Kyber on ARM64: Compact Implementations of Kyber on 64-Bit ARM Cortex-A Processors

Pakize Sanal[1(✉)], Emrah Karagoz[1], Hwajeong Seo[2], Reza Azarderakhsh[1,4], and Mehran Mozaffari-Kermani[3]

[1] Florida Atlantic University, Boca Raton, USA
{psanal2018,ekaragoz2017,razarderakhsh}@fau.edu
[2] Hansung University, Seoul, South Korea
hwajeong84@gmail.com
[3] University of South Florida, Tampa, USA
mehran2@usf.edu
[4] PQSecure Technologies, LLC, Boca Raton, FL, USA

Abstract. Public-key cryptography based on the lattice problem is efficient and believed to be secure in a post-quantum era. In this paper, we introduce carefully-optimized implementations of Kyber encryption schemes for 64-bit ARM Cortex-A processors. Our research contribution includes optimizations for Number Theoretic Transform (NTT), noise sampling, and AES accelerator based symmetric function implementations. The proposed Kyber512 implementation on ARM64 improved previous works by 1.79×, 1.96×, and 2.44× for key generation, encapsulation, and decapsulation, respectively. Moreover, by using AES accelerator in the proposed Kyber512-90s implementation, it is improved by 8.57×, 6.94×, and 8.26× for key generation, encapsulation, and decapsulation, respectively.

Keywords: Post-quantum cryptography · Kyber · ARM64 · Vectorized implementation

1 Introduction

The integer factorization and discrete logarithm problems, where RSA and Elliptic Curve Cryptography (ECC) are based on, can be easily solved by using Shor's algorithm [21] on a quantum computer. For this reason, the Post-Quantum Cryptography (PQC) standardization process is initiated by NIST in 2016 to choose quantum-resistant algorithms for the upcoming quantum era. In this process, Crystals-Kyber (shortly *Kyber*) [5] is one of the promising candidates among the third round finalists announced in 2020. It is an IND-CCA2-secure lattice-based key-encapsulation mechanism (KEM), and its security is based on the hardness of solving the learning-with-errors problem in module lattices (MLWE problem). In addition, it is comparably fast due to the small parameter size. It is easier to

J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 399, pp. 424–440, 2021.
https://doi.org/10.1007/978-3-030-90022-9_23

implement because the main primitives are modular reductions, small polynomial multiplications, and Number Theoretic Transformation (NTT) operations.

Several works devoted to improve the performance of a primitive of Kyber or of the scheme itself in several platforms. NTT operations are optimized on Advanced Vector Extensions 2 (AVX2) (see [16,20]) and on Cortex-M4 (see [1,7]). It is also accelerated by using GPU (see [11,18]) and hardware accelerators (see [2,4,8,10,12,14,22,23]). However, an efficient implementation of Kyber on high-end ARM processor (i.e. ARMv8 Cortex-A) was not conducted. Since the high-end ARM is widely used in smartphone, smartwatch, and laptop computer, the efficient implementation should be highly considered.

Our Contribution. We propose an optimized implementation of Kyber on 64-bit ARMv8 processors. Primitive operations of Kyber are fully vectorized in ASIMD instructions. The reduction and NTT operations are improved by 3.0–5.0× and 4.0–6.0×, respectively, compared to its optimized C implementation. Moreover, we implement full parameters for Kyber schemes. For example, Kyber512 implementation outperforms by 1.79×, 1.96×, and 2.44×, for key generation, encapsulation, and decapsulation, respectively. Lastly, we use acceleration of symmetric functions through cryptography extension of 64-bit ARMv8 processors. Results show that Kyber512-90s w/accelerator is faster than w/o accelerator by 8.57×, 6.94×, and 8.26×, for key generation, encapsulation, and decapsulation, respectively.

Our code is available at https://github.com/psanal2018/kyber-arm64.

Outline. The paper is organized as follows. Section 2 presents an overview of the Kyber algorithm. In Sect. 3, we introduce the ARMv8-A architecture. In Sect. 4, proposed implementations of Kyber on 64-bit ARM Cortex-A processors are presented. In Sect. 5, the performance evaluation of proposed implementations is described. Finally, the conclusion is given in Sect. 6.

2 Kyber

In this section, we give a brief description of the functions included in Kyber. The details of the algorithm is given in its specification document [19].

2.1 Mathematical Background

The basic elements in Kyber are the polynomials in the ring $\mathbb{Z}_q[X]/(X^n+1)$, denoted by R_q, with $n = 256$ and $q = 3329$ in all variants of Kyber. The polynomials can be represented as a vector of linear polynomials by using Number-Theoretic Transform (NTT): for a polynomial $f = \sum_{i=0}^{255} f_i X^i$ in R_q,

$$\mathsf{NTT}(f) = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X)$$

where

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2i+1)j} \quad \text{and} \quad \hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2i+1)j}$$

with $\zeta = 17$ being the 256-th primitive root of unity. Two polynomials f and g in R_q can be efficiently multiplied by using NTT:

$$\mathsf{NTT}(f) \circ \mathsf{NTT}(g) = \hat{f} \circ \hat{g} = \hat{h}$$

where $\circ$ is the component-wise multiplication of linear polynomials, that is,

$$\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1}X)(\hat{g}_{2i} + \hat{g}_{2i+1}X) \mod (X^2 - \zeta^{2i+1})$$

for $i = 0, 1, \ldots, 127$. Then, the product of f and g is

$$fg = \mathsf{NTT}^{-1}(\hat{h}) = \mathsf{NTT}^{-1}(\mathsf{NTT}(f) \circ \mathsf{NTT}(g)).$$

2.2 Compression and Encoding

An element $x \in \mathbb{Z}_q$ is converted to an d-bit integer by $\mathtt{Compress}_q(x, d)$. An d-bit integer x is converted to a $\mathbb{Z}_q$ element by $\mathtt{Decompress}_q(x, d)$. They are defined as follows:

$$\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d \text{ and } \mathsf{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor$$

where $\lceil a \rfloor$ is the closest integer to a. When each function is applied to a polynomial (or a vector/matrix of polynomials), it is applied to each coefficient individually. In addition, a polynomial (or a vector/matrix of polynomials) is serialized to byte arrays by using $\mathtt{Encode}_\ell()$ function, where ℓ is the bit-length of each coefficient. On the other hand, $\mathtt{Decode}_\ell()$ is the inverse of $\mathtt{Encode}_\ell()$, and it deserializes the byte arrays to polynomials. Lastly, `Parse()` converts a byte stream to the `NTT` representation of a polynomial in R_q.

2.3 Sampling

The noise is sampled from a centered binomial distribution (CBD), denoted by B_η, for $\eta = 2$ or $\eta = 3$. For a sample $(a_1, \ldots, a_\eta, b_1, \ldots, b_\eta) \leftarrow \{0,1\}^{2\eta}$, the output is computed as $\sum_{i=1}^{\eta}(a_i - b_i)$. Using B_η, a polynomial $f = \sum_{i=0}^{255} f_i X^i$ in R_q can be sampled by sampling each coefficient f_i deterministically from 512η-bit output $(\beta_0, \ldots, \beta_{512\eta-1})$ of a pseudo-random function:

$$f_i = \sum_{j=0}^{\eta-1}(\beta_{2i\eta+j} - \beta_{2i\eta+j+\eta}) \quad i = 0, 1, \ldots, 255.$$

For this purpose, Kyber uses a function namely $\mathtt{CBD}_\eta$, which takes 512η-bit input and outputs the corresponding polynomial.

2.4 Parameters

The fixed parameters are $n = 256$ and $q = 3329$. The parameter k represents the dimension of the matrix of polynomials in R_q. The parameter pair (η_1, η_2) are used in $\mathtt{CBD}_\eta$ function for sampling. The parameter pair (d_u, d_v) is used in `Compress` and `Decompress` functions. The list of parameters are given in Table 1.

Table 1. Kyber parameters.

Algorithm	NIST-Level	n	q	k	(η_1, η_2)	(d_u, d_v)
KYBER512	1 (AES-128)	256	3329	2	(3,2)	(10,3)
KYBER768	3 (AES-192)	256	3329	3	(2,2)	(10,4)
KYBER1024	5 (AES-256)	256	3329	4	(2,2)	(11,5)

2.5 Symmetric Functions

Kyber makes a use of a pseudo-random function (PRF), an extendable output function (XOF), two hash functions H, and G, and a key-derivation function (KDF). These functions are specified in Table 2. At this point, Kyber has an alternative version Kyber-90s which uses SHA-2 hash functions and AES, while Kyber uses SHA-3 hash functions.

Table 2. Symmetric primitives in Kyber.

Symmetric primitive	Kyber	Kyber-90s
XOF	SHAKE-128	AES-256 in CTR mode
H and G	SHA3-256 and SHA3-512	SHA-256 and SHA-512
PRF (s, b)	SHAKE-256$(s\|\|b)$	AES-256 in CTR mode (key $= s$ and nonce $= b$)
KDF	SHAKE-256	SHAKE-256

2.6 Kyber-PKE and Kyber-KEM

Kyber-PKE is an IND-CPA-secure public-key encryption scheme. It encrypts messages of a fixed length of 32 bytes. It contains three algorithms: Key Generation, Encryption, and Decryption.

In Kyber-PKE Key Generation, the polynomial matrix $\mathbf{A}$ is randomly generated, and the polynomial vectors $\mathbf{s}$ and $\mathbf{e}$ are sampled according to B_{η_1}. Then, normally, the secret key is $\mathbf{s}$ and the public key is $\mathbf{As} + \mathbf{e}$. However, for efficient implementation purposes, the multiplication $\mathbf{As}$ is performed in NTT domain by generating $\mathbf{A}$ in NTT domain (i.e. $\hat{\mathbf{A}}$) and transforming $\mathbf{s}$ to $\hat{\mathbf{s}} = \mathsf{NTT}(s)$. To avoid NTT^{-1} operation, $\mathbf{e}$ is also transformed to $\hat{\mathbf{e}}$ and added to $\hat{\mathbf{A}} \circ \hat{\mathbf{s}}$. Therefore, the values of secret and public keys are left in NTT domain and encoded to sk and pk, respectively. In addition, the seed for randomness is appended to the public key for letting the recipient generate the matrix $\mathbf{A}$.

In Kyber-PKE Encryption, the message m is encrypted to the ciphertext $c = (c_1, c_2)$ by using the public key pk and random coins r. The polynomial vector $\mathbf{t}$ and the matrix $\mathbf{A}$ are obtained using the public key. The polynomial vector $\mathbf{r}$ is sampled according to B_{η_1} using r. The polynomial vector $\mathbf{e}_1$ and the polynomial e_2 are sampled according to B_{η_2} using r. Then, normally, the ciphertext $c = (c_1, c_2)$ is $(\mathbf{A}^T\mathbf{r} + \mathbf{e}_1, \mathbf{t}^T r + e_2 + m)$. However, multiplications are performed in NTT domain and then transformed to the normal domain by using NTT^{-1}. Moreover, the ciphertext is compressed and encoded.

In Kyber-PKE Decryption, the polynomial vector $\mathbf{u}$ and the polynomial v are obtained from the ciphertext by decoding and decompressing. The vector $\mathbf{s}$ is obtained from the secret key. Then, the message m is $v - \mathbf{s}^T\mathbf{u}$. Again, the multiplications are performed in NTT domain and then transformed to the normal domain by using NTT^{-1}.

Nonce values (which are 0 in the beginning of the algorithms) are incremental in each computation. Algorithms are given in Algorithm 1, 2, and 3.

Algorithm 1: Kyber-PKE Key Generation

Output : secret key and public key pair (pk, sk)

$d \xleftarrow{\$} \{0,1\}^{256}$
$\rho, \sigma \leftarrow G(d)$
$\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho, \mathtt{nonce{+}{+}}))$ ▷ Generate matrix $\mathbf{A} \in R_q^{k \times k}$ (in NTT domain)
$\mathbf{s} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, \mathtt{nonce{+}{+}}))$ ▷ Sample $\mathbf{s} \in R_q^k$
$\mathbf{e} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, \mathtt{nonce{+}{+}}))$ ▷ Sample $\mathbf{e} \in R_q^k$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ $\mathbf{t} := \mathbf{A}\mathbf{s} + \mathbf{e}$ (in NTT domain)
$pk \leftarrow \mathsf{Encode}_{12}(\hat{\mathbf{t}}) \| \rho$
$sk \leftarrow \mathsf{Encode}_{12}(\hat{\mathbf{s}})$
return (pk, sk)

Algorithm 2: Kyber-PKE Encryption

Input : public key pk, message m, random coins $r \in \{0,1\}^{256}$
Output : ciphertext $c = (c_1, c_2)$

$\hat{\mathbf{t}} \leftarrow \mathsf{Decode}_{12}(pk)$
$\rho \leftarrow pk$
$\hat{\mathbf{A}} \leftarrow \mathsf{Parse}(\mathsf{XOF}(\rho, \mathtt{nonce{+}{+}}))$ ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
$\mathbf{r} \leftarrow \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, \mathtt{nonce{+}{+}}))$ ▷ Sample $\mathbf{r} \in R_q^k$
$\mathbf{e}_1 \leftarrow \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, \mathtt{nonce{+}{+}}))$ ▷ Sample $\mathbf{e}_1 \in R_q^k$
$e_2 \leftarrow \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, \mathtt{nonce{+}{+}}))$ ▷ Sample $e_2 \in R_q$
$\hat{\mathbf{r}} \leftarrow \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v := \mathbf{t}^T\mathbf{r} + e_2 + m$
$c_1 \leftarrow \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 \leftarrow \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
return $c = (c_1, c_2)$

Algorithm 3: Kyber-PKE Decryption

Input : secret key sk, ciphertext $c = (c_1, c_2)$
Output : message m

$\mathbf{u} \leftarrow \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c_1), d_u)$
$v \leftarrow \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c_2), d_v)$
$\hat{\mathbf{s}} \leftarrow \mathsf{Decode}_{12}(sk)$
$m \leftarrow \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1)$
return m

On the other hand, Kyber-KEM is an IND-CCA2-secure KEM and it is constructed from Kyber-PKE using (a slightly tweaked) Fujisaki-Okamoto transform. It contains three steps: Key Generation, Encapsulation, and Decapsulation. In the first step, Alice generates the public and secret keys by using Kyber-PKE Key Generation algorithm, and shares her public key with Bob. In the second step, Bob encrypts the message to the ciphertext by using Kyber-PKE Encryption algorithm, and sends the ciphertext to Alice. He also computes the shared secret by using the message, Alice's public key, and the ciphertext. In the last step, Alice decrypts the ciphertext to the message by using Kyber-PKE Decryption algorithm, and then verifies whether it can be encrypted to the same ciphertext (sent by Bob) by following similar steps as Bob did by using Kyber-PKE Encryption algorithm. If ciphertexts match, Alice computes the shared secret by using the message, her public key, and the ciphertext. Otherwise, she computes the shared secret by using a random value and the ciphertext. Details of Kyber-KEM are illustrated in Fig. 1.

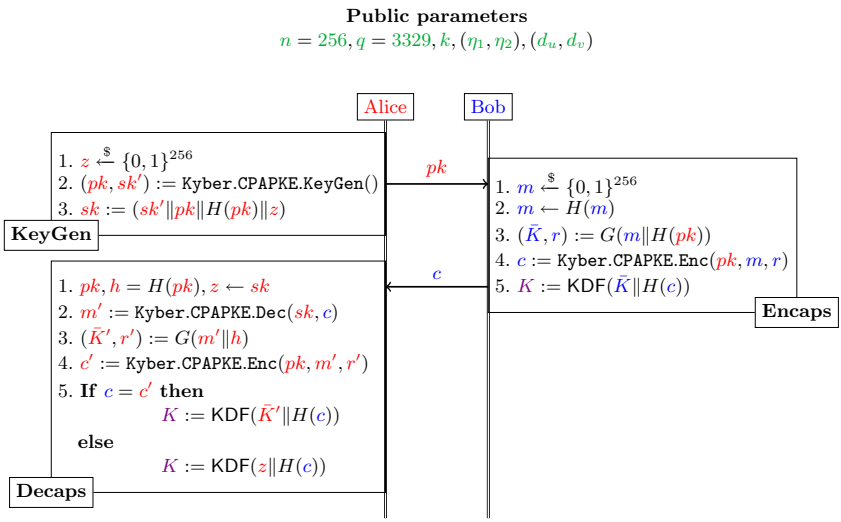

Fig. 1. Kyber-CCA-KEM.

3 ARMv8-A Architecture

ARMv8-A is a 64-bit architecture. It provides 31 general purpose registers which can hold 32-bit values in registers `w0-w30` or 64-bit values in registers `x0-x30`. It provides SIMD (Single Instruction Multiple Data) instruction set, which can process 128 bit data per instruction on average. The SIMD vectorization is possible for same data per vector registers and it does not allow carry handling. There are 32 128-bit registers (`v0-v31`), which can be divided into lanes which are 8, 16, 32, or 64 bits wide. They are defined via operand suffix `b`, indicated byte, `h` indicates half-word, `s` indicates word, `d` indicated double-word. For instance, `v0.8h` create a vector with eight 16-bit elements. Single element of a vector can be accessed via square brackets (e.g. `v0.4s[0]` is the first 32-bit element of the vector `v0`, and `v1.8h[2]` is the third 16-bit element of the vector `v1`).

The ARMv8-A has a various SIMD instructions. Load and store operations are performed by using `LD` and `ST` operations. Each has 4 types according to the degree of interleaving: `LD1/ST1`, `LD2/ST2`, `LD3/ST3` and `LD4/ST4`. For example, `LD1` fills the vector `va` first, and continues to fill the vector `vb` later. However, `LD2` fills the vectors `va` and `vb` simultaneously, that is, one element for `va` and the next element for `vb`, another element for `va` again and so on. `LD3` follows a similar order for the vectors `va`, `vb` and `vc`. `ZIP1/ZIP2` zip two vectors into a single vector according to even/odd indices. `UZP1/UZP2` concatenate even or odd elements from two vectors. The `SXTL/SXTL2` instructions widens the lower/upper halfs of the source register (e.g. widens 8-bit elements to 16-bit elements). `TBL` is an instruction used for permutation according the indices given in a look-up table. `SSHR/USHR` performs vectorized signed/unsigned right shift operations. `AND/ORR` are bitwise and/or operations. `ADD/SUB` performs vectorized addition/subtraction. `MUL` performs vectorized multiplication restricted to the vector element size, however, `SMULL/SMULL2` perform the actual multiplication and widens the vector element. All of `MUL/SMULL/SMULL2` also support multiplication by a scalar, that is, all the vector elements are multiplied with a single scalar element. Moreover, multiplication can be combined with addition or subtraction within a single instruction. For example, `MLA/MLS` adds/subtracts the product to/from the original value (i.e. MLS c, a, b performs $c \leftarrow c + ab$ and MLA c, a, b performs $c \leftarrow c - ab$). The details of ARMv8-A architecture can be found in [3].

4 Implementation Details

Kyber performs the mathematical operations on the polynomials in R_q. Each polynomial, and its representation in NTT-domain, can be serialized as a vector of length 256 as follows:

$$f = \sum_{i=0}^{255} f_i X^i \rightarrow (f_0, f_1, \ldots, f_{255})$$

and

$$\hat{f} = \mathsf{NTT}(f) = (\hat{f}_0 + \hat{f}_1 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X) \rightarrow (\hat{f}_0, \ldots, \hat{f}_{255}).$$

The addition and subtraction of two polynomials f and g can be performed component-wise on their serialized forms: $f_i \pm g_i$ or $\hat{f}_i \pm \hat{g}_i$. However, the multiplication is only performed in NTT domain (for efficiency) by multiplying component-wise pairs: $(\hat{f}_{2i}, \hat{f}_{2i+1}) \circ (\hat{g}_{2i}, \hat{g}_{2i+1})$. Modular reductions can also be performed component-wise: $f_i \mod q$, or $\hat{f}_i \mod q$.

In our implementation, the basic goal is to vectorize the input and take the advantage of SIMD operations on ARM. As $q = 3329$ is a 12-bit integer, the input values can be stored in 16-bit (or multiples of 16-bit). Later, 16-bit values are vectorized in `vx.8h` registers. In addition, 32-bit values are vectorized in `vx.4s` registers, if needed. Here, `x` is the vector index in $\{0, 1, \ldots, 31\}$.

4.1 Reduction

For a given 16-bit integer a, Barrett reduction computes the centered representative congruent to $a \bmod q$, that is, the unique integer x in the interval $\left[-\frac{q-1}{2}, ..., \frac{q-1}{2}\right]$ such that $x = a \bmod q$. It uses a special constant value $r = \lfloor (2^{26} + \lfloor q/2 \rfloor)/q \rfloor$, which is 20159 as $q = 3329$. On the other hand, for a given 32-bit integer a, Montgomery reduction computes 16-bit integer congruent to $aR^{-1} \bmod q$, where $R = 2^{16}$, in the interval $[-q + 1, \ldots, q - 1]$.

We use the vectorized form of Barrett reduction as given in Listing 1. We use the vectorized form of Montgomery reduction inplaced in `tomont` (see Listing 2) and `fqmul` (see Listing 3) functions. The `tomont` function performs the conversion of polynomial coefficients from normal domain to Montgomery domain by multiplying them with $t = 2^{32} \bmod q$ first and by applying the Montgomery reduction later. As $q = 3329$, the constant values in `tomont` are $q' = 62209 = q^{-1} \bmod 2^{16}$ and $t = 1353$. Moreover, the `fqmul` function performs the multiplication of two $\mathbb{Z}_q$-elements and then apply the Montgomery reduction. It uses the constant value q' as defined before. In the comments of the listings, $(x)_{hi}$ and $(x)_{lo}$ refer to the most significant and the least significant 16-bit of a 32-bit integer x, respectively.

Listing 1: `BARR`: Vectorized Barrett Reduction ($r = \lfloor (2^{26} + \lfloor q/2 \rfloor)/q \rfloor$)

Input	: `va.8h` $= [\mathtt{a}_0, \mathtt{a}_1, ..., \mathtt{a}_7]$ and `vq.8h` $= [\mathtt{q}, \mathtt{r}, ...]$ (`vx` is an intermediate vector)	
Output	: `va.8h` $= [\mathtt{a}_0, \mathtt{a}_1, ..., \mathtt{a}_7]$	$(a_0, a_1, \ldots, a_7)$
1: `SQDMULH`	`vx.8h, va.8h, vq.h[1]`	$\triangleright\ x \leftarrow (2 \cdot a \cdot r)_{hi}$
2: `SSHR`	`vx.8h, vx.8h, 11`	$\triangleright\ x \leftarrow x \gg 11$
3: `MLS`	`va.8h, vx.8h, vq.h[0]`	$\triangleright\ a \leftarrow a - q \cdot x$

Listing 2: TOMONT: Vectorized conversion of polynomial coefficients from normal domain to Montgomery domain ($q' = q^{-1} \bmod 2^{16}$ and $t = 2^{32} \bmod q$)

Input : va.8h $= [a_0, a_1, ..., a_7]$ and vq.8h $= [q, q', t, ...]$
(vx, vy, vz, vt are intermediate vectors)
Output : va.8h $= [a_0, a_1, ..., a_7]$

1: MUL	vx.8h, va.8h, vq.h[2]	$\triangleright\ x \leftarrow (t \cdot a)_{lo}$
2: SQDMULH	vy.8h, va.8h, vq.h[2]	$\triangleright\ y \leftarrow (2 \cdot a \cdot t)_{hi}$
3: MUL	vz.8h, vx.8h, vq.h[1]	$\triangleright\ z \leftarrow (q^{-1} \cdot x)_{lo}$
4: SQDMULH	vt.8h, vz.8h, vq.h[0]	$\triangleright\ t \leftarrow (2 \cdot q \cdot z)_{hi}$
5: SHSUB	va.8h, vy.8h, vt.8h	$\triangleright\ a \leftarrow (y - t)/2$

Listing 3: FQMUL: Vectorized Multiplication followed by Montgomery Reduction

Input : Vectors va.8h $= [a_0, ..., a_7]$, vb.8h $= [b_0, ..., b_7]$ and vq.8h $= [q, q', ...]$
(vx, vy, vz, vt are intermediate vectors)
Output : vc.8h $= [c_0, c_1, ..., c_7]$

1: MUL	vx.8h, va.8h, vb.8h	$\triangleright\ x \leftarrow (a \cdot b)_{lo}$
2: SQDMULH	vy.8h, va.8h, vb.8h	$\triangleright\ y \leftarrow (2 \cdot a \cdot b)_{hi}$
3: MUL	vz.8h, vx.8h, vq.h[1]	$\triangleright\ z \leftarrow (q^{-1} \cdot x)_{lo}$
4: SQDMULH	vt.8h, vz.8h, vq.h[0]	$\triangleright\ t \leftarrow (2 \cdot q \cdot z)_{hi}$
5: SHSUB	vc.8h, vy.8h, vt.8h	$\triangleright\ c \leftarrow (y - t)/2$

4.2 NTT Operations

In NTT, the state-of-art computation is performed using Butterfly operations. As $n = 256 = 2^8$ in Kyber, the Butterfly operations are performed in 7 levels. In each level, the serialized representation $(f_0, f_1, \ldots, f_{255})$ are filled into the 8×16-bit vectors, and each two vectors (according to some distance in each level) are updated using Butterfly operation in NTT (see Listing 4). In the end, the its NTT representation (i.e. $(\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{255})$) is obtained. Vice versa, a vector in NTT domain can also be transformed to the normal domain by using the Butterfly operation in NTT^{-1} (see Listing 5).

4.3 Polynomial Operations

Polynomials stored in the vectors are simply added or subtracted using ADD or SUB instructions as many times as needed. For the multiplication, two linear polynomials $a_0 + a_1X$ and $b_0 + b_1X$ are multiplied to compute their product $c_0 + c_1X$ in modulo $X^2 - \zeta_k$ as mentioned in Sect. 2.1. For this purpose, we use the BASEMUL function (see Listing 6) as the vectorized multiplication of two linear polynomials.

Listing 4: Butterfly operation in NTT

Input	: va.8h $= [a_0, a_1, ..., a_7]$	$(a_0, a_1, \ldots, a_7)$
	vb.8h $= [b_0, b_1, ..., b_7]$	$(b_0, b_1, \ldots, b_7)$
	vz.8h $= [z_0, z_1, ..., z_7]$	$(\zeta_0, \zeta_1, \ldots, \zeta_7)$
	(vc is an intermediate vector)	
Output	: va.8h $= [a_0, a_1, ..., a_7]$	$(a_0, a_1, \ldots, a_7)$
	vb.8h $= [b_0, b_1, ..., b_7]$	$(b_0, b_1, \ldots, b_7)$
1: FQMUL	vc.8h, vz.8h, vb.8h	
2: SUB	vb.8h, va.8h, vc.8h	$\triangleright\ b \leftarrow a - b \cdot \zeta$
3: ADD	va.8h, va.8h, vc.8h	$\triangleright\ a \leftarrow a + b \cdot \zeta$

Listing 5: Butterfly operation in NTT^{-1}

Input	: va.8h $= [a_0, a_1, ..., a_7]$	$(a_0, a_1, \ldots, a_7)$
	vb.8h $= [b_0, b_1, ..., b_7]$	$(b_0, b_1, \ldots, b_7)$
	vz.8h $= [z_0, z_1, ..., z_7]$	$(\zeta_0, \zeta_1, \ldots, \zeta_7)$
	(vc is an intermediate vector)	
Output	: va.8h $= [a_0, a_1, ..., a_7]$	$(a_0, a_1, \ldots, a_7)$
	vb.8h $= [b_0, b_1, ..., b_7]$	$(b_0, b_1, \ldots, b_7)$
1: MOV	vc.16b, va.16b	$\triangleright\ c \leftarrow a$
2: ADD	va.8h, vc.8h, vb.8h	$\triangleright\ a \leftarrow b + c$
3: BARR	va.8h	$\triangleright\ a \leftarrow$ BarrettRed(a)
4: SUB	vb.8h, vc.8h, vb.8h	$\triangleright\ b \leftarrow b - c$
5: FQMUL	vb.8h, vz.8h, vb.8h	$\triangleright\ b \leftarrow b \cdot \zeta$

4.4 Noise Sampling

Vectors are sampled according to B_2 or B_3 since $\eta \in \{2, 3\}$. We use CBD2 (see Listing 7) and CBD3 (see Listing 8) when $\eta = 2$ and $\eta = 3$, respectively. We initialize some vector registers (as many as needed) in the beginning: the vectors vmi are used for masking and the vector vs is used for shuffling. As mentioned in Sect. 2.3, every 4 bits (resp. 6 bits) produce an output when $\eta = 2$ (resp. $\eta = 3$). Therefore, CBD2 takes a 128-bit input and produces 32 output values (which are stored in vc0.8h, vc1.8h, vc2.8h and vc3.8h). Similarly, CBD3 takes a 96-bit input and produces 16 output values (which are stored in vc0.8h and vc1.8h). For these functions, we mainly followed the steps in Kyber's AVX implementation given in [6].

4.5 Symmetric Functions

As described in Table 2, Kyber requires SHAKE-128/256, SHA-256/512, SHA3-256/512, and AES-256 for XOF, H, G, PRF, and KDF. For hash functions, we utilized the implementation provided by PQClean [13]. For the AES implementation, we utilized the AES accelerator in the target board. If the board does not support the AES accelerator, we utilized PQClean AES implementations.

Listing 6: BASEMUL: Vectorized multiplication of two linear polynomials

Input : va0.8h $= [\mathtt{a0}_0, ..., \mathtt{a0}_7]$ and va1.8h $= [\mathtt{a1}_0, ..., \mathtt{a1}_7]$ as $a_0 + a_1X$
vb0.8h $= [\mathtt{b0}_0, ..., \mathtt{b0}_7]$ and vb1.8h $= [\mathtt{b1}_0, ..., \mathtt{b1}_7]$ as $b_0 + b_1X$
vz.8h $= [\mathtt{z}_0, -\mathtt{z}_0, ..., \mathtt{z}_3, -\mathtt{z}_3]$ as ζ values
(vd is an intermediate vector)
Output : vc0.8h $= [\mathtt{c0}_0, ..., \mathtt{c0}_7]$ and vc1.8h $= [\mathtt{c1}_0, ..., \mathtt{c1}_7]$ as $c_0 + c_1X$

```
1: FQMUL  vc0.8h, va1.8h, vb1.8h
2: FQMUL  vc0.8h, vc0.8h, vz.8h          ▷ c0 ← a1 · b1 · ζ
3: FQMUL  vd.8h, va0.8h, vb0.8h
4: ADD    vc0.8h, vc0.8h, vd.8h          ▷ c0 ← c0 + a0b0
5: FQMUL  vc1.8h, va0.8h, vb1.8h
6: FQMUL  vd.8h, va1.8h, vb0.8h
7: ADD    vc1.8h, vc1.8h, vd.8h          ▷ c1 ← a0 · b1 + a1 · b0
```

Listing 7: CBD2: Vectorized noise sampling for $\eta = 2$

Input : va.16b $= [\mathtt{a}_0, \mathtt{a}_1, ..., \mathtt{a}_{15}]$, (input values)
vm0.16b $= [\mathtt{0x55}, ..., \mathtt{0x55}]$, vm1.16b $= [\mathtt{0x33}, ..., \mathtt{0x33}]$,
vm2.16b $= [\mathtt{0x03}, ..., \mathtt{0x03}]$, vm3.16b $= [\mathtt{0x0F}, ..., \mathtt{0x0F}]$ (masking)
(vd, ve, vf are intermediate vectors)
Output : vc0.8h $= [\mathtt{c0}_0, \mathtt{c0}_1, ..., \mathtt{c0}_7]$ vc1.8h $= [\mathtt{c1}_0, \mathtt{c1}_1, ..., \mathtt{c1}_7]$
vc2.8h $= [\mathtt{c2}_0, \mathtt{c2}_1, ..., \mathtt{c2}_7]$ vc3.8h $= [\mathtt{c3}_0, \mathtt{c3}_1, ..., \mathtt{c3}_7]$

```
 1: USHR   vd.8h, va.8h, 1
 2: AND    va.16b, va.16b, vm0.16b
 3: AND    vd.16b, vd.16b, vm0.16b
 4: ADD    va.16b, va.16b, vd.16b
 5: USHR   vd.8h, va.8h, 2
 6: AND    va.16b, va.16b, vm1.16b
 7: AND    vd.16b, vd.16b, vm1.16b
 8: ADD    va.16b, va.16b, vm1.16b
 9: SUB    va.16b, va.16b, vd.16b
10: USHR   vd.8h, va.8h, 4
11: AND    va.16b, va.16b, vm3.16b
12: AND    vd.16b, vd.16b, vm3.16b
13: SUB    va.16b, va.16b, vm2.16b
14: SUB    vd.16b, vd.16b, vm2.16b
15: ZIP1   ve.16b, va.16b, vd.16b
16: ZIP2   vf.16b, va.16b, vd.16b
17: SXTL   vc0.8h, ve.8b
18: SXTL2  vc1.8h, ve.16b
19: SXTL   vc2.8h, vf.8b
20: SXTL2  vc3.8h, vf.16b
```

Listing 8: CBD3: Vectorized noise sampling for $\eta = 3$

Input : va.16b $= [a_0, a_1, ..., a_7]$
vs.16b $= [-1, 11, 10, 9, -1, 8, 7, 6, -1, 5, 4, 3, -1, 2, 1, 0]$, (shuffle)
vm0.4s $=$ [0x00249249, ...], vm1.4s $=$ [0x006DB6DB, ...]
vm2.4s $=$ [0x00000007, ...], vm3.4s $=$ [0x00070000, ...],
vm4.4s $=$ [0x00030003, ...], (masking)
(vd is an intermediate vectors)

Output : vc0.4s $= [c0_0, c0_1, c0_2, c0_3]$ and vc1.4s $= [c1_0, c1_1, c1_2, c1_3]$

```
 1: TBL   va.16b, va.16b, vs.16b
 2: USHR  vd.4s, va.4s, 1
 3: USHR  vc0.4s, va.4s, 2
 4: AND   va.16b, va.16b, vm0.16b
 5: AND   vd.16b, vd.16b, vm0.16b
 6: AND   vc0.16b, vc0.16b, vm0.16b
 7: ADD   va.4s, va.4s, vd.4s
 8: ADD   va.4s, va.4s, vc0.4s
 9: USHR  vd.4s, va.4s, 3
10: ADD   va.4s, va.4s, vm1.4s
11: SUB   va.4s, va.4s, vd.4s
12: SHL   vd.4s, va.4s, 10
13: USHR  vc0.4s, va.4s, 12
14: USHR  vc1.4s, va.4s, 2
15: AND   va.16b, va.16b, vm2.16b
16: AND   vd.16b, vd.16b, vm3.16b
17: AND   vc0.16b, vc0.16b, vm2.16b
18: AND   vc1.16b, vc1.16b, vm3.16b
19: ADD   va.8h, va.8h, vd.8h
20: ADD   vd.8h, vc0.8h, vc1.8h
21: SUB   va.8h, va.8h, vm4.8h
22: SUB   vd.8h, vd.8h, vm4.8h
23: ZIP1  vc0.4s, va.4s, vd.4s
24: ZIP2  vc1.4s, va.4s, vd.4s
```

5 Performance Results

Benchmark results were measured both ARM and Apple chips. The ARM board is on Google Pixel 3 Android smartphone. The processor (Snapdragon 845) on it has 8 cores including 4 of ARM Cortex-A53 (@1.77 GHz) and 4 of ARM Cortex-A75 (@2.8 GHz) based. Performance results are taken by using Cortex-A75 processor. The executable is `aarch64` cross-compiled on Linux operating system (Ubuntu 20.04) with `gcc-9`.

The Apple board is on iPad mini 5-th generation. The processor (A12 Bionic) on it has 6 cores including 2 of Vortex (@2.49 GHz) and 4 of Tempest (@1.54 GHz) based. Performance results are taken by using Vortex processor on Apple operating system (iPadOS 14.3).

The reference C code is originally obtained from [13] as the clean format of Kyber Round 3 submission [6]. Then, cycle count function is changed as how is written in Microsoft's SIDH code [17]. The clock is set as `CLOCK_MONOTONIC` which gives more accurate results than `CLOCK_REALTIME`.

Results shown in the Tables 3 and 4 are median values for 1,000 tests. The Table 3 shows reference and optimized implementation performance results for the arithmetic functions in Kyber. Notice that these results are same for all Kyber variants, because each Kyber variant has the same number of polynomial coefficients (e.g. $n = 256$). The overall performance results of key generation (K), encapsulation (E) and decapsulation (D) for all Kyber variants (including Kyber-90s) are presented in Tables 4. They show that the optimized implementation is $\sim$2$\times$ faster than reference implementation even though the arithmetic functions are optimized $\sim$5$\times$ faster. The main reason here is that the hashing operations mainly in the matrix generation part and in other various sums up to a big portion of the timing results as it is also indicated in the paper [1]. Detailed percentages of these functions are illustrated in the Fig. 2.

Table 3. Comparison of clock cycles for functions of Kyber schemes on 64-bit ARM Cortex-A75@2.8 GHz. (Ref-C: Reference C implementation [13]. Opt: Our optimized implementation.)

Functions	Timing [cc] Ref-C [13]	Opt	Ref-C [13]/Opt
Reduction			
poly_tomont (Montgomery Red)	1,896	437	4.34
poly_reduce (Barrett Red)	2,187	294	7.44
NTT			
poly_ntt (NTT+Barrett Red)	11,228	1750	6.42
poly_invntt_tomont (InvNTT)	17,500	2624	6.67
poly_basemul_montgomery	5,396	1168	4.62

5.1 Cryptography Extension for Kyber-90s

64-bit ARMv8 Cortex-A processor supports cryptography extension, which accelerates AES encryption, SHA-1, SHA-224, and SHA-256[1]. In CT-RSA'15, compact implementations of AES-GCM were presented [9]. They utilized new cryptography instructions including 64-bit polynomial multiplication (e.g. `PMULL` and `PMULL2`) and AES operations (e.g. `AESE` (AddRoundKey, SubBytes, and ShiftRows) and `AESMC` (MixColumns)) for high-performance. In PQCrypto'18, SPHINCS with different cryptographic hash functions on ARMv8-A platform

[1] Recent ARM architecture even supports SHA-3, SHA-512, SM3, and SM4 functions.

was presented [15]. The implementation of SHA256 is optimized with cryptography extension (`SHA256H`, `SHA256H2`, `SHA256SU0`, and `SHA256SU1`). HARAKA implementation is optimized with AES extension. These dedicated instruction sets are also beneficial for a variant of Kyber, namely Kyber-90s, suggested by Kyber team. This new scheme utilizes AES-256 in counter mode and SHA2 instead of SHAKE. Kyber512-90s can be further optimized with AES-256 accelerator. We evaluated Kyber512-90s on 64-bit Apple A12 processors@2.49 GHz. Reference implementations require 279,751, 292,742, and 305,511 clock cycles for key generation, encryption, and decryption while optimized implementations with ARM64 assembly and AES-256 accelerator require 32,640, 42,158, and 36,982 clock cycles for key generation, encryption, and decryption, respectively. The implementation with accelerator shows 8.57×, 6.94×, and 8.26× faster than the implementation without the AES accelerator.

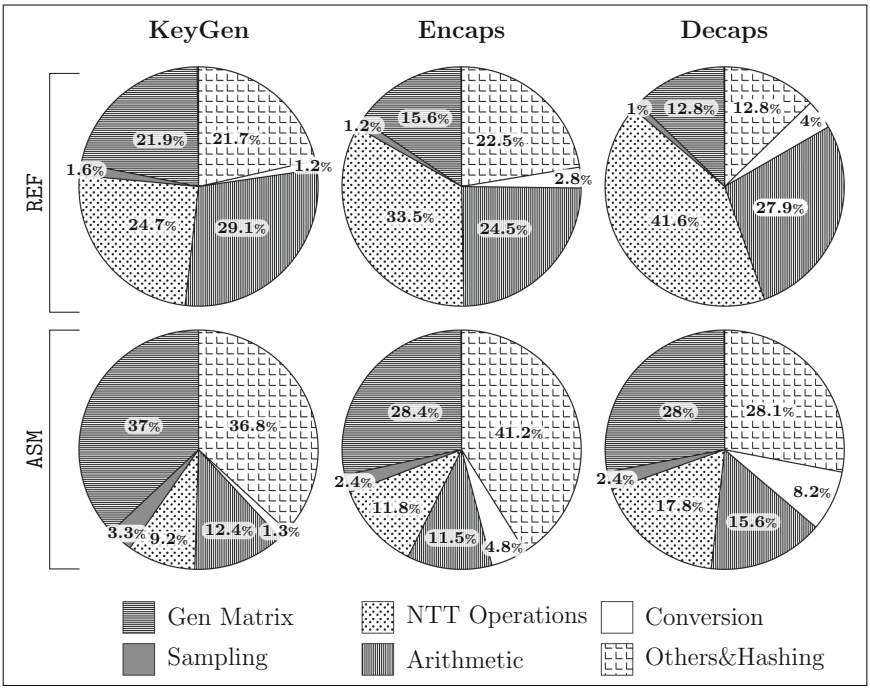

Fig. 2. Percentages of used functions in Keygen, Encapsulation and Decapsulation

Table 4. Comparison of clock cycles for Kyber schemes. (Ref-C: Reference C implementation [13]. Opt: Our optimized implementation.)

		ARM Cortex-A75 @2.8 GHz			Apple A12 @2.49 GHz w/AES accelerator		
Schemes		Timing [cc] Ref-C	Opt	Ref-C/Opt	Timing [cc] Ref-C	Opt	Ref-C/Opt
Kyber512	K	145.8	81.7	1.79	60.4	34.9	1.78
	E	205.2	104.9	1.96	77.7	37.7	2.06
	D	248.5	101.9	2.44	94.6	37.2	2.53
Kyber768	K	247.5	138.0	1.79	106.0	62.2	1.70
	E	327.8	173.4	1.89	131.9	60.8	2.16
	D	383.0	168.6	2.27	146.7	60.0	2.44
Kyber1024	K	385.1	222.7	1.73	171.2	95.2	1.79
	E	476.7	262.8	1.81	182.2	93.0	1.95
	D	546.0	257.7	2.12	209.1	91.0	2.29
Kyber512-90s	K	270.5	205.6	1.32	279.7	32.6	8.57
	E	334.5	236.7	1.41	292.7	42.1	6.94
	D	375.1	230.7	1.63	305.5	37.0	8.26
Kyber768-90s	K	491.7	379.2	1.30	554.2	56.4	9.82
	E	581.4	426.1	1.36	576.0	64.5	8.92
	D	632.9	417.1	1.52	590.7	57.0	10.35
Kyber1024-90s	K	790.3	625.8	1.26	941.9	87.1	10.80
	E	897.3	680.5	1.32	964.8	93.8	10.28
	D	959.6	669.4	1.43	983.0	83.5	11.76

6 Conclusion

This paper presented several optimization techniques to efficiently implement Kyber-KEM on 64-bit ARM processors. We proposed optimizations for primitive operations of Kyber and symmetric functions to accelerate the execution time. A combination of these optimizations achieved 1.79×, 1.96×, and 2.44× faster than previous Kyber512 implementations for key generation, encapsulation, and decapsulation, which set new speed records for Kyber-KEM on an 64-bit ARM processor.

Acknowledgment. The authors would like to thank the reviewers for their comments. This work is supported in parts by a grant from NSF-2101085.

References

1. Alkim, E., Alper Bilgin, Y., Cenk, M., Gérard, F.: Cortex-M4 optimizations for $\{R, M\}$ LWE schemes. IACR Trans. Crypt. Hardware Embed. Syst. **2020**(3), 336–357 (2020). https://doi.org/10.13154/tches.v2020.i3.336-357, https://tches.iacr.org/index.php/TCHES/article/view/8593
2. Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., Petri, R.: ISA extensions for finite field arithmetic: accelerating Kyber and NewHope on RISC-V. IACR Trans. Crypt. Hardware Embed. Syst. **2020**(3), 219–242 (2020). https://doi.org/10.13154/tches.v2020.i3.219-242, https://tches.iacr.org/index.php/TCHES/article/view/8589
3. ARM: ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/fc/. Accessed 15 Jan 2021
4. Bisheh-Niasar, M., Azarderakhsh, R., Mozaffari-Kermani, M.: High-speed NTT-based polynomial multiplication accelerator for CRYSTALS-kyber post-quantum cryptography. Cryptology ePrint Archive, Report 2021/563 (2021). https://eprint.iacr.org/2021/563
5. Bos, J., et al.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 353–367. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00032
6. Bos, J., et al.: Kyber project. https://github.com/pq-crystals/kyber. Accessed 12 Dec 2020
7. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-efficient high-speed implementation of Kyber on Cortex-M4. In: Buchmann, J., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2019. LNCS, vol. 11627, pp. 209–228. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23696-0_11
8. Chen, Z., Ma, Y., Chen, T., Lin, J., Jing, J.: Towards efficient Kyber on FPGAs: a processor for vector of polynomials. In: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 247–252 (2020). https://doi.org/10.1109/ASP-DAC47756.2020.9045459
9. Gouvêa, C.P.L., López, J.: Implementing GCM on ARMv8. In: Nyberg, K. (ed.) Topics in Cryptology — CT-RSA 2015. LNCS, vol. 9048, pp. 167–180. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16715-2_9
10. Greconici, D.: Kyber on RISC-V. Master's Thesis (2020). https://www.ru.nl/publish/pages/769526/denisa_greconici.pdf
11. Gupta, N., Jati, A., Chauhan, A.K., Chattopadhyay, A.: PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber. IEEE Trans. Parallel Distrib. Syst. **32**(3), 575–586 (2021). https://doi.org/10.1109/TPDS.2020.3025691
12. Huang, Y., Huang, M., Lei, Z., Wu, J.: A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. IEICE Electron. Exp. **17**(17), 20200234 (2020). https://doi.org/10.1587/elex.17.20200234
13. Kannwischer, M., Rijneveld, J., Schwabe, P., Stebila, D., Wiggers, T.: The PQClean project. https://github.com/PQClean/PQClean. Accessed 10 Dec 2020
14. Karabulut, E., Aysu, A.: RANTT: a RISC-V architecture extension for the number theoretic transform. In: 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), pp. 26–32 (2020). https://doi.org/10.1109/FPL50879.2020.00016
15. Kölbl, S.: Putting wings on SPHINCS. In: Lange, T., Steinwandt, R. (eds.) PQCrypto 2018. LNCS, vol. 10786, pp. 205–226. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3_10

16. Longa, P., Naehrig, M.: Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Foresti, S., Persiano, G. (eds.) CANS 2016. LNCS, vol. 10052, pp. 124–139. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_8
17. Microsoft: PQCrypto-SIDH project. https://github.com/microsoft/PQCrypto-SIDH. Accessed 13 Dec 2020
18. Ono, T., Bian, S., Sato, T.: Automatic parallelism tuning for module learning with errors based post-quantum key exchanges on GPUs. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5 (2021). https://doi.org/10.1109/ISCAS51556.2021.9401575
19. Schwabe, P., et al.: CRYSTALS-KYBER algorithm specifications and supporting documentation. Technical report, National Institute of Standards and Technology (2020). https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
20. Seiler, G.: Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039 (2018). https://eprint.iacr.org/2018/039
21. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE (1994). https://doi.org/10.1109/SFCS.1994.365700
22. Xing, Y., Li, S.: A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. IACR Trans. Cryptogr. Hardware Embed. Syst. **2021**(2), 328–356 (2021). https://doi.org/10.46586/tches.v2021.i2.328-356, https://tches.iacr.org/index.php/TCHES/article/view/8797
23. Yaman, F., Mert, A.C., Ö-ztürk, E., Savaş, E.: A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER. PQC scheme. Cryptology ePrint Archive, Report 2021/485 (2021). https://eprint.iacr.org/2021/485