Cortex-M4 Optimizations for {R,M}LWE Schemes*

Erdem Alkim[1,2], Yusuf Alper Bilgin[3,4], Murat Cenk[4], and François Gérard[5]

[1] Department of Computer Engineering, Ondokuz Mayıs University, Samsun, Turkey, erdemalkim@gmail.com
[2] Fraunhofer SIT, Darmstadt, Germany
[3] Aselsan Inc., Ankara, Turkey, y.alperbilgin@gmail.com
[4] Institude of Applied Mathematics, Middle East Technical University, Ankara, Turkey, mcenk@metu.edu.tr
[5] Université libre de Bruxelles, Brussels, Belgium, fragerar@ulb.ac.be

Abstract. This paper proposes various optimizations for lattice-based key-encapsulation mechanisms (KEM) using the Number Theoretic Transform (NTT) on the popular ARM Cortex-M4 microcontroller. Improvements come in the form of a faster code using more efficient modular reductions, small polynomial multiplications and more aggressive layer merging in the NTT but also reduced stack usage. We test those optimizations in software implementations of Kyber and NewHope, both round 2 candidates in the NIST post-quantum project and also NewHope-Compact, a recently proposed derivative of NewHope with smaller parameters. Our software is the first implementation of NewHope-Compact on Cortex-M4 and shows speed improvements over previous high-speed implementations on the same platform for Kyber and NewHope. Moreover, it gives a common framework to compare those algorithms with the same level of optimization. Our results show that NewHope-Compact is the faster algorithm, followed by Kyber and finally NewHope that seems to suffer from its large modulus and error distribution for small dimensions.

1 Introduction

Post-quantum cryptography, i.e., cryptography resisting adversaries equipped with both classical and quantum computers, has grown significantly among the research community in the last few years. This growth is partially driven by the NIST post-quantum standardization project aiming to create a formal environment in which concrete instantiations of several post-quantum techniques for signature and key encapsulation can be analyzed and compared to each other

* The work of EA was partially supported by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity, and was partially carried out during his tenure of the ERCIM 'Alain Bensoussan' Fellowship Programme.

with respect to several metrics. The first round of the project took place mainly during the year 2018 with the goal of assessing the different possible quantum-safe algorithms. In early 2019, 26 out of the 69 initial algorithms advanced to the second round of the project. In the meantime, it has been announced that, in this second round, practical performances of the candidates will play a more important role in the selection for a hypothetical future standardization.
While part of the candidates already had an optimized version of their code targeting instruction sets of large CPU such as AVX2, such piece of software often live in an environment where they are not the bottleneck in term of computation time or memory usage. On the other hand, small embedded devices are the ones that risk to be the most impacted by the switch toward a post-quantum paradigm. Indeed, they often offer low memory, low computing power and even have the drawback to be more exposed to side-channel attacks. Hence, in recent papers, researchers focused more and more on small devices such has the popular ARM Cortex-M4 to assess performances in an embedded world. This microcontroller has the advantage of being large enough to support public-key algorithms while being still reasonably small and cheap in the grand scheme of computing. From its popularity was born a library called `pqm4` [16] aiming to offer a common framework to benchmark implementations of post-quantum algorithms on this platform.

Contributions: In this paper, we describe an optimized Cortex-M4 implementation of NewHope, Kyber, and a recently proposed derivative of those schemes called NewHope-Compact. We present various optimizations, mainly in terms of speed and stack usage on the ARM microcontroller. Since those schemes naturally share structural similarities, general improvements are applicable to all of them. Our implementation outperforms the current state-of-the art for Kyber and NewHope and gives a unified framework to compare the three schemes since they use the same level of optimization, which was not the case in previous works [16]. Our contributions are listed as follows:

- We propose 2 cycles modular reduction implementation for Montgomery arithmetic, which translates subtraction to addition to be able to use special instructions.
- We show that small polynomial multiplications can be implemented efficiently using lazy reduction techniques. Hence, we show that early termination of NTT can be implemented more efficiently when the base multiplication has degree more than 2.
- We show that even the target architecture has only 13 usable registers, 16 coefficients can be used during butterfly layers. This allowed us to merge up to 4 layers of the NTT and reduce the number of load and store instructions.
- We provide trade-offs between stack usage and speed of the implementation for computing addition of two polynomials after an NTT based polynomial multiplication.

Availability of the software: All source codes are available at `https://github.com/erdemalkim/NewHope-Compact-M4`

Organization of this paper: Section 2 gives a background information on key encapsulation schemes NewHope, NewHope-Compact, and Kyber. It also describes the recent advances on NTT, and the reduction algorithms mostly preferred for constant time and efficient implementation of reductions inside the NTT. Section 3 provides our implementation details and optimizations to achieve a faster implementation while using as few as possible stack space. It also gives a tradeoff between secret key size and speed. Finally, our performance results for NewHope, NewHope-Compact, and Kyber and a comparison with previous implementations of those schemes are presented in Section 4.

2 Preliminaries

2.1 Notation

Let $\mathcal{R}$ be the ring of integer polynomials modulo $X^n + 1$ and denoted as $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$. Then, we define $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ as a special case of $\mathcal{R}$ such that every coefficient is reduced modulo q. Note that n and q are selected in such a way that performing an efficient NTT to an element in $\mathcal{R}_q$ is possible. For that reason, n is selected as a power of 2 such that $X^n + 1$ is the $2n$-th cyclotomic polynomial and q is selected as a prime. We represent an element $a \in \mathcal{R}_q$ as $a = \sum_{i=0}^{n-1} a_i X^i$ where a_i is in $\mathbb{Z}_q$. Moreover, the bold lower-case letters such as $\mathbf{v}$ denote a column vector and the bold upper-case letters such as $\mathbf{A}$ denote a matrix with entries in $\mathcal{R}_q$. The representations of polynomials, vectors and matrices in NTT domain are denoted as $\hat{a}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{A}}$, respectively.

2.2 NewHope

NewHope [1,4] is one of the NIST post-quantum standardization candidates whose security is based on the hardness of solving ring learning with errors (RLWE) problem [19,20]. This cryptosystem includes both adaptive chosen plaintext attacks (CPA) secure key encapsulation mechanism (KEM), referred to as NewHope-CPA-KEM, and adaptive chosen ciphertext attacks (CCA) secure KEM, referred to as NewHope-CCA-KEM, which are based on previously proposed NewHope-Simple [3] designed as semantically secure public-key encryption (PKE) scheme with respect to CPA that is referred to as NewHope-CPA-PKE. Key generation, encryption, and decryption functions of NewHope-CPA-PKE are represented in Algorithm 1, 2, and 3. The constructions of NewHope-CPA-KEM and NewHope-CCA-KEM by using NewHope-CPA-PKE are out of the scope of this work, so we refer to [1, Alg. 16-21] for more information.

The most time-consuming parts of all learning with errors (LWE) variant cryptosystems are the randomness generation and hashing. These are vastly used inside GenA and Sample functions. Moreover, CPA-KEM and CCA-KEM constructions also require hashing. Note that encode, decode, compress and decompress functions perform bits or bytes manipulation. Thus, they do not cost

Algorithm 1 NewHope-CPA-PKE key generation

Output: public key $pk = (\hat{b}', \rho)$
Output: secret key $sk = \hat{s}$

1: $seed \xleftarrow{\$} \{0, \cdots, 255\}^{32}$
2: $\rho, \sigma \leftarrow \mathsf{SHAKE256}(64, seed)$
3: $\hat{a} \leftarrow \mathsf{GenA}(\rho)$
4: $s \leftarrow \mathsf{Sample}(\sigma, 0)$
5: $e \leftarrow \mathsf{Sample}(\sigma, 1)$
6: $\hat{b} \leftarrow \hat{a} \circ \mathsf{NTT}(s) + \mathsf{NTT}(e)$
7: **return** $pk = (\hat{b}, \rho), sk = \hat{s}$

Algorithm 2 NewHope-CPA-PKE encryption

Input: public key $pk = (\hat{b}, \rho)$
Input: message μ encoded in $\mathcal{R}_q$
Input: seed $coin \in \{0, \cdots, 255\}^{32}$
Output: ciphertext $(\hat{u}', h)$

1: $\hat{a} \leftarrow \mathsf{GenA}(\rho)$
2: $s' \leftarrow \mathsf{Sample}(coin, 0)$
3: $e' \leftarrow \mathsf{Sample}(coin, 1)$
4: $e'' \leftarrow \mathsf{Sample}(coin, 2)$
5: $\hat{t} \leftarrow \mathsf{NTT}(s')$
6: $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \mathsf{NTT}(e')$
7: $v' \leftarrow \mathsf{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + \mu$
8: **return** $c = (\hat{u}, \mathsf{Compress}(v'))$

Algorithm 3 Decryption of NewHope-CPA-PKE decryption

Input: ciphertext $c = (\hat{u}, h)$
Input: secret key $sk = \hat{s}$
Output: message $\mu \in \{0, \cdots, 255\}^{32}$

1: $v' \leftarrow \mathsf{Decompress}(h)$
2: **return** $\mu = \mathsf{Decode}(v' - \mathsf{NTT}^{-1}(\hat{u} \circ \hat{s}))$

much. Apart from these operations, the main cost of NewHope is multiplication in $\mathcal{R}_q$. NewHope selects its parameters n and q to enable a fast polynomial multiplication in $\mathcal{R}_q$ by utilizing NTT. The parameter sets are provided in Table 1. As can be seen, q is selected such that $q \equiv 1 \pmod{2n}$ so that n-th root of unity ω and $2n$-th root of unity $\gamma = \sqrt{\omega}$ exist, and a fast NTT is possible.

Table 1. Parameters of NewHope512 and NewHope1024 and derived high level properties [1]

Parameter Set	NewHope512	NewHope1024
Dimension n	512	1024
Modulus q	12289	12289
Noise Parameter k	8	8
NTT parameter γ	10968	7
Decryption error probability	2^{-213}	2^{-216}
Claimed post-quantum bit security	101	233
NIST Security Strength Category	1	5

Polynomial Multiplication Utilizing NTT: The forward NTT is performed to transform all of the coefficients to NTT domain, and the inverse of this operation

NTT^{-1} is performed to carry all coefficents to normal domain again. The formulae for these two operations are given as follows:

$$\mathsf{NTT}(a) = \hat{a} = \sum_{i=0}^{n-1} \hat{a}_i X^i, \text{ where } \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \mod q,$$

$$\mathsf{NTT}^{-1}(\hat{a}) = a = \sum_{i=0}^{n-1} a_i X^i, \text{ where } a_i = \left(n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij}\right) \mod q.$$

The multiplication of $a, b \in \mathcal{R}_q$ can be computed as $ab = \mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$ where $\circ$ denotes the coefficient-wise multiplication. To perform NTT and NTT^{-1} operations faster, there are two commonly used approaches that utilize so-called *butterfly*. The first approach makes use of Cooley-Tukey butterflies [12] in NTT and Gentleman-Sande butterflies [14] in NTT^{-1}, while the other one exploits Gentleman-Sande butterflies in both ways. However, the second approach requires an extra bit-reversal. NewHope selects the second one since it allows more lazy reductions when unsigned integers are used in the implementation.

NewHope-Compact*:* Alkim, Bilgin, and Cenk proposed a compact and fast instantiation of NewHope which is called as NewHope-Compact in [2]. They presented three new parameter sets which are shown in Table 2. As can be seen from the new parameter sets, they reduced the modulus q from 12289 to 3329 by preserving the same security level with the adjustment of the noise parameter k. Due to the change in q, $2n$-th root of unity γ does not exist anymore, i.e., $q \not\equiv 1 \pmod{2n}$. However, even in this situation the works in [2,7,21,27] show that a fast NTT is possible. The use of NTT in this situation is achieved in [2] by selecting γ as the 256-th root of unity so that a 7-level NTT is possible. Then, at the end, instead of having 512 (for $n = 512$) or 1024 (for $n = 1024$) integer coeffcients, i.e., degree zero polynomials, NewHope-Compact has 128 degree three (for $n = 512$) or degree seven (for $n = 1024$) polynomials. Then, coefficient-wise multiplications in NTT domain are performed on small polynomials (degree three for $n = 512$ or degree seven for $n = 1024$) in modulo $(X^4 - r)$ ($n = 512$) or $(X^8 - r)$ ($n = 1024$) where r is a power of γ instead of integers coefficients. They used a one-iterative Karatsuba method [26] for the multiplication of small polynomials whose pseducode can be found in [2, Alg. 5]. Note that the only changing parts of the NewHope are the definition of NTT and the coefficient-wise multiplication. Therefore, one can still refer to Algorithm 1, 2, and 3 for the definition of key generation, encryption and decryption respectfully, since these algorithms are written in a high-level perspective, and the mentioned changes are hidden inside internal functions.

Another contribution of [2], as can be seen in Table 2, the proposal of a new security level for NewHope which is referred to as NewHope-Compact768. This is possible by using a different ring structure that is $\mathbb{Z}_q[X]/(X^{768} - x^{384} + 1)$, firstly proposed by [21]. NewHope-Compact selects q as 3457 which allows a

Table 2. Parameters of NewHope-Compact512, NewHope-Compact768 and NewHope-Compact1024 derived high level properties [2]

Parameter Set	NH-Compact512	NH-Compact768	NH-Compact1024
Dimension n	512	768	1024
Modulus q	3329	3457	3329
Noise Parameter k	2	2	2
NTT parameter γ	17	55	17
Decryption error probability	2^{-256}	2^{-170}	2^{-181}
Claimed post-quantum bit security	100	163	230
NIST Security Strength Category	1	3	5

similar implementation with other parameter sets. They applied a trick at the first level of NTT to switch the regular ring structure $\mathcal{R}_q$ and a similar one at the last level of NTT^{-1}. This trick can be viewed by factorizing $(X^2 - X + 1)$ as $(X - \zeta_1)$ and $(X - \zeta_2)$ where ζ_1 and ζ_2 are both sixth roots of unity. We refer to [2, Appendix A] or [21, Sec. 4.1] for more details. At the end, one-iterative Karatsuba method is applied to perform coefficient-wise multiplications for the small polynomials at degree five in modulo $(X^6 - r)$ where r is a power of γ.

2.3 Kyber

Kyber [7,9] is a post-quantum KEM whose security relies on the hardness of module learning with errors (MLWE) problem [18]. Kyber constructs a CCA secure KEM Kyber.CCAKEM by using a CPA secure PKE, which is referred to as Kyber.CPAPKE, with a variant of the Fujisaki-Okamoto transform [13]. We refer to [7, Alg. 7-9] for Kyber.CCAKEM. The key generation, encryption, and decryption functions of Kyber.CPAPKE are presented in Algorithm 4, 5, and 6 from a high-level perspective.

The most time-consuming parts of Kyber are the randomness generation and hashing similar to other LWE variants such as NewHope. Aside from them, the most costly operation is the multiplication in $\mathcal{R}_q$. Kyber also utilizes NTT in order to speed up this operation. The modulus q and the dimension n are selected as 3329 and 256 for all parameter sets of Kyber. This enables a 7-level NTT with the parameter $\gamma = 17$. After performing 7-level NTT, there are 128 degree one polynomials. Therefore, coefficient-wise multiplications are performed on these 128 degree one polynomials in modulo $(X^2 - r)$ where r is a power of γ by using schoolbook method.

Algorithm 4 KYBER.CPAPKE key generation

Output: public key $pk = (\hat{\mathbf{b}}, \rho)$
Output: secret key $sk = \hat{\mathbf{s}}$

1: $seed \xleftarrow{\$} \{0, \cdots, 255\}^{32}$
2: $\rho, \sigma \leftarrow \mathsf{SHAKE256}(64, seed)$
3: $\hat{\mathbf{A}} \leftarrow \mathsf{GenMatrixA}(\rho)$
4: $\mathbf{s} \leftarrow \mathsf{SampleVec}(\sigma, 0)$
5: $\mathbf{e} \leftarrow \mathsf{SampleVec}(\sigma, 1)$
6: $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{s}) + \mathsf{NTT}(\mathbf{e})$
7: **return** $pk = (\hat{\mathbf{b}}, \rho), sk = \hat{\mathbf{s}}$

Algorithm 5 KYBER.CPAPKE encryption

Input: public key $pk = (\hat{\mathbf{b}}, \rho)$
Input: message $\mu \in \mathcal{R}_q$
Input: seed $coin \in \{0, \cdots, 255\}^{32}$
Output: ciphertext $(\hat{\mathbf{u}}', h)$

1: $\hat{\mathbf{A}} \leftarrow \mathsf{GenMatrixA}(\rho)$
2: $\mathbf{s}' \leftarrow \mathsf{SampleVec}(coin, 0)$
3: $\mathbf{e}' \leftarrow \mathsf{SampleVec}(coin, 1)$
4: $e'' \leftarrow \mathsf{SampleVec}(coin, 2)$
5: $\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
6: $\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{t}}) + \mathbf{e}'$
7: $v' \leftarrow \mathsf{NTT}^{-1}((\hat{\mathbf{b}}^T \circ \hat{\mathbf{t}}) + e'' + \mu$
8: **return** $(\mathsf{Compress}(\mathbf{u}), \mathsf{Compress}(v'))$

Algorithm 6 KYBER.CPAPKE decryption

Input: ciphertext $c = (\mathbf{u}', h)$
Input: secret key $sk = \hat{\mathbf{s}}$
Output: message $\mu \in \mathcal{R}_q$

1: $\mathbf{u} \leftarrow \mathsf{Decompress}(\mathbf{u}')$
2: $v' \leftarrow \mathsf{Decompress}(h)$
3: **return** $\mu = v' - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u}))$

2.4 FFT Trick

The NTT used in this work is known in the recent literature as the FFT trick [24]. The idea is to map the ring

$$\mathbb{Z}_q[X]/\langle X^n - \gamma^n \rangle$$

to

$$\mathbb{Z}_q[X]/\langle X^{n/2} - \gamma^{n/2} \rangle \times \mathbb{Z}_q[X]/\langle X^{n/2} + \gamma^{n/2} \rangle$$

by computing the straightforward CRT map

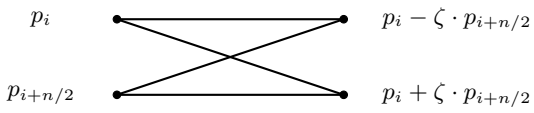
$$p \mapsto (p \bmod X^{n/2} - \gamma^{n/2}, p \bmod X^{n/2} + \gamma^{n/2}).$$

Since $X^{n/2} + \gamma^{n/2} = X^{n/2} - \gamma^{n+n/2}$, the same map can be computed again on both components until reaching a product of rings in which the base multiplication is cheap.

The core operation to reduce a polynomial p modulo $X^{n/2} \pm \zeta$ (for ζ a specific power of γ) is called a butterfly and is depicted in Figure 1. The computation of the NTT consists in applying $n/2$ butterflies to pairs of coefficients of the whole polynomial iteratively between 1 and $\log_2 n$ times, each iteration being referred to as a layer. Figure 2 in appendix A pictures a full NTT consisting of 9 layers mapping the ring $\mathbb{Z}_q[X]/\langle X^{512} + 1 \rangle$ to a product of rings of the form

$\mathbb{Z}_q[X]/\langle X-\zeta\rangle$. In NewHope-Compact and Kyber, the NTT is stopped earlier because $\mathbb{Z}_q$ does not offer high order enough roots of unity to compute all the layers. This means that the NTT itself is less expensive but the base operation in the product of rings is more computationally intensive. This base operation being polynomial multiplication in rings of the form $\mathbb{Z}_q[X]/\langle X^a - b\rangle$ for a in $\{2, 4, 6, 8\}$, depending on the algorithm and parameter set used.

Fig. 1. Cooley-Tukey Butterfly



2.5 Montgomery and Barrett Reductions

Montgomery [22] and Barrett [8] reductions are very useful for efficient and constant time implementation of reductions in $\mathcal{R}_q$. Efficient versions for signed integers of these reduction algorithms were presented in [24, Alg. 3, 5], and they are recalled in Algorithms 12 and 13, in Appendix B. Moreover, assembly implementations of these reduction algorithms on Cortex-M4 are provided in Algorithm 7, [11, Alg. 7], and 8. Note that there are two important things to consider while using these efficient reduction algorithms. Firstly, while Montgomery reduction gives output between $-q$ and q, Barrett reduction gives output between 0 and q. Secondly, Montgomery reduction can not handle all signed words. It accepts input in the range of $-\frac{\beta}{2}q$ to $\frac{\beta}{2}q$ where β is selected as 2^{16} for efficient implementation.

Algorithm 7 Signed Montgomery reduction [11]; using Montgomery factor $\beta = 2^{16}$

Input: a where $-\frac{\beta}{2}q \leq a < \frac{\beta}{2}q$
Output: reduced a $\rightarrow r'$ where $r' = \beta^{-1}a \pmod q$, and $-q < r' < q$

1: `smulbb` t, a, q^{-1}	$\triangleright\ t \leftarrow (a \mod \beta) \cdot q^{-1}$
2: `smulbb` t, t, q	$\triangleright\ t \leftarrow (t \mod \beta) \cdot q$
3: `usub16` a, a, t	$\triangleright\ a_{top} \leftarrow \lfloor \frac{a}{2^{16}} \rfloor - \lfloor \frac{t}{2^{16}} \rfloor$

2.6 ARM Cortex-M4

Our target platform is STM32F4DISCOVERY [25], featuring 32-bit ARM Cortex-M4 [6] with FPU core, which is the selected platform by NIST in order

Algorithm 8 Barrett reduction on packed argument; using $\beta = 2^{16}$

Input: a (32 bit signed integer where a_{top} and a_{bottom} contains two different coefficents)
Output: r = $r_{top} \mid r_{bottom}$ where $r_{top} \equiv a_{top} \pmod q$, $r_{bottom} \equiv a_{bottom} \pmod q$, $0 \leq r_{top}, r_{bottom} \leq q$ and $0 \leq q < 2^{15}$

1: v $\leftarrow \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1} \cdot 2^{\beta}}{q} \right\rceil$	▷ precomputed
2: `smulbb` t_1, a, v	▷ $t_1 \leftarrow a_{bottom} \cdot v$
3: `smultb` t_2, a, v	▷ $t_2 \leftarrow a_{top} \cdot v$
4: `asr` t_1, t_1, #(log(β) + $\lfloor$log(q)$\rfloor$ − 1)	▷ $t_1 \leftarrow t_1 >> (\log(\beta) + \lfloor \log(q) \rfloor - 1)$
5: `asr` t_2, t_2, #(log(β) + $\lfloor$log(q)$\rfloor$ − 1)	▷ $t_2 \leftarrow t_2 >> (\log(\beta) + \lfloor \log(q) \rfloor - 1)$
6: `smulbb` t_1, t_1, q	▷ $t_1 \leftarrow t_1 \cdot q$
7: `smulbb` t_2, t_2, q	▷ $t_2 \leftarrow t_2 \cdot q$
8: `pkhbt` t, t_1, t_2, lsl#16	▷ $t \leftarrow (t_1 \& 0xFFFFu) \mid (t_2 << 16)$
9: `usub16` r, a, t	▷ $r_{top} \leftarrow a_{top} - t_{top}$ and $r_{bottom} \leftarrow a_{bottom} - t_{bottom}$

to evaluate the post-quantum candidates on microcontrollers. It implements ARMv7E-M instruction sets, which provide Digital Signal Processing (DSP) instructions. These instructions include saturating and unsigned Single Instruction Multiple Data (SIMD) instructions that can perform arithmetic operations on two halfwords or four bytes in parallel. These instructions have been shown to be very beneficial to speed up post-quantum algorithms [5,10,11,15,16,17,23], although this architecture comes with the restriction of a limited number of registers, which is 16 general purpose 32-bit registers and only 14 of them are available for the developer. This platform is also used by the benchmarking and testing framework `pqm4` [16].

3 Implementation Details

This section first describes our optimizations to speed up the computation of polynomial multiplication in $\mathcal{R}_q$ that includes both the computation of NTT and NTT^{-1}, and coefficient-wise multiplication for NewHope, corresponding to multiplications of integers, Kyber, corresponding to multiplications of degree one polynomials modulo $(X^2 - r)$, and NewHope-Compact, corresponding to multiplications of degree three polynomials modulo $(X^4 - r)$ (for $n = 512$), degree five modulo $(X^6 - r)$ (for $n = 768$) or degree seven modulo $(X^8 - r)$ (for $n = 1024$). Then, we present the implementation techniques in order to decrease the stack usage of NewHope and NewHope-Compact by following a similar approach to the Kyber implementation of [11]. Finally, we provide a tradeoff between secret key size and performance for Kyber and NewHope-Compact.

3.1 Optimization of Polynomial Multiplication for Speed

Polynomial multiplication in $\mathcal{R}_q$ is one of the most time-consuming part of KEMs whose security relies on RLWE/RLWR or MLWE/MLWR problems. Some of them, specifically NewHope and Kyber, utilize NTT for ring arithmetic in

order to have a fast and efficient implementation. We present an optimized assembly implementation of polynomial multiplication on Cortex-M4, which can be used by all RLWE/RLWR-MLWE/MLWR schemes utilizing NTT for ring arithmetic and have a modulus smaller than 2^{15}. Although some of the techniques described in this section are specific to the ring presented in NewHope, NewHope-Compact or Kyber, adapting the implementation of one to another with some minor changes is possible. We indicate such points when appropriate.

Representation of Polynomials and Packing: We represent polynomials in $\mathcal{R}_q$ as an array composed of signed 16-bit integers similar to [11], which was firstly introduced in AVX2 implementation of Kyber by [24]. Signed representation frees us from adding a multiple of q after subtraction of two coefficients. Hence, using Cooley-Tukey butterflies in forward NTT does not have the downside coming from the subtraction anymore. It already has an advantage in addition over Gentleman-Sande butterflies since the coefficients grow by q after each additions while they doubled in Gentleman-Sande butterflies. Moreover, Cooley-Tukey butterflies do not need a bit-reversal while we need to perform bit-reversal permutation before and after Gentleman-Sande butterflies, i.e., Cooley-Tukey butterflies take input in normal order and give result in bit-reversed order while Gentleman-Sande butterflies take input in bit-reversed order and give result in normal order. However, using Cooley-Tukey butterflies increases the code size, since different methods are used for NTT and NTT^{-1}. By considering these, we decided to use Cooley-Tukey butterfly in forward NTT, unlike the optimized implementation of NewHope in [5]. Moreover, Cortex-M4 is a 32-bit architecture and polynomial coefficients are below 16 bits. Therefore, in order to utilize the properties of the Cortex-M4 platform as much as possible, we packed two coefficients into one register. Thereby, we can utilize SIMD instructions and perform addition/subtraction on two halfwords in parallel by using `uadd16` or `usub16`. Moreover, similar to [11], we implement double butterfly which takes a packed register as input and returns a packed butterfly result.

Montgomery, Barrett and Lazy Reductions: Montgomery reduction of [11], which is given in Algorithm 7, is implemented in three clock cycles. In this work, we reduce the implementation of Montgomery reduction such that it can be performed in only two clock cycles by storing $-q^{-1}$ instead of q^{-1} and using the `smlabb` instruction which multiplies two halfwords and adds the 32-bit result to another 32-bit value in one clock cycle. This implementation is given in Algorithm 9. Kyber implementation of [11] performs 1856 Montgomery reductions (896 in the two NTT, 448 in NTT^{-1}, 512 in base multiplication) in a full polynomial multiplication ($\mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$). Therefore, this change saves 1856 clock cycles for a full polynomial multiplication in $\mathcal{R}_q$.

Similarly to [11], we used Barrett reduction given by [24, Alg. 5]. The assembly implementation of this reduction on packed argument is given in Algorithm 8. We can see that it requires nine clock cycles on our Cortex-M4 microcontroller. We also implemented a Montgomery reduction on packed argument that is provided in Algorithm 10. It is implemented in eight clock

Algorithm 9 Signed Montgomery reduction; using Montgomery factor $\beta = 2^{16}$

Input: a where $-\frac{\beta}{2}q \leq a < \frac{\beta}{2}q$
Output: reduced a $\rightarrow r'$ where $r' = \beta^{-1}a \pmod{q}$, and $-q < r' < q$

1: `smulbb` t, a, -q^{-1}	$\triangleright\ t \leftarrow (a \mod \beta) \cdot (-q^{-1})$
2: `smlabb` a, t, q, a	$\triangleright\ a_{top} \leftarrow \lfloor \frac{(t \mod \beta) \cdot q}{2^{16}} \rfloor + \lfloor \frac{a}{2^{16}} \rfloor$

cycles, which is slightly faster than Barrett reduction. Note that their outputs are not in the same range. Hence, when we need our output to be between 0 and q, we use Barrett reduction. In other case, we use the faster Montgomery reduction.

Algorithm 10 Signed Montgomery reduction on packed argument; using Montgomery factor $\beta = 2^{16}$

Input: a (32 bit signed integer where a_{top} and a_{bottom} contains two different coefficients)
Output: r = r$_{top}$ | r$_{bottom}$ where $r_{top} \equiv \beta^{-1}a_{top} \pmod{q}$, $r_{bottom} \equiv \beta^{-1}a_{bottom} \pmod{q}$

1: v $\leftarrow \beta \pmod{q}$	$\triangleright$ precomputed
2: `smulbb` t$_1$, a, v	
3: `smulbb` r$_1$, t$_1$, -q^{-1}	$\triangleright\ r_1 \leftarrow (t_1 \mod \beta) \cdot (-q^{-1})$
4: `smlabb` r$_1$, r$_1$, q, t$_1$	$\triangleright\ r1_{top} \leftarrow \lfloor \frac{(r_1 \mod \beta) \cdot q}{2^{16}} \rfloor + \lfloor \frac{t_1}{2^{16}} \rfloor$
5: `smultb` t$_2$, a, v	
6: `smulbb` r$_2$, t$_2$, -q^{-1}	$\triangleright\ r_2 \leftarrow (t_2 \mod \beta) \cdot (-q^{-1})$
7: `smlabb` r$_2$, r$_2$, q, t$_2$	$\triangleright\ r2_{top} \leftarrow \lfloor \frac{(r_2 \mod \beta) \cdot q}{2^{16}} \rfloor + \lfloor \frac{t_2}{2^{16}} \rfloor$
8: `pkhtb` r, r$_2$, r$_1$, asr #16	$\triangleright\ r \leftarrow (r2_{top} \vert (r1_{top} >> 16))$

Depending on the size of the modulus and the register size of the underlying architecture, it is not always necessary to reduce the results after an addition or subtraction. Skipping unnecessary reductions is called lazy reductions. It is usual that optimized NTT implementations heavily use this technique to speed up the code. However, those lazy reductions are mostly performed after an addition or subtraction as stated before. In this work, we also perform lazy reductions after component-wise multiplications, also referred to as base multiplications, for modulus 3329 and 3457 used in Kyber and NewHope-Compact. Base multiplication for Kyber is given in Algorithm 11. Each (mod q) in Algorithm 11 corresponds to Montgomery reduction. As we can see from Algorithm 11, five Montgomery reductions are needed in one base multiplication function. We noticed that if both of the coefficients that are multiplied are already reduced modulo q, then the result is way lower than the value that Montgomery reduction can handle which is $2^{15} \cdot q$, (See Proposition 1). Then, we can add the results of several multiplications before performing a Montgomery reduction.

Hence, we can compute $(c[0] \leftarrow ((a[0] \cdot b[0]) + ((a[1] \cdot b[1]) \bmod q) \cdot r) \bmod q)$ instead of applying line 2 of Algorithm 11. Therefore, we save one Montgomery reduction per coefficient so two per one base multiplication. In total, there are 128 base multiplications in a polynomial multiplication in $\mathcal{R}_q$. Thus, we save 256 Montgomery reductions, that is to say 768 clock cycles for the implementation of [11] and 512 clock cycles for our implementation, for a polynomial multiplication of Kyber. We can also use these lazy reductions for NewHope-Compact. The base multiplications for NewHope-Compact512, NewHope-Compact768, and NewHope-Compact1024 require 4, 6, and 8 sequential additions of the multiplication results which can be handled as shown in Proposition 1. Therefore, we can skip 3, 5, or 7 Montgomery reductions per coefficient so 1536, 3840, or 7168 Montgomery reductions in total for NewHope-Compact512, NewHope-Compact768 or NewHope-Compact1024, respectively. Note that we can also omit the Montgomery reductions after the multiplications in the first Cooley-Tukey butterflies where the inputs are small secret or error polynomials sampled from centered binomial distribution. The results of the first multiplications can only grow up to $-2q$ or $2q$ for Kyber and NewHope-Compact and $-q$ or q for NewHope. However, this technique can not be used if the input polynomial is not a small polynomial, i.e., not sampled from centered binomial distribution. We have such cases mainly due to the stack usage optimization explained in Section 3.2 for Kyber, NewHope, and NewHope-Compact. Therefore, we prefer not to use this lazy reduction to prevent having two very similar NTT implementations.

Algorithm 11 Multiplication of polynomials in $\mathbb{Z}_q[X]/(X^2 - r)$ for Kyber

Input: a and $b \in \mathbb{Z}_q[X]/(X^2 - r)$ where r is a power of γ.
Output: $c \in \mathbb{Z}_q[X]/(X^2 - r)$.

1: **function** basemul(a, b)
2: $\quad c[0] \leftarrow (a[0] \cdot b[0]) \bmod q + ((a[1] \cdot b[1]) \bmod q) \cdot r) \bmod q$
3: $\quad c[1] \leftarrow (a[0] \cdot b[1]) \bmod q + (a[1] \cdot b[0]) \bmod q$
4: $\quad$ **return** c
5: **end function**

Proposition 1. *Let* $-3329 < a_i, b_i < 3329$ *where* $0 \leq i \leq 8$, *and* $c = \sum_{i=0}^{8} a_i \cdot b_i$. *c is in the range of* $(-2^{15} \cdot 3329)$ *to* $(2^{15} \cdot 3329)$

Proof. Let $a_i = 3328$ and $b_i = 3328$ for $0 \leq i \leq 8$ be the maximum allowed values. Then, $\sum_{i=0}^{8} a_i \cdot b_i = 99680256$. This is very close to $2^{15} \cdot 3329$ which is the maximum value for the input of Montgomery reduction. In fact, if we add $3328 \cdot 3328$ to the sum (99680256), it will be (110755840) and it will exceed $2^{15} \cdot 3329 = 109084672$.

Merging NTT Layers: Merging multiple NTT layers reduces loads and stores and gives a noticeable performance improvement on Cortex-M4 [5,11]. While

[5] uses eight registers for eight coefficients and performs three layers of NTT, [11] uses only four registers for eight coefficients and performs two layers of NTT without storing and reloading. They use the remaining registers to keep the constants required in Montgomery and Barrett reductions, and the loop counter to keep track of the loop rolling. In this work, we use eight registers to keep 16 coefficients and perform three or four layers of NTT, depending on the distance between the coefficients being used in the same butterfly on the next layer. In other words, we load coefficients to these eight registers in such a way that a maximum level of NTT can be performed. Thanks to the structure of the NTT of NewHope and NewHope-Compact, we can merge four layers together, since on the last layers, we need coefficients with distance one, and having distance one coefficients on the memory is free with `ldr` instruction. Note that NewHope512 does not need four layers merging since it contains nine layers that can be merged efficiently as 3+3+3. Moreover, Kyber includes seven layers of 256-point NTT, which is different from NewHope-Compact having seven layers of 128-point NTT. Consequently, while NewHope-Compact performs NTT with distance one coefficients on the last layer, Kyber requires distance two coefficients. Thus, the last four layers can not be merged. Therefore, we merged seven layers as 3+3+1 for Kyber. Although eight registers are used to store coefficients, we can still spare some registers to store the constants required for Montgomery reductions, specifically for q and $-q^{-1}$. However, we have to reload Barrett constant (line 1 of Algorithm 8) or Montgomery constant (line 1 of Algorithm 10) at every use, but note that we do not need them heavily thanks to lazy reductions. We follow a different approach for loop counter which will be described in the loop unrolling paragraph. Hence, we save more than we lose, i.e. loading more coefficients and performing more layers in every loop is better than loading Barrett constant only once and keeping it in the memory to be used for all Barrett reductions.

Precomputation of Twiddle Factors: The powers of the NTT constant γ are often referred to as twiddle factors. It is a common approach to precompute all of these twiddle factors in the Montgomery domain and store them in flash memory. In this work, we use Montgomery factor $\beta = 2^{16}$ similar to [11,24]. We reorder these constants before storing them in flash to have them appearing in memory in the same order as they are used. Hence, we can easily load the next one without computing its address. The loading instruction on Cortex-M4 has the ability to move the pointer to the next twiddle factor while fetching the current value from memory. Thus moving to the next one has no extra cost. Moreover, we perform half of the division with n during the last butterfly by just multiplying the last twiddle factor with n^{-1}.

Unrolling: We unroll the outer loop of NTT and iterate over the layers as usual. [5,11] spare one register for the loop counter. While [5] uses this loop counter both to check the remaining loop and to decide which precomputed twiddle factor to use, [11] uses it only to detect when to end the loop. We decided not to spare a register for this loop counter and instead use this freed register to

load more coefficients in every loop and merge more NTT layers. Then, we can naively use `.rept` directive instead of this loop counter. However, since `.rept` only repeats the same code, it increases the code size dramatically. Hence, we went back to the loop counter idea again. But instead of keeping it in a register, we spill it to the stack. Consequently, the code size stays reasonable, and we can still load more coefficients in every loop. As an obvious observation, using `.rept` directive is faster, but again it increases the code-size. Hence, it might be useful for some applications where there is plenty of empty memory space.

Link-time Optimization: Link-time optimization is an important option to control optimization, although it may increase the code size. The usual approach without link-time optimization to form an executable file is that each source files is compiled with some optimization level to generate different object files. Then, these optimized object files are linked together to compose an executable file. Although this approach does a good job optimizing source codes, it turns out that linker can perform even better optimization when link-time optimization (`-flto`) is enabled and can give a performance boost of around 10%. The key performance gain is achieved from cross-module function inlining, which is not directly possible without `-flto`. Hence, it will tend to increase the code size since inlining functions across source files introduces code duplication. However, it should also be noted that link-time optimization is more effective at identifying unused codes or codes which have no impact on the output.

`pqm4` platform does not have `-flto` as default option since it increases stack consumption or results in a slower computation of some schemes as stated by [11]. However, they showed that it improves the performance of Kyber. Therefore, we have also tested the effect of `-flto` on performance for our implementations of Kyber, NewHope, and NewHope-Compact and realized that they all benefit from it and have a performance boost. However, since assembly optimized polynomial multiplication was already implemented carefully by inlining all necessary functions such as modular reductions, adding `-flto` has no effect on its performance.

3.2 Optimization of NewHope and NewHope-Compact for Stack Usage

On embedded devices, it is often the case that memory usage is a real bottleneck. Outside of real-time systems, one can always wait for a slow algorithm, but if the algorithm needs more memory than what can be found on the device, it is completely unusable. While the Cortex-M4 on our board offers quite a large amount of memory, we decided to optimize for stack usage as well.

The approach we took was to reduce the minimum amount of stack space required to compute the cipher while keeping performances mostly unaffected. One looking to reduce stack usage to the bare minimum aggressively would write an implementation considerably slower than ours. The three main metrics regarding implementation are speed, stack usage, and code size. The one to

optimize severely depends on the context in which the cipher will be used. In our work, we tried to optimize the two firsts while keeping the last one reasonable.

Key generation: The core of the key generation (Algorithm 1) is the computation of $\hat{b} \leftarrow \hat{a} \circ \mathsf{NTT}(s) + \mathsf{NTT}(e)$. Since each coefficient of the output of the NTT depends on *all* the coefficients of the input, all the coefficients of s and e must have been generated before proceeding the addition. Hence, at least two full polynomials should be stored in memory. To reduce the memory usage, we used the observation that polynomial multiplication can be performed on-the-fly in the NTT domain and, likewise, adding error to a polynomial can be performed on-the-fly in *normal* domain. Indeed, the operation $\circ$ works sequentially on parts of its inputs (one coefficient at the time for NewHope and four, six or eight for NewHope-Compact depending on the parameter set used) and do not need all of them in memory at the same time. Similarly, the error polynomial can be computed coefficient by coefficient and added directly but only if the addition considered is in normal domain. This is why instead of computing

$$\hat{b} \leftarrow \hat{a} \circ \mathsf{NTT}(s) + \mathsf{NTT}(e),$$

we computed

$$\hat{b} \leftarrow \mathsf{NTT}(\mathsf{NTT}^{-1}(\hat{a} \circ \mathsf{NTT}(s)) + e)$$

and performed the multiplication and the addition on-the-fly. This requires one more NTT^{-1} but allows to only store *one* polynomial in memory, containing s and $\hat{b}$ subsequently. Doing this way reduced significantly stack usage and since our benchmarks showed that computing one extra optimized NTT^{-1} would only increase the key generation time by around 5%, we decided that this was a good trade-off. This trick can be similarly applied to Kyber. Note that the small *relative* cost of this technique is specific to our context and is mainly due to the fact that hashing is the main performance bottleneck [6]. One using a faster hash function would have a decrease in performance higher than 5%. That being said, the *absolute* cost of the trick is always the same and is one NTT^{-1}.

Encryption: The encryption procedure (Algorithm 2) is mainly driven by the following computations:

1. $\hat{t} \leftarrow \mathsf{NTT}(s')$
2. $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \mathsf{NTT}(e')$
3. $v' \leftarrow \mathsf{NTT}^{-1}(\mathsf{DecodePoly}(\hat{b'}) \circ \hat{t}) + e'' + \mathsf{Encode}(\mu)$

The two firsts yield a situation similar as the key generation but unfortunately require two polynomials in the stack frame. Indeed, since $\hat{t}$ appears in the second and last computation, the result of $\hat{a} \circ \hat{t}$ *cannot* be stored in the same memory space as $\hat{t}$ (and since it would need to go through a NTT^{-1}, it does need to be fully stored). Once $\hat{u}$ is computed, it can be packed in the ciphertext and free one of the two polynomials. The last computation is quite friendly for stack usage.

[6] This issue will be discussed in more details in the result section

Indeed, since both the base multiplication and the addition operate on small portions of the polynomial, that $e'' + \mathsf{Encode}(\mu)$ can be computed coefficient by coefficient, and that $\hat{b}$ can be partially unpacked from the inputs, it could technically be computed with one polynomial plus a small overhead in the stack frame. Since two polynomials where already allocated previously, we actually fully unpack $\hat{b}$ as only maximal stack usage is relevant. Finally, the stack usage is greater than the one of the key generation because of the extra polynomial stored.

Decryption: The decryption of NewHope (Algorithm 3) is quite lightweight in terms of stack usage. Unfortunately, the algorithms introduced in the preliminaries are the CPA version of the cipher. Since the CCA transform is running the encryption procedure during decryption, the stack usage is essentially the same as for encryption.

3.3 Tradeoffs between Secret Key Size and Speed

There are different tradeoffs between secret key size and the performance of the algorithm. Some of these tradeoffs are also discussed in [7,9]. If the secret key size critical, one can only store the seed used for all randomness in key generation. However, this requires to perform key generation again during decapsulation and gives a significant performance penalty. As also stated by [7,9], another optimization could be storing the secret key in the normal domain instead of the NTT domain. Hence, each coefficients can be compressed to 3 bits, since their possible values are in between -2 and 2. Note that NTT used in Kyber and NewHope-Compact are fast on Cortex-M4 so that we decided such optimizations are good trade-offs. We also observed that sampling the secret polynomial s is fast enough, although it is usually stated that the most time consuming part of such algorithms is the randomness generation and hashing. Note that sampling the secret key from the centered binomial distribution is lightweight in comparison to generation of the public parameter a, since we can extract two coefficients from only one byte by using the centered binomial distribution while uniform sampling needs two bytes to extract only one coefficients. Hence, we decided to store only the seed, whose size is 32 bytes, to sample the secret key. Then, the secret key is sampled again during decapsulation, and it is transformed to NTT domain. These operations reduce the secret key size by 736 bytes for Kyber512 and NewHope-Compact512, 1120 bytes for Kyber768 and NewHope-Compact768, and 1504 bytes for Kyber1024 and NewHope-Compact1024. On the other hand, they increase the decapsulation time by around 7% for Kyber and 9% for NewHope-Compact while decreasing the key generation time slightly.

4 Results and Comparison

Our optimizations were implemented in the three siblings schemes NewHope, NewHope-Compact, and Kyber. Comparing different schemes across pa-

rameter sets is often complicated because performances are always strongly correlated with the targeted security. We decided to group them in security levels corresponding roughly to what NIST refers to as security levels 1, 3 and 5 which themselves correspond to 128, 192 and 256 bits of security. Fortunately, since all the schemes involved in our tests are similar and based on {R,M}LWE, the dimension of the underlying lattice problem can be roughly translated into NIST security levels. Hence, we compare them for dimensions 512, 768 (if available) and 1024, which correspond to the three security levels aforementioned.

Scheme		Dimension	KeyGen	Encaps	Decaps
NewHope	**(This work)**	512	2 056	2 872	2 888
		1024	3 088	4 928	4 944
	([16])	512	5 960	9 168	10 296
		1024	11 080	17 360	19 576
NewHope-Compact		512	2 168	2 984	3 000
		768	2 616	3 952	3 960
		1024	3 200	5 048	5 056

Table 3. Stack usage.

Scheme		Dimension 512	Dimension 768	Dimension 1024
NewHope	**(This work)**	**G**: 578 890 **E**: 858 982 **D**: 806 300	-	**G**: 1 157 222 **E**: 1 674 899 **D**: 1 587 107
	[16]	**G**: 588 683 **E**: 918 558 **D**: 904 800	-	**G**: 1 161 112 **E**: 1 777 918 **D**: 1 760 470
NewHope-Compact	**(This work)**	**G**: 343 196 **E**: 524 840 **D**: 478 135	**G**: 516 762 **E**: 775 393 **D**: 710 457	**G**: 673 499 **E**: 1 010 961 **D**: 926 801
Kyber	**(This work)**	**G**: 455 191 **E**: 586 334 **D**: 543 500	**G**: 864 008 **E**: 1 032 540 **D**: 969 867	**G**: 1 404 695 **E**: 1 605 707 **D**: 1 525 805
	[11]	**G**: 514 291 **E**: 652 769 **D**: 621 245	**G**: 976 757 **E**: 1 146 556 **D**: 1 094 849	**G**: 1 575 052 **E**: 1 779 848 **D**: 1 709 348

Table 4. Cycle counts comparison for the {R,M}LWE schemes improved by our work. **G**: key generation, **E**: encapsulation, **D**: decapsulation

4.1 Speed comparison

The results of our benchmarks in terms of speed can be found in Table 4. The code was compiled and run in the same conditions as the schemes benchmarked in `pqm4` [16]. We compare the two candidates NewHope and Kyber against themselves in their previous Cortex-M4 optimized version available in `pqm4` and also add the newcomer NewHope-Compact. One can see that NewHope and Kyber perform around 10% better with our optimizations. Furthermore, NewHope-Compact is more than 40% faster when comparing with NewHope and more than 25% faster when comparing with Kyber for all security levels. This is explained by the two following observations:

- NewHope-Compact is a derivative of NewHope using a smaller modulus and distribution, which means increased performances during polynomial multiplication because of lazy reductions and less hashing need to sample the error distribution.
- NewHope-Compact is based on RLWE while Kyber is based on MLWE. Hence, even though they share similar parameter sets, the inherent performance penalty of using the less structured version of LWE hurts Kyber.

Scheme	Dimension 512	Dimension 768	Dimension 1024
NewHope	**G**: 75% **E**: 80% **D**: 73%	-	**G**: 74% **E**: 79% **D**: 71%
NewHope-Compact	**G**: 75% **E**: 79% **D**: 68%	**G**: 74% **E**: 78% **D**: 67%	**G**: 74% **E**: 78% **D**: 67%
Kyber	**G**: 76% **E**: 79% **D**: 69%	**G**: 77% **E**: 80% **D**: 72%	**G**: 78% **E**: 80% **D**: 73%

Table 5. Time spent hashing. **G**: key generation, **E**: encapsulation, **D**: decapsulation.

4.2 Dominance of hashing

The speed difference showed in Table 4 might look slim at first sight. Actually, this is due to the fact that, as pointed out by previous works, those schemes have been optimized so much that the bottleneck is now the generation of random numbers through hashing instead of the polynomial multiplication procedure. Table 5 shows the time spent hashing for all algorithms and parameter sets. As we can see, with a minimum of 67% for the decapsulation of NewHope-Compact,

all the algorithms are severely dominated by hashing. Even if polynomial multiplications were somehow instantaneous, the results of Table 4 would be somewhat similar.

Scheme	Dimension	NTT	NTT^{-1}	$\circ$
NewHope	512	31217	23439	3157
	1024	68131	51231	6229
NewHope-Compact	512	13319	13059	7060
	768	20424	21235	12756
	1024	26568	26048	18517
Kyber	256	6855	6983	2325
	256 ([11])	7754	9377	3076

Table 6. Comparison of the polynomial multiplication functions of all the schemes. Kyber actually uses the exact same NTT code for all dimensions.

Scheme	Dimension	KeyGen	Encaps	Decaps
NewHope	512	89030	92187	26596
	1024	193722	199951	57460
NewHope-Compact	512	46757	53817	20119
	768	74839	87595	33991
	1024	97701	116218	44565
Kyber	512	50686	48609	25343
	768	83004	76397	34523
	1024	119972	108835	43703

Table 7. Total time spent in polynomial multiplication subroutines (NTT, NTT^{-1} and $\circ$).

4.3 Comparing polynomial multiplications

The reader might wonder why to bother optimizing polynomial multiplications further if it is not the bottleneck anymore. The reason is twofold: first, `Keccak` is used to expand the seed in every algorithm as it is the default choice since the end of the `SHA-3` competition. However, the choice of the seed expansion algorithm is somewhat orthogonal to the scheme and does not affect post-quantum assumptions. Hence, using a faster hash function would reduce the impact of hashing on speed performances. Furthermore, it might be unnecessary to use a cryptographic hash function to generate the public parameter. For instance, [10] uses a faster, non-cryptographic RNG to speed-up a scheme base on LWE. Second, even if `Keccak` is used, since its usage will likely grow in all future cryptographic applications, we will eventually see hardware accelerations

for it on a lot of architecture. This would naturally drastically decrease the time spent hashing in our schemes and make polynomial multiplication the most important optimization target again. Recall that, as stated in Section 3.2, this would increase the relative cost of the reduced stack usage trick used in the key generation. Nevertheless, we think that outside of unrealistically fast polynomial generation, the trade-off can still be useful.

Since our work is the first Cortex-M4 implementation of NewHope-Compact, we do not have any point of comparison for our technique for this scheme. Table 6 shows the speed-up for the dimension 256 NTT used in all parameter sets of Kyber and the cycle counts of all subroutines of the polynomial multiplication for each algorithm and dimension. The total cost of multiplication operations for each scheme is presented in Table 7. This table was obtained by summing all the time spent in the three multiplication subroutines: NTT, NTT^{-1} and $\circ$. It can be seen that for dimension 512, both Kyber and NewHope perform similarly while NewHope-Compact is sensibly faster, whilst for dimension 1024 Kyber and NewHope-Compact became similar while NewHope is slower. This is mainly due to the extra layers of NTT and the increased number of reductions caused by the larger modulus.

References

1. Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope - Algorithm Specifications And Supporting Documentation (version 1.03). Technical report, NIST Post-Quantum Cryptography Standardization Project, 2019. `https://newhopecrypto.org/`.
2. Erdem Alkim, Yusuf Alper Bilgin, and Murat Cenk. Compact and simple RLWE based key encapsulation mechanism. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*, volume 11774 of *Lecture Notes in Computer Science*, pages 237–256, Santiago de Chile, Chile, October 2–4 2019. Springer, Heidelberg, Germany. `https://doi.org/10.1007/978-3-030-30530-7_12`.
3. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. `http://eprint.iacr.org/2016/1157`.
4. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 327–343, Austin, TX, USA, August 10–12 2016. USENIX Association. `https://eprint.iacr.org/2015/1092`.
5. Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. Newhope on ARM cortex-m. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076

of *Lecture Notes in Computer Science*, pages 332–349. Springer, 2016. https://eprint.iacr.org/2016/758.

6. ARM. ARM Cortex-M4. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4.
7. Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER - Algorithm Specifications And Supporting Documentation (version 2.0). Technical report, NIST Post-Quantum Cryptography Standardization Project, 2019. https://pq-crystals.org/kyber/.
8. Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986. https://doi.org/10.1007/3-540-47721-7_24.
9. Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018. https://eprint.iacr.org/2017/634.
10. Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! faster frodo for the arm cortex-m4. Cryptology ePrint Archive, Report 2018/1116, 2018. https://eprint.iacr.org/2018/1116.
11. Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228, Rabat, Morocco, July 9–11 2019. Springer, Heidelberg, Germany. https://eprint.iacr.org/2019/489.
12. 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. http://www.jstor.org/stable/2003354.
13. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999. https://doi.org/10.1007/3-540-48405-1_34.
14. 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, New York, NY, USA, 1966. ACM. http://doi.acm.org/10.1145/1464291.1464352.
15. 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 Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 281–301, Bogota, Colombia, June 5–7 2019. Springer, Heidelberg, Germany. https://eprint.iacr.org/2018/1018.

16. 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.
17. Angshuman Karmakar, Jose M. Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM cca-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, 2018. https://eprint.iacr.org/2018/682.
18. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015. https://eprint.iacr.org/2012/090.
19. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010. https://doi.org/10.1007/978-3-642-13190-5_1.
20. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013. https://doi.org/10.1145/2535925.
21. Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast ntru using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, 2019. https://eprint.iacr.org/2019/040.
22. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf.
23. Markku-Juhani O. Saarinen, Sauvik Bhattacharya, Óscar García-Morchón, Ronald Rietman, Ludo Tolhuizen, and Zhenfei Zhang. Shorter messages and faster post-quantum encryption with round5 on cortex M. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers.*, volume 11389 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2018. https://eprint.iacr.org/2018/723.
24. Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.
25. STMicroelectronics. STM32F4DISCOVERY kit. https://www.st.com/en/evaluation-tools/stm32f4discovery.html.
26. André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006. http://eprint.iacr.org/2006/224.
27. Shuai Zhou, Haiyang Xue, Daode Zhang, Kunpeng Wang, Xianhui Lu, Bao Li, and Jingnan He. Preprocess-then-NTT Technique and Its Applications to Kyber and NewHope. In Fuchun Guo, Xinyi Huang, and Moti Yung, editors, *Information Security and Cryptology - 14th International Conference, Inscrypt 2018, Fuzhou, China, December 14-17, 2018, Revised Selected Papers*, volume 11449 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 2018. https://eprint.iacr.org/2018/995.

A NTT on dimension 512 polynomial

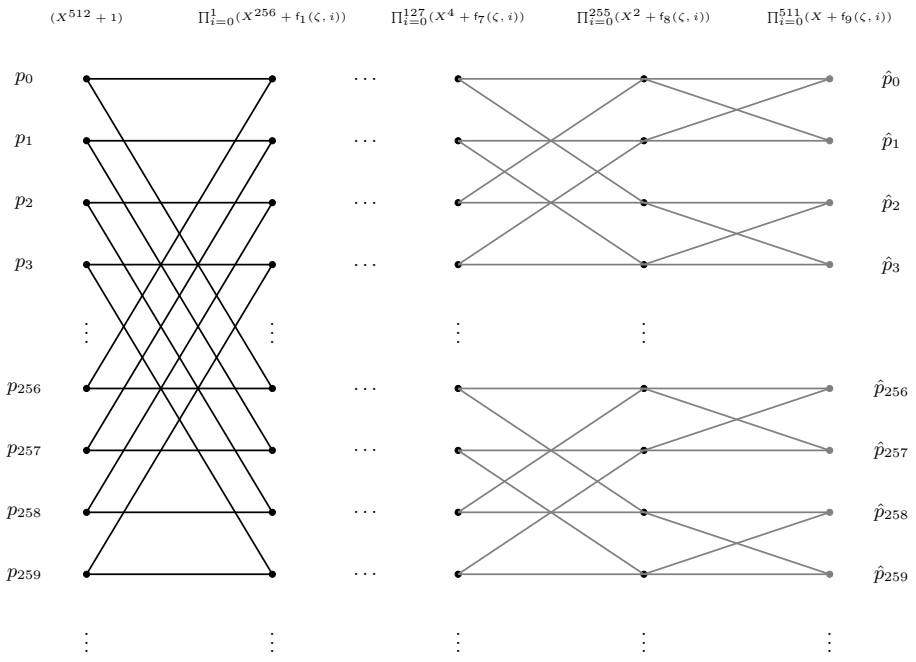


Fig. 2. Full NTT on a dimension 512 polynomial. The function $f_j(\zeta, i) = \zeta^{\mathsf{brv}(2^j-1+i)}$ selects the correct root to compute the isomorphism. All those roots are usually precomputed and correctly ordered in a table. Techniques to reduce q skip some levels: for example, using $q = 3329$ as in NewHope-Compact requires to skip the two lasts (gray) layers.

B Montgomery and Barrett reductions

Algorithm 12 Signed Montgomery reduction [24]; using Montgomery factor $\beta = 2^{16}$

Input: odd q where $0 < q < \frac{\beta}{2}$, and a where $-\frac{\beta}{2}q \leq a = a_1\beta + a_0 < \frac{\beta}{2}q$ and $0 < a_0 < \beta$
Output: r' where $r' = \beta^{-1}a \pmod{q}$, and $-q < r' < q$

1: $m \leftarrow a_0 q^{-1} \pmod^{\pm} \beta$ ▷ signed low product, q^{-1} precomputed
2: $t_1 \leftarrow \left\lfloor \frac{mq}{\beta} \right\rfloor$ ▷ signed high product
3: $r' \leftarrow a_1 - t_1$

Algorithm 13 Signed Barrett reduction [24]; using $\beta = 2^{16}$

Input: odd q where $0 < q < \frac{\beta}{2}$, and a where $-\frac{\beta}{2} \leq a < \frac{\beta}{2}$
Output: r where $r = a \pmod{q}$, and $0 \leq r \leq q$

1: $v \leftarrow \left\lfloor \frac{2^{\log(q)-1} \cdot \beta}{q} \right\rceil$ ▷ precomputed
2: $t \leftarrow \left\lfloor \frac{av}{2^{\log(q)-1} \cdot \beta} \right\rceil$ ▷ signed high product and arithmetic right shift
3: $t \leftarrow tq \mod \beta$ ▷ signed low product
4: $r \leftarrow a - t$