10.6342/NTU202200738

2. $(a'_0, \dots, a'_7) \mapsto (a''_0, \dots, a''_7)$ where

$$\begin{aligned}
(a''_0, a''_4) &= (a'_0, a'_4) + (a'_2, a'_6) \\
(a''_1, a''_5) &= ((a'_1, a'_5) + (a'_3, a'_7))\, \omega_4 \\
(a''_2, a''_6) &= (a'_0, a'_4) - (a'_2, a'_6) \\
(a''_3, a''_7) &= ((a'_1, a'_5) - (a'_3, a'_7))\, \omega_4
\end{aligned}$$

3. $(a''_0, \dots, a''_7) \mapsto (a'''_0, \dots, a'''_7)$ where

$$\begin{aligned}
(a'''_0, a'''_2, a'''_4, a'''_6) &= (a''_0, a''_2, a''_4, a''_6) + (a''_1, a''_3, a''_5, a''_7) \\
(a'''_1, a'''_3, a'''_5, a'''_7) &= (a''_0, a''_2, a''_4, a''_6) - (a''_1, a''_3, a''_5, a''_7)
\end{aligned}$$

We can re-write the computation $(a_0, \dots, a_7) \mapsto (a''_0, \dots, a''_7)$ as

$$\begin{cases}
(a_0, a_2, a_4, a_6) \mapsto (a'_0, a'_2, a'_4, a'_6) \\
(a_1, a_3, a_5, a_7) \mapsto (a'_1, a'_3, a_1 - a_5, a_3 - a_7)
\end{cases}$$

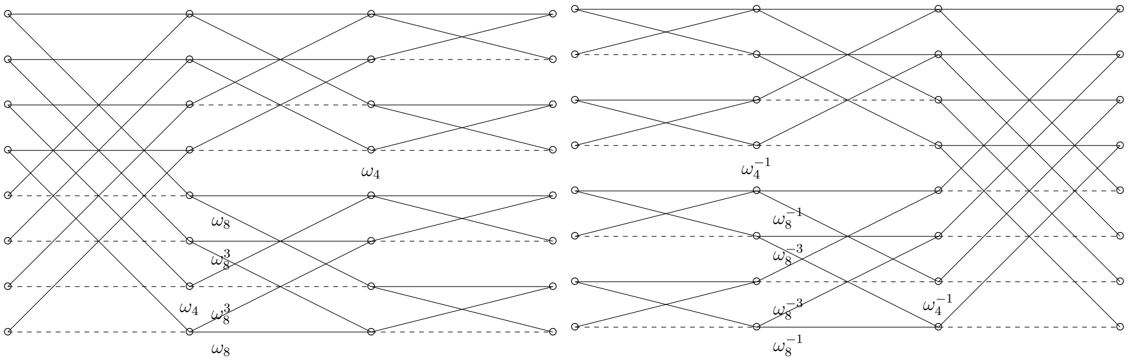
followed by

$$\begin{cases}
(a'_0, a'_2, a'_4, a'_6) \mapsto (a''_0, a''_2, a''_4, a''_6) \\
(a'_1, a'_3) \mapsto (a''_1, a''_3) \\
(a_1 - a_5, a_3 - a_7) \mapsto (a''_5, a''_7)
\end{cases}$$

Now if we compute $(a_1 - a_5, a_3 - a_7) \mapsto (a''_5, a''_7)$ as

$$\begin{aligned}
(a_1 - a_5, a_3 - a_7) &\mapsto ((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, (a_1 - a_5)\omega_8^3 + (a_3 - a_7)\omega_8) \\
&= ((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, (a_1 - a_5)\omega_8^3 - (a_3 - a_7)\omega_8^5) \\
&= ((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, ((a_1 - a_5)\omega_8 - (a_3 - a_7)\omega_8^3)\, \omega_8^2) \\
&= ((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, ((a_1 - a_5)\omega_8 - (a_3 - a_7)\omega_8^3)\, \omega_4) \\
&= (a''_5, a''_7)
\end{aligned}$$

then we derive the same shape of computation.



(a) CT–GS butterflies for NTT over $x^8 - 1$. (b) CT–GS butterflies for iNTT over $x^8 - 1$.

Figure 7.2: CT–GS butterflies [ACC+22].

7.2.6 Improving Non-Radix-2 Butterflies on Cortex-M4

We implement the improved radix-3 butterfly in Section 4.3.5 on Cortex-M4.

doi:10.6342/NTU202200738

Recall that radix-2 CT butterfly maps (c_0, c_1, ζ) to

$$(c_0 + \zeta c_1, c_0 - \zeta c_1),$$

and radix-3 CT butterfly maps (c_0, c_1, c_2, ζ) to

$$(\hat{c}_0, \hat{c}_1, \hat{c}_2) \coloneqq (c_0 + \zeta c_1 + \zeta^2 c_2, c_0 + \zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2, c_0 + \zeta\omega_3^2 c_1 + \zeta^2\omega_3^4 c_2).$$

[CHK+21] implemented 32-bit radix-3 butterflies for `ntruhps4096821` with Algorithm 31, which requires 15 cycles if $\zeta \neq 1$ and 12 cycles if $\zeta = 1$.

Algorithm 31 Radix-3 butterfly [CHK+21].

Input:
$(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}) = (c_0, c_1, c_2)$
Output:

$$\begin{aligned} \mathtt{c0'} = \hat{\mathtt{c0}} &\coloneqq & c_0 + \zeta c_1 + \zeta^2 c_2 \\ \mathtt{c1'} = \hat{\mathtt{c1}} &\coloneqq & c_0 + \zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2 \\ \mathtt{c2'} = \hat{\mathtt{c2}} &\coloneqq & c_0 + \zeta\omega_3^2 c_1 + \zeta^2\omega_3^4 c_2 \end{aligned}$$

```
smull t1, c0’, c1, $\zeta$
smlal t1, c0’, c2, $\zeta^2$
mul t0, t1, $q'$
smlal t1, c0’, t0, $q$                              ▷ c0’ $= \zeta c_1 + \zeta^2 c_2$
                    ▷ If $\zeta = 1$, we compute c0’ $= c_1 + c_2$ with add instead
smull t1, c1’, c1, $\zeta\omega_3$
smlal t1, c1’, c2, $\zeta^2\omega_3^2$
mul t0, t1, $q'$
smlal t1, c1’, t0, $q$                       ▷ c1’ $= \zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2$
smull t1, c2’, c1, $\zeta\omega_3^2$
smlal t1, c2’, c2, $\zeta^2\omega_3^4$
mul t0, t1, $q'$
smlal t1, c2’, t0, $q$                       ▷ c2’ $= \zeta\omega_3^2 c_1 + \zeta^2\omega_3^4 c_2$
add.w c2’, c2’, c0                                          ▷ c2’ $= \hat{c}_2$
add c1’, c0                                                 ▷ c1’ $= \hat{c}_1$
add c0’, c0                                                 ▷ c0’ $= \hat{c}_0$
```

As mentioned in Section 4.3.5, we first compute $\zeta c_1 + \zeta^2 c_2$ and $\zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2$. For computing $\hat{c}_2$, we compute $(\zeta c_1 + \zeta^2 c_2) + (\zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2)$ with *one addition* and subtract the result from c_0. Now we have $c_0 - ((\zeta c_1 + \zeta^2 c_2) + (\zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2)) = \hat{c}_2$ as desired. Finally, we add c_0 to $\zeta c_1 + \zeta^2 c_1$ and $\zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_1$ to dervie $\hat{c}_0$ and $\hat{c}_1$. In total, 12 cycles are required. If $\zeta = 1$, then 9 cycles are required. Algorithm 32 is an illustration. Comparing to Algorithm 31, 3 cycles are saved for each radix-3 butterfly. We also believe that the 16-bit radix-3 and radix-5 butterflies by [ACC+21], and the AVX2 implementaion of `ntruhps4096821` by [CHK+21] can be improved with our idea.

doi:10.6342/NTU202200738

Algorithm 32 Improved radix-3 butterfly.

```
Input:
(c0, c1, c2) = $(c_0, c_1, c_2)$
Output:
(c0, c1, c2) = $(\hat{c_0}, \hat{c_1}, \hat{c_2})$
smull t1, t0, c1, $\zeta$
smlal t1, t0, c2, $\zeta^2$
mul t2, t1, $q'$
smlal t1, t0, t2, $q$                         ▷ t0 = $\zeta c_1 + \zeta^2 c_2$
                 ▷ If $\zeta = 1$, we compute t0 = $c_1 + c_2$ with add instead
smull t1, c1, c1, $\zeta\omega_3$
smlal t1, c1, c2, $\zeta^2\omega_3^2$
mul t2, t1, $q'$
smlal t1, c1, t2, $q$                  ▷ c1 = $\zeta\omega_3 c_1 + \zeta^2\omega_3^2 c_2$
add c2, c1, t0
rsb c2, c2, c0                                ▷ c2 = $\hat{c_2}$
add c1, c0                                    ▷ c1 = $\hat{c_1}$
add c0, t0                                    ▷ c0 = $\hat{c_0}$
```

7.2.7 Dedicated Vector–Radix Butterflies on Cortex-M4

This section introduces how to implement radix-$(2, 3)$ butterflies while permuting the coefficients with Good–Thomas FFT. For simplicty, we illustrate the idea for $R[x]/\langle x^6 - 1\rangle$.

Let e_0 and e_1 be idempotent elements in $\mathbb{Z}_6$ realizing

$$i = (e_0(i \bmod 2) + e_1(i \bmod 3)) \bmod 6.$$

Given coefficients $(c_0, \ldots, c_5)$, we define c_{i_0,i_1} as $c_{e_0 i_0 + e_1 i_1}$. It is clear that introducing the equivalence $x \sim x^{(0)}x^{(1)}$ converts the polynomial $\sum_{i=0}^{5} c_i x^i$ into the polynomial $\sum_{i_0=0}^{2}\sum_{i_1=0}^{1} c_{i_0,i_1}\left(x^{(0)}\right)^{i_0}\left(x^{(1)}\right)^{i_1}$. If $c_3 = c_4 = c_5 = 0$, then we have $c_{1,0} = c_{0,1} = c_{1,2} = 0$. We define the characteristic vector of (c_{i_0,i_1}) as $(\llbracket (e_0 i_0 + e_1 i_1) \bmod 6 \geq 3 \rrbracket)$ where $\llbracket\rrbracket$ is the Iverson bracket. If the larger half of the input are 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 the NTTs. Since the characteristic vector is determined by $(c_{0,0}, c_{0,1}, c_{0,2})$, we name the computation as `_6_ntt_XXX` where `XXX` is the characteristic vector of $(c_{0,0}, c_{0,1}, c_{0,2})$. Algorithm 49 implements `_6_ntt_100` in 12 cycles, Algorithm 50 implements `_6_ntt_010` in 17 cycles, Algorithm 51 implements `_6_ntt_001` in 18 cycles, Algorithm 53 implements `_6_ntt_011` in 12 cycles, Algorithm 52 implements `_6_ntt_101` in 17 cycles, and Algorithm 54 implements `_6_ntt_110` in 17 cycles.

If we permute with Good–Thomas FFT, asymptotically, $\frac{1}{3}$ of `_6_ntt_XXX` are 12 cycles, and $\frac{2}{3}$ of `_6_ntt_XXX` are 17 or 18 cycles.

doi:10.6342/NTU202200738

Chapter 8

Implementations of Schemes

This chapter introduces the implementations of several lattice-based cryptosystems for the CPUs Cortex-M3, Cortex-M4, and Cortex-A72.

Implementations and ideas from the following works are contributions of this thesis.

- Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127-151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292. Reference [ACC+22]. Full version available at https://eprint.iacr.org/2021/995.

- Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221-224, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. Reference [BHK+22]. Full version available at https://eprint.iacr.org/2021/986.

Furthermore, Sections 8.2 and 8.5.1 are contributions of the submitted work:

- Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. Submitted in April 2022.

Table 8.1 summarizes the state-of-the-art implementations of lattice-based cryptosystems related to this thesis on target CPUs.

doi:10.6342/NTU202200738

Table 8.1: Summary of the state-of-the-art implementations of lattice-based cryptosystems in the 3rd round of NIST Post-Quantum Standardization. Submitted work is included in this table. Starred works are contributions of this thesis.

Parameter	Cortex-M3	Cortex-M4	Cortex-A72
Dilithium			
`dilithium2`	[GKS21]	[AHKS22]	[BHK+22]*
`dilithium3`	[GKS21]	[AHKS22]	[BHK+22]*
`dilithium5`	[GKS21]	[AHKS22]	[BHK+22]*
Kyber			
`kyber512`	[GKS21]	[AHKS22]	[BHK+22]*
`kyber768`	[GKS21]	[AHKS22]	[BHK+22]*
`kyber1024`	[GKS21]	[AHKS22]	[BHK+22]*
NTRU			
`ntruhps2048509`	-	[IKPC22]	[NG21]
`ntruhps2048677`	-	This thesis*	[NG21]
`ntruhrss701`	-	This thesis*	[NG21]
`ntruhps4096821`	-	This thesis*	[NG21]
`ntruhps40961229`	-	-	-
`ntruhrss1373`	-	-	-
NTRU Prime			
`ntrulpr653/sntrup653`	-	[Che21]	-
`ntrulpr761/sntrup761`	-	[Che21]	-
`ntrulpr857/sntrup857`	-	This thesis*	-
`ntrulpr953/sntrup953`	-	-	-
`ntrulpr1013/sntrup1013`	-	This thesis*	-
`ntrulpr1277/sntrup1277`	-	This thesis*	-
Saber			
`lightsaber`	[ACC+22]*	[ACC+22]*	[BHK+22]*
`saber`	[ACC+22]*	[ACC+22]*	[BHK+22]*
`firesaber`	[ACC+22]*	[ACC+22]*	[BHK+22]*

doi:10.6342/NTU202200738

8.1 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 lifting a polynomial modulus to a larger polynomial modulus.

8.2 More on Incomplete NTTs

We first describe some important observations about incomplete NTTs.

8.2.1 Incomplete NTTs: Review

Let m and n be positive integers with $n \perp \mathrm{char}(R)$, ζ^n be an invertible element in the ring R, and ω_n be a principal n-th root of unity in R. Incomplete NTTs maps a polynomial $\boldsymbol{a}x \in R[x]/\langle x^{mn} - \zeta^n \rangle$ to the n-tuple $\left(\boldsymbol{a}^{(0)}(\zeta\omega_n^i)\right)_i$ where $\boldsymbol{a}^{(0)}(x^{(0)})$ is the polynomial $\sum_{i=0}^{n-1} \left(\sum_{j=0}^{m-1} a_{im+j} x^j\right) \left(x^{(0)}\right)^i$ in the polynomial ring $\frac{\bar{R}[x^{(0)}]}{\left\langle \left(x^{(0)}\right)^n - \zeta^n \right\rangle}$ and $\bar{R} = R[x]/\left\langle x^m - x^{(0)} \right\rangle$.

8.2.2 Code Size Consideration of Good–Thomas FFT

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

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

Let's take the parameter `ntruhps4096821` for NTRU as an example. [CHK+21] computed size-1728 convolution with mixed-radix CT FFT. They implemented size-576 CT FFT and computed 3×3 schoolbooks. As described in 4.4.4, Good–Thomas FFT is more favorable than Cooley–Tukey FFT because cyclic NTTs are cheaper than acyclic ones. Good–Thomas FFT for `ntrulpr761` and `sntrp761` by [ACC+21] and `ntruhps2048677` and `ntruhrss701` by [CHK+21] suggested to transform the

doi:10.6342/NTU202200738

size-1728 convolution into a 2-dimensional convolution with dimensions 27×64. For computing the 2-dimensional FFT, applying dedicated radix-$(2,3)$ butterflies will result in a blow of code size since it is dominated by $3^{2\cdot 3-1} = 243$ dedicated radix-$(2,3)$ butterflies, including memory operations. There is no way to control the code size and permute the coefficients on-the-fly as suggested by [ACC+21] and [CHK+21].

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

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

Let $\omega_{q_0q_1}$ be a principal q_0q_1-th root of unity, $\omega_{q_0} := \omega_{q_0q_1}^{e_0}$, and $\omega_{q_1} := \omega_{q_0q_1}^{e_1}$ where e_0 and e_1 are orthogonal idempotent elements realizing $i \sim e_0(i \bmod q_0) + e_1(i \bmod q_1) \pmod{q_0q_1}$. We introduce the equivalence $x^v \sim x^{(0)}x^{(1)}$ and adopt the following abbreviations:

- $\mathcal{R} := \frac{R[x]}{\left\langle x^{q_0\tilde{q}q_1v}-1\right\rangle}$.
  - $\forall i = 0, \ldots, q_0q_1 - 1, \mathcal{R}_i := \frac{R[x]}{\left\langle x^{\tilde{q}v}-\omega_{q_0q_1}^{i}\right\rangle}$.
- $\bar{R} := \frac{R[x]}{\left\langle x^{v}-x^{(0)}x^{(1)}\right\rangle}$.
- $\bar{\mathcal{R}}^{(0)} := \frac{\bar{R}[x^{(0)}]}{\left\langle \left(x^{(0)}\right)^{q_0\tilde{q}}-1\right\rangle}$.
  - $\forall i_0 = 0, \ldots, q_0 - 1, \bar{\mathcal{R}}_{i_0} := \frac{\bar{R}[x^{(0)}]}{\left\langle \left(x^{(0)}\right)^{\tilde{q}}-\omega_{q_0}^{i_0}\right\rangle}$.
- $\bar{\mathcal{R}}^{(1)} := \frac{\bar{R}[x^{(1)}]}{\left\langle \left(x^{(1)}\right)^{q_1}-1\right\rangle}$.
  - $\forall i_1 = 0, \ldots, q_1 - 1, \bar{\mathcal{R}}_{i_1} := \frac{\bar{R}[x^{(1)}]}{\left\langle x^{(1)}-\omega_{q_1}^{i_1}\right\rangle}$.
- $\bar{\mathcal{R}} := \frac{\bar{R}[x^{(0)},x^{(1)}]}{\left\langle \left(x^{(0)}\right)^{q_0\tilde{q}}-1,\left(x^{(1)}\right)^{q_1}-1\right\rangle}$.
  - $\forall i_0 = 0, \ldots, q_0 - 1, \forall i_1 = 0, \ldots, q_1 - 1, \bar{\mathcal{R}}_{i_0,i_1} := \frac{\bar{R}[x^{(0)},x^{(1)}]}{\left\langle \left(x^{(0)}\right)^{\tilde{q}}-\omega_{q_0}^{i_0},x^{(1)}-\omega_{q_1}^{i_1}\right\rangle}$.

We compute the result of $\mathsf{NTT}_{\mathcal{R}:\omega_{q_0q_1}} : \mathcal{R} \to \prod_{i=0}^{q_0q_1-1} \mathcal{R}_i$ as follows. We first convert $\mathcal{R}$ into $\bar{\mathcal{R}}^{(0)} \otimes \bar{\mathcal{R}}^{(1)}$ with the equivalence $x^v \sim x^{(0)}x^{(1)}$. Then, we apply $\mathsf{NTT}_{\bar{\mathcal{R}}^{(0)}:\omega_{q_0}} \otimes \mathsf{NTT}_{\bar{\mathcal{R}}^{(1)}:\omega_{q_1}}$ transforming $\bar{\mathcal{R}}^{(0)} \otimes \bar{\mathcal{R}}^{(1)}$ into $\left(\prod_{i_0=0}^{q_0-1} \bar{\mathcal{R}}_{i_0}^{(0)}\right) \otimes \left(\prod_{i_1=0}^{q_1-1} \bar{\mathcal{R}}_{i_1}^{(1)}\right)$. Now $\pi[x^v \sim x^{(0)}x^{(1)}] \left(\prod_{i=0}^{q_0q_1-1} \mathcal{R}_i\right) = \left(\prod_{i_0=0}^{q_0-1} \bar{\mathcal{R}}_{i_0}^{(0)}\right) \otimes \left(\prod_{i_1=0}^{q_1-1} \bar{\mathcal{R}}_{i_1}^{(1)}\right)$ where $\pi[x^v \sim x^{(0)}x^{(1)}]$ is the permutation realizing $x^v \sim x^{(0)}x^{(1)}$.

For computing $\mathsf{NTT}_{\bar{\mathcal{R}}^{(0)}:\omega_{q_0}} \otimes \mathsf{NTT}_{\bar{\mathcal{R}}^{(1)}:\omega_{q_1}}$, we look at $\gcd(v_2(q_0), v_3(q_1))$ for determining the number of layers of radix-$(2, 3)$ butterflies where v_2 and v_3 are valuations. We then replace the 0-th layer of radix-$(2, 3)$ butterflies and the permutation $x^v \sim x^{(0)}x^{(1)}$ with dedicated radix-$(2, 3)$ butterflies according to the assumption that the upper half of the coefficients are all zeros. After $\gcd(v_2(q_0), v_3(q_1))$ layers of radix-$(2, 3)$ butterflies, we compute the remaining layers of radix-2 butterflies.

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

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

8.3 `MatrixVectorMul`

Matrix-to-vector multiplications are heavily used in Dilithium, Kyber, and Saber. Let M be a $k \times l$ matrix, and v be an $l \times 1$ column vector. `MatrixVectorMul` computes $M \cdot_{k \times l} v$ where the operation $\cdot_{k \times l}$ multiplies the components and accumulates the results. In particular, the output is the column vector v' with k components where

$$\forall j = 0, \ldots, k - 1, v'_j = \sum_{i=0}^{l-1} M_{j,i} \cdot v_i$$

and $\cdot$ is the ring multiplication. For Dilithium, Kyber, and Saber, $\cdot$ is the multiplication in the ring $\mathbb{Z}_q[x]/\langle x^n + 1 \rangle$ where $(q, n) = (8380417, 256)$ for Dilithium [ABD+20a], $(q, n) = (3329, 256)$ for Kyber [ABD+20b], and $(q, n) = (8192, 256)$ for Saber [DKRV20]. For each scheme, (k, l) varies for security levels. Table 8.2 summarizes the parameters of `MatrixVectorMul`.

Table 8.2: `MatrixVectorMul` in Dilithium, Kyber, and Saber.

Scheme		Dilithium	Kyber	Saber
(q, n)		$(8380417, 256)$	$(3329, 256)$	$(8192, 256)$
(k, l)	Level 1	$(4, 4)$	$(2, 2)$	$(2, 2)$
	Level 3	$(5, 6)$	$(3, 3)$	$(3, 3)$
	Level 5	$(7, 8)$	$(4, 4)$	$(4, 4)$

doi:10.6342/NTU202200738

8.3.1 Computing MatrixVectorMul with Monomorphisms

For multiplying two ring elements $a, b \in R$, if there is a ring monomorphism Φ from R to S, we can compute with

$$ab = \Phi^{-1}\left(\Phi(a)\Phi(b)\right)$$

where Φ^{-1} is only defined on $\Phi(R)$.

If we apply a monomorphism Φ to the components of M and v, then we see that computing v'_js with

$$\forall j = 0, \ldots, k-1, v'_j = \Phi^{-1}\left(\sum_{i=0}^{l-1} \Phi(M_{j,i})\Phi(v_i)\right)$$

requires $kl + l$ Φs and k Φ^{-1}s. Let C_R be the cost of multiplying two elements in R, C_S be the cost of multiplying two elements in S, C_Φ be the cost of Φ, and $C_{\Phi^{-1}}$ be the cost of Φ^{-1}. If

$$klC_R > klC_S + (kl + l)C_\Phi + kC_{\Phi^{-1}},$$

we improve the computation of $M \cdot_{k\times l} v$. In general, since id is one of the choices for Φ, we cannot degrade the implementations if we are careful.

For Saber, we have the freedom to choose Φ (and hence S). For Dilithium and Kyber, Φ and S are already chosen in the specifications. The result of the 8-layer CT FFT is chosen for Dilithium and the result of the 7-layer CT FFT is chosen for Kyber. However, in Dilithium and Kyber, since the matrix M is already transformed by definition [ABD+20b, ABD+20a], we can remove klC_Φ from the right-hand side.

For Saber, many different choices had been proposed for various platforms. We focus on software implementations. On Cortex-M4, [KRS19] implemented Toom–Cook without exploiting the negacyclic property of $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and the homomorphic characterization of MatrixVectorMul. [MKV20] later realized that Toom–Cook is a choice of Φ and reduced the number of Φ. [IKPC20] introduced Toeplitz-matrix-based approach and [BMK+22] introduced striding Toom–Cook. Both approaches exploit the negacyclic property and are choices for Φ. [CHK+21] and [ACC+22] implemented NTT-based approaches. On Cortex-A72, [NG21] implemented Toom–Cook and NTT-based approaches, and [BHK+22] implemented NTT-based approaches.

In this thesis, we report the works by [ACC+22] and [BHK+22], which are the state-of-the-art of Saber.

8.3.2 Asymmetric Multiplication

We discuss a more advanced approach for improving MatrixVectorMul with NTTs at the level of computation without modifying the underlying mathematics. This approach is called “asymmetric multiplication”. Asymmetric multiplication was introduced by [BHK+22] and also implemented by [AHKS22]. This thesis reports the work by [BHK+22].

We recall from previous section that by choosing $\Phi = \mathsf{NTT}$, we can compute $M \cdot_{k\times l} v$ as

$$M \cdot_{k\times l} v = \mathsf{NTT}^{-1}\left(\mathsf{NTT}(M) \cdot_m \mathsf{NTT}(v)\right)$$

doi:10.6342/NTU202200738

where $\cdot_m$ is the multiplication in the product ring $\prod_{i=0}^{\frac{n}{m}-1} \mathbb{Z}_q[x] \Big/ \left\langle x^m - \zeta^{\frac{n}{m}} \omega_{\frac{n}{m}}^{i} \right\rangle$.

Algorithm 33 4×4 convolution [BHK+22].

```
Inputs:
$\boldsymbol{a}(x), \boldsymbol{b}(x)$
Temporary registers:
$\mathtt{c}, \mathtt{l}, \mathtt{h}, \mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{T0}, \ldots, \mathtt{T3}$
Output:
$c_0 + c_1 x + c_2 x^2 + c_3 x^3 = (\boldsymbol{a}(x) * \boldsymbol{b}(x) \bmod (x^4 - 1))\, \mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_0$
    $(\mathtt{a0}, \ldots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \ldots, \mathtt{T3}) = (b_0, b_3, b_2, b_1)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
$\mathtt{c} = \mathsf{montgomery}_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$    ▷ $c_0 = (a_0 b_0 + a_1 b_3 + a_2 b_2 + a_3 b_1)\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_1$
    $(\mathtt{a0}, \ldots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \ldots, \mathtt{T3}) = (b_1, b_0, b_3, b_2)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
$\mathtt{c} = \mathsf{montgomery}_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$    ▷ $c_1 = (a_0 b_1 + a_1 b_0 + a_2 b_3 + a_3 b_2)\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_2$
    $(\mathtt{a0}, \ldots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \ldots, \mathtt{T3}) = (b_2, b_1, b_0, b_3)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
$\mathtt{c} = \mathsf{montgomery}_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$    ▷ $c_2 = (a_0 b_2 + a_1 b_1 + a_2 b_0 + a_3 b_3)\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_3$
    $(\mathtt{a0}, \ldots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \ldots, \mathtt{T3}) = (b_3, b_2, b_1, b_0)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
$\mathtt{c} = \mathsf{montgomery}_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$    ▷ $c_3 = (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0)\mathtt{R}^{-1}$
```

Asymmetric multiplication is an improved technique caching the computations for later use. It is applicable whenever incomplete NTTs are cached and reused several times. For simplicity, we illustrate the idea for a specific $\zeta' = \zeta^{\frac{n}{m}} \omega_{\frac{n}{m}}^{i}$. For multiplying two size-m polynomials $\boldsymbol{a}(x) = \sum_{j=0}^{m-1} a_j x^j$ and $\boldsymbol{b}(x) = \sum_{j=0}^{m-1} b_j x^j$ in $\mathbb{Z}_q[x]/\langle x^m - \zeta' \rangle$, we want to compute

$$\boldsymbol{a}(x)\boldsymbol{b}(x) = \sum_{j=0}^{m-1} \left(\sum_{h=0}^{j} a_h b_{j-h} + \zeta' \sum_{h=j+1}^{m-1} a_h b_{j-h+m} \right) x^j.$$

doi:10.6342/NTU202200738

If we instead compute

$$\boldsymbol{a}(x)\boldsymbol{b}(x) = \sum_{j=0}^{m-1}\left(\sum_{h=0}^{j} a_h b_{j-h} + \sum_{h=j+1}^{m-1} a_h(\zeta' b_{j-h+m})\right) x^j,$$

we can regard the computation as a function mapping $\boldsymbol{a}(x), \boldsymbol{b}(x), \boldsymbol{b}'(x)$ to $\boldsymbol{a}(x)\boldsymbol{b}(x)$ where $\boldsymbol{b}'(x) = b_0 + \zeta' \sum_{j=1}^{m-1} b_j x^j$. The number of multiplications is therefore the same as $\zeta' = 1$. Algorithm 33 is an illustration for $\zeta' = 1$ and Algorithm 34 is an illustration for $\zeta' \neq 1$ where $\boldsymbol{b}'(x)$ is already computed. For computing $\boldsymbol{b}'(x)$ from $\boldsymbol{b}(x)$, we have to perform $m-1$ multiplications and this is exactly the difference of the number of multiplications between $\zeta' = 1$ and $\zeta' \neq 1$. So asymmetric multiplication improves nothing if $\boldsymbol{a}(x)$ and $\boldsymbol{b}(x)$ are used only once.

Algorithm 34 4×4 asymmetric multiplication [BHK+22].

```
Inputs:
$\boldsymbol{a}(x), \boldsymbol{b}(x), \boldsymbol{b}'(x)$
Temporary registers:
c, l, h, a0, ..., a3, T0, ..., T3
Output:
$c_0 + c_1 x + c_2 x^2 + c_3 x^3 = (\boldsymbol{a}(x) * \boldsymbol{b}(x) \bmod (x^4 - \zeta))\,\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_0$
    $(\mathtt{a0}, \dots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \dots, \mathtt{T3}) = (b_0, \zeta b_3, \zeta b_2, \zeta b_1)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
c = montgomery$_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$        ▷ $c_0 = (a_0 b_0 + \zeta(a_1 b_3 + a_2 b_2 + a_3 b_1))\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_1$
    $(\mathtt{a0}, \dots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \dots, \mathtt{T3}) = (b_1, b_0, \zeta b_3, \zeta b_2)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
c = montgomery$_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$        ▷ $c_1 = (a_0 b_1 + a_1 b_0 + \zeta(a_2 b_3 + a_3 b_2))\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_2$
    $(\mathtt{a0}, \dots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \dots, \mathtt{T3}) = (b_2, b_1, b_0, \zeta b_3)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
c = montgomery$_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$        ▷ $c_2 = (a_0 b_2 + a_1 b_1 + a_2 b_0 + \zeta a_3 b_3)\mathtt{R}^{-1}$
Name
    $\mathtt{c} = c_3$
    $(\mathtt{a0}, \dots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$
    $(\mathtt{T0}, \dots, \mathtt{T3}) = (b_3, b_2, b_1, b_0)$
$\mathtt{l} + \mathtt{hR} = \mathtt{a0} \cdot \mathtt{T0} + \mathtt{a1} \cdot \mathtt{T1} + \mathtt{a2} \cdot \mathtt{T2} + \mathtt{a3} \cdot \mathtt{T3}$
c = montgomery$_{q,\mathtt{R}}(\mathtt{l} + \mathtt{hR})$        ▷ $c_3 = (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0)\mathtt{R}^{-1}$
```

doi:10.6342/NTU202200738

Things become quite different if $\boldsymbol{a}(x)$ or $\boldsymbol{b}(x)$ are reused several times. Let `asymmetric_mul` be the function mapping $\boldsymbol{a}(x), \boldsymbol{b}(x), \boldsymbol{b}'(x)$ to $\boldsymbol{a}(x)\boldsymbol{b}(x)$. We can compute $M \cdot_{k\times l} v$ as

$$M \cdot_{k\times l} v = \mathsf{NTT}^{-1}\left(\texttt{asymmetric\_mul}_{k\times l}(\mathsf{NTT}(M), \mathsf{NTT}(v), \mathsf{NTT}(v)')\right)$$

at the expense of computing $\mathsf{NTT}(v)'$ from $\mathsf{NTT}(v)$ [BHK+22]. Let $\mathsf{NTT}_{\mathsf{heavy}} := v \mapsto (\mathsf{NTT}(v), \mathsf{NTT}(v)')$. In summary, we have a cheaper computation for $k \times l$ base multiplications at the cost of l maps of the form $\boldsymbol{b}(x) \mapsto (\boldsymbol{b}(x), \boldsymbol{b}'(x))$. The number of multiplications is therefore the same as the sum of l base multiplications with $\zeta' \neq 1$ and $(k-1)l$ base multiplications with $\zeta' = 1$.

Note that no restrictions are attached in order to apply `asymmetric_mul`. In particular, we do not require the existence of an m-th root of ζ'. This is difference from the isomorphsim $\mathbb{Z}_q[x]/\langle x^m - \zeta'\rangle \cong \mathbb{Z}_q[x]/\langle x^m - 1\rangle$ which requires taking an m-th root of ζ'.

Lastly, for accumulating several products of `asymmetric_mul`, we can accumulate double-width results and only reduce them to the single-width value after computing all the corresponding `asymmetric_mul`s.

A Toeplitz matrix view of asymmetric multiplication. The asymmetric multiplication can be seen as an application of the low-degree Toeplitz matrix. Recall that computing $\boldsymbol{a}(x)\boldsymbol{b}(x)$ over $R[x]/\langle x^n - \zeta\rangle$ can be seen as applying a Toeplitz matrix with the array form $(b_{n-1}, \dots, b_0, \zeta b_{n-1}, \dots, \zeta b_1)$. It turns out that the asymmetrict multiplication is exactly applying the Toeplitz matrix $(b_{n-1}, \dots, b_0, \zeta b_{n-1}, \dots, \zeta b_1)$ to the column vector

$$\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Therefore, the performance gain of asymmetric multiplication of incomplete-NTT-based `MatrixVectorMul` can be seen as caching the array forms of Toeplitz matrices for $\zeta \neq \pm 1$.

8.3.3 Time-Memory Tradeoffs for Saber

For Saber, the most critical operations is the `MatrixVectorMul` $A^T s$ in key generation and As' in encryption. As mentioned in Section 8.3.1, we can implement `MatrixVectorMul` with NTTs. This implies an increase of precision of the arithmetic. In particular, the input coefficients are 16-bit integers but we have to compute the precision up to 25 bits as an injection to $\mathbb{Z}[x]$.

The rest of this section is to analyze the impact of memory usage for computing `MatrixVectorMul` as an monomorphism.

We first analyze the bottleneck for computing $A^T s$ and As'. In all implementations, we employ on-the-fly generation of A suggested in [MKV20], and consequently, only need one polynomial of A in memory. For computing $A^T s$ in key generation, we can compute the NTTs for s on-the-fly and accumulate the result in the NTT domain with l accumulators. This is because the first component of the result only depends on the first column of A and the first component of s. For computing As' during encryption, we compute the entire NTT of s with l polynomial buffers but

doi:10.6342/NTU202200738

hold only one buffer for accumulations. This is because a component of the result depends only on a row of A and s'. In summary, for computing $A^T s$, the most memory-consuming part is the accumulation in the NTT domain. And for computing As', the most memory-consuming part is transforming s' into the NTT domain. In the most speed-optimized and the most stack-optimized implementations, there is no downside to this. But they result in different time-memory trade-offs as shown below.

This thesis reports the findings by [ACC+22]. [ACC+22] showed four ways for computing the `MatrixVectorMul`, and named them strategies A, B, C, and D. They are distinguished by caching the NTTs of s or not and accumulating in the NTT domain or not.

A. We cache $\texttt{NTT}(s)$ and accumulate values in the NTT domain.

B. we cache $\texttt{NTT}(s)$ and accumulate values in normal domain.

C. we re-compute $\texttt{NTT}(s)$ and accumulate values in the NTT domain.

D. we re-compute $\texttt{NTT}(s)$ and accumulate values in normal domain.

All four strategies apply to $A^T s$ and As'. A is the fastest, and D consumes the least amount of memory. B and C run in comparable cycles but result in different degrees of tradeoff for memory. For reducing the memory usage of $A^T s$, B is much better than C since B effectively reduces the size of accumulators. On the other hand, for reducing the memory usage of As', C is much better than B, since C avoids caching the NTTs of s'.

8.4 Stack Optimizations on Cortex-M4

This thesis reports an implementation aspect of stack optimization on implemeting NTTs for NTT-unfriendly rings on Cortex-M4 by [ACC+22]. Let q_0 and q_1 be coprime moduli for 16-bit NTTs, then we can compute a 32-bit NTT over $\mathbb{Z}_{q_0 q_1}$. On Cortex-M4, since long multiplications take 1 cycle each, a 32-bit NTT over $q_0 q_1$ easily outpaces two 16-bit NTTs. We apply this observation for multiplying polynomials. We denote NTT_q for an NTT defined over a ring with the coefficient ring $\mathbb{Z}_q$ and $\cdot_q$ be the ring multiplication of the product ring after applying NTT_q.

For multiplying two size-n polynomials, if the coefficients of the result are smaller than a product of k 16-bit primes, then we only need $(k+1) \cdot n \cdot 2 = 2n(k+1)$ bytes of storage by interleaving the computation of 16-bit NTTs as shown in Algorithm 35.

Algorithm 35 Computing with k 16-bit NTTs.

```
Declare int16_t buf[$k+1$][$n$]
for $i = 0, \ldots, k-1$ do
    buf[$i$] $= \mathsf{NTT}_{q_i}(a)$
    buf[$i+1$] $= \mathsf{NTT}_{q_i}(b)$
    buf[$i$] $=$ buf[$i$] $\cdot_{q_i}$ buf[$i+1$]
    buf[$i$] $= \mathsf{NTT}_{q_i}^{-1}($buf[$i$]$)$
end for
res $= \mathrm{CRT}($buf[0]$, \ldots,$ buf[$k-1$]$)$
```

doi:10.6342/NTU202200738

However, since 32-bit NTT is significantly faster than two 16-bit NTTs, we should replace every two 16-bit NTTs with a 32-bit NTT. If k is odd, then we can process the multiplicands by computing $\frac{k-1}{2}$ 32-bit NTTs and one 16-bit NTT for each as shown in Algorithm 36. If k is even, for the first multiplicand, we compute $\frac{k}{2}$ 32-bit NTTs and transform the last one into the result of two 16-bit NTTs, while for the second multiplicand, we compute $\frac{k}{2} - 1$ 32-bit NTTs and two 16-bit NTTs. Algorithm 37 is an illustration.

Algorithm 36 Improving with 32-bit NTTs for odd k [ACC+22].

```
Declare int16_t buf[$k+1$][$n$]
Declare int32_t *buf_int32 = (int32_t *)(&buf[0][0])
for $i = 0, \ldots, \frac{k-1}{2} - 1$ do
    buf_int32 + $in = \mathsf{NTT}_{q_{2i}q_{2i+1}}(a)$
    buf_int32 + $(i+1)n = \mathsf{NTT}_{q_{2i}q_{2i+1}}(b)$
    buf_int32 + $in =$ (buf_int32 + $in$) $\cdot_{q_{2i}q_{2i+1}}$ (buf_int32 + $(i+1)n$)
    buf_int32 + $in = \mathsf{NTT}^{-1}_{q_{2i}q_{2i+1}}$(buf_int32 + $in$)
end for
buf[$k-1$] $= \mathsf{NTT}_{q_{k-1}}(a)$
buf[$k$] $= \mathsf{NTT}_{q_{k-1}}(b)$
buf[$k-1$] $=$ buf[$k-1$] $\cdot_{q_{k-1}}$ buf[$k$]
buf[$k-1$] $= \mathsf{NTT}^{-1}_{q_{k-1}}$(buf[$k-1$])
res $= \mathsf{CRT}($buf_int32, buf_int32 + $n, \ldots,$ buf_int32 + $\left(\frac{k-1}{2} - 1\right) n$, buf[$k-1$]$)$
```

Algorithm 37 Improving with 32-bit NTTs for even k [ACC+22].

```
Declare int16_t buf[$k+1$][$n$]
Declare int32_t *buf_int32 = (int32_t *)(&buf[0][0])
for $i = 0, \ldots, \frac{k}{2} - 2$ do
    buf_int32 + $in = \mathsf{NTT}_{q_{2i}q_{2i+1}}(a)$
    buf_int32 + $(i+1)n = \mathsf{NTT}_{q_{2i}q_{2i+1}}(b)$
    buf_int32 + $in =$ (buf_int32 + $in$) $\cdot_{q_{2i}q_{2i+1}}$ (buf_int32 + $(i+1)n$)
    buf_int32 + $in = \mathsf{NTT}^{-1}_{q_{2i}q_{2i+1}}$(buf_int32 + $in$)
end for
buf_int32 + $\left(\frac{k}{2} - 1\right) n = \mathsf{NTT}_{q_{k-2}q_{k-1}}(a)$
buf[$k$] $=$ (buf_int32 + $\left(\frac{k}{2} - 1\right) n$) mod $q_{k-1}$
buf[$k-2$] $=$ (buf_int32 + $\left(\frac{k}{2} - 1\right) n$) mod $q_{k-2}$
buf[$k-1$] $= \mathsf{NTT}_{q_{k-1}}(b)$
buf[$k$] $=$ buf[$k$] $\cdot_{q_{k-1}}$ buf[$k-1$]
buf[$k-1$] $= \mathsf{NTT}_{q_{k-2}}(b)$
buf[$k-1$] $=$ buf[$k-2$] $\cdot_{q_{k-2}}$ buf[$k-1$]
buf_int32 + $\left(\frac{k}{2} - 1\right) n = \mathsf{CRT}($buf[$k-1$], buf[$k$]$)$
buf_int32 + $\left(\frac{k}{2} - 1\right) n = \mathsf{NTT}^{-1}_{q_{k-2}q_{k-1}}$(buf_int32 + $\left(\frac{k}{2} - 1\right) n$)
res $= \mathsf{CRT}($buf_int32, buf_int32 + $n, \ldots,$ buf_int32 + $\left(\frac{k}{2} - 1\right) n)$
```

doi:10.6342/NTU202200738

8.5 Cortex-M4:NTRU, NTRU Prime, and Saber

8.5.1 NTRU and NTRU Prime

Throughout this section, we assume $R = \mathbb{Z}_{q'}$ is a suitable coefficient ring for NTTs. There are six paramter sets for NTRU and twelve parameter sets for NTRU Prime. This thesis only focus on implementing polynomial multiplications. The strategies are summarized in Table 8.3.

Table 8.3: Strategies for polynomial multiplications in NTRU and NTRU Prime on Cortex-M4.

Convolution	NTRU	NTRU Prime
Size-1024	`ntruhps2048509`	-
Size-1440	`ntruhps2048677/ntruhrss701`	`ntrulpr653/sntrup653`
Size-1536	`ntruhps2048677/ntruhrss701`	`ntrulpr761/sntrup761`
Size-1728	`ntruhps4096821`	`ntrulpr857/sntrup857`
Size-2048	-	`ntrulpr1013/sntrup1013`
Size-2560	-	`ntrulpr1277/sntrup1277`

The sizes of convolutions are chosen to be size-$2^{k_0}3^{k_1}5^{k_2}$ convolutions covering the maximum degrees of the results as in $\mathbb{Z}[x]$ where $k_2 = 0, 1$. For `ntruhps2048509`, we choose a size-1024 convolution. For `ntruhps2048677` and `ntruhrss701`, we choose size-1440 and size-1536 convolutions. The size-1440 convolution also supports `ntrulpr653` and `sntrup653`, and the size-1536 convolution also supports `ntrulpr761` and `sntrup761`. For `ntruhps4096821`, `ntrulpr857`, and `sntrup857`, we choose size-1728 convolutions.

We report the implementations in the increasing order of the sizes of the convolutions.

Size-1024 **convolution for `ntruhps2048509`.** We implement a size-1024 convolution for `ntruhps2048509` with two levels of 4-layer radix-2 butterflies as follows.

$$
\begin{aligned}
\frac{R[x]}{\left\langle x^{1024} - 1 \right\rangle} &\overset{\eta_0}{\cong} \prod_{i=0}^{15} \frac{R[x]}{\left\langle x^{128} - \omega_{256}^{16\mathrm{rev}_{(2:4)}(i)} \right\rangle} \\
&\overset{\eta_1}{\cong} \prod_{i=0}^{255} \frac{R[x]}{\left\langle x - \omega_{256}^{\mathrm{rev}_{(2:8)}(i)} \right\rangle}
\end{aligned}
$$

where η_0 and η_1 are 4-layer radix-2 butterflies.

Size-1440 **convolutions for `ntruhps2048677`, `ntruhrss701`, `ntrulpr653`, and `sntrup653`.** We introduce the equivalences $x \sim x^{(0)}x^{(1)}$, $\left(x^{(0)}\right)^9 - 1$, and $\left(x^{(1)}\right)^{160} - 1$ and compute a 2-dimensional transformation 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. The computation is as follows.

doi:10.6342/NTU202200738

$$
\begin{aligned}
\frac{R[x]}{\langle x^{1440}-1\rangle} &\overset{\eta_0}{\cong} \prod_{i_0=0}^{2}\prod_{i_1=0}^{1} \frac{R[x^{(0)},x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, \left(x^{(0)}\right)^3-\omega_9^{3i_0}, \left(x^{(1)}\right)^{80}-\omega_{32}^{16i_1}\right\rangle} \\
&\overset{\eta_1}{\cong} \prod_{i_0}^{8}\prod_{i_1=0}^{3} \frac{R[x^{(0)},x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, x^{(0)}-\omega_9^{\mathrm{rev}_{(3:2)}(i_0)}, \left(x^{(1)}\right)^{40}-\omega_{32}^{8\mathrm{rev}_{(2:2)}(i_0)}\right\rangle} \\
&\overset{\eta_2}{\cong} \prod_{i_0}^{8}\prod_{i_1=0}^{31} \frac{R[x^{(0)},x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, x^{(0)}-\omega_9^{\mathrm{rev}_{(3:2)}(i_0)}, \left(x^{(1)}\right)^{5}-\omega_{32}^{\mathrm{rev}_{(2:5)}(i_0)}\right\rangle}
\end{aligned}
$$

where η_0 is the level of dedicated radix-$(2,3)$ butterflies, η_1 is the level of (generic) radix-$(2,3)$ butterflies, and η_2 is the level of 3-layer radix-2 butterflies.

Size-1536 convolutions for `ntruhps2048677`, `ntruhrss701`, `ntrulpr761`, and `sntrup761`. For computing a size-1536 convolution, [ACC+21] and [CHK+21] computed with Good–Thomas FFT by introducing the equivalences $x \sim x^{(0)}x^{(1)}$, $\left(x^{(0)}\right)^3 \sim 1$, and $\left(x^{(1)}\right)^{512} \sim 1$. They then compute three size-512 NTTs and 512 3×3 convolutions. We introduce the same equivalences but proceed differently. We compute a 2-dimensional transformation with one level of dedicated radix-$(2,3)$ butterflies and two levels of 3-layer radix-2 butterflies. The computation is as follows.

$$
\begin{aligned}
\frac{\mathbb{Z}_{q'}[x]}{\langle x^{1536}-1\rangle} &\overset{\eta_0}{\cong} \prod_{i_0=0}^{2}\prod_{i_1=0}^{1} \frac{\mathbb{Z}_{q'}[x^{(0)},x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, x^{(0)}-\omega_3^{i_0}, \left(x^{(1)}\right)^{256}-\omega_{128}^{64\mathrm{rev}_{(2:1)}(i_1)}\right\rangle} \\
&\overset{\eta_1}{\cong} \prod_{i_0=0}^{2}\prod_{i_1=0}^{15} \frac{\mathbb{Z}_{q'}[x^{(0)},x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, x^{(0)}-\omega_3^{i_0}, \left(x^{(1)}\right)^{32}-\omega_{128}^{8\mathrm{rev}_{(2:4)}(i_1)}\right\rangle} \\
&\overset{\eta_2}{\cong} \prod_{i_0=0}^{2}\prod_{i_1=0}^{127} \frac{\mathbb{Z}_{q'}[x^{(0)},x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, x^{(0)}-\omega_3^{i_0}, \left(x^{(1)}\right)^{4}-\omega_{128}^{\mathrm{rev}_{(2:7)}(i_1)}\right\rangle}
\end{aligned}
$$

where η_0 is the level of dedicated radix-$(2,3)$ butterflies, η_1 is the level of radix-$(2,3)$ butterflies, and η_2 is the level of 3-layer radix-2 butterflies.

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

We first assume that dedicated radix-$(2,3)$ butterflies take $15\frac{1}{3}$ cycles on average (this is not true for a size-24 convolution, but it is true for our size-1536 and size-1728 convolutions), a 4×4 schoolbook takes 30 cycles, and a radix-$(2,3)$ butterfly

doi:10.6342/NTU202200738

takes $9 \cdot 2 + 2 \cdot 3 = 24$ cycles. Then we only need $15\frac{1}{3} \cdot 4 \cdot 2 + 30 \cdot 6 + 24 \cdot 4 \approx 399$ cycles, only 92% of the approach by [ACC+21]. The actual saving is larger since after computing dedicated radix-$(2, 3)$ butterflies, we reach the 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 two levels of 3-layer radix-2 butterflies.

Size-1728 convolutions for `ntruhps4096821`, `ntrulpr857`, `sntrup857`. First of all, we introduce the equivalences $x^3 \sim x^{(0)}x^{(1)}$ to perform an incomplete permutation for Good–Thomas FFT. We now regard $\bar{R} = \frac{\mathbb{Z}_{q'}[x]}{\left\langle x^3 - x^{(0)}x^{(1)} \right\rangle}$ as the coefficient ring. Since $\frac{1728}{3} = 9 \cdot 64$, we perform a 2-dimensional FFT defined over the ring $\frac{\bar{R}[x^{(0)}, x^{(1)}]}{\left\langle \left(x^{(0)}\right)^9 - 1, \left(x^{(1)}\right)^{64} - 1 \right\rangle}$ with vector-radix FFT. Our vector-radix FFT is built upon the tensor product of a size-9 CT FFT and a size-64 CT FFT. 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. 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 is as follows.

$$
\begin{aligned}
\frac{R[x]}{\left\langle x^{1728} - 1 \right\rangle} &\overset{\eta_0}{\cong} \prod_{i_0=0}^{2} \prod_{i_1=0}^{1} \frac{\bar{R}[x^{(0)}, x^{(1)}]}{\left\langle \left(x^{(0)}\right)^3 - \omega_9^{3i_0}, \left(x^{(1)}\right)^{32} - \omega_{64}^{32\mathrm{rev}_{(2:1)}(i_1)} \right\rangle} \\
&\overset{\eta_1}{\cong} \prod_{i_{0,0}, i_{0,1}=0}^{2} \prod_{i_1=0}^{3} \frac{\bar{R}[x^{(0)}, x^{(1)}, y^{(0)}]}{\left\langle x^{(0)} - \omega_9^{i_{0,0}} y^{(0)}, y^{(0)} - \omega_9^{3i_{0,1}}, \left(x^{(1)}\right)^{16} - \omega_{64}^{16\mathrm{rev}_{(2:2)}(i_1)} \right\rangle} \\
&= \prod_{i_0=0}^{8} \prod_{i_1=0}^{3} \frac{\bar{R}[x^{(0)}, x^{(1)}]}{\left\langle x^{(0)} - \omega_9^{\mathrm{rev}_{(3:2)}(i_0)}, \left(x^{(1)}\right)^{16} - \omega_{64}^{16\mathrm{rev}_{(2:2)}(i_1)} \right\rangle} \\
&\overset{\eta_2}{\cong} \prod_{i_0=0}^{8} \prod_{i_1=0}^{63} \frac{\bar{R}[x^{(0)}, x^{(1)}]}{\left\langle x^{(0)} - \omega_9^{\mathrm{rev}_{(3:2)}(i_0)}, x^{(1)} - \omega_{64}^{\mathrm{rev}_{(2:6)}(i_1)} \right\rangle} \\
&= \prod_{i_0=0}^{8} \prod_{i_1=0}^{63} \frac{R[x]}{\left\langle x^3 - \omega_9^{\mathrm{rev}_{(3:2)}(i_0)} \omega_{64}^{\mathrm{rev}_{(2:6)}(i_1)} \right\rangle}
\end{aligned}
$$

where η_0 is the level of dedicated radix-$(2, 3)$ butterflies, η_1 is the level of radix-$(2, 3)$ butterflies, and η_2 is the level of 4-layer radix-2 butterflies.

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 the twiddle factors in the NTTs, they required $9 + 31 + 63 \cdot 8 = 544$ distinct twiddle factors. We instead require $9 + 9 \cdot 1 + 30 = 48$ distinct twiddle factors where the $9 \cdot 1$ are the twiddles $\omega_9^{i_0} \omega_{64}^{16}$ used in the vector–radix FFT. This implies less memory operations. For the second difference, we compute the result of

doi:10.6342/NTU202200738

a 1-*dimensional* size-576 NTT with a 2-*dimensional* FFT. [CHK+21] computed the result of a 1-*dimensional* size-576 NTT with a 1-*dimensional* FFT instead. There are two benefits: the 2-dimensional FFT requires less multiplications than the 1-dimensional FFT regardless if half of the inputs are all 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-2048 **convolution for `ntrulpr1013` and `sntrup1013`.** We implement a size-2048 convolution for `ntrulpr1013` and `sntrup1013` with 3 levels of 3-layer radix-2 butterflies as follows.

$$\begin{aligned}
\frac{R[x]}{\langle x^{2048}-1\rangle} &\overset{\eta_0}{\cong} \prod_{i=0}^{7} \frac{R[x]}{\left\langle x^{256}-\omega_{512}^{64\mathrm{rev}_{(2:3)}(i)}\right\rangle} \\
&\overset{\eta_1}{\cong} \prod_{i=0}^{63} \frac{R[x]}{\left\langle x^{32}-\omega_{512}^{8\mathrm{rev}_{(2:6)}(i)}\right\rangle} \\
&\overset{\eta_2}{\cong} \prod_{i=0}^{511} \frac{R[x]}{\left\langle x^{4}-\omega_{512}^{\mathrm{rev}_{(2:9)}(i)}\right\rangle}
\end{aligned}$$

where η_0, η_1 and η_2 are 3-layer radix-2 butterflies.

Size-2560 **convolution for `ntrulpr1277` and `sntrup1277`.** We implement a size-2560 convolution for `ntrulpr1277` and `sntrup1277` with 3 levels of 3-layer radix-2 butterflies as follows.

$$\begin{aligned}
\frac{R[x]}{\langle x^{2560}-1\rangle} &\overset{\eta_0}{\cong} \prod_{i=0}^{7} \frac{R[x^{(0)}, x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, \left(x^{(0)}\right)^5-1, \left(x^{(1)}\right)^{64}-\omega_{512}^{64\mathrm{rev}_{(2:3)}(i)}\right\rangle} \\
&\overset{\eta_1}{\cong} \prod_{i=0}^{63} \frac{R[x^{(0)}, x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, \left(x^{(0)}\right)^5-1, \left(x^{(1)}\right)^{8}-\omega_{512}^{8\mathrm{rev}_{(2:6)}(i)}\right\rangle} \\
&\overset{\eta_2}{\cong} \prod_{i=0}^{511} \frac{R[x^{(0)}, x^{(1)}]}{\left\langle x-x^{(0)}x^{(1)}, \left(x^{(0)}\right)^5-1, x^{(1)}-\omega_{512}^{\mathrm{rev}_{(2:9)}(i)}\right\rangle}
\end{aligned}$$

where η_0 consists of dedicated 3-layer radix-2 butterflies, and η_1 and η_2 are 3-layer radix-2 butterflies.

8.5.2 Saber

This section introduces the implementations of Saber on Cortex-M4. This thesis reports the unmasked Saber by [ACC+22]. For simplicity, we illustrate our strategies only for `MatrixVectorMul` As' in encryption. However, the ideas apply analogously to $A^T s$ in key generation.

The polynomial ring used in Saber is $\mathbb{Z}_{8192}[x]/\langle x^{256}+1\rangle$. As mentioned in Section 8.3.1, we choose an NTT-friendly modulus and compute As' and $A^T s$ with monomorphisms. In this section, we choose $q' = p_0 p_1$ as a product of two NTT-friendly primes. Our implementations are built upon two levels of 3-layer radix-2

doi:10.6342/NTU202200738

butterflies. In general, we compute the following isomorphism with different approaches.

$$\frac{\mathbb{Z}_{q'}[x]}{\langle x^{256}+1\rangle} \cong \prod_{i=0}^{63} \frac{\mathbb{Z}_{q'}[x]}{\left\langle x^4 - \omega_{128}^{1+2\mathrm{rev}_{(2:6)}(i)} \right\rangle}.$$

For unmasked Saber, we choose $(p_0, p_1) = (3329, 7681)$ enabling flexible time-memory tradeoffs. For masked Saber, we choose $(p_0, p_1) = (44683393, 769)$ for correctness.

Speed-optimized `MatrixVectorMul` for unmasked Saber. We implement Strategy A for speed-optimized `MatrixVectorMul` with the NTT defined over the ring $\mathbb{Z}_{q'}[x]/\langle x^{256}+1\rangle$.

Stack-optimized `MatrixVectorMul` for unmasked Saber. Before going through the implementation of stack-optimized `MatrixVectorMul`, we first discuss a concrete implementation of the stack-optimized polynomial multiplication described in Section 8.4. Let $\omega_{p_0:128}$ be a principal 128-th roots of unity in $\mathbb{Z}_{p_0}$, $\omega_{p_1:128}$ be a principal 128-th roots of unity in $\mathbb{Z}_{p_1}$, and $\omega_{p_0p_1:128}$ be $\mathsf{CRT}(\omega_{p_0:128}, \omega_{p_1:128})$. Let

- $\mathcal{R} := \mathbb{Z}_{p_0p_1}[x]/\langle x^{256}+1\rangle$.
- $\mathcal{R}^{(0)} := \mathbb{Z}_{p_0}[x]/\langle x^{256}+1\rangle$.
- $\mathcal{R}^{(1)} := \mathbb{Z}_{p_1}[x]/\langle x^{256}+1\rangle$.
- $\forall i = 0, \ldots, 63,$
  - $\mathcal{R}_i := \frac{\mathbb{Z}_{p_0p_1}[x]}{\left\langle x^4-\omega_{p_0p_1:128}^{1+2\mathrm{rev}_{(2:6)}(i)}\right\rangle}$.
  - $\mathcal{R}_i^{(0)} := \frac{\mathbb{Z}_{p_0}[x]}{\left\langle x^4-\omega_{p_0:128}^{1+2\mathrm{rev}_{(2:6)}(i)}\right\rangle}$.
  - $\mathcal{R}_i^{(1)} := \frac{\mathbb{Z}_{p_1}[x]}{\left\langle x^4-\omega_{p_1:128}^{1+2\mathrm{rev}_{(2:6)}(i)}\right\rangle}$.

  By the CRT for rings, we have the following isomorphisms:

- $\mathbb{Z}_{p_0p_1} \cong \mathbb{Z}_{p_0} \times \mathbb{Z}_{p_1}$.
- $\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega^2_{p_0:128}} : \mathcal{R}^{(0)} \cong \prod_{i=0}^{63} \mathcal{R}_i^{(0)}$.
- $\mathsf{NTT}_{\mathcal{R}^{(1)}:\omega^2_{p_1:128}} : \mathcal{R}^{(1)} \cong \prod_{i=0}^{63} \mathcal{R}_i^{(1)}$.

Together, we have $\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}$:

$$\mathcal{R} \cong \prod_{i=0}^{63} \mathcal{R}_i.$$

Figure 8.1 is an illustration of the isomorphisms.

doi:10.6342/NTU202200738

Figure 8.1: Split of polynomial rings with $\mathsf{CRT}$for incomplete NTTs for Saber. Blue arrows are isomorphisms via NTT. The red arrows are isomorphisms via CRT with $\omega_{p_0p_1:128} = \mathsf{CRT}(\omega_{p_0:128}, \omega_{p_1:128})$.

$$\begin{tikzcd}
\mathcal{R} \arrow[rr, leftrightarrow, "\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}"] \arrow[d, leftrightarrow, "\mathsf{CRT}"'] & & \prod_{i=0}^{63} \mathcal{R}_i \arrow[d, leftrightarrow, "\mathsf{CRT}"'] \\
\mathcal{R}^{(0)} \times \mathcal{R}^{(1)} \arrow[rr, leftrightarrow, "\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega^2_{p_0:128}} \times \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega^2_{p_1:128}}"] & & \prod_{i=0}^{63} \left(\mathcal{R}_i^{(0)} \times \mathcal{R}_i^{(1)}\right)
\end{tikzcd}$$

Instead of implementing $\boldsymbol{a}(x)\boldsymbol{b}(x)$ in $\mathcal{R}$ as applying $\mathsf{NTT}^{-1}_{\mathcal{R}:\omega^2_{p_0p_1:128}}$ on the result of

$$\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}\left(\boldsymbol{a}(x)\right) \cdot_{\prod \mathcal{R}_i} \mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}\left(\boldsymbol{b}(x)\right),$$

for saving memory, we apply $\mathsf{NTT}^{-1}_{\mathcal{R}:\omega^2_{p_0p_1:128}}$ on the $\mathsf{CRT}$of

$$\begin{cases} \left(\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x)) \bmod p_0\right) \cdot_{\prod \mathcal{R}_i^{(0)}} \mathsf{NTT}_{\omega^2_{p_0:128}}(\boldsymbol{b}(x)) \\ \left(\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x)) \bmod p_1\right) \cdot_{\prod \mathcal{R}_i^{(1)}} \mathsf{NTT}_{\omega^2_{p_1:128}}(\boldsymbol{b}(x)). \end{cases}$$

The workflow is outlined in Algorithm 38. We declare 16-bit arrays in the order

$$\begin{cases} \texttt{buff1\_16[256]} \\ \texttt{buff2\_16[256]} \\ \texttt{buff3\_16[256]} \end{cases}$$

and 32-bit pointers

$$\begin{cases} {}^*\texttt{buff1\_32} = \texttt{(uint32\_t}^*\texttt{)buff1\_16} \\ {}^*\texttt{buff2\_32} = \texttt{(uint32\_t}^*\texttt{)buff2\_16} \end{cases}$$

for accessing the memory as 32-bit arrays at some point.

First, we compute $\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x))$ and store the result to the 32-bit array `buff1_32`. We then compute and put `buff1_32` mod p_1 in the 16-bit array `buff3_16`. For computing `buff1_32` mod p_0, we see that the result in `buff1_32` won't be needed after reducing to modp_0, so we compute and put `buff1_32` mod p_0 in the 16-bit array `buff1_16`. This is doable if we reduce to $\mathbb{Z}_{p_0}$ from the beginning. We proceed with computing $\mathsf{NTT}_{\mathcal{R}^{(1)}:\omega^2_{p_1:128}}(\boldsymbol{b}(x))$ in the 16-bit array `buff2_16`, $\cdot_{\prod \mathcal{R}_i^{(1)}}$ to `buff3_16`, $\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega^2_{p_0:128}}(\boldsymbol{b}(x))$ in the 16-bit array `buff2_16`, and $\cdot_{\prod \mathcal{R}_i^{(0)}}$ to `buff2_16`. Next we compute the CRT, *giving 32-bit coefficients as in the NTT domain with the coefficient ring* $\mathbb{Z}_{p_0p_1}$, and put the result in the 32-bit array `buff1_32`. Finally, we compute $\mathsf{NTT}^{-1}_{\mathcal{R}:\omega^2_{p_0p_1:128}}$ and reduce the coefficient ring to $\mathbb{Z}_q$. Algorithm 38 is an illustration.

Application to `MatrixVectorMul`. For stack-optimized `MatrixVectorMul` in encapsulation of unmasked Saber, we employ a variant of Strategy D. We first declare an array

`uint16_t acc_16[256]`

doi:10.6342/NTU202200738

Algorithm 38 16-bit `polymul` using 1 536 bytes of memory [ACC+22].

```
Declare arrays uint16_t buff1_16[256], buff2_16[256], buff3_16[256]
Declare pointers { uint32_t *buff1_32 = (uint32_t*)buff1_16
                 { uint32_t *buff2_32 = (uint32_t*)buff1_16
buff1_32[0-255] = $\mathsf{NTT}_{\mathcal{R}:\omega^2_{p_0p_1}:128}$(src1[0-255])
buff3_16[0-255] = buff1_32[0-255] mod $p_1$
buff1_16[0-255] = buff1_32[0-255] mod $p_0$
buff2_16[0-255] = $\mathsf{NTT}_{\mathcal{R}^{(1)}:\omega^2_{p_1}:128}$(src2[0-255])
buff3_16[0-255] = buff3_16[0-255] $\cdot_{\prod \mathcal{R}^{(1)}_i}$ buff2_16[0-255]
buff2_16[0-255] = $\mathsf{NTT}_{\mathcal{R}^{(0)}:\omega^2_{p_0}:128}$(src2[0-255])
buff2_16[0-255] = buff1_16[0-255] $\cdot_{\prod \mathcal{R}^{(0)}_i}$ buff2_16[0-255]
buff1_32[0-255] = $\mathsf{CRT}$(buff2_16[0-255], buff3_16[0-255])
des[0-255] = $\mathsf{NTT}^{-1}_{\mathcal{R}:\omega^2_{p_0p_1}:128}$(buff1_32[0-255]) mod $q$
```

and multiply an element of A by an element of s' with the above strategy. We then accumulate the result to `acc_16` and derive an element of b'. In total, only 1536 bytes are needed if the accumulator is excluded.

Comparison with previous stack-optimized implementation. We compare the memory usage of polynomial multiplication to the currently most stack-optimized implementation – 4 levels of memory efficient Karatsuba [MKV20]. Ignoring the extra $O(\log n)$ memory overhead for Karatsuba, we focus on the buffers for the multiplicands and the result. For the Karatsuba approach, one needs 512 bytes for the accumulator, 512 bytes for holding a component of A, and 1022 bytes for the degree-510 result – almost the same as the NTT approach with composite modulus. Essentially, *any algorithm not exploiting the negacyclic property requires such amount of memory.* We only find the work by [IKPC20] giving a non-NTT-based approach exploiting the negacyclic property, but the authors reported that they were not able to achieve a smaller footprint than the Karatsuba by [MKV20].

8.6 Cortex-M3:Saber

8.6.1 Saber

Due to the more limited instruction set and the early terminating long multiplications on the Cortex-M3, the 32-bit butterflies from the previous section can only be used with some restrictions. In general, there are two approaches to still benefit from NTTs on the Cortex-M3: One can either implement 32-bit NTTs, but avoid the early terminating multiplication instructions for secret inputs, or one exclusively uses 16-bit NTTs and computes the CRTof the results. The former approach resembles the Cortex-M4 approach from [CHK+21] and the previous section, while the latter is similar to the AVX2 implementation from [CHK+21]. This thesis reports the 16-bit approach by [ACC+22].

doi:10.6342/NTU202200738

16-bit NTTs for `MatrixVectorMul`. We implement strategies A, C, and D with the 16-bit NTT approach for `MatrixVectorMul` on Cortex-M3. For strategy A, this corresponds to the AVX2 implementation from [CHK+21]. We also carry out the stack optimization on Cortex-M4 and implement strategies C and D.

8.7 Cortex-A72:Dilithium, Kyber, and Saber

This section introduces the implementations of Dilithium, Kyber, and Saber on the Cortex-A72. This thesis reports the work by [BHK+22].

Section 8.7.1 summarizes the implementations of `MatrixVectorMul` in Dilithium, Kyber, and Saber. We first discuss the potential permutation issues in the NTTs in Section 8.7.2. Then, we illustrate the implementations of NTTs of Saber in Section 8.7.3, Dilithium in Section 8.7.4, and Kyber in Section 8.7.5.

8.7.1 `MatrixVectorMul` in Dilithium, Kyber, and Saber

As discussed in Section 8.3, Dilithium and Kyber write certain NTTs into the specification while Saber only requires the computation of `MatrixVectorMul` defined by the arithmetic of $\mathbb{Z}_q[x]/\langle x^{256}+1\rangle$.

For Dilithium, we apply 8-layer 32-bit NTTs for computing `MatrixVectorMul` and compute 1×1 basemulwith the improved accumulation introduced by [CHK+21]. For Kyber, we apply 7-layer 16-bit NTTs and compute 2×2 `asymmetric_mul` with improved accumulation. Finally, we apply 6-layer 32-bit NTTs and compute 4×4 `asymmetric_mul` wiht improved accumulation for Saber.

8.7.2 The Need for Permutations in the NTTs

For vectorized implementations of NTTs, an important consideration is the overhead of permuting coefficients within SIMD registers. This is required when the distance between the inputs of a butterfly is less than the size of a SIMD register. Since each SIMD register in Neon holds 128 bits (cf. Section 5.2.2), any butterfly taking inputs at distance larger than 128 bits doesn't require any permutation.

After each layer, the distance between the butterfly inputs is halved. For Dilithium and Saber, we compute 32-bit NTTs for size-256 polynomials. Initially, we compute butterflies with inputs distancing by $32\cdot 128 = 4096$ bits. At the kth layer, we compute butterflies with inputs distancing by $32\cdot 2^{7-k} = \frac{4096}{2^k}$ bits. Therefore, after the 5th layer, any follow-up butterflies require permutations. As for Kyber with 16-bit NTTs for size-256 polynomials, it is after the 4th layer that one needs permutations for the follow-up butterflies.

8.7.3 NTTs in Saber

First of all, since Armv8-A provides instructions for both 16-bit and 32-bit arithmetic, we found no reasons to employ 16-bit NTTs as in [NG21]. We implement the 32-bit NTT as implemented for Cortex-M4 in [ACC+22, CHK+21]. Both of our $\mathsf{NTT}$and NTT^{-1} are implemented with two levels of interleaved 3-layer CT butterflies. Since a 128-bit SIMD register contains four 32-bit elements, there is no need for permuting the coefficients in $\mathsf{NTT}$and NTT^{-1}.

doi:10.6342/NTU202200738

8.7.4 NTTs in Dilithium

An 8-layer NTT defined over the ring $\mathbb{Z}_q[x]/\langle x^{256} - \zeta^{256}\rangle$ is required for Dilithium where $\zeta^{256} = -1$ and $512|\mathbf{O}(q)$. We implement NTTwith CT butterflies and NTT^{-1} with GS butterflies for Dilithium. For NTT, after four layers of radix-2 butterflies, the distance of the butterfly inputs for the next layer is 256 bits.

For the next 4 layers of NTTs, we first load with `ld1` and apply two layers of radix-2 splits on four SIMD registers as usual. Next, we transpose the four registers with Algorithm 5. At the end of butterflies, we store with `st4` to cancel out the transpose (cf. Figure 5.7). For the NTT^{-1}, we invert the entire process: for the initial 4 layers of NTT^{-1}, we load with `ld4`, compute two layers with GS, invert Algorithm 5, compute two layers with GS, and store with `st1`. For the last 4 layers, we invert CT butterflies with GS butterflies. One must note that the layer of `trn1` and `trn2` forbids the 4-layer interleaving of butterflies. Currently, there is no known solutions to overcome this.

8.7.5 NTTs in Kyber

In Kyber, a 7-layer NTT defined over the ring $\mathbb{Z}_q[x]/\langle x^{256} - \zeta^{128}\rangle$ is required where $\zeta^{128} = -1$ and $256|\mathbf{O}(q)$. We implement NTTwith CT butterflies and NTT^{-1} with GS butterflies for Kyber. For the NTT, after four layers of radix-2 butterflies, the distance of the butterfly inputs for the next layer is 128 bits.

At this point, we could choose to compute a layer and permute at the 4th layer. As mentioned in the previous section, this forbids the multi-layer interleaving of butterflies. Instead, we proceed for the bottom three layers with a combination of `ld4`, `trn1`, and `trn2`. First, we `ld4` each 4-byte to eight SIMD registers. Next, we use `trn1` and `trn2` to separate the upper 32-bit from the lower 32-bit for butterflies as shown in Algorithm 62. In this way, we can then apply three layers of butterflies without intermediate permutations, which prohibits an aggressive interleaving of instructions. For NTT^{-1}, we invert the entire NTT.

Table 8.4 summarizes the butterflies for NTTand NTT^{-1}.

Table 8.4: Summary of butterflies for NTTs and NTT^{-1}s on Cortex-A72.

	NTT	NTT^{-1}
Kyber	4-layer-CT + 3-layer-CT	3-layer-GS + 4-layer-GS
Saber	2× 3-layer-CT	2× 3-layer-CT
Dilithium	2× 4-layer-CT	2× 4-layer-GS

doi:10.6342/NTU202200738

Chapter 9

Results

This chapter presents the results based on the following works.

- Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127-151, 2022. `https://tches.iacr.org/index.php/TCHES/article/view/9292`. Reference [ACC+22]. Full version available at `https://eprint.iacr.org/2021/995`.

- Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221-224, 2022. `https://tches.iacr.org/index.php/TCHES/article/view/9295`. Reference [BHK+22]. Full version available at `https://eprint.iacr.org/2021/986`.

Furthermore, Sections 9.1.2, 9.1.3, and 9.1.4 are contributions of the submitted work:

- Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. Submitted in April 2022.

Table 9.1 summarizes the results reported in this thesis. For brevity, we abbreviate key generation as **K**, encapsulation as **E**, and decapsulation as **D** while presenting the overall performance numbers.

doi:10.6342/NTU202200738

Table 9.1: Overview of the results.

Parameter	Cortex-M3	Cortex-M4	Cortex-A72
		Dilithium	
dilithium2	-	-	9.3
dilithium3	-	-	9.3
dilithium5	-	-	9.3
		Kyber	
kyber512	-	-	9.3
kyber768	-	-	9.3
kyber1024	-	-	9.3
		NTRU	
ntruhps2048509	-	-	-
ntruhps2048677	-	9.1.2 and 9.1.3	-
ntruhrss701	-	9.1.2 and 9.1.3	-
ntruhps4096821	-	9.1.2 and 9.1.3	-
ntruhps40961229	-	-	-
ntruhrss1373	-	-	-
		NTRU Prime	
ntrulpr653/sntrup653	-	9.1.2 and 9.1.4	-
ntrulpr761/sntrup761	-	9.1.2 and 9.1.4	-
ntrulpr857/sntrup857	-	9.1.2 and 9.1.4	-
ntrulpr953/sntrup953	-	-	-
ntrulpr1013/sntrup1013	-	9.1.2 and 9.1.4	-
ntrulpr1277/sntrup1277	-	9.1.2 and 9.1.4	-
		Saber	
lightsaber	9.2.2 and 9.2.3	9.1.5 and 9.1.6	9.3
saber	9.2.2 and 9.2.3	9.1.5 and 9.1.6	9.3
firesaber	9.2.2 and 9.2.3	9.1.5 and 9.1.6	9.3

9.1 Cortex-M4 Results

We present the performance numbers of Cortex-M4 implementations in this Section. We present the performance numbers of polynomial multiplications in NTRU and NTRU Prime in Section 9.1.2. The overall performance of NTRU is shown in Section 9.1.3 and the overall performance of NTRU Prime is shown in Section 9.1.4. Next, we present the performance numbers of `MatrixVectorMul` and `InnerProd` in Saber in Section 9.1.5. The overall performance of Saber in shown in Section 9.1.6.

doi:10.6342/NTU202200738

9.1.1 Benchmark Environment

We target the board STM32F407-DISCOVERY featuring an STM32F407VG Cortex-M4 with 196 kB of SRAM and 1 MB of flash. For benchmarking individual functions, we clock at both the loweset frequency 24 MHz and the highest frequency 168 MHz. For benchmarking the entire scheme, we follow the benchmarking setup by [KRSS] which clocks at 24 MHz. One can benchmark at 168 MHz, but a large portion of optimizations that are not described in this thesis do not have code-size-optimized implementations. The benchmark at 168 MHz for the schemes are meaningless.

9.1.2 Polynomial Multiplications in NTRU and NTRU Prime

We first recall the parameters related to polynomial multiplications in NTRU and NTRU Prime in Table 9.2.

Table 9.2: Parameters of NTRU and NTRU Prime.

NTRU			NTRU Prime		
Ring $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$			Field $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$		
Parameter	n	q	Parameter	p	q
ntruhps2048509	509	2048	ntrulpr653/sntrup653	653	4621
ntruhps2048677	677	2048	ntrulpr761/sntrup761	761	4591
ntruhrss701	701	8192	ntrulpr857/sntrup857	857	5167
ntruhps4096821	821	4096	ntrulpr953/sntrup953	953	6343
ntruhps40961229	1229	4096	ntrulpr1013/sntrup1013	1013	7177
ntruhps1373	1373	8192	ntrulpr1277/sntrup1277	1277	7879

Detailed numbers of polynomial multiplications. Table 9.3 shows the detailed performance numbers polynomial multiplications in NTRU and NTRU Prime. There are two rows for each implementations. The first row is benchmarked at 24 MHz and the second row is benchmarked at 168 MHz. Each of our convolutions supports at least one parameter set of NTRU and one parameter set of NTRU Prime. Our size-1440 convolution supports ntruhps2048677, ntruhrss701, and ntrulpr653/sntrup653. The implementations are exactly the same except for the final_maps, which are tuned with the reduction to target polynomial rings. Our size-1536 convolution supports ntruhps2048677, ntruhrss701, and ntrulpr761/sntrup761 and our size-1728 convolution supports ntruhps4096821, and ntrulpr857/sntrup857.

Next, we compare polynomial multiplications in this thesis to existing works. Table 9.4 is a summary. We present numbers at both 24 MHz and 168 MHz while existing works reported numbers at 24 MHz. After carefully examining the implementations by existing works, programs by [Che21, ACC$^+$21, CHK$^+$21] have compact code size while programs by [IKPC22] suffer from large code size. This implies after raising the frequency to 168 MHz, the cyclces of programs by [IKPC22] will increase drastically.

doi:10.6342/NTU202200738

Table 9.3: Performance of polymuls in NTRU and NTRU Prime on Cortex-M4.

NTRU							
(n, q)	Size	polymul	NTT	NTT_small	basemul	iNTT	final_map
$(677, 2048)$	1440	140 444	34 102	33 241	27 690	36 756	8 835
		143 016	34 963	34 093	27 825	37 214	9 208
$(677, 2048)$	1536	147 126	37 485	36 573	23 322	41 437	8 489
		149 174	38 076	37 139	23 506	42 001	8 717
$(701, 8192)$	1440	140 577	34 102	33 241	27 690	36 756	8 968
		143 239	34 957	34 087	27 819	37 208	9 431
$(701, 8192)$	1536	147 670	37 485	36 573	23 322	41 437	9 033
		149 771	38 076	37 139	23 506	42 001	9 314
$(821, 4096)$	1728	181 534	48 629	47 627	21 848	53 098	10 512
		186 197	49 480	48 507	22 349	55 569	10 564
NTRU Prime							
(p, q)	Size	polymul	NTT	NTT_small	basemul	iNTT	final_map
$(653, 4621)$	1440	142 244	34 104	33 244	27 690	36 756	10 629
		146 665	34 992	34 095	27 813	37 214	12 823
$(761, 4591)$	1536	151 374	37 487	36 573	23 322	41 435	12 739
		153 299	38 069	37 138	23 510	42 001	12 861
$(857, 5167)$	1728	184 714	48 629	47 623	21 848	53 099	13 695
		189 523	49 483	48 499	22 336	55 720	13 743
$(1013, 7177)$	2048	223 968	57 430	55 383	32 858	139 838	17 028
		227 308	58 406	56 345	33 248	141 911	17 156
$(1277, 7879)$	2560	284 895	73 460	72 592	36 435	81 040	21 542
		289 903	75 315	74 358	36 509	82 101	21 846

Comparing polynomial multiplications in NTRU. For ntruhps2048677, our size-1440 convolution outperforms the size-677 convolution from [IKPC22] by 4.2% and the size-1536 convolution from [CHK+21] by 2.8%. Our size-1536 convolution is slower than [IKPC22] but still faster than [CHK+21]. For ntruhrss701, we have similar results. Next, we compare ntruhps4096821. Our size-1728 convolution outperforms the size-821 convolution from [IKPC22] by 7.8% and the size-1728 convolution from [CHK+21] by 10.6%. Notice that when benchmarking at full speed (168 MHz), we only pay around 2% additional cycles. After carefully examining the implementations by [IKPC22], their implementations are fully unrolled. Although we believe that their implementations can be made much more compact with some care, our implementations at 168 MHz are already faster than their implementations at 24 MHz. This implies for practical deployment, users have a much wider range of frequency to fit the implementations into their use without bothering the sacrifice of performance. Additionally, no algebraic properties are exploited in our implementations, while [IKPC22] only applies to (weighted) convolutions. Therefore, with little

doi:10.6342/NTU202200738

modifications, our implementations support polynomial multiplications in NTRU Prime.

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

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

Table 9.4: Comparisons of `polymul` in NTRU and NTRU Prime on Cortex-M4.

NTRU				
(n, q)	Convolution	This work	[CHK+21]	[IKPC22]
$(677, 2048)$	Size-677	−/−	−/−	144k/−
	Size-1440	140k/143k	−/−	−/−
	Size-1536	147k/149k	156k/−	−/−
$(701, 8192)$	Size-701	−/−	−/−	144k/−
	Size-1440	141k/143k	−/−	−/−
	Size-1536	148k/150k	156k/−	−/−
$(821, 4096)$	Size-821	−/−	−/−	193k/−
	Size-1728	178k/182k	199k/−	−/−
NTRU prime				
(p, q)	Convolution	This work	[ACC+21]	[Che21][1]
$(653, 4621)$	Size-1320	−/−	−/−	120k/−
	Size-1440	142k/147k	−/−	−/−
$(761, 4591)$	Size-1530	−/−	152k/−	142k/−
	Size-1536	151k/153k	159k/−	−/−
	Size-1620	−/−	185k/−	−/−
$(857, 5167)$	Size-1722	−/−	−/−	203k/−
	Size-1728	182k/186k	−/−	−/−
$(1013, 7177)$	Size-2048	224k/227k	−/−	−/−
$(1277, 7879)$	Size-2560	285k/290k	−/−	−/−

[1] Method 2.

doi:10.6342/NTU202200738

9.1.3 Performance of NTRU

We compare the overall performance of NTRU to existing works. Fastest approaches in this thesis are selected for comparison. 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 9.5.

We first compare the encapsulations. For `ntruhps2048677` and `ntruhps4096821`, we outperform [IKPC22] by $36.0 - 36.6\%$. 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 2.3% for `ntruhrss701`.

For decapsulations, the differences between our implementations and [IKPC22] are one big by small polynomial multiplication and one polynomial multiplication in $\mathbb{Z}_3$. We outperform [IKPC22] by $1.3 - 2\%$.

Our NTT-based multiplications have limited impact to key generations. 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 the work by [Li21] and [IKPC22], 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 9.5: Performance of NTRU on Cortex-M4.

Parameter		[CHK+21]	[IKPC22]	[Li21][2]	This thesis
`ntruhps2048677`	**K**	143 725k	142 378k	4 625k	3 906k
	E	821k	816k	820k	523k
	D	818k	729k	812k	714k
`ntruhrss701`	**K**	153 403k	153 479k	4 233k	3 816k
	E	377k	369k	376k	359k
	D	871k	787k	868k	774k
`ntruhps4096821`	**K**	207 495k	212 377k	6 116k	5 208k
	E	1 027k	1 026k	1 027k	651k
	D	1 030k	914k	1 031k	902k

9.1.4 Performance of NTRU Prime

We compare the overall performance of NTRU Prime to existing works. We replace AES with secret-dependent input by the fixslicing AES by [AP21]. This increases

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

doi:10.6342/NTU202200738

the overall performance numbers of NTRU LPRime. Furthermore, we improve the `crypto_sort`. Table 9.6 summarizes the overall performance numbers.

Table 9.6: Performance of NTRU Prime on Cortex-M4.

Parameter		[ACC$^+$21][3]	[Che21][4]	This thesis
`ntrulpr653`	**K**	-	678k	667k
	E	-	1 158k	1 127k
	D	-	1 233k	1 226k
`sntrup653`	**K**	-	6 715k	6 673k
	E	-	632k	619k
	D	-	487k	522k
`ntrulpr761`	**K**	731k	727k	710k
	E	1 102k	1 312k	1 266k
	D	1 200k	1 394k	1 365k
`sntrup761`	**K**	10 778k	7 951k	7 937k
	E	694k	684k	666k
	D	572k	538k	563k
`ntrulpr857`	**K**	-	-	882k
	E	-	-	1 460k
	D	-	-	1 588k
`sntrup857`	**K**	-	-	10 189k
	E	-	-	809k
	D	-	-	679k
`ntrulpr1013`	**K**	-	-	1 059k
	E	-	-	1 742k
	D	-	-	1 899k
`sntrup1013`	**K**	-	-	13 841k
	E	-	-	981k
	D	-	-	827k
`ntrulpr1277`	**K**	-	-	1 360k
	E	-	-	2 207k
	D	-	-	2 401k
`sntrup1277`	**K**	-	-	22 756k
	E	-	-	1 253k
	D	-	-	1 058k

[3] Secret-dependant table lookup AES for `ntrulpr761`, see `https://github.com/mupq/pqm4/pull/173` for details.

[4] The method 2 integrated into `pqm4` in commit `844e7cafdb5df8416abef3c03b49edb810b7e396`.

doi:10.6342/NTU202200738

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

Next, we compare Streamlined NTRU Prime. In Streamlined NTRU Prime, we need crypto_sort in key generations and encapsulations. Since we optimize the crypto_sort, our key generations are faster than [Che21] even though our polynomial multiplications are slower than [Che21]. For decapsulations, the only differences between [Che21] and our work are two big by small polynomial multiplications. This explains why our decapsulations are slower.

Finally, we present the performance numbers of ntrulpr857, sntrup857, ntrulpr953, sntrup953, ntrulpr1013, sntrup1013, ntrulpr1277, and sntrup1277.

9.1.5 MatrixVectorMul and InnerProd in Saber

We present the performance numbers of NTT-related functions implementing the core operations MatrixVectorMul and InnerProd in Saber. Table 9.7 summarizes the performance of NTT-based polynomial multiplication and Table 9.8 summarizes the performance of MatrixVectorMul and InnerProd with various strategies trading speed for stack usage. The implementations are adapted from [ACC+22] for range issues.

Polynomial multiplications in Saber. We present the numbers of speed-optimized and stack-optimized polynomial multiplications in Saber. The speed-optimized polynomial multiplications consists of two 32-bit NTTs, one basemul, and one 32-bit NTT^{-1}. The stack-optimized polynomial multiplications consists of one 32-bit NTT, one 16-bit NTTover 7681, one 16-bit NTTover 3329, one reduction of a 32-bit polynomial to a 16-bit polynomial, one 16-bit basemul, one computation merging reduction to a 16-bit polynomial and 16-bit basemul, one CRT, and one 32-bit NTT^{-1}. Table 9.7 summarizes the number of each operations and the performance numbers.

MatrixVectorMul and InnerProd in Saber. Table 9.8 summarizes the performance of MatrixVectorMul and InnerProd with strategies A, B, C, and D.

doi:10.6342/NTU202200738

Table 9.7: The operation counts and performance of `polymul` in Saber on Cortex-M4.

Operation	Performance		Operation Count	
	24 MHz	168 MHz	`polymul` (speed)	`polymul` (stack)
32-bit NTT	5 855	6 118	2	1
16-bit NTT(mod7681)	4 918	5 163	0	1
16-bit NTT(mod3329)	4 470	4 707	0	1
32-bit basemul	4 186	4 300	1	0
32-bit to 16-bit	1 181	1 250	0	1
16-bit basemul	2 966	3 051	0	1
32 × 16-bit basemul	3 734	3 831	0	1
CRT	2 439	2 521	0	1
32-bit NTT^{-1}	7 318	7 661	1	1
Overall performance				
Frequency			`polymul` (speed)	`polymul` (stack)
24 MHz			23 077	32 555
168 MHz			23 969	33 857

doi:10.6342/NTU202200738

Table 9.8: Performance of MatrixVectorMul and InnerProd in Saber on Cortex-M4.

	lightsaber	saber	firesaber
MatrixVectorMul (Enc, A)	67 624	133 587	221 006
	70 172	138 386	228 741
MatrixVectorMul (Enc, B)	81 844	176 277	306 387
	85 109	183 335	318 609
MatrixVectorMul (Enc, C)	79 225	168 417	290 681
	82 177	174 556	301 200
MatrixVectorMul (Enc, D)	131 373	296 370	527 770
	136 671	308 516	549 413
InnerProd (Enc, A)	28 009	38 736	49 456
	29 046	40 156	51 165
InnerProd (Dec, A)	39 621	56 172	72 708
	41 119	58 311	75 404
InnerProd (Dec, B)	46 728	70 397	94 049
	48 591	73 288	97 855
InnerProd (Dec, C)	39 630	56 170	72 706
	41 132	58 301	75 354
InnerProd (Dec, D)	65 698	98 847	131 993
	68 389	102 920	137 372

9.1.6 Performance of Saber

We present the performance numbers of Saber. Table 9.9 summarizes seven implementations. In the literature, [CHK+21] implemented the 32-bit NTT-based multiplication, [IKPC20] implemented the Toeplitz matrix-vector product approach, [BMK+22] implemented the striding Toom–Cook approach, and [MKV20] implemented Toom–Cook as the speed-optimized implementation and Karatsuba as the stack-optimized implementation.

Table 9.9 shows that NTT-based multiplications are superior than non-NTT-based approaches. The fastest implementation is the strategy A in this thesis and the most stack-optimized implementation is the strategy C in this thesis. For strategy A, it caches the results of NTTs by exploiting the structure of MatrixVectorMul as explained in Section 8.3. For strategy D, it consume a small amount of memory by exploiting the negacyclic property of the polynomial ring in Saber. Due to the powerful 32-bit multipliers on Cortex-M4, strategy D also outperforms all non-NTT-based implementations in terms of cycle count.

doi:10.6342/NTU202200738

Table 9.9: Performance of Saber on Cortex-M4.

		lightsaber		saber		firesaber	
		Cycle	Stack	Cycle	Stack	Cycle	Stack
This thesis (speed, A)	K	**352k**	5 612	**644k**	6 644	**989k**	7 668
	E	**480k**	6 276	**819k**	7 316	**1 199k**	8 340
	D	**451k**	6 284	**772k**	7 324	**1 144k**	8 348
[CHK+21]	K	360k	14 604	658k	23 284	1 008k	37 116
	E	513k	16 252	864k	32 620	1 255k	40 484
	D	498k	16 996	835k	33 824	1 227k	41 964
This thesis (stack, D)	K	424k	**3 276**	823k	**3 788**	1 326k	**4 300**
	E	594k	**3 052**	1 063k	**3 180**	1 624k	**3 316**
	D	584k	**3 060**	1 043k	**3 188**	1 605k	**3 324**
[IKPC20]	K	440k	7 956	825k	12 616	1 319k	20 144
	E	615k	9 684	1 060k	14 896	1 621k	22 992
	D	622k	10 428	1 073k	15 992	1 649k	24 472
[BMK+22]	K	433k	4 296	840k	4 272	1 350k	4 280
	E	607k	4 808	1 086k	4 784	1 654k	4 792
	D	612k	5 328	1 091k	5 296	1 674k	5 304
[MKV20] (speed)	K	466k	14 208	853k	19 824	1 340k	26 448
	E	653k	15 928	1 103k	22 088	1 642k	29 228
	D	678k	16 672	1 127k	23 184	1 679k	30 768
[MKV20] (stack)	K	612k	3 564	1 230k	4 348	2 046k	5 116
	E	880k	3 148	1 616k	3 412	2 538k	3 668
	D	976k	3 164	1 759k	3 420	2 740k	3 684

9.2 Cortex-M3 Results

We present the performance numbers of Cortex-M3 implementations in this Section. The implementations are adapted from [ACC+22] for range issues.

9.2.1 Benchmark Environment

We target the board `NUCLEO-F207ZG` featuring an `STM32F207ZG` Cortex-M3 core with 128 kB of SRAM and 1 MB of flash. We clock at both 30 MHz and 120 MHz for benchmarking individual functions. We clock at 30 MHz for benchmarking the schemes.

9.2.2 `MatrixVectorMul` and `InnerProd` in Saber

Table 9.10 summarizes the detailed numbers of `polymul` and Table 9.11 summarizes the performance of `MatrixVectorMul` and `InnerProd` with various strategies.

doi:10.6342/NTU202200738

Table 9.10: Performance of `polymul` in Saber on Cortex-M3.

Frequency	`polymul`	NTT	basemul	NTT^{-1}	CRT
30 MHz	70 089	8 688	5 987	9 553	4 639
120 MHz	70 618	8 775	6 035	9 679	4 656

Table 9.11: Performance of `MatrixVectorMul` and `InnerProd` in Saber on Cortex-M3.

	`lightsaber`	`saber`	`firesaber`
`MatrixVectorMul` (Enc, A)	204 452	407 020	635 269
	205 940	409 710	639 462
`MatrixVectorMul` (Enc, B)	249 100	537 685	904 660
	251 113	542 077	912 688
`MatrixVectorMul` (Enc, C)	239 013	510 864	832 063
	240 680	514 063	837 424
`MatrixVectorMul` (Enc, D)	283 649	641 482	1 097 587
	285 857	646 297	1 106 774
`InnerProd` (Enc, A)	84 938	117 071	143 856
	85 586	117 863	144 810
`InnerProd` (Dec, A)	119 488	168 931	210 355
	120 330	170 043	211 701
`InnerProd` (Dec, B)	141 837	213 624	280 127
	142 960	215 277	282 419
`InnerProd` (Dec, C)	119 499	168 902	210 352
	120 331	169 991	211 692
`InnerProd` (Dec, D)	141 835	213 579	280 071
	142 997	215 303	282 335

9.2.3 Performance of Saber

Table 9.12 compares the overall performance of Saber. Strategy A is the fastest approach and strategy D consumes the least amount of memory.

doi:10.6342/NTU202200738

Table 9.12: Performance of Saber on Cortex-M3.

		lightsaber		saber		firesaber	
		Cycle	Stack	Cycle	Stack	Cycle	Stack
This thesis (speed, A)	**K**	**526k**	5 756	**973k**	6 780	**1 519k**	7 804
	E	**714k**	6 428	**1233k**	7 460	**1 839k**	8 484
	D	**758k**	6 428	**1288k**	7 460	**1 916k**	8 484
This thesis (C)	**K**	570k	4 724	1 106k	5 236	1 789k	5 748
	E	795k	3 716	1 412k	3 844	2 152k	3 972
	D	842k	3 716	1 469k	3 844	2 233k	3 972
This thesis (stack, D)	**K**	618k	**3 412**	1 239k	**3 924**	2 042k	**4 436**
	E	868k	**3 204**	1 603k	**3 332**	2 508k	**3 460**
	D	913k	**3 340**	1 657k	**3 340**	2 585k	**3 460**
pqm3 (Toom–Cook)	**K**	710k	9 652	1 328k	13 252	2 171k	20 116
	E	967k	11 372	1 738k	15 516	2 688k	22 964
	D	1 081k	12 116	1 902k	16 612	2 933k	24 444

9.3 Cortex-A72 Results

We present the performance numbers of Cortex-A72 implementations in this Section.

9.3.1 Benchmark Environment

Cortex-A72 implements the Armv8.0-A architecture (cf. Section 5.2) and has a triple-issue out-of-order pipeline (cf. Section 6.3). Specifically, we use the Raspberry Pi 4 Model B featuring the quad-core Broadcom BCM2711 chipset. It comes with a 32 kB L1 data cache, a 48 kB L1 instruction cache, and a 1 MB L2 cache and runs at 1.5 GHz. For hashing, we use the parallelized SHA-3/SHAKE Neon implementation by [NG21]. For benchmarking individual functions, we make use of the cycle counter of the PMU. For benchmarking the full cryptographic schemes, we use SUPERCOP. We use gcc version 10.3.0 with -O3. For individual functions, we report the cycle count of median of 10 000 executions. For the schemes, we report the median cycle count of 10 000 executions for Kyber and Saber, and 100 000 executions for Dilithium.

9.3.2 Benchmark of NTT-Related Functions

Table 9.13 summarizes our results of NTT-related functions in Dilithium, Kyber, and Saber. dim refers to the module dimension. Some implementations implement the basemulfor each polynomial separately, while we implement it for an entire vector so we can optimize the accumulation (cf. Section 8.3.2).

For kyber768, we outperform the NTTand NTT^{-1} by [NG21] by 19% each, even though we include the Barrett reduction which is not reported by [NG21]. For saber, our 32-bit NTTis 23% faster than the 16-bit NTTby [NG21]. For Dilithium, we obtain a vast speed-up over the reference implementation.

doi:10.6342/NTU202200738

Table 9.13: Performance of `NTT`, $\mathsf{NTT}_{\mathsf{heavy}}$, $\mathsf{basemul}$, and NTT^{-1} in `kyber768`, `saber`, and `dilithium3` on Cortex-A72. The performance of `dim` × `base_mul` and CRTby [NG21] are our own benchmarks.

	NTT	$\mathsf{NTT}_{\mathsf{heavy}}$	dim × basemul	NTT^{-1}	CRT
`kyber768` [BHK+22]	1 200	1 434	952	1 338	–
`kyber768` [NG21]	1 473[5]	–	3 040	1 661	–
`saber` 32-bit [BHK+22]	1 529	2 031	2 689	1 896	–
`saber` 16-bit [NG21]	1 991	–	1 500	1 893	813
`dilithium3` [BHK+22]	2 241	–	1 378	2 821	–
`dilithium3` (ref)	9 302	–	5 × 2 325	11 633	–

9.3.3 `MatrixVectorMul` and `InnerProd` in Dilithium, Kyber, and Saber

For Kyber and Saber, in encapsulation (Enc), `InnerProd` reuse the intermediate results from the `MatrixVectorMul` while in decapsulation (Dec), a full `InnerProd` is needed. Dilithium does not use an `InnerProd`.

Table 9.14 presents the results for the core arithmetic operations `MatrixVectorMul` and `InnerProd` for all parameter sets. For Kyber, our `MatrixVectorMul` is 1.56 – 1.74× faster than the previous implementations [NG21]. For `InnerProd` (Dec), our code is 1.47–1.52× faster. For Saber, we speed up `MatrixVectorMul` by 2.00–2.03×. The speed-up for `InnerProd` (Dec) is 1.62–1.63×. Also note that for the `InnerProd` in encapsulation of Kyber and Saber, one can re-use $\mathsf{NTT}_{\mathsf{heavy}}(s)$ to obtain a much faster `InnerProd` (Enc).

9.3.4 Performance of Dilithium, Kyber, and Saber

Table 9.15 summarizes the overall performance of Dilithium, Kyber, and Saber. Kyber runs 9–14% faster on Cortex-A72 than previous work by [NG21]. For Saber, the speed-ups are even more significant with 24 – 35% fewer cycles. Unsurprisingly, our Dilithium implementation performs much better than the reference implementation.

[5] [NG21] reports cycles without a final reduction, while our implementation includes a reduction.

doi:10.6342/NTU202200738

Table 9.14: Performance of MatrixVectorMul and InnerProd in Dilithium, Kyber, and Saber on Cortex-A72. MV denotes the MatrixVectorMul, IP (Enc) denotes the InnerProd in encryption, and IP (Dec) denotes the InnerProd in decryption.

		MV	IP(Enc)	IP (Dec)
kyber512	[BHK+22]	6 849	2 000	4 844
	[NG21]	10 700	–	7 100
kyber768	[BHK+22]	11 077	2 242	6 518
	[NG21]	19 300	–	9 900
kyber1024	[BHK+22]	16 338	2 758	8 487
	[NG21]	–	–	–
lightsaber	[BHK+22]	18 149	7 038	11 113
	[NG21] (NTT)	37 000	–	22 500
	[NG21] (TC)	40 200	–	18 100
saber	[BHK+22]	35 730	9 284	15 452
	[NG21] (NTT)	71 300	–	31 500
	[NG21] (TC)	81 000	–	25 000
firesaber	[BHK+22]	56 109	11 783	20 112
	[NG21] (NTT)	–	–	–
	[NG21] (TC)	–	–	–
dilithium2	[BHK+22]	26 268	–	–
	(ref)	135 182	–	–
dilithium3	[BHK+22]	38 107	–	–
	(ref)	215 503	–	–
dilithium5	[BHK+22]	54 759	–	–
	(ref)	334 865	–	–

doi:10.6342/NTU202200738

國立臺灣大學 愛國 愛學 勤學

Table 9.15: Performance of Dilithium, Kyber, and Saber on Cortex-A72. We benchmark the fastest implementations by [NG21] in SUPERCOP.

		K	**E**	**D**
`kyber512`	[BHK+22]	62 459	80 710	76 443
	[NG21]	67 903	88 906	87 563
`kyber768`	[BHK+22]	99 201	127 453	120 665
	[NG21]	110 784	141 312	138 984
`kyber1024`	[BHK+22]	156 694	192 280	184 161
	[NG21]	176 809	215 665	214 076
`lightsaber`	[BHK+22]	64 181	87 272	92 813
	[NG21]	83 960	118 583	136 203
`saber`	[BHK+22]	109 192	140 103	147 925
	[NG21]	158 757	206 337	226 304
`firesaber`	[BHK+22]	175 104	211 382	222 317
	[NG21]	245 249	304 128	330 750
		K	**S**	**V**
`dilithium2`	[BHK+22]	269 724	649 230	272 824
	Ref	410 312	1 353 753	449 633
`dilithium3`	[BHK+22]	515 776	1 089 387	447 460
	Ref	743 166	2 308 598	728 866
`dilithium5`	[BHK+22]	782 752	1 436 988	764 886
	Ref	1 151 504	2 903 604	1 198 723

doi:10.6342/NTU202200738

Chapter 10

Future Works

In this chapter, I list some directions for future research. The most urgent ones are to finalize the implementations of `ntrulpr953/sntrup953` and derive a more refined range analysis. Below I'll go through some open problems with great potential.

10.1 Directions for Future Research

The scheduling problem of FFTs is interesting. Several works on scheduling instructions exist. [ACD74] proposed height-based list scheduling, and [Rau94] proposed iterative modulo scheduling to handle the general scheduling problems[1]. Recently, [RMD+15] proposed a systematic approach to building conflict-free schedules for arbitrary mixed-radix FFT, and [BBELG16] proposed a heuristic for very long instruction word (VLIW) schedules. The questions about FFTs schedules are the following:

- How different the scheduling derived from the premutation $\mathrm{rev}^{\mathrm{rev}}_{(2:k)}$ is from the systematic approach by [RMD+15]?
- Is the scheduling with $\mathrm{rev}^{\mathrm{rev}}_{(2:k)}$ a certain kind of height-based list scheduling proposed by [ACD74]?
- Instruction scheduling is NP-hard in general. However, for scheduling FFTs, is it possible to derive a provable optimal scheduling after some modeling?

In terms of the arithmetic of FFTs, several works exist for the complex case. The cost metric of complex-valued FFT is based on the floating-point arithmetic realizing real number computations (since $\mathbb{C} \cong \mathbb{R}[x]/\langle x^2+1\rangle$). Bruun's FFT and split-radix FFT are known for their low number of real arithmetic. Furthermore, for real inputs, discrete Hartley transform and a variant of split-radix FFT effectively reduce the number of real arithmetic. However, modular arithmetic defined on a predefined integer ring is used for most lattice-based cryptosystems. The cost metric is the number of additions, subtractions, multiplications, and modular reductions.

Section 7.2.5 introduces the CT–GS butterflies achieving a smaller cycle count for cyclic radix-2 NTTs by looking at several layers of computations simultaneously. The CT—GS butterflies reduce the number of modular reductions by moving twiddle factors to the earlier layers. There are two natural questions.

[1] We use the word "handle" and it does not mean the problem is solved optimially.

doi:10.6342/NTU202200738

- How to generalize the CT–GS butterflies to acyclic radix-2 NTTs?
- What is the practical interpretation of moving twiddle factors to earlier layers? This thesis proposes an improvement for the näive computations of non-radix-2 butterflies by extending the existence of subtractions (cf. Sections 4.3.5 and 7.2.6). Is it possible to improve non-radix-2 butterflies further by generalizing the CT–GS butterflies?

Vectorized implementations are also an exciting research direction. This thesis studies the vectorization of Cooley–Tukey FFTs for powers of two NTTs. A natural question is the vectorization of polynomial multiplications with sizes that are not powers of two. [BBCT21] implemented truncated Schönhage FFT for computing polynomial multiplication with bounded degree $1536 = 3 \cdot 2^9$. Truncated FFT was implied in [CF94, Section 7] and formally introduced in [vdH04].

Suppose the degree of a polynomial product is bounded by n. [CF94] introduced the computation defined over the polynomial modulus

$$\frac{x^{2^{\lceil \log_2 n \rceil}} - 1}{\prod_{i=0}^{\lceil \log_2 n \rceil - 1} \left(x^{2^i} + 1\right)^{\tilde{b}_i}}$$

where the tuple $(\tilde{b}_0, \tilde{b}_1, \dots, \tilde{b}_{\lceil \log_2 n \rceil - 1})$ is the binary representation of the 2's complement of $-n$ as a $\lceil \log_2 n \rceil$-bit number. [vdH04] computed the product modulo the polynomial

$$\prod_{i=0}^{n-1} (x - \omega_{2^{\lceil \log_2 n \rceil}}^{\mathrm{rev}(2:i)}).$$

Notice that this is a restriction of the computation defined over the polynomial $x^{2^{\lceil \log_2 n \rceil}} - 1 = \prod_{i=0}^{2^{\lceil \log_2 n \rceil}-1} (x - \omega_{2^{\lceil \log_2 n \rceil}}^{\mathrm{rev}(2:i)})$. To see where "truncation" comes from, one first implement the transformation for the polynomial modulus $x^{2^{\lceil \log_2 n \rceil}} - 1$ and discard the computations defined over

$$\prod_{i=n}^{2^{\lceil \log_2 n \rceil}-1} (x - \omega_{2^{\lceil \log_2 n \rceil}}^{\mathrm{rev}(2:i)}).$$

If the coefficeint ring does not exhibit a principal $2^{\lceil \log_2 n \rceil}$-th root of unity, [Sch77] and [Nus80] introduced approaches for crafting symbolic roots of unity. Schönhage FFT is more cache-friendly and vectorization-friendly. We first look at the computation for the ring

$$\frac{R[x]}{\left\langle x^{2^{\lceil \log_2 n \rceil}} - 1 \right\rangle}.$$

Schönhage FFT introduces the equivalences $x^l \sim y$ and $y^{\frac{2^{\lceil \log_2 n \rceil}}{l}} \sim 1$. We now regard the computation as the multiplication in the ring

$$\frac{\frac{R[x]}{\langle x^l - y \rangle}[y]}{\left\langle y^{\frac{2^{\lceil \log_2 n \rceil}}{l}} - 1 \right\rangle},$$

doi:10.6342/NTU202200738

and apply an injection to the ring

$$\frac{\frac{R[x]}{\left\langle x^{2l}+1\right\rangle}[y]}{\left\langle y^{\frac{2^{\lceil \log_2 n\rceil}}{l}} - 1\right\rangle}.$$

If $\frac{2^{\lceil \log_2 n\rceil}}{l} | 4l$, x is a principal $4l$-th root of unity for applying FFT defined over the polynomial $y^{\frac{2^{\lceil \log_2 n\rceil}}{l}} - 1$.

Truncated Schönhage FFT can be derived by truncating the Schönhage defined over $x^{2^{\lceil \log_2 n\rceil}} - 1$. We ask the following questions.

- What if the original coefficient ring R already admits principal roots of unity for certian sizes of FFTs?
- In this thesis, a vectorization-friendly Good–Thomas FFT is drafted in Section 8.2.3. Is it possible to derive a vectorization-friendly approach based on Good–Thomas FFT and Schönhage FFT? In particular, what can we say about the already existing principal 3rd root of unity in $R = \mathbb{Z}_{4591}$ for `ntrulpr761/sntrup761`?
- What is the practical interpretation of the dedicated radix-$(2, 3)$ butterflies in the symbolic setting?

10.2 On Systemization

The ultimate goal of implementations is to design a system choosing between combinations of Cooley–Tukey FFT, Good–Thomas FFT, vector–radix FFT, Schönhage FFT, Nussbaumer FFT, and Toeplitz matrix for a large spectrum of platforms with emphases on speed, code size, vectorization, and stack consumption. We are still at the very beginning of a long journey for scientifically evaluating the pros and cons of approaches multiplying polynomials. As hardware rapidly evolves, implementations in this thesis will soon become outdated. Systematic approaches are required for pairing algorithms with future platforms. I hope the data points in this thesis will become helpful for those making decisions for future platforms, provided that similar scenarios are covered by this thesis, particularly when cryptosystems are to be deployed on platforms without compilers specifically optimized for cryptographic uses.

Since great effort is required to develop a system supporting several platforms, I describe a possible systemization for implementing several NTT-based polynomial multiplications below.

1. Implement cyclic radix-2 NTTs since they can be applied to almost all scenarios.
2. Implement acyclic radix-2 NTTs with customizable tables of twiddle factors.
3. Implement cyclic convolutions of sizes $2^k \cdot 3$ and $2^k \cdot 5$ with Good–Thomas FFT.
4. Implement cyclic convolutions of sizes $2^k \cdot 3^2$ and $2^k \cdot 3 \cdot 5$ with Good–Thomas FFT.

doi:10.6342/NTU202200738

5. Implement cyclic convolutions of sizes $2^k \cdot 3^3$, $2^k \cdot 3^2 \cdot 5$ with Good–Thomas FFT.

To see where the three's and five's come from, we extract the following fractions from the Stern–Brocot tree

$$\frac{9}{8}, \frac{5}{4}, \frac{45}{32}, \frac{3}{2}, \frac{27}{16}, \frac{15}{8}.$$

If we multiply all the fractions by 2^k, we have

$$2^k < 2^{k-3} \cdot 3^2 < 2^{k-2} \cdot 5 < 2^{k-5} \cdot 3^2 \cdot 5 < 2^{k-1} \cdot 3 < 2^{k-4} \cdot 3^3 < 2^{k-3} \cdot 3 \cdot 5 < 2^{k+1}.$$

They are

$$1024 < 1152 < 1280 < 1440 < 1536 < 1728 < 1920 < 2048$$

for $2^k = 1024$, and

$$2048 < 2304 < 2560 < 2880 < 3072 < 3456 < 3840 < 4096$$

for $2^k = 2048$. In this thesis, size-1440, size-1536, size-1728, size-2048, and size-2560 convolutions are explored on Cortex-M4.

We give some notes on implementing Good–Thomas FFT.

1. We first check if vectorization is supported. When vectorization is supported, the number v of coefficients contained in a vector is usually a power of two. For this case, divide the size of convolution by v and perform Good–Thomas FFT.

2. If vectorization is not supported, we next look at if there is a code size issue. Analysis in Section 8.2.2 shows that for applying dedicated radix-$(2, 3)$ butterflies supporting the on-the-fly permutation, the radix-3 splits should be at most $3^2 = 9$ on Cortex-M4. If there is another power of three, we should move this power of three to the incompleteness of NTTs. Section 8.2.2 provides analysis for determining the platform-dependent criterion.

3. After determining which Good–Thomas FFT to be implemented, we then apply as many levels of radix-$(2, 3)$ butterflies as possible. Note that the initial level should consist of dedicated radix-$(2, 3)$ butterflies.

10.3 On Verification

Experiments show that modern compilers are far from producing satisfactory programs. One may argue that portable implementations suffice, and the reality is that cryptosystems are indeed implemented at the assembly level for each architecture nowadays[2]. Although not mentioned in this thesis, there are many reasons supporting assembly-level implementations other than performance concerns. For example, the correctness of implementations is crucial for cryptographic implementations.

[2] Elliptic curves for various architectures at https://github.com/openssl/openssl/tree/master/crypto/ec/asm.

doi:10.6342/NTU202200738

For implementations in general-purpose programming languages, the state-of-the-art verified optimizing C compiler CompCert [LBK+16][3] offers performance on a par with GCC at optimization level `-O1`. Since there is already a considerable gap between highly-optimized assembly implementations and assembly programs produced by GCC with `-O3`, a huge sacrifice of performance is expected if we use a verified C compiler for compiling performance-critical subroutines within cryptosystems.

If one chooses to program in a general-purpose programming language and compile with an unverified optimizing compiler, the only way to verify the correctness is to verify the assembly programs generated by the compiler. Experience has shown that deciphering and translating the compiler-generated assembly programs into formal expressions for verification requires a great effort. On the other hand, programming in assembly, though challenging, implies that the author must know how the programs work. To verify the correctness, the author can either learn about verification or communicate with experts from the verification domain. I believe this is a more reasonable way of producing highly optimized and verified assembly programs for cryptographic uses.

[3] CompCert compiles the source code in C into assembly programs while preserving the semantic in C.

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Chapter 11

Bibliography

[ABB$^+$20] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zemor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. BIKE. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://bikesuite.org/. 3

[ABC$^+$20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://classic.mceliece.org/. 3

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8593. xvii, xviii, 68, 74, 76, 140

[ABD$^+$20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/dilithium/. 2, 9, 89, 90

[ABD$^+$20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/kyber/. 2, 9, 89, 90

[ACC$^+$21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben

doi:10.6342/NTU202200738

Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733. xvii, xviii, 4, 7, 48, 66, 67, 74, 83, 87, 88, 97, 98, 107, 109, 111, 112, 136, 137, 138, 139

[ACC$^+$22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292. xiii, xvii, xviii, 5, 61, 73, 76, 77, 80, 82, 85, 86, 90, 94, 95, 99, 102, 103, 105, 112, 115, 141, 142, 143, 144, 145

[ACD74] Thomas L Adam, K. Mani Chandy, and JR Dickson. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM*, 17(12):685–690, 1974. 121

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Dann Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. 2022. To appear at ACNS 2022, available as https://eprint.iacr.org/2022/112. 7, 67, 86, 90

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like Ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8739. 110, 112

[ARM10a] ARM. *Cortex-M3 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0337/h. 58

[ARM10b] ARM. *Cortex-M4 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0439/b/. 57

[ARM15] ARM. *Cortex-A72 Software Optimization Guide*, 2015. https://developer.arm.com/documentation/uan0016/a/. 58, 78

[ARM21a] ARM. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*, 2021. https://developer.arm.com/documentation/ddi0487/gb/?lang=en. 48, 49, 50, 51, 52, 54, 55

[ARM21b] ARM. *Armv7-M Architecture Refernce Manual*, 2021. https://developer.arm.com/documentation/ddi0403/ed. 43, 44, 45, 46, 47, 48

[ARM21c] ARM. *Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)*, 2021. https://github.com/ARM-software/abi-aa/releases. 48

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO 1986*, LNCS, pages 311–323. SV, 1986. 62

doi:10.6342/NTU202200738

[BBC$^+$20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntruprime.cr.yp.to/. 2, 10

[BBCT21] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. *arXiv preprint arXiv:2106.08759*, 2021. 122

[BBELG16] Mounir Bahtat, Said Belkouch, Philippe Elleaume, and Philippe Le Gall. Instruction scheduling heuristic for an efficient FFT in VLIW processors with balanced resource usage. *EURASIP Journal on Advances in Signal Processing*, 2016(1):1–21, 2016. 121

[BDL$^+$12] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. 6

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 2001. 4

[BHK$^+$22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. xvii, xviii, 6, 61, 62, 63, 64, 70, 71, 72, 77, 78, 85, 86, 90, 91, 92, 93, 103, 105, 118, 119, 120, 152, 153, 154, 155, 156, 157, 160

[BMK$^+$22] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhedeg. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9305. 90, 114, 115

[Bou89] Nicolas Bourbaki. *Algebra I*. Springer, 1989. 27

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8298. 110

[CDH$^+$20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntru.org/. 2, 10

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994. 26, 29, 122

doi:10.6342/NTU202200738

[CF13] Jean Cardinal and Samuel Fiorini. On Generalized Comparison-Based Sorting Problems. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 164–175. 2013. 80

[CFJ+10] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M Jungers, and J Ian Munro. An Efficient Algorithm for Partial Order Production. *SIAM journal on computing*, 39(7):2927–2940, 2010. 80

[CFJ+13] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M Jungers, and J Ian Munro. Sorting under Partial Information (without the Ellipsoid Algorithm). *Combinatorica*, 33(6):655–697, 2013. 80

[CFMR+20] A. Casanova, J.-C. Faugere, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem. GeMSS. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www-polsys.lip6.fr/Links/NIST/GeMSS.html. 2

[Che21] Yun-Li Cheng. Number Theoretic Transform for Polynomial Multiplication in Lattice-based Cryptography on ARM Processors. Master's thesis, 2021. https://github.com/dean3154/ntrup_m4. 86, 107, 109, 111, 112

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8791. xvii, 4, 7, 75, 83, 87, 88, 90, 97, 98, 99, 102, 103, 107, 108, 109, 110, 114, 115

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. 34

[DCP+20] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias Kannwischer, and Jacques Patarin. Rainbow. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www.pqcrainbow.org/. 2

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/. 2, 11, 89

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8683. 87

[Für09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. https://doi.org/10.1137/070711761. 28, 29

doi:10.6342/NTU202200738

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: a Foundation for Computer Science*. Addison-Wesley, second edition, 1994. 69

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2021. `https://tches.iacr.org/index.php/TCHES/article/view/8725`. xvii, 65, 66, 76, 86

[Goo58] Irving John Good. The Interaction Algorithm and Practical Fourier Analysis. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):361–372, 1958. 36

[GS66] W. M. Gentleman and G. Sande. Fast Fourier Transforms: For Fun and Profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578. Association for Computing Machinery, 1966. `https://doi.org/10.1145/1464291.1464352`. 35

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014. 6, 63

[HBD$^+$20] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS$^+$. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. `https://sphincs.org/`. 3

[HMCS77] D. Harris, J. McClellan, D. Chan, and H. Schuessler. Vector Radix Fast Fourier Transform. In *ICASSP'77. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 548–551, 1977. 39

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time O (n log n). *Annals of Mathematics*, 193(2):563–617, 2021. 28, 29

[IKPC20] Írem Keskinkurt Paksoy and Murat Cenk. TMVP-based Multiplication for Polynomial Quotient Rings and Application to Saber on ARM Cortex-M4. *Cryptology ePrint Archive*, 2020. `https://eprint.iacr.org/2020/1302`. 26, 90, 102, 114, 115

[IKPC22] Írem Keskinkurt Paksoy and Murat Cenk. Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication. 2022. `https://ia.cr/2022/300`. 26, 86, 107, 108, 109, 110

[Jac12] Nathan Jacobson. *Basic Algebra I*. Courier Corporation, 2012. 17, 20, 22

doi:10.6342/NTU202200738

[JAC$^+$20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://sike.org/. 3

[Knu14] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, third edition, 2014. 33

[KRS19] Matthias J Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates. In *International Conference on Applied Cryptography and Network Security*, pages 281–301. Springer, 2019. 90

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4. 107

[LBK$^+$16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016. 125

[Li21] Ching-Lin Li. Implementation of Polynomial Modular Inversion in Lattice-based cryptography on ARM. Master's thesis, 2021. https://github.com/trista5658321/polyinv-m4. 110

[MAB$^+$20] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. HQC. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. http://pqc-hqc.org/. 3

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8550. 90, 93, 102, 114, 115

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. 62, 63

[NAB$^+$20] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebil. FrodoKEM. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://frodokem.org/. 2

doi:10.6342/NTU202200738

[NG21] Duc Tri Nguyen and Kris Gaj. Optimized Software Implementations of CRYSTALS-Kyber, NTRU, and Saber Using NEON-Based Special Instructions of ARMv8, 2021. Third PQC Standardization Conference. 86, 90, 103, 117, 118, 119, 120

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography. 9, 10, 11, 127, 129, 130, 131, 132, 133

[Nus80] Henri Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980. 122

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://falcon-sign.info/. 2

[Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, 1994. 121

[RMD+15] Stephen Richardson, Dejan Marković, Andrew Danowitz, John Brunhaver, and Mark Horowitz. Building Conflict-Free FFT Schedules. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(4):1146–1155, 2015. 121

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977. 122

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. 2018. https://eprint.iacr.org/2018/039. 63

[Shi20] Cheng-Jhih Shih. Personal communication, June 2020. xvii, 75

[Sho94] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *FOCS 1994*, pages 124–134. IEEE, 1994. https://ieeexplore.ieee.org/abstract/document/365700. 2

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971. 28

[vdH04] Joris van der Hoeven. The Truncated Fourier Transform and Applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 290–296, 2004. 122

[Win78] Shmuel Winograd. On Computing the Discrete Fourier Transform. *Mathematics of computation*, 32(141):175–199, 1978. 37

doi:10.6342/NTU202200738

doi:10.6342/NTU202200738

Appendix A

Macros

doi:10.6342/NTU202200738

A.1 32-bit Butterflies on Cortex-M4

Algorithm 39 32-bit CT butterflies for $\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle :\omega_8}$ (Cortex-M4) [ACC+21].

Signature:

```
_3_layer_CT_32 c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi1, xi2, xi3,
xi4, xi5, xi6, twiddle, t0, t1
```

Inputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = (c_0, \dots, c_7)$$

$$(\mathtt{xi0}, \dots, \mathtt{xi6}) = (\zeta^4, \zeta^2, \zeta^2\omega_8^2, \zeta, \zeta\omega_8^2, \zeta\omega_8, \zeta\omega_8^3)2^{32} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle :\omega_8}\right) \left(\sum_{i=0}^{7} c_i x^i\right)$$

```
vmov twiddle, xi0
montgomery_mul_32 c4, twiddle, t0, t1
montgomery_mul_32 c5, twiddle, t0, t1
montgomery_mul_32 c6, twiddle, t0, t1
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c4, c1, c5, c2, c6, c3, c7
    ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle :\omega_2}\left(\sum_{i=0}^{7} c_i x^i\right)$
vmov twiddle, xi1
montgomery_mul_32 c2, twiddle, t0, t1
montgomery_mul_32 c3, twiddle, t0, t1
vmov twiddle, xi2
montgomery_mul_32 c6, twiddle, t0, t1
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c2, c1, c3, c4, c6, c5, c7
    ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c4}, \mathtt{c5}, \mathtt{c2}, \mathtt{c3}, \mathtt{c6}, \mathtt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle :\omega_4}\left(\sum_{i=0}^{7} c_i x^i\right)$
vmov twiddle, xi3
montgomery_mul_32 c1, twiddle, t0, t1
vmov twiddle, xi4
montgomery_mul_32 c3, twiddle, t0, t1
vmov twiddle, xi5
montgomery_mul_32 c5, twiddle, t0, t1
vmov twiddle, xi6
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c1, c2, c3, c4, c5, c6, c7
    ▷ $(\mathtt{c0}, \mathtt{c4}, \mathtt{c2}, \mathtt{c6}, \mathtt{c1}, \mathtt{c5}, \mathtt{c3}, \mathtt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle :\omega_8}\left(\sum_{i=0}^{7} c_i x^i\right)$
    ▷ $(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle :\omega_8}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 40 32-bit CT butterflies for $\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8}$ (Cortex-M4) [ACC+21].

Signature:
`_3_layer_CT_light_32 c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi1, xi2, xi3, xi4, xi5, xi6, twiddle, tmp0, tmp1`

Inputs:

$$(\mathtt{c0}, \ldots, \mathtt{c7}) = (c_0, \ldots, c_7)$$

$$(\mathtt{xi0}, \ldots, \mathtt{xi6}) = (1, 1, \omega_8^2, 1, \omega_8^2, \omega_8, \omega_8^3)2^{32} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \ldots, \mathtt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$$

```
add_sub4 c0, c4, c1, c5, c2, c6, c3, c7
        ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_2}\left(\sum_{i=0}^{7} c_i x^i\right)$
vmov twiddle, xi2
montgomery_mul_32 c6, twiddle, t0, t1
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c2, c1, c3, c4, c6, c5, c7
        ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c4}, \mathtt{c5}, \mathtt{c2}, \mathtt{c3}, \mathtt{c6}, \mathtt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_4}\left(\sum_{i=0}^{7} c_i x^i\right)$
montgomery_mul_32 c3, twiddle, t0, t1
vmov twiddle, xi5
montgomery_mul_32 c5, twiddle, t0, t1
vmov twiddle, xi6
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c1, c2, c3, c4, c5, c6, c7
        ▷ $(\mathtt{c0}, \mathtt{c4}, \mathtt{c2}, \mathtt{c6}, \mathtt{c1}, \mathtt{c5}, \mathtt{c3}, \mathtt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8}\left(\sum_{i=0}^{7} c_i x^i\right)$
        ▷ $(\mathtt{c0}, \ldots, \mathtt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 41 32-bit CT butterflies for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$ (Cortex-M4) [ACC+21].

Signature:
`_3_layer_inv_CT_32 c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi1, xi2, xi3, xi4, xi5, xi6, twiddle, t0, t1`

Inputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \mathrm{rev}_{(2:3)}(c_0, \dots, c_7)$$

$$(\mathtt{xi0}, \dots, \mathtt{xi6}) = (\zeta^{-4}, \zeta^{-2}, \zeta^{-2}\omega_8^{-2}, \zeta^{-1}, \zeta^{-1}\omega_8^{-2}, \zeta^{-1}\omega_8^{-1}, \zeta^{-1}\omega_8^{-3})2^{32} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$$

```
vmov twiddle, xi0
montgomery_mul_32 c1, twiddle, t0, t1
montgomery_mul_32 c3, twiddle, t0, t1
montgomery_mul_32 c5, twiddle, t0, t1
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c1, c2, c3, c4, c5, c6, c7
    ▷ $(\mathtt{c0}, \mathtt{c4}, \mathtt{c2}, \mathtt{c6}, \mathtt{c1}, \mathtt{c5}, \mathtt{c3}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_2^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
vmov twiddle, xi1
montgomery_mul_32 c2, twiddle, t0, t1
montgomery_mul_32 c6, twiddle, t0, t1
vmov twiddle, xi2
montgomery_mul_32 c3, twiddle, t0, t1
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c2, c1, c3, c4, c6, c5, c7
    ▷ $(\mathtt{c0}, \mathtt{c2}, \mathtt{c4}, \mathtt{c6}, \mathtt{c1}, \mathtt{c3}, \mathtt{c5}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_4^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
vmov twiddle, xi3
montgomery_mul_32 c4, twiddle, t0, t1
vmov twiddle, xi4
montgomery_mul_32 c5, twiddle, t0, t1
vmov twiddle, xi5
montgomery_mul_32 c6, twiddle, t0, t1
vmov twiddle, xi6
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c4, c1, c5, c2, c6, c3, c7
    ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 42 32-bit CT butterflies for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$ (Cortex-M4) [ACC+21].

Signature:

`_3_layer_inv_CT_32_light c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi1, xi2, xi3, xi4, xi5, xi6, twiddle, t0, t1`

Inputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \mathrm{rev}_{(2:3)}\,(c_0, \dots, c_7)$$

$$(\mathtt{xi0}, \dots, \mathtt{xi6}) = (1, 1, \omega_8^{-2}, 1, \omega_8^{-2}, \omega_8^{-1}, \omega_8^{-3})2^{32} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$$

```
add_sub4 c0, c1, c2, c3, c4, c5, c6, c7
    ▷ $(\mathtt{c0}, \mathtt{c4}, \mathtt{c2}, \mathtt{c6}, \mathtt{c1}, \mathtt{c5}, \mathtt{c3}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_2^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
vmov twiddle, xi2
montgomery_mul_32 c3, twiddle, t0, t1
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c2, c1, c3, c4, c6, c5, c7
    ▷ $(\mathtt{c0}, \mathtt{c2}, \mathtt{c4}, \mathtt{c6}, \mathtt{c1}, \mathtt{c3}, \mathtt{c5}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_4^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
montgomery_mul_32 c6, twiddle, t0, t1
vmov twiddle, xi4
montgomery_mul_32 c5, twiddle, t0, t1
vmov twiddle, xi6
montgomery_mul_32 c7, twiddle, t0, t1
add_sub4 c0, c4, c1, c5, c2, c6, c3, c7
    ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

A.2 16-bit Butterflies on Cortex-M4

Algorithm 43 16-bit CT butterflies for $\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle:\omega_8}$ (Cortex-M4) [ABCG20].

Signature:

```
_3_layer_double_CT_16 c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi12,
xi34, xi56, twiddle, t0, t1
```

Inputs:

$$(\texttt{c0},\ldots,\texttt{c7}) = (c_{2i+1}||c_{2i})_{i=0,\ldots,7}$$

$$(\texttt{xi0},\texttt{xi12},\ldots,\texttt{xi56}) = (\zeta^4||0, \zeta^2\omega_8^2||\zeta^2, \zeta\omega_8^2||\zeta, \zeta\omega_8^3||\zeta\omega_8)2^{16} \bmod {}^{\pm}q$$

Outputs:

$$(\texttt{c0},\ldots,\texttt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle:\omega_8}\right)\left(\sum_{i=0}^{7} c_{2i+1}||c_{2i}x^i\right)$$

```
vmov twiddle, xi0
doublebutterfly_16 t, c0, c4, twiddle, t0, t1
doublebutterfly_16 t, c1, c5, twiddle, t0, t1
doublebutterfly_16 t, c2, c6, twiddle, t0, t1
doublebutterfly_16 t, c3, c7, twiddle, t0, t1
        ▷ $(\texttt{c0},\texttt{c1},\texttt{c2},\texttt{c3},\texttt{c4},\texttt{c5},\texttt{c6},\texttt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle:\omega_2}\left(\sum_{i=0}^{7} c_{2i+1}||c_{2i}x^i\right)$
vmov twiddle, xi12
doublebutterfly_16 b, c0, c2, twiddle, t0, t1
doublebutterfly_16 b, c1, c3, twiddle, t0, t1
doublebutterfly_16 t, c4, c6, twiddle, t0, t1
doublebutterfly_16 t, c5, c7, twiddle, t0, t1
        ▷ $(\texttt{c0},\texttt{c1},\texttt{c4},\texttt{c5},\texttt{c2},\texttt{c3},\texttt{c6},\texttt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle:\omega_4}\left(\sum_{i=0}^{7} c_{2i+1}||c_{2i}x^i\right)$
vmov twiddle, xi34
doublebutterfly_16 b, c0, c1, twiddle, t0, t1
doublebutterfly_16 t, c2, c3, twiddle, t0, t1
vmov twiddle, xi56
doublebutterfly_16 b, c4, c5, twiddle, t0, t1
doublebutterfly_16 t, c6, c7, twiddle, t0, t1
        ▷ $(\texttt{c0},\texttt{c4},\texttt{c2},\texttt{c6},\texttt{c1},\texttt{c5},\texttt{c3},\texttt{c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle:\omega_8}\left(\sum_{i=0}^{7} c_{2i+1}||c_{2i}x^i\right)$
            ▷ $(\texttt{c0},\ldots,\texttt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle:\omega_8}\right)\left(\sum_{i=0}^{7} c_{2i+1}||c_{2i}x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 44 16-bit CT butterflies for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$ (Cortex-M4) [ACC$^+$22].

Signature:
`_3_layer_double_inv_CT_16 c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi12, xi34, xi56, twiddle, t0, t1`

Inputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \mathrm{rev}_{(2:3)} \left(c_{2i+1} \,||\, c_{2i}\right)_{i=0,\dots,7}$$

$$(\mathtt{xi0}, \dots, \mathtt{xi56}) = (\zeta^{-4} \,||\, 0, \zeta^{-2}\omega_8^{-2} \,||\, \zeta^{-2}, \zeta^{-1}\omega_8^{-2} \,||\, \zeta^{-1}, \zeta^{-1}\omega_8^{-3} \,||\, \zeta^{-1}\omega_8^{-1})2^{16} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right) \left(\sum_{i=0}^{7} c_{2i+1} \,||\, c_{2i} x^i\right)$$

```
vmov twiddle, xi0
doublebutterfly_16 t, c0, c1, twiddle, t0, t1
doublebutterfly_16 t, c2, c3, twiddle, t0, t1
doublebutterfly_16 t, c4, c5, twiddle, t0, t1
doublebutterfly_16 t, c6, c7, twiddle, t0, t1
        ▷ $(\mathtt{c0}, \mathtt{c4}, \mathtt{c2}, \mathtt{c6}, \mathtt{c1}, \mathtt{c5}, \mathtt{c3}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_2^{-1}} \circ \mathrm{rev}_{(2:3)}\right) \left(\sum_{i=0}^{7} c_{2i+1} \,||\, c_{2i} x^i\right)$
vmov twiddle, xi12
doublebutterfly_16 b, c0, c2, twiddle, t0, t1
doublebutterfly_16 t, c1, c3, twiddle, t0, t1
doublebutterfly_16 b, c4, c6, twiddle, t0, t1
doublebutterfly_16 t, c5, c7, twiddle, t0, t1
        ▷ $(\mathtt{c0}, \mathtt{c2}, \mathtt{c4}, \mathtt{c6}, \mathtt{c1}, \mathtt{c3}, \mathtt{c5}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_4^{-1}} \circ \mathrm{rev}_{(2:3)}\right) \left(\sum_{i=0}^{7} c_{2i+1} \,||\, c_{2i} x^i\right)$
vmov twiddle, xi34
doublebutterfly_16 b, c0, c4, twiddle, t0, t1
doublebutterfly_16 t, c1, c5, twiddle, t0, t1
vmov twiddle, xi56
doublebutterfly_16 b, c2, c6, twiddle, t0, t1
doublebutterfly_16 t, c3, c7, twiddle, t0, t1
        ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right) \left(\sum_{i=0}^{7} c_{2i+1} \,||\, c_{2i} x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 45 16-bit CT butterflies for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$ (Cortex-M4) [ACC+22].

Signature:
`_3_layer_double_inv_CT_16_light c0, c1, c2, c3, c4, c5, c6, c7, xi0, xi12, xi34, xi56, twiddle, t0, t1`

Inputs:

$$(\texttt{c0},\dots,\texttt{c7}) = \mathrm{rev}_{(2:3)}\,(c_{2i+1}\,||\,c_{2i})_{i=0,\dots,7}$$

$$(\texttt{xi0},\dots,\texttt{xi56}) = (1\,||\,0, \omega_8^{-2}\,||\,1, \omega_8^{-2}\,||\,1, \omega_8^{-3}\,||\,\omega_8^{-1})2^{16} \bmod {}^{\pm}q$$

Outputs:

$$(\texttt{c0},\dots,\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_{2i+1}\,||\,c_{2i}x^i\right)$$

```
sadd16 t0, c0, c1
ssub16 c1, c0, c1
sadd16 t1, c2, c3
ssub16 c3, c2, c3
sadd16 c0, t0, t1
ssub16 c2, t0, t1
        ▷ $(\texttt{c0},\texttt{c4},\texttt{c2},\texttt{c6},\texttt{c1},\texttt{c5},\texttt{c3},\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_2^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_{2i+1}\,||\,c_{2i}x^i\right)$
sadd16 t0, c4, c5
ssub16 c5, c4, c5
sadd16 t1, c6, c7
ssub16 c7, c6, c7
sadd16 c4, t0, t1
ssub16 c6, t0, t1
vmov twiddle, xi12
doublebutterfly_16 t, c1, c3, twiddle, t0, t1
doublebutterfly_16 t, c5, c7, twiddle, t0, t1
        ▷ $(\texttt{c0},\texttt{c2},\texttt{c4},\texttt{c6},\texttt{c1},\texttt{c3},\texttt{c5},\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_4^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_{2i+1}\,||\,c_{2i}x^i\right)$
vmov twiddle, xi34
doublebutterfly_16 b, c0, c4, twiddle, t0, t1
doublebutterfly_16 t, c1, c5, twiddle, t0, t1
vmov twiddle, xi56
doublebutterfly_16 b, c2, c6, twiddle, t0, t1
doublebutterfly_16 t, c3, c7, twiddle, t0, t1
        ▷ $(\texttt{c0},\texttt{c1},\texttt{c2},\texttt{c3},\texttt{c4},\texttt{c5},\texttt{c6},\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_{2i+1}\,||\,c_{2i}x^i\right)$
```

doi:10.6342/NTU202200738

A.3 16-bit Butterflies on Cortex-M3

Algorithm 46 16-bit CT butterflies for $\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_8}$ (Cortex-M3) [ACC+22].

Signature:

`_3_layer_CT_16 c0, c1, c2, c3, c4, c5, c6, c7, twiddle_ptr, twiddle, t`

Inputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = (c_0, \dots, c_7),$$

$$\mathtt{twiddle\_ptr[0\text{-}6]} = (\zeta^4, \zeta^2, \zeta^2\omega^2, \zeta, \zeta\omega^2, \zeta\omega, \zeta\omega^3)2^{16} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_8}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$$

```
ldr.w twiddle, [twiddle_ptr, #0]
montgomery_mul_16_addSub c0, c4, twiddle, t
montgomery_mul_16_addSub c1, c5, twiddle, t
montgomery_mul_16_addSub c2, c6, twiddle, t
montgomery_mul_16_addSub c3, c7, twiddle, t
                    ▷ $(\mathtt{c0,c1,c2,c3,c4,c5,c6,c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_2}\left(\sum_{i=0}^{7} c_i x^i\right)$
ldr.w twiddle, [twiddle_ptr, #4]
montgomery_mul_16_addSub c0, c2, twiddle, t
montgomery_mul_16_addSub c1, c3, twiddle, t
ldr.w twiddle, [twiddle_ptr, #8]
montgomery_mul_16_addSub c4, c6, twiddle, t
montgomery_mul_16_addSub c5, c7, twiddle, t
                    ▷ $(\mathtt{c0,c1,c4,c5,c2,c3,c6,c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_4}\left(\sum_{i=0}^{7} c_i x^i\right)$
ldr.w twiddle, [twiddle_ptr, #12]
montgomery_mul_16_addSub c0, c1, twiddle, t
ldr.w twiddle, [twiddle_ptr, #16]
montgomery_mul_16_addSub c2, c3, twiddle, t
ldr.w twiddle, [twiddle_ptr, #20]
montgomery_mul_16_addSub c4, c5, twiddle, t
ldr.w twiddle, [twiddle_ptr, #24]
montgomery_mul_16_addSub c6, c7, twiddle, t
                    ▷ $(\mathtt{c0,c4,c2,c6,c1,c5,c3,c7}) = \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_8}\left(\sum_{i=0}^{7} c_i x^i\right)$
                         ▷ $(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathrm{rev}_{(2:3)} \circ \mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^8\rangle : \omega_8}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 47 16-bit CT butterflies for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$ (Cortex-M3) [ACC+22].

Signature:
`_3_layer_inv_CT_16 c0, c1, c2, c3, c4, c5, c6, c7, twiddle_ptr, twiddle, t`

Inputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = (c_0, \dots, c_7),$$

$$\mathtt{twiddle\_ptr[0\text{-}6]} = (\zeta^{-4}, \zeta^{-2}, \zeta^{-2}\omega^{-2}, \zeta^{-1}, \zeta^{-1}\omega^{-2}, \zeta^{-1}\omega^{-1}, \zeta^{-1}\omega^{-3})2^{16} \bmod {}^{\pm}q$$

Outputs:

$$(\mathtt{c0}, \dots, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)$$

```
ldr.w twiddle, [twiddle_ptr, #0]
montgomery_mul_16_addSub c0, c1, twiddle, t
montgomery_mul_16_addSub c2, c3, twiddle, t
montgomery_mul_16_addSub c4, c5, twiddle, t
montgomery_mul_16_addSub c6, c7, twiddle, t
    ▷ $(\mathtt{c0}, \mathtt{c4}, \mathtt{c2}, \mathtt{c6}, \mathtt{c1}, \mathtt{c5}, \mathtt{c3}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_2^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
ldr.w twiddle, [twiddle_ptr, #4]
montgomery_mul_16_addSub c0, c2, twiddle, t
montgomery_mul_16_addSub c4, c6, twiddle, t
ldr.w twiddle, [twiddle_ptr, #8]
montgomery_mul_16_addSub c1, c3, twiddle, t
montgomery_mul_16_addSub c5, c7, twiddle, t
    ▷ $(\mathtt{c0}, \mathtt{c2}, \mathtt{c4}, \mathtt{c6}, \mathtt{c1}, \mathtt{c3}, \mathtt{c5}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_4^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
ldr.w twiddle, [twiddle_ptr, #12]
montgomery_mul_16_addSub c0, c4, twiddle, t
ldr.w twiddle, [twiddle_ptr, #16]
montgomery_mul_16_addSub c1, c5, twiddle, t
ldr.w twiddle, [twiddle_ptr, #20]
montgomery_mul_16_addSub c2, c6, twiddle, t
ldr.w twiddle, [twiddle_ptr, #24]
montgomery_mul_16_addSub c3, c7, twiddle, t
    ▷ $(\mathtt{c0}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3}, \mathtt{c4}, \mathtt{c5}, \mathtt{c6}, \mathtt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-\zeta^{-8}\rangle : \omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

Algorithm 48 16-bit CT butterflies for $\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}$ with initial reductions (Cortex-M3) [ACC+22].

Signature:
`_3_layer_inv_CT_16_light c0, c1, c2, c3, c4, c5, c6, c7, twiddle_ptr, twiddle, t`

Inputs:

$$(\texttt{c0},\dots,\texttt{c7}) = (c_0,\dots,c_7),$$

$$\texttt{twiddle\_ptr[0-6]} = (1,1,\omega^{-2},1,\omega^{-2},\omega^{-1},\omega^{-3})2^{16} \bmod {}^{\pm}q$$

Outputs:

$$(\texttt{c0},\dots,\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$$

```
ldr.w twiddle, [twiddle_ptr, #0]
montgomery_mul_16_addSub c0, c1, twiddle, t
montgomery_mul_16_addSub c2, c3, twiddle, t
montgomery_mul_16_addSub c4, c5, twiddle, t
montgomery_mul_16_addSub c6, c7, twiddle, t
    ▷ $(\texttt{c0},\texttt{c4},\texttt{c2},\texttt{c6},\texttt{c1},\texttt{c5},\texttt{c3},\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_2^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
addSub2 c0, c2, c4, c6
ldr.w twiddle, [twiddle_ptr, #8]
montgomery_mul_16_addSub c1, c3, twiddle, t
montgomery_mul_16_addSub c5, c7, twiddle, t
    ▷ $(\texttt{c0},\texttt{c2},\texttt{c4},\texttt{c6},\texttt{c1},\texttt{c3},\texttt{c5},\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_4^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
addSub1 c0, c4
montgomery_mul_16_addSub c2, c6, twiddle, t
ldr.w twiddle, [twiddle_ptr, #16]
montgomery_mul_16_addSub c1, c5, twiddle, t
ldr.w twiddle, [twiddle_ptr, #24]
montgomery_mul_16_addSub c3, c7, twiddle, t
    ▷ $(\texttt{c0},\texttt{c1},\texttt{c2},\texttt{c3},\texttt{c4},\texttt{c5},\texttt{c6},\texttt{c7}) = \left(\mathsf{NTT}_{\mathbb{Z}_q[x]/\langle x^8-1\rangle:\omega_8^{-1}} \circ \mathrm{rev}_{(2:3)}\right)\left(\sum_{i=0}^{7} c_i x^i\right)$
```

doi:10.6342/NTU202200738

A.4 Dedicated Vector–Radix Butterflies on Cortex-M4

Algorithm 49 _6_ntt_100 (Cortex-M4).

Input:

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

$$\begin{cases} \mathcal{R}^{(0)} & = \frac{R[x^{(0)}]}{\left\langle \left(x^{(0)}\right)^{2} - 1 \right\rangle} \\ \mathcal{R}^{(1)} & = \frac{R[x^{(1)}]}{\left\langle \left(x^{(1)}\right)^{3} - 1 \right\rangle} \end{cases}$$

Output:

$$\begin{pmatrix} \texttt{c0} & \texttt{c2} & \texttt{c4} \\ \texttt{c1} & \texttt{c3} & \texttt{c5} \end{pmatrix} = \begin{pmatrix} \hat{c}_{0,0} & \hat{c}_{0,1} & \hat{c}_{0,2} \\ \hat{c}_{1,0} & \hat{c}_{1,1} & \hat{c}_{1,2} \end{pmatrix} = \mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_2} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_3} \begin{pmatrix} c_{0,0} & 0 & 0 \\ 0 & c_{1,1} & c_{1,2} \end{pmatrix}$$

```
smull c4, c2, c3, $\omega_3$
smlal c4, c2, c5, $\omega_3^2$
mul c1, c4, $q'$
smlal c4, c2, c1, $q$
add.w c1, c3, c5
add.w c5, c2, c1
sub.w c4, c0, c5
add.w c5, c5, c0
sub.w c3, c0, c2
add.w c2, c2, c0
add.w c0, c0, c1
sub.w c1, c0, c1, lsl #1
```

doi:10.6342/NTU202200738

Algorithm 50 _6_ntt_010 (Cortex-M4).

Input:

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

$$\begin{cases} \mathcal{R}^{(0)} & = \frac{R[x^{(0)}]}{\left\langle \left(x^{(0)}\right)^2 - 1 \right\rangle} \\ \mathcal{R}^{(1)} & = \frac{R[x^{(1)}]}{\left\langle \left(x^{(1)}\right)^3 - 1 \right\rangle} \end{cases}$$

Output:

$$\begin{pmatrix} \texttt{c0} & \texttt{c2} & \texttt{c4} \\ \texttt{c1} & \texttt{c3} & \texttt{c5} \end{pmatrix} = \begin{pmatrix} \hat{c}_{0,0} & \hat{c}_{0,1} & \hat{c}_{0,2} \\ \hat{c}_{1,0} & \hat{c}_{1,1} & \hat{c}_{1,2} \end{pmatrix} = \mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_2} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_3} \begin{pmatrix} 0 & c_{0,1} & 0 \\ c_{1,0} & 0 & c_{1,2} \end{pmatrix}$$

```
montgomery_mul_des_32 t, c4, c5, $\omega_3$, c0
add.w c3, c4, c5
sub.w c3, c1, c3
add.w c4, c4, c1
add.w c1, c1, c5
montgomery_mul_des_32 c0, t, c2, $\omega_3$, c5
add.w c5, c2, t
add.w c0, c2, c1
sub.w c1, c0, c1, lsl #1
add.w c2, t, c3
sub.w c3, c2, c3, lsl #1
sub.w c4, c4, c5
add.w c5, c4, c5, lsl #1
neg.w c5, c5
```

doi:10.6342/NTU202200738

Algorithm 51 _6_ntt_001 (Cortex-M4).

Input:

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

Output:

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

```
montgomery_mul_des_32 t, c2, c3, $\omega_3$, c0
add.w c5, c2, c3
sub.w c5, c1, c5
add.w c2, c2, c1
add.w c1, c1, c3
montgomery_mul_des_32 c0, t, c4, $\omega_3$ c3
add.w c3, c4, t
add.w c0, c4, c1
sub.w c1, c0, c1, lsl #1
add.w c4, t, c5
sub.w c5, c4, c5, lsl #1
sub.w c2, c2, c3
add.w c3, c2, c3, lsl #1
neg.w c3, c3
```

doi:10.6342/NTU202200738

Algorithm 52 `_6_ntt_101` (Cortex-M4).

Input:

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

Output:

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

```
montgomery_mul_des_32 t, c5, c4, $\omega_3$, c1        ▷ $\mathtt{c5} = \omega_3 c_{0,2}$
add c2, c4, c5                                        ▷ $\mathtt{c2} = -\omega_3^2 c_{0,2}$
sub c2, c0, c2                                        ▷ $\mathtt{c2} = c_{0,0} + \omega_3^2 c_{0,2}$
add c5, c5, c0                                        ▷ $\mathtt{c5} = c_{0,0} + \omega_3 c_{0,2}$
add c0, c0, c4                                        ▷ $\mathtt{c0} = c_{0,0} + c_{0,2}$
montgomery_mul_des_32 c1, t, c3, $\omega_3$, c4        ▷ $\mathtt{t} = \omega_3 c_{1,1}$
add c4, c3, t                                         ▷ $\mathtt{c4} = -\omega_3^2 c_{1,1}$
add c0, c0, c3                                        ▷ $\mathtt{c0} = \hat{c}_{0,0}$
sub c1, c0, c3, lsl #1                                ▷ $\mathtt{c1} = \hat{c}_{1,0}$
add c2, c2, t                                         ▷ $\mathtt{c2} = \hat{c}_{0,1}$
sub c3, c2, t, lsl #1                                 ▷ $\mathtt{c3} = \hat{c}_{1,1}$
add c5, c5, c4                                        ▷ $\mathtt{c5} = \hat{c}_{1,2}$
sub c4, c5, c4, lsl #1                                ▷ $\mathtt{c4} = \hat{c}_{0,2}$
```

doi:10.6342/NTU202200738

Algorithm 53 `_6_ntt_011` (Cortex-M4).

Input:

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

Output:

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

```
smull c5, c3, c2, $\omega_3$
smlal c5, c3, c4, $\omega_3^2$
mul c0, c5, $q'$
smlal c5, c3, c0, $q$
add.w c0, c2, c4
add.w c4, c3, c0
add.w c5, c1, c4
neg.w c5, c5
sub.w c4, c1, c4
add.w c2, c3, c1
sub.w c3, c3, c1
add.w c0, c0, c1
sub.w c1, c0, c1, lsl #1
```

doi:10.6342/NTU202200738

Algorithm 54 `_6_ntt_110` (Cortex-M4).

Input:

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

Output:

$$\begin{pmatrix} \texttt{c0} & \texttt{c2} & \texttt{c4} \\ \texttt{c1} & \texttt{c3} & \texttt{c5} \end{pmatrix} = \begin{pmatrix} \hat{c}_{0,0} & \hat{c}_{0,1} & \hat{c}_{0,2} \\ \hat{c}_{1,0} & \hat{c}_{1,1} & \hat{c}_{1,2} \end{pmatrix} = \mathsf{NTT}_{\mathcal{R}^{(0)}:\omega_2} \otimes \mathsf{NTT}_{\mathcal{R}^{(1)}:\omega_3} \begin{pmatrix} c_{0,0} & c_{0,1} & 0 \\ 0 & 0 & c_{1,2} \end{pmatrix}$$

```
montgomery_mul_des_32 t, c3, c2, $\omega_3$, c1
add.w c4, c2, c3
sub.w c4, c0, c4
add.w c3, c3, c0
add.w c0, c0, c2
montgomery_mul_des_32 c1, t, c5, $\omega_3$, c2
add.w c2, c5, t
add.w c0, c0, c5
sub.w c1, c0, c5, lsl #1
add.w c4, c4, t
sub.w c5, c4, t, lsl #1
add.w c3, c3, c2
sub.w c2, c3, c2, lsl #1
```

doi:10.6342/NTU202200738

A.5 Reductions and Multiplications on Cortex-A72

Algorithm 55 Interleaved single-width Barrett reduction (Cortex-A72) [BHK+22].

Signature:
`wrap_oX_barrett a[0-3], t[0-3], a[4-7], t[4-7], barrett_const, shrv, wX, nX`

Inputs:
`a[0-3]` $= (c_0, \ldots, c_3)$,
`a[4-7]` $= (c_4, \ldots, c_7)$,
`barrett_const` $= \left\lfloor \frac{\mathtt{R}}{q} \right\rceil$, `shrv` $= \mathtt{R} - \mathtt{w} + 1$

Outputs:
`a[0-3]` $= (c_0 \bmod {}^{\pm}q, \ldots, c_3 \bmod {}^{\pm}q)$,
`a[4-7]` $= (c_4 \bmod {}^{\pm}q, \ldots, c_7 \bmod {}^{\pm}q)$

```
sqdmulh t0wX, a0wX, barrett_const
sqdmulh t1wX, a1wX, barrett_const
sqdmulh t2wX, a2wX, barrett_const
sqdmulh t3wX, a3wX, barrett_const
srshr t0wX, t0wX, shrv
sqdmulh t4wX, a4wX, barrett_const
srshr t1wX, t1wX, shrv
sqdmulh t5wX, a5wX, barrett_const
srshr t2wX, t2wX, shrv
sqdmulh t6wX, a6wX, barrett_const
srshr t3wX, t3wX, shrv
sqdmulh t7wX, a7wX, barrett_const
mls a0wX, t0wX, $q$
srshr t4wX, t4wX, shrv
mls a1wX, t1wX, $q$
srshr t5wX, t5wX, shrv
mls a2wX, t2wX, $q$
srshr t6wX, t6wX, shrv
mls a3wX, t3wX, $q$
srshr t7wX, t7wX, shrv
mls a4wX, t4wX, $q$
mls a5wX, t5wX, $q$
mls a6wX, t6wX, $q$
mls a7wX, t7wX, $q$
```

doi:10.6342/NTU202200738

A.6 Butterflies on Cortex-A72

Algorithm 56 Parallelized butterflies (top, Cortex-A72) [BHK+22].

Signature:

`wrap_qX_butterfly_top a[0-3], b[0-3], t[0-3], {z, l, h}[0-3], wX, nX`

Inputs:

`b[0-3]` $= (b_0, \dots, b_3)$,

`(z0[h0],z0[l0])` $= \left(\psi_0, \frac{\left\lfloor \frac{\psi_0 \mathrm{R}}{q} \right\rceil_{2,q}}{2}\right), \dots,$ `(z3[h3],z3[l3])` $= \left(\psi_3, \frac{\left\lfloor \frac{\psi_3 \mathrm{R}}{q} \right\rceil_{2,q}}{2}\right)$

Outputs: `t[0-3]` $= (b_0\psi_0, \dots, b_3\psi_3)$

```
mul t0wX, b0wX, z0nX[h0]
mul t1wX, b1wX, z1nX[h1]
mul t2wX, b2wX, z2nX[h2]
mul t3wX, b3wX, z3nX[h3]
sqrdmulh b0wX, b0wX, z0nX[l0]
sqrdmulh b1wX, b1wX, z1nX[l1]
sqrdmulh b2wX, b2wX, z2nX[l2]
sqrdmulh b3wX, b3wX, z3nX[l3]
mls t0wX, b0wX, $q$
mls t1wX, b1wX, $q$
mls t2wX, b2wX, $q$
mls t3wX, b3wX, $q$
```

Algorithm 57 Parallelized butterflies (bot, Cortex-A72) [BHK+22].

Signature:

`wrap_qX_butterfly_bot a[0-3], b[0-3], t[0-3], {z, l, h}[0-3], wX, nX`

Inputs:

`a[0-3]` $= (a_0, \dots, a_3)$,

`t[0-3]` $= (b_0, \dots, b_3)$

Outputs:

`a[0-3]` $= (a_0 + b_0, \dots, a_3 + b_3)$,

`b[0-3]` $= (a_0 - b_0, \dots, a_3 - b_3)$

```
sub b0wX, a0wX, t0wX
sub b1wX, a1wX, t1wX
sub b2wX, a2wX, t2wX
sub b3wX, a3wX, t3wX
add a0wX, a0wX, t0wX
add a1wX, a1wX, t1wX
add a2wX, a2wX, t2wX
add a3wX, a3wX, t3wX
```

doi:10.6342/NTU202200738

Algorithm 58 Interleaved CT butterflies (Cortex-A72) [BHK+22].

Signature:
`wrap_qX_butterfly_mixed a[0-3], b[0-3], t[0-3], a[4-7], b[4-7], t[4-7], {z, l, h}[0-3], {z, l, h}[4-7], wX, nX`

Inputs:
`a[0-3]` $= (a_0, \ldots, a_3)$,
`t[0-3]` $= (b_0\psi_0, \ldots, b_3\psi_3)$,
`b[4-7]` $= (b_4, \ldots, b_7)$,
`(z4[h4],z4[l4])` $= \left(\psi_4, \frac{\left\lfloor \frac{\psi_4 \mathrm{R}}{q} \right\rceil_{2,q}}{2}\right), \ldots,$ `(z7[h7],z7[l7])` $= \left(\psi_7, \frac{\left\lfloor \frac{\psi_7 \mathrm{R}}{q} \right\rceil_{2,q}}{2}\right)$

Outputs:
`a[0-3]` $= (a_0 + \psi_0 b_0, \ldots, a_3 + \psi_3 b_3)$,
`b[0-3]` $= (a_0 - \psi_0 b_0, \ldots, a_3 - \psi_3 b_3)$,
`t[4-7]` $= (b_4\psi_4, \ldots, b_7\psi_7)$

```
sub b0wX, a0wX, t0wX
mul t4wX, b4wX, z4nX[h4]
sub b1wX, a1wX, t1wX
mul t5wX, b5wX, z5nX[h5]
sub b2wX, a2wX, t2wX
mul t6wX, b6wX, z6nX[h6]
sub b3wX, a3wX, t3wX
mul t7wX, b7wX, z7nX[h7]
add a0wX, a0wX, t0wX
sqrdmulh b4wX, b4wX, z4nX[l4]
add a1wX, a1wX, t1wX
sqrdmulh b5wX, b5wX, z5nX[l5]
add a2wX, a2wX, t2wX
sqrdmulh b6wX, b6wX, z6nX[l6]
add a3wX, a3wX, t3wX
sqrdmulh b7wX, b7wX, z7nX[l7]
mls t4wX, b4wX, $q$
mls t5wX, b5wX, $q$
mls t6wX, b6wX, $q$
mls t7wX, b7wX, $q$
```

doi:10.6342/NTU202200738

Algorithm 59 Interleaved GS butterflies (Cortex-A72) [BHK+22].

Signature:

`wrap_qX_butterfly_mixed_rev a[0-3], b[0-3], t[0-3], a[4-7], b[4-7], t[4-7], {z, l, h}[0-3], {z, l, h}[4-7], wX, nX`

Inputs:

`b[0-3]` $= (a_0 - b_0, \ldots, a_3 - b_3)$,

`(z0[h0],z0[l0])` $= \left(\psi_0, \frac{\left\lfloor \frac{\psi_0 \mathtt{R}}{q} \right\rceil_{2,q}}{2}\right), \ldots,$ `(z3[h3],z3[l3])` $= \left(\psi_3, \frac{\left\lfloor \frac{\psi_3 \mathtt{R}}{q} \right\rceil_{2,q}}{2}\right)$

`a[4-7]` $= (a_4, \ldots, a_7)$,

`t[4-7]` $= (b_4, \ldots, b_7)$

Outputs:

`t[0-3]` $= ((a_0 - b_0)\psi_0, \ldots, (a_3 - b_3)\psi_3)$,

`a[4-7]` $= (a_4 + b_4, \ldots, a_7 + b_7)$,

`b[4-7]` $= (a_4 - b_4, \ldots, a_7 - b_7)$

```
mul t0wX, b0wX, z0nX[h0]
sub b4wX, a4wX, t4wX
mul t1wX, b1wX, z1nX[h1]
sub b5wX, a5wX, t5wX
mul t2wX, b2wX, z2nX[h2]
sub b6wX, a6wX, t6wX
mul t3wX, b3wX, z3nX[h3]
sub b7wX, a7wX, t7wX
sqrdmulh b0wX, b0wX, z0nX[l0]
add a4wX, a4wX, t4wX
sqrdmulh b1wX, b1wX, z1nX[l1]
add a5wX, a5wX, t5wX
sqrdmulh b2wX, b2wX, z2nX[l2]
add a6wX, a6wX, t6wX
sqrdmulh b3wX, b3wX, z3nX[l3]
add a7wX, a7wX, t7wX
mls t0wX, b0wX, $q$
mls t1wX, b1wX, $q$
mls t2wX, b2wX, $q$
mls t3wX, b3wX, $q$
```

doi:10.6342/NTU202200738

Algorithm 60 4 layers of 32-bit CT butterflies over 32 SIMD registers (Cortex-A72) [BHK+22].

Inputs:
Coefficients: (v0, . . . , v15) = ($a_0, \ldots, a_{15}$)
Auxiliary registers: v16, . . . , v19, v28, . . . , v31
Twiddle factors: v20 = h0||l0||0||Q, v21 = h2||l2||h1||l1, v22 = h4||l4||h3||l3, . . . , v27 = h14||l14||h13||l13

Symbols:

```
L0_0  =  v0, v2,  v4,  v6,  v8, v10, v12, v14
L0_1  =  v1, v3,  v5,  v7,  v9, v11, v13, v15
L1_0  =  v0, v2,  v8,  v10, v4, v6,  v12, v14
L1_1  =  v1, v3,  v9,  v11, v5, v7,  v13, v15
L2_0  =  v0, v4,  v8,  v12, v2, v6,  v10, v14
L2_1  =  v1, v5,  v9,  v13, v3, v7,  v11, v15
L3_0  =  v0, v2,  v4,  v6,  v1, v3,  v5,  v7
L3_1  =  v8, v10, v12, v14, v9, v11, v13, v15
T0    = v16, v17, v18, v19
T1    = v28, v29, v30, v31
mod   = v20
W0    = v20, 2, 3, v20, 2, 3, v20, 2, 3, v20, 2, 3
W1    = v21, 0, 1, v21, 0, 1, v21, 2, 3, v21, 2, 3
W2    = v22, 0, 1, v22, 2, 3, v23, 0, 1, v23, 2, 3
W3_0  = v24, 0, 1, v24, 2, 3, v25, 0, 1, v25, 2, 3
W3_1  = v26, 0, 1, v26, 2, 3, v27, 0, 1, v27, 2, 3
```

Outputs: (v0, . . . , v15) = NTT($a_0, \ldots, a_{15}$)

```
qq_butterfly_top L0_1, T0, mod, W0
qq_butterfly_bot L0_1, T0, mod, W0
qq_butterfly_top L0_0, T1, mod, W0
qq_butterfly_bot L0_0, T1, mod, W0
qq_butterfly_top L1_1, T0, mod, W1
qq_butterfly_bot L1_1, T0, mod, W1
qq_butterfly_top L1_0, T1, mod, W1
qq_butterfly_bot L1_0, T1, mod, W1
qq_butterfly_top L2_1, T0, mod, W2
qq_butterfly_bot L2_1, T0, mod, W2
qq_butterfly_top L2_0, T1, mod, W2
qq_butterfly_bot L2_0, T1, mod, W2
qq_butterfly_top L3_0, T0, mod, W3_0
qq_butterfly_bot L3_0, T0, mod, W3_0
qq_butterfly_top L3_1, T1, mod, W3_1
qq_butterfly_bot L3_1, T1, mod, W3_1
```

doi:10.6342/NTU202200738

Algorithm 61 4 layers of interleaved 32-bit CT butterflies over 32 SIMD registers (Cortex-A72) [BHK+22].

Inputs:

Coefficients: $(\texttt{v0}, \ldots, \texttt{v15}) = (a_0, \ldots, a_{15})$

Auxiliary registers: $\texttt{v16}, \ldots, \texttt{v19}$, $\texttt{v28}, \ldots, \texttt{v31}$

Twiddle factors: $\texttt{v20} = \texttt{h0}||\texttt{l0}||\texttt{0}||\texttt{Q}$, $\texttt{v21} = \texttt{h2}||\texttt{l2}||\texttt{h1}||\texttt{l1}$, $\texttt{v22} = \texttt{h4}||\texttt{l4}||\texttt{h3}||\texttt{l3}$, $\ldots$, $\texttt{v27} = \texttt{h14}||\texttt{l14}||\texttt{h13}||\texttt{l13}$

Symbols:

```
L0_0 =  v0, v2,  v4,  v6,  v8, v10, v12, v14
L0_1 =  v1, v3,  v5,  v7,  v9, v11, v13, v15
L1_0 =  v0, v2,  v8,  v10, v4, v6,  v12, v14
L1_1 =  v1, v3,  v9,  v11, v5, v7,  v13, v15
L2_0 =  v0, v4,  v8,  v12, v2, v6,  v10, v14
L2_1 =  v1, v5,  v9,  v13, v3, v7,  v11, v15
L3_0 =  v0, v2,  v4,  v6,  v1, v3,   v5,  v7
L3_1 =  v8, v10, v12, v14, v9, v11, v13, v15
T0 = v16, v17, v18, v19
T1 = v28, v29, v30, v31
mod = v20
W0   = v20, 2, 3, v20, 2, 3, v20, 2, 3, v20, 2, 3
W1   = v21, 0, 1, v21, 0, 1, v21, 2, 3, v21, 2, 3
W2   = v22, 0, 1, v22, 2, 3, v23, 0, 1, v23, 2, 3
W3_0 = v24, 0, 1, v24, 2, 3, v25, 0, 1, v25, 2, 3
W3_1 = v26, 0, 1, v26, 2, 3, v27, 0, 1, v27, 2, 3
```

Outputs: $(\texttt{v0}, \ldots, \texttt{v15}) = \mathsf{NTT}(a_0, \ldots, a_{15})$

```
qq_butterfly_top L0_1, T0, mod, W0
qq_butterfly_mixed L0_1, T0, L0_0, T1, mod, W0, W0
qq_butterfly_mixed L0_0, T1, L1_1, T0, mod, W0, W1
qq_butterfly_mixed L1_1, T0, L1_0, T1, mod, W1, W1
qq_butterfly_mixed L1_0, T1, L2_1, T0, mod, W1, W2
qq_butterfly_mixed L2_1, T0, L2_0, T1, mod, W2, W2
qq_butterfly_mixed L2_0, T1, L3_0, T0, mod, W2, W3_0
qq_butterfly_mixed L3_0, T0, L3_1, T1, mod, W3_0, W3_1
qq_butterfly_bot L3_1, T1, mod, W3_1
```

doi:10.6342/NTU202200738

A.7 Wrapping Macros for Cortex-A72

Parallel add-sub pairs, reductions and multiplications.
`qq`-type.

- `qq_montgomery_mul`: `wrap_qX_montgomery_mul` with `.4S, .S`
- `qq_montgomery`: `wrap_qX_montgomery` with `.2S, .4S, .2D`
- `qq_sub_add`: `wrap_qX_sub_add` with `.4S`
- `qo_barrett`: `wrap_qX_barrett` with `.8H, .H`
- `oo_barrett`: `wrap_oX_barrett` with `.8H, .H`
- `qo_barrett_vec`: `wrap_qo_barrett_vec` with `.8H, .H`
- `oo_barrett_vec`: `wrap_oo_barrett_vec` with `.8H, .H`

Parallel vector-scalar butterflies.
`qo`-type.

- `qo_butterfly_top`: `wrap_qX_butterfly_top` with `.8H, .H`
- `qo_butterfly_bot`: `wrap_qX_butterfly_bot` with `.8H, .H`
- `qo_butterfly_mixed`: `wrap_qX_butterfly_mixed` with `.8H, .H`
- `qo_butterfly_mixed_rev`: `wrap_qX_butterfly_mixed_rev` with `.8H, .H`

`dq`-type.

- `dq_butterfly_top`: `wrap_dX_butterfly_top` with `.4S, .S`
- `dq_butterfly_bot`: `wrap_dX_butterfly_bot` with `.4S, .S`
- `dq_butterfly_mixed`: `wrap_dX_butterfly_mixed` with `.4S, .S`
- `dq_butterfly_mixed_rev`: `wrap_dX_butterfly_mixed_rev` with `.4S, .S`

`qq`-type.

- `qq_butterfly_top`: `wrap_qX_butterfly_top` with `.4S, .S`
- `qq_butterfly_bot`: `wrap_qX_butterfly_bot` with `.4S, .S`
- `qq_butterfly_mixed`: `wrap_qX_butterfly_mixed` with `.4S, .S`
- `qq_butterfly_mixed_rev`: `wrap_qX_butterfly_mixed_rev` with `.4S, .S`

Parallel vector-vector butterflies.
`do`-type.

- `do_butterfly_vec_top`: `wrap_dX_butterfly_vec_top` with `.8H, .H`

doi:10.6342/NTU202200738

- do_butterfly_vec_bot: wrap_dX_butterfly_vec_bot with .8H, .H
- do_butterfly_vec_mixed: wrap_dX_butterfly_vec_mixed with .8H, .H
- do_butterfly_vec_mixed_rev: wrap_dX_butterfly_vec_mixed_rev with .8H, .H

dq-type.

- dq_butterfly_vec_top: wrap_dX_butterfly_vec_top with 4S, .S
- dq_butterfly_vec_bot: wrap_dX_butterfly_vec_bot with 4S, .S
- dq_butterfly_vec_mixed: wrap_dX_butterfly_vec_mixed with 4S, .S
- dq_butterfly_vec_mixed_rev: wrap_dX_butterfly_vec_mixed_rev with 4S, .S

doi:10.6342/NTU202200738

A.8 Permutation on Cortex-A72

Algorithm 62 Permutation of the bottom 3 layers of Kyber NTT (Cortex-A72) [BHK+22].

Inputs:

$$\begin{aligned} *\mathtt{src0} &= (\mathtt{a0a1}, \mathtt{a2a3}, \ldots, \mathtt{a30a31}) \\ *\mathtt{src1} &= (\mathtt{a64a65}, \mathtt{a66a67}, \ldots, \mathtt{a94a95}) \end{aligned}$$

Outputs:

$$\begin{aligned} \mathtt{v24} &= (\mathtt{a0a1}, \mathtt{a64a65}, \mathtt{a16a17}, \mathtt{a80a81}) \\ \mathtt{v25} &= (\mathtt{a2a3}, \mathtt{a66a67}, \mathtt{a18a19}, \mathtt{a82a83}) \\ \mathtt{v26} &= (\mathtt{a4a5}, \mathtt{a68a69}, \mathtt{a20a21}, \mathtt{a84a85}) \\ \mathtt{v27} &= (\mathtt{a6a7}, \mathtt{a70a71}, \mathtt{a22a23}, \mathtt{a86a87}) \\ \mathtt{v28} &= (\mathtt{a8a9}, \mathtt{a72a73}, \mathtt{a24a25}, \mathtt{a88a89}) \\ \mathtt{v29} &= (\mathtt{a10a11}, \mathtt{a74a75}, \mathtt{a26a27}, \mathtt{a90a91}) \\ \mathtt{v30} &= (\mathtt{a12a13}, \mathtt{a76a77}, \mathtt{a28a29}, \mathtt{a92a93}) \\ \mathtt{v31} &= (\mathtt{a14a15}, \mathtt{a78a79}, \mathtt{a30a31}, \mathtt{a94a95}) \end{aligned}$$

```
ld4 {v16.4S, v17.4S, v18.4S, v19.4S}, [src0]
ld4 {v20.4S, v21.4S, v22.4S, v23.4S}, [src1]
trn1 v24.4S, v16.4S, v20.4S
trn2 v28.4S, v16.4S, v20.4S
trn1 v25.4S, v17.4S, v21.4S
trn2 v29.4S, v17.4S, v21.4S
trn1 v26.4S, v18.4S, v22.4S
trn2 v30.4S, v18.4S, v22.4S
trn1 v27.4S, v19.4S, v23.4S
trn2 v31.4S, v19.4S, v23.4S
```

doi:10.6342/NTU202200738

Appendix B

Toolkit for Development

This chapter docements the C functions developed during the development of the selected works. There are four header files and four source files.

- Section B.1 describes `tools.h` and `tools.c`.
  - `tools.h` provides the functions defining the coefficient rings, and the C structure `struct compress_profile` enabling flexible merging strategies.
  - There is no dependencies except for the C standard library.
- Section B.2 describes `naive_mult.h` and `naive_mult.c`.
  - `naive_mult.h` provides functions for convolutions and polynomial-by-coefficient multiplications.
  - There is no dependencies except for the C standard library.
- Section B.3 describes `gen_table.h` and `gen_table.c`.
  - `gen_table.h` provides functoins for generating tables of twiddle factors.
  - Two files are required:
    - `tools.h`.
    - `NTT_params.h`. We require the following symbols to be defined:
      - `NTT_N`.
      - `LOGNTT_N`.
- Section B.4 describes `ntt_c.h` and `ntt_c.c`.
  - `ntt_c.h` provides functions computing NTTs.
  - Two files are required:
    - `tools.h`.
    - `NTT_params.h`. We require the following symbols to be defined:
      - `NTT_N`.
      - `LOGNTT_N`.
      - `ARRAY_N`.

doi:10.6342/NTU202200738

B.1 Basic Constructs

This section describes the C functions declared in tools.h and defined in tools.c

B.1.1 struct compress_profile

We define a C structure struct compress_profile as a control block containing information of layer merging. In a struct compress_profile, there are a variable compressed_layers and an array merged_layers[16]. compressed_layers indicates the number of layers after the merging. The array merged_layers then specifies for each compressed layers, how many layers are merged. Only the first compressed_layers elements of merged_layers will be used.

Listing B.1: struct compress_profile.

```
struct compress_profile{
    size_t compressed_layers;
    size_t merged_layers[16];
};
```

B.1.2 Coefficient Ring Operations

We define some functions realizing the additions, and multiplications. For convenience, we define functions realizing the subtractions and exponentiations.

We define the reductions to $\mathbb{Z}_{\texttt{*mod}}$. cmod_{type} reduces the value *src to the representation in $\mathbb{Z}_{\texttt{*mod}}$.

Algorithm 63 cmod_{type}(void *des, void *src, void *mod)

```
$\texttt{*des} = \texttt{*src} \bmod {}^{\pm}\texttt{*mod}$
```

For convenience, reductions with data types int16_t, int32_t, and int64_t are provided.

Listing B.2: cmod_int16, cmod_int32, and cmod_int64.

```
void cmod_int16(void *des, void *src, void *mod);
void cmod_int32(void *des, void *src, void *mod);
void cmod_int64(void *des, void *src, void *mod);
```

We define the additions with reductions to $\mathbb{Z}_{\texttt{*mod}}$. addmod_{type} adds the values *src1 and *src2 and reduces the result to the representation in $\mathbb{Z}_{\texttt{*mod}}$.

Algorithm 64 addmod_{type}(void *des, void *src1, void *src2, void *mod)

```
$\texttt{*des} = ((\texttt{*src1}) + (\texttt{*src2})) \bmod {}^{\pm}\texttt{*mod}$
```

For convenience, additions with reductions of data types int16_t and int32_t are provided.

Listing B.3: addmod_int16 and addmod_int32.

```
void addmod_int16(void *des,
    void *src1, void *src2, void *mod);
void addmod_int32(void *des,
    void *src1, void *src2, void *mod);
```

doi:10.6342/NTU202200738

We define the subtractions with reductions to $\mathbb{Z}_{\texttt{*mod}}$. `submod_{type}` subtract `*src2` from `*src1` and reduces the result to the representation in $\mathbb{Z}_{\texttt{*mod}}$.

Algorithm 65 `submod_{type}(void *des, void *src1, void *src2, void *mod)`

$\texttt{*des} = ((\texttt{*src1}) - (\texttt{*src2})) \bmod {}^{\pm}\texttt{*mod}$

For convenience, subtractions with reductions of data types `int16_t` and `int32_t` are provided.

Listing B.4: `submod_int16` and `submod_int32`.

```
void submod_int16(void *des,
    void *src1, void *src2, void *mod);
void submod_int32(void *des,
    void *src1, void *src2, void *mod);
```

We define the multiplications with reductions to $\mathbb{Z}_{\texttt{*mod}}$. `mulmod_{type}` multiplies `*src1` and `*src2` and reduces the result to the representation in $\mathbb{Z}_{\texttt{*mod}}$.

Algorithm 66 `mulmod_{type}(void *des, void *src1, void *src2, void *mod)`

$\texttt{*des} = ((\texttt{*src1}) \cdot (\texttt{*src2})) \bmod {}^{\pm}\texttt{*mod}$

For convenience, multiplications with reductions of data types `int16_t` and `int32_t` are provided.

Listing B.5: `mulmod_int16` and `mulmod_int32`.

```
void mulmod_int16(void *des,
    void *src1, void *src2, void *mod);
void mulmod_int32(void *des,
    void *src1, void *src2, void *mod);
```

We define the exponentiations with reductions to $\mathbb{Z}_{\texttt{*mod}}$. `expmod_{type}` computes $(\texttt{*src} \bmod {}^{\pm}\mathbb{Z}_{\texttt{*mod}})^{\texttt{e}}$ and reduces the result to the representation in $\mathbb{Z}_{\texttt{*mod}}$.

Algorithm 67 `expmod_{type}(void *des, void *src, size_t e, void *mod)`

$\texttt{*des} = (\texttt{*src} \bmod {}^{\pm}\mathbb{Z}_{\texttt{*mod}})^{\texttt{e}} \bmod {}^{\pm}\texttt{*mod}$

For convenience, exponentiations with reductions of data types `int16_t` and `int32_t` are provided.

Listing B.6: `expmod_int16` and `expmod_int32`.

```
void expmod_int16(void *des,
    void *src, size_t e, void *mod);
void expmod_int32(void *des,
    void *src, size_t e, void *mod);
```

B.1.3 Permutation

Additionally, we provide the functin `bitreverse` for applying in-place bit-reversal.

Listing B.7: `bitreverse`.

```
void bitreverse(void *src, size_t len, size_t size);
```

doi:10.6342/NTU202200738

B.2 Näive Polynomial Multiplications

This section describes the functions declared in `naive_mult.h` and defined in `naive_mult.c`.

`naive_mulR` computes the convolution of two polynomials. In other words, given two polynomials `src1` and `src2` of length `len`, we compute

$$\texttt{src1[0 - (len - 1)]} \cdot \texttt{src2[0 - (len - 1)]} \bmod (x^{\texttt{len}} - 1).$$

The result is written to the array `des`.

Listing B.8: `naive_mulR`.

```
void naive_mulR(
    void *des,
    void *src1, void *src2,
    size_t len, void *twiddle,
    void *mod,
    size_t size,
    void (*addmod)(void *_des,
        void *_src1, void *_src2, void *_mod),
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod)
    );
```

Additionally, we provide the function `point_mul` for point-wise multiplications. Given a size-`len`$*$`jump` polynomial `src1` and a size-`len` polynomial `src2`, we compute

$$\texttt{src1}[i * \texttt{jump} + j] * \texttt{src2}[i]$$

for each $i = 0, \ldots, \texttt{len} - 1$ and $j = 0, \ldots, \texttt{jump} - 1$.

Listing B.9: `point_mul`.

```
void point_mul(
    void *des,
    void *src1, void *src2,
    size_t len, size_t jump,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod)
    );
```

doi:10.6342/NTU202200738

B.3 Generating Twiddle Factors

This section describes the functions declared in `gen_table.h` for generating tables of twiddle factors. Below are the required symbols and function types.

- `NTT_params.h`
  - `NTT_N`
  - `LOGNTT_N`
- `tools.h`
  - `void (*mulmod)(void*, void*, void*, void*)`
  - `void (*expmod)(void*, void*, size_t, void*)`
  - `struct compress_profile`

B.3.1 Generating Basic Tables

We first describe functions that do not depend on `struct compress_profiles`.

`gen_CT_table_generic` generates the twiddle factors

$$\left((\texttt{*scale}) \cdot (\texttt{*omega})^{\mathrm{rev}_{(2:(\texttt{LOGNTT\_N}-1))}(i)}\right)$$

for $i = 0, \ldots, \frac{\texttt{NTT\_N}}{2} - 1$.

Listing B.10: `gen_CT_table_generic`.

```
void gen_CT_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod)
    );
```

`gen_CT_negacyclic_table_generic` generates the twiddle factors

$$\left((\texttt{*scale}) \cdot (\texttt{*omega})^{\mathrm{rev}_{(2:\texttt{LOGNTT\_N})}(i)}\right)$$

for $i = 1, \ldots, \texttt{NTT\_N} - 1$.

Listing B.11: `gen_CT_negacyclic_table_generic`.

```
void gen_CT_negacyclic_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod)
    );
```

`gen_inv_CT_table_generic` generates the twiddle factors

$$\left((\texttt{*scale}) \cdot (\texttt{*omega})^{j_i 2^{\frac{\texttt{LOGNTT\_N}}{i+1}}}\right)$$

where each $j_i = 0, \ldots, 2^i - 1$ and $i = 0, \ldots, \texttt{LOGNTT\_N} - 1$.

doi:10.6342/NTU202200738

Listing B.12: `gen_inv_CT_table_generic`.

```
void gen_inv_CT_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod),
    void (*expmod)(void *_des,
        void *_src, size_t _size, void *_mod)
    );
```

`gen_twist_table_generic` generates the twiddle factors

$$((\texttt{*scale}) \cdot (\texttt{*omega})^i)$$

for $i = 0, \ldots, \texttt{NTT\_N} - 1$.

Listing B.13: `gen_twist_table_generic`.

```
void gen_twist_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod)
    );
```

`gen_mul_table_generic` generates the twiddle factors

$$((\texttt{*scale}) \cdot (\texttt{*omega})^i)$$

for $i = 0, \ldots, \frac{\texttt{NTT\_N}}{2} - 1$.

Listing B.14: `gen_mul_table_generic`.

```
void gen_mul_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *mod)
    );
```

B.3.2 Generating Streamlined Tables

Now we describe functions that depend on `struct compress_profile`.

`gen_streamlined_CT_table_generic` generates the twiddle factors for Cooley–Tukey FFT for $\mathsf{NTT}_{R[x]/\langle x^{\texttt{NTT\_N}}-1\rangle:\texttt{*omega}}$. The twiddle factors are reordered according to `*_profile`.

Listing B.15: `gen_streamlined_CT_table_generic`.

```
void gen_streamlined_CT_table_generic(
    void *des,
    void *scale, void *omega,
```

doi:10.6342/NTU202200738

```
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod),
    struct compress_profile *_profile, bool pad
    );
```

gen_streamlined_CT_negacyclic_table_generic generates the twiddle factors for Cooley–Tukey FFT for $\mathsf{NTT}_{R[x]/\langle x^{\texttt{NTT\_N}}-(\texttt{*omega})^{\texttt{NTT\_N}}\rangle:(\texttt{*omega})^2}$. The twiddle factors are reordered according to *_profile.

Listing B.16: gen_streamlined_CT_negacyclic_table_generic.

```
void gen_streamlined_CT_negacyclic_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod),
    struct compress_profile *_profile, bool pad
    );
```

gen_streamlined_inv_CT_table_generic generates the twiddle factors for Cooley–Tukey FFT for $\mathsf{NTT}^{-1}_{R[x]/\langle x^{\texttt{NTT\_N}}-1\rangle:(\texttt{*omega})^{-1}}$. The twiddle factors are reordered according to *_profile. Notice that an inverted principal NTT_N-th root of unity is passed to the function.

Listing B.17: gen_streamlined_inv_CT_table_generic.

```
void gen_streamlined_inv_CT_table_generic(
    void *des,
    void *scale, void *omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod),
    void (*expmod)(void *_des,
        void *_src, size_t e, void *_mod),
    struct compress_profile *_profile, bool pad
    );
```

gen_streamlined_inv_CT_nagecyclic_table_generic generates the twiddle factors for Cooley–Tukey FFT for $\mathsf{NTT}^{-1}_{R[x]/\langle x^{\texttt{NTT\_N}}-(\texttt{*twist\_omega})^{-\texttt{NTT\_N}}\rangle:(\texttt{*omega})^{-2}}$. The twiddle factors are reordered according to *_profile. Notice that an inverted principal NTT_N-th root of unity and an NTT_N-th root for twisting are passed to the function.

Listing B.18: gen_streamlined_inv_CT_negacyclic_table_generic.

```
void gen_streamlined_inv_CT_negacyclic_table_generic(
    void *des,
    void *scale1, void *omega,
    void *scale2, void *twist_omega,
    void *mod,
    size_t size,
    void (*mulmod)(void *_des,
        void *_src1, void *_src2, void *_mod),
    void (*expmod)(void *_des,
        void *_src, size_t e, void *_mod),
    struct compress_profile *_profile, bool pad
```

doi:10.6342/NTU202200738

```
);
```

doi:10.6342/NTU202200738

B.4 Portable NTTs

This section describes the functions declared in ntt_.h and defined in ntt_c.c. We require the following symbols.

- ARRAY_N.
- LOGNTT_N.
- NTT_N.

B.4.1 Cooley–Tukey Butterflies

We first describe the basic building blocks of Cooley–Tukey FFTs. This section requires the following functions for defining commutative rings.

- void (*addmod)(void*, void*, void*, void*)
- void (*submod)(void*, void*, void*, void*)
- void (*mulmod)(void*, void*, void*, void*)

Based on the functions defining commutative rings, CT_butterfly_generic computes the Cooley–Tukey butterfly defined on the coefficients src[indx_a] and src[indx_b], and the twiddle factor *twiddle. In other words, we compute

$$\begin{cases} \texttt{src[indx\_a]} = \texttt{src[indx\_a]} + (\texttt{*twiddle}) \cdot \texttt{src[indx\_b]}, \\ \texttt{src[indx\_b]} = \texttt{src[indx\_a]} - (\texttt{*twiddle}) \cdot \texttt{src[indx\_b]}. \end{cases}$$

Listing B.19: CT_butterfly_generic.

```
void CT_butterfly_generic(
    void *src,
    size_t indx_a, size_t indx_b,
    void *twiddle,
    void *mod,
    size_t size,
    void (*addmod)(void *des,
        void *src1, void *src2, void *mod),
    void (*submod)(void *des,
        void *src1, void *src2, void *mod),
    void (*mulmod)(void *des,
        void *src1, void *src2, void *mod)
    );
```

For convenience, we provide Cooley–Tukey butterflies with the data types int16_t and int32_t.

Listing B.20: CT_butterfly_int16.

```
void CT_butterfly_int16(
    void *src,
    size_t indx_a, size_t indx_b,
    void *twiddle,
    void *mod,
    size_t size
    );
```

doi:10.6342/NTU202200738

Listing B.21: `CT_butterfly_int32`.

```
void CT_butterfly_int32(
    void *src,
    size_t indx_a, size_t indx_b,
    void *twiddle,
    void *mod,
    size_t size
    );
```

B.4.2 Multi-Layer Cooley–Tukey Butterflies

This section describes the functions implementing multi-layer Cooley–Tukey butterflies. We require the following function.

- `void (*CT_butterfly)(void*, size_t, size_t, void*, void*, size_t)`.

`m_layer_CT_butterfly_generic` computes `layers` Cooley–Tukey butterflies defined on the coefficients

$$\texttt{src[0]}, \texttt{src[step]}, \ldots, \texttt{src[}(2^{\texttt{layers}} - 1) \cdot \texttt{step]}.$$

The table of twiddle factors `_root_table` is in the streamlined order.

Listing B.22: `m_layer_CT_butterfly_generic`.

```
void m_layer_CT_butterfly_generic(
    void *src,
    size_t layers, size_t step,
    void *_root_table,
    void *mod,
    size_t size,
    void (*CT_butterfly)(
        void *src,
        size_t indx_a, size_t indx_b,
        void *twiddle,
        void *mod,
        size_t size
        )
    );
```

For convenience, we provide functions with data types `int32_t` and `int16_t`.

Listing B.23: `m_layer_CT_butterfly_int16`.

```
void m_layer_CT_butterfly_int16(
    void *src,
    size_t layers, size_t step,
    void *_root_table,
    void *mod,
    size_t size
    );
```

Listing B.24: `m_layer_CT_butterfly_int32`.

```
void m_layer_CT_butterfly_int32(
    void *src,
    size_t layers, size_t step,
    void *_root_table,
```

doi:10.6342/NTU202200738

```
    void *mod,
    size_t size
    );
```

Next, we provide functions for computing the inverse of radix-2 Cooley–Tukey FFTs with Cooley–Tukey butterflies. `m_layer_CT_ibutterfly_generic` computes `layers` Cooley–Tukey butterflies defined on the coefficients

$$\texttt{src[0]}, \texttt{src[step]}, \ldots, \texttt{src[}(2^{\texttt{layers}} - 1) \cdot \texttt{step]}.$$

Note that the input coefficients are assumed in the bit-reversed order. The table of twiddle factors `_root_table` is in the streamlined order.

Listing B.25: `m_layer_CT_ibutterfly_generic`.

```
void m_layer_CT_ibutterfly_generic(
    void *src,
    size_t layers, size_t step,
    void *_root_table,
    void *mod,
    size_t size,
    void (*CT_butterfly)(
        void *src,
        size_t indx_a, size_t indx_b,
        void *twiddle,
        void *mod,
        size_t size
        )
    );
```

For convenience, we provide functions with the data type `int32_t`.

Listing B.26: `m_layer_CT_ibutterfly_int32`.

```
void m_layer_CT_ibutterfly_int32(
    void *src,
    size_t layers, size_t step,
    void *_root_table,
    void *mod,
    size_t size
    );
```

B.4.3 Cooley–Tukey FFTs

Finally, this section describes functions computing NTTs and NTT^{-1}s. The functions access the memory exactly as what is planned to be implemented in assembly. To facilitate the development of assembly programs, the C functions provide separable calls for NTTs and NTT^{-1}s. A common use case is that one can replace a multi-layer computation with assembly implementations for testing their efficiency in a timely manner.

The basic building blocks of generic NTTand NTT^{-1}are the following.

- `void (*m_layer_CT_butterfly)(void*, size_t, size_t, void*, void*, size_t)`.

- `void (*m_layer_CT_ibutterfly)(void*, size_t, size_t, void*, void*, size_t)`.

doi:10.6342/NTU202200738

We provide the following functions for computing NTTand NTT^{-1}.

Listing B.27: compressed_CT_NTT_generic.

```
void compressed_CT_NTT_generic(
    void *src,
    size_t start_level, size_t end_level,
    void *_root_table,
    void *mod,
    struct compress_profile *_profile,
    size_t size,
    void (*m_layer_CT_butterfly)(
        void *src,
        size_t layers, size_t step,
        void *_root_table,
        void *mod,
        size_t size
        )
    );
```

Listing B.28: compressed_CT_iNTT_generic.

```
void compressed_CT_iNTT_generic(
    void *src,
    size_t start_level, size_t end_level,
    void *_root_table,
    void *mod,
    struct compress_profile *_profile,
    size_t size,
    void (*m_layer_CT_ibutterfly)(
        void *src,
        size_t layers, size_t step,
        void *_root_table,
        void *mod,
        size_t size
        )
    );
```

compressed_CT_NTT_generic computes NTTand compressed_CT_iNTT_generic computes NTT^{-1}according to *_profile. As long as the tables of twiddle factors are generated with the same *_profile or the same merging strategy for the same layers, the result will be correct.

The _root_table in compressed_CT_NTT_generic should be generated by the following functions:

- gen_streamlined_CT_table_generic, and
- gen_streamlined_CT_negacyclic_table_generic.

The _root_table in compressed_CT_iNTT_generic should be generated by the following function:

- gen_streamlined_inv_CT_table_generic.

doi:10.6342/NTU202200738