Vectorized Implementation of Crystals-Kyber on NEON extension

YoungBeom Kim and Seog Chung Seo*

Department of Financial Information Security, Kookmin University, Seoul, South Korea
{darania, scseo}@kookmin.ac.kr

Abstract

Compared to Elliptic Curve Cryptography, Post-Quantum Cryptography generally demands lower performance and higher memory resources. Recently, there has been active research on neon-based parallel implementations for the 64-bit ARMv8-based Cortex-A series. However, investigation into implementing Post-Quantum Cryptography on the widely used ARMv7-based Cortex-A series, which is prevalent in the industry, has been lacking. Therefore, in this paper, we present the first optimized implementation of Crystals-Kyber, a key encapsulation mechanism standard selected by NIST, specifically tailored for the 32-bit ARMv7-based Cortex-A series environment. We finely tune the widely used signed Montgomery multiplication and Barrett multiplication in order to take full advantage of ARMv7's NEON capability. Specifically, we propose modifications to the internal parameters and propose changes to operations of Montgomery and Barrett arithmetic in order to preserve the parallelism logic. Additionally, to accelerate NTT-based polynomial multiplication, we present a merging technique tailored for the NEON engine of ARMv7. Finally, in comparison with the vectorized Reference code, our proposed approach realizes performance enhancements of 62%, 50%, and 56% for NTT, Point multiplication, and NTT^{-}1, respectively. Within the Kyber scheme, benchmarked on Kyber768, we achieve performance improvements of 50%, 43%, and 52% for the KeyGen, EnCapsulation, and DeCapsulation processes, respectively.

Keyword: PQC, Crystals-Kyber, Number Theoretic Transform (NTT), NEON

1 Introduction

The development of quantum computers is threatening the security of currently used public-key cryptosystems such as RSA, DSA, and ECC. Since 2016, the National Institute of Standards and Technology (NIST) has begun a competition for standardization of post-quantum cryptography (PQC) and four algorithms (Crystals-Kyber [1], Crystals-Dilithium [2], Falcon [3], and SPHINCS+ [4]) were selected last year. Among four algorithms, Crystals-Kyber (Kyber) is the only Key Encapsulation Mechanism (KEM) for key establishment, while other algorithms are digital signature schemes.

The security of Kyber is based on Module-LWE (Learning with Error) problem and its main computations are polynomial multiplication over Ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ and rejection-based error sampling process. Since key establishment is one of the most important objectives of public-key schemes and Kyber is the sole KEM algorithm selected by NIST, some researches on optimizing Kyber on several devices have been actively conducted. Until now, Kyber and

The 7th International Conference on Mobile Internet Security (MobiSec'23), Dec. 19-21, 2023, Okinawa, Japan
Article No. 9

*Corresponding author: Department of Financial Information Security, Kookmin University, Seoul, 02707, Republic of Korea, Tel: +82-02-910-5127

Dilithium has been optimized on x86-64-bit CPUs with AVX2 instructions [5], 32-bit Cortex-M0/M3/M4 [6, 7, 8], and 64-bit ARMv8 [9, 10] with NEON instructions.

There are two main concerns regarding optimization: performance optimization and memory usage optimization. In case of devices having sufficient memory, such as 64-bit ARMv8 MCUs and x86-64 CPUs, performance optimization is the most important objective. On the other hand, memory-constrained devices like 32-bit Cortex-M0/M3/M4, not only performance but also memory optimization is also an important factor. Regarding computation efficiency, the main target is to optimize the polynomial multiplication. Kyber specification makes use of number-theoretic transform (NTT)-based method for efficient polynomial multiplication and NTT-based polynomial multiplication consists of three steps: NTT conversion, point-wise multiplication, and inverse NTT (NTT^{-1}) conversion. The main computations of these three steps are modular multiplication and modular addition/subtraction over $\mathbb{Z}_q$, for $q = 3329$. For efficient constant-time modular reduction, Montgomery method and Barrett method have been widely used. In 2018, Seiler's work has shown that signed Montgomery method and signed Barrett method are superior to their unsigned versions [5] in the context of NTT-based polynomial multiplication. From the proposal of signed reduction methods, NTT-based polynomial multiplication using them has been actively optimized on aforementioned devices from 32-bit Cortex-M0/M3/M4 [6, 7, 8] to x84-64-bit CPUs [5] and 64-bit ARMv8 MCUs [9]. Regarding memory usage efficiency, some streaming approaches have been applied to PQM4 projects. However, until now, there is no Kyber optimization research on 32-bit ARMv7-based Cortex-A series, which are widely used for IoT devices.

Motivation: The 32-bit ARMv7-based Cortex A series has prominently found its place in a wide range of IoT applications, from automotive systems and home appliances to medical devices. As Kyber has been earmarked for standardization by NIST, it is necessary to integrate Kyber into the ARMv7-based Cortex-A environments becomes increasingly inevitable. In light of quantum threats, particularly Shor's algorithm, it's paramount to maintain secure communication across these devices, underscoring the importance of dedicated implementation optimization studies. While there have been efforts to optimize Kyber on the ARMv8-based Cortex-A series, the distinctions in their instruction sets, register counts, and sizes call for needing unique implementation strategies in ARMv7-based Cortex A series. Therefore, in this paper, our focus narrows down to pinpointing and discussing these optimization techniques for Kyber implementation. Our research contributions can be summarized as:

Contribution: In this paper, we present the first implementation of Kyber on the ARMv7-based Cortex-A series. Initially, we survey existing arithmetic, subsequently opting for those most amenable to parallel implementation within the ARMv7 environment, namely Montgomery multiplication and Barrett reduction. To facilitate vectorized arithmetic, we recalibrate the parameter sets and adapt each arithmetic operation in alignment with the NEON instruction set. Furthermore, in the context of NTT-based multiplication, we introduce a merging strategy tailored for the ARMv7-based Cortex-A series. Ultimately, in comparison with the vectorized Reference code, our proposed approach realizes performance enhancements of 62%, 50%, and 56% for NTT, Point multiplication, and NTT^{-1}, respectively. Within the Kyber scheme, benchmarked on Kyber768, we achieve performance improvements of 50%, 43%, and 52% for the KeyGen, EnCapsulation, and DeCapsulation processes, respectively.

Paper Organization: This paper is structured as follows. Section 2 goes through the preliminaries, Section 3 describes our insights on modular multiplications using NEON extension of

ARMv7-based Cortex A series, and Section 4 shows the performance numbers of NTT/NTT^{-1} and the overall impact on Kyber. Finally, Section 5 we conclude this paper.

2 Preliminaries

2.1 Crystals-Kyber

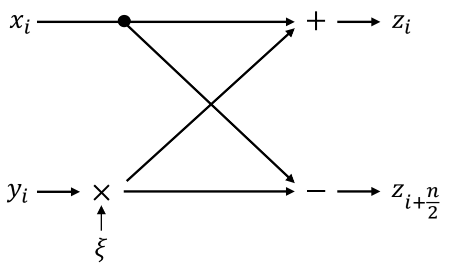
Crystals-Kyber (Kyber) is the only KEM algorithm selected by NIST and its security is based on Module Learning With Error (Module-LWE) problem. Since characteristic of Module-LWE-based kyber, each element in a matrix and a vector is a polynomial over Ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where $n = 256$ and $q = 3329$. The public matrix of Kyber exhibits a size of $\ell \times \ell$, while the secret vector and noise each possess a size of $\ell \times 1$. For security levels 1, 3, and 5, the corresponding values of ℓ are 2, 3, and 4, respectively. Kyber's Public Key Encryption (PKE) consists of three operations: Key generation, Encryption, and Decryption, and Kyber-KEM utilizes Kyber-PKE with Fujisaki-Okamoto transform for providing IND-CCA2 security. For the complete pseudocode structure of Kyber PKE/KEM, please refer to Algorithm 4, 5, and 6 for PKE, and Algorithm 7, 8, and 9 for KEM in Appendix A and B.

Except for the random sampling based on hash function, the core operation of each Kyber-PKE algorithm is either matrix by vector multiplication ($\mathbf{A} \circ \mathbf{s}$) or vector by vector multiplication ($\hat{\mathbf{s}}^T \circ \mathbf{u}$). For example, multiplication with an $\ell \times \ell$ matrix and a $\ell \times 1$ vector requires ℓ^2 polynomial multiplications (vector by vector multiplication requires ℓ polynomial multiplications). In the algorithm description, bold upper-case letters (e.g., $\mathbf{A}$) and bold lower-case letters (e.g., $\mathbf{s}$) denote matrices and vectors, respectively.

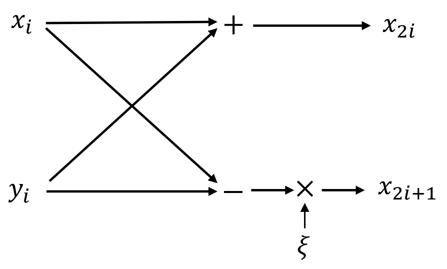
In the framework of Kyber, GenMatrixA generates a public matrix $\hat{\mathbf{A}}$, size of $\ell \times \ell$. The result of this process hinges on the input of a 32-bit parameter known as ρ, which leads to the formation of R_q elements via two established methods: SHAKE and Rejection-sampling. Considering the fact that sampling is performed randomly within the R_q, it can be stated that the public matrix $\mathbf{A}$ is derived from the NTT domain. Hence, it is depicted as $\hat{\mathbf{A}}$, eliminating the necessity for explicit NTT. Simultaneously, SampleVec algorithm plays a pivotal role in Kyber, tasked with producing a secret vector and its corresponding noise, both sized $\ell \times 1$. Kyber leverages the Central Binomial Distribution (CBD) in its sampling process, a decision based on the system's predilection for noise and secret vectors that comprise small coefficients. Compress and Decompress serve as pivotal mechanisms in the efficiency of data transmission and optimization of network resources. These procedures sequentially execute compression and restoration on 8-bit data. The details of the Kyber description can be found in the Kyber specification document [1].

2.2 Number Theoretic Transform (NTT)

Polynomial multiplication typically has a complexity of $\mathcal{O}(n^2)$ when implemented based on the schoolbook method, but in lattice-based Schemes (LBS) systems, there is a preference for multiplication algorithms with lower complexity. The Number Theoretic Transform (NTT) algorithm is one variation of the Discrete Fourier Transform (DFT) and enables efficient polynomial multiplication in Galois Field. In LBS, NTT can be seen as an evaluation of polynomials and, generally, in a $\mathbb{Z}_q[X]/(X^n - 1)$, n evaluation values are required to determine a polynomial with n coefficients. Hence, the primitive n-th root of unity existing in $\mathbb{Z}_q[X]$ is used to generate these evaluation values. For Cyclic NTT that employs the factor polynomial $X^n - 1$, a primitive n-th root of unity (ζ_n) must exist in Z_q to facilitate NTT usage. Meanwhile, Negacyclic



(a) Cooley-Tukey [11]



(b) Gentleman-Sande [12]

Figure 1: Butterfly diagrams of FFT/NTT algorithms

NTT, which uses the factor polynomial $X^n + 1$, necessitates a primitive $2n$-th root of unity (ζ_{2n}). Commonly, the Chinese Remainder Theorem (CRT) and the Fast Fourier Transform (FFT) algorithm are applied for an efficient NTT implementation, utilizing only $\mathcal{O}(n \log n)$ operations.

Through the CRT algorithm, one can generate an isomorphic mapping of Negacyclic NTT $\mathbb{Z}_q[X]/(X^n - 1) \rightarrow \prod_i \mathbb{Z}_q[X]/(X - \zeta_{2n}^{2i+1})$ for $i = 1, 2, \cdots, n-1$, when iterated $\log n$ times, fully decomposes an $n-1$ degree polynomial into n single coefficients. Each iteration is commonly referred to as an NTT layer, where the result of the j-th layer is the remainder of any polynomial a f mod $(X^{2^{j-1}} \pm \zeta_{2n}^{2i+1})$ for some i. The polynomial elements of Kyber belong to $\mathbb{Z}_q[X]/(X^n + 1)$ with $q = 3329$ and $n = 256$, and since there is no 512-th root of unity within the modulus 3329, Kyber employs an incomplete Negacyclic NTT. In other words, an NTT with 7 layers using the primitive 256-th root of unity $\zeta = 17$ is actually implemented, eventually decomposing into a first-degree polynomial. Through the NTT, the original polynomial $X^{256} + 1$ is decomposed as follows:

$$X^{256} + 1 = \prod_{i=0}^{127} (X^2 - \zeta_{2n}^{2i+1}) = \prod_{i=0}^{127} (X^2 - \zeta_{2n}^{2br_7(i)+1})$$

In here $br_7(i)$ for $i \in \{0, \cdots 127\}$ is the bit reversal of the unsigned 7-bit integer i. Finally, in Kyber, any polynomial $f \in R_q$ is represented as follows by the NTT:

$$f \bmod (X^2 - \zeta^{2br_7(0)+1}), f \bmod (X^2 - \zeta^{2br_7(1)+1}), \cdots, f \bmod (X^2 - \zeta^{2br_7(127)+1})$$

To compute the NTT layers efficiently, butterfly operations, typically used in FFT, are utilized. Generally, the Cooley-Tukey (CT) butterfly [13] is used in the NTT and the Gentleman-Sande (GS) butterfly [12] is used in the NTT^{-1} (Please See Figure 1 for illustrations). Since the split polynomials are in bit-reverse order, rearrangement is required in the implementation. However, by using the CT butterfly, one can transition from the normal order to the bit-reverse order at no additional cost. Similarly, using the GS butterfly in the NTT^{-1} enables us to convert the polynomial back to the normal order without incurring any extra cost.

2.3 Modular Arithmetic

Three steps in an NTT-based polynomial multiplication require a number of multiplication over $\mathbb{Z}_q$ (Namely, $a \cdot b \bmod q$). In other words, in NTT conversion (CT method) and inverse NTT conversion (GS method), each coefficient in a polynomial is multiplied with a twiddle factor ζ over $\mathbb{Z}_q$ for factoring into smaller polynomials or combining into lager polynomial and in point-wise multiplication, coefficients of two polynomials of degree 1 are multiplied over $\mathbb{Z}_q$. In Kyber, an element over $\mathbb{Z}_q$ is expressed with a 16-bit data type. Thus, the result of multiplying two elements is 32-bit and it needs to be reduced by q. In other words, $c \bmod q = c - \lfloor c/q \rfloor \cdot q$ ($c = a \cdot b$), where the division operation is typically non-constant-time in modern computers. Thus, an efficient and constant-time reduction is required to prevent timing-based side-channel analysis. First, we introduce the previously proposed modular reduction algorithms in lattice-based cryptography.

As efficient and constant-time reduction algorithms, Montgomery method [14] and Barrett method [15] have been widely used. In addition, there is also research that has recently effectively improved Plantard reduction and applied it to Kyber [16]. In the context of LBS using small prime q, the signed Montgomery method (Algorithm 1) and signed Barrett method (Algorithm 2) were proposed [5] and have been applied to the implementations of Kyber and Dilithium. Signed reduction is denoted by $\text{mod}^{\pm}\ q$ and $c \text{ mod}^{\pm}\ q$ reduces into $(-\frac{q}{2}, \frac{q}{2})$. Actually, a signed reduction is more efficient than an unsigned reduction in the context of LBS, because the result of the reduction does not need to be unsigned [5, 6, 17, 7].

Algorithm 1 Signed Montgomery reduction [5]

```
Input: $c = a \cdot b$ such that $c = c_1\beta + c_0$ and $c \in (-\frac{\beta}{2}q, \frac{\beta}{2}q)$, where $\beta = 2^{16}$ if $q < 2^{16}$, the odd modulus $q \in (0, \frac{\beta}{2})$
Output: $r \equiv ab\beta^{-1} \bmod q$, $r \in (-q, q)$
1: $m = c_0 \cdot q^{-1} \text{ mod}^{\pm}\beta$        ▷ signed low product, $q^{-1}$ is a precomputed
2: $t_1 = \lfloor m \cdot q/\beta \rfloor$        ▷ signed high product, shift right operation
3: $r = c_1 - t_1$
4: return $r$
```

Signed Montgomery reduction is given by Algorithm 1 and the constant β is typically set to l-bit word size such that the modulus q fits in one word. It replaces division with multiplication, shift, and subtraction operations. Note that in the case of subtraction, only the high part ($c_1 - t_1$) of subtraction is required. Since the result r of Algorithm 1 is in the range of $(-q, q)$, it does not need to be unsigned.

Algorithm 2 Signed Barrett reduction [5]

```
Input: $q$ with $0 < q < \frac{\beta}{2}, 2 \nmid q$ and $a$ with $a \in [-\frac{\beta}{2}, \frac{\beta}{2})$, where $\beta = 2^{16}$
Output: $r \equiv a \bmod q$, $r \in [0, q]$
1: $v \leftarrow \lfloor \frac{2^{\log(q)-1} \cdot \beta}{q} \rceil$        ▷ precomputed
2: $t \leftarrow \lfloor \frac{av}{2^{\log(q)-1} \cdot \beta} \rceil$        ▷ signed high product
3: $t \leftarrow tq \bmod \beta$        ▷ signed low product
4: return $r \leftarrow a - t$
```

Barrett reduction [15] replaces the computation of $\lfloor c/q \rfloor$ requiring a division with an approximated quotient computation $\lfloor (c \cdot \lambda)/R) \rfloor$ requiring efficient shift operation where $\lambda = \lfloor R/q \rfloor$ is a precomputed constant, with $R = 2^l$. Algorithm 2 is signed Barrett method [5] used by Kyber reference code and note that the input a can be signed value and the result r is in the range of $0 \leq r \leq q$.

Algorithm 3 Improved Plantard reduction [8]

```
Input: $c = a \cdot b$, where $a, b \in [-q2^\alpha,\ q2^\alpha]$, $q < 2^{l-\alpha-1}$, and $q' = q^{-1} \text{mod}^{\pm} 2^{2l}$
Output: $r = c(-2^{-2l})\ \text{mod}^{\pm} q$, where $r \in (-\frac{q}{2}, \frac{q}{2})$
1: $r = \left[ \left( [[cq']_{2l}]^l + 2^\alpha \right) q \right]^l$
2: return $r$
```

The basic idea of Plantard reduction [16] is similar to Montgomery reduction, aiming to find a value of t such that $(tq - ab)$ is divisible by $R = 2^{2l}$. Ultimately, it should satisfy $(tq - ab)^{-2l} \equiv ab(-2^{-2l}) \bmod q$. Recently, an improved version of Plantard multiplication was proposed in [8] for Cortex-M4, extending the method to a signed system and expanding the input range (Algorithm 3). The improved Plantard reduction method provides a range 2^α times larger ($[-q^2 2^{2\alpha},\ q^2 2^{2\alpha}]$) compared to Montgomery reduction, while maintaining half of the output range $(-\frac{q}{2}, \frac{q}{2})$. Leveraging Cortex-M4's `smulwb` and `smlabb` instructions, and when one of the operands is constant, only two multiplications occur during the modular multiplication process.

In the ARMv8 environment, [9] proposed an advanced approximation, building on the foundational Barrett reduction. Through his research, Becker illustrated a linkage between Barrett and Montgomery multiplications, conclusively identifying the result as a 32-bit value. This breakthrough facilitated vectorization when deploying NEON in the ARMv8 context. However, the efficiency of this method largely hinges on the robust capabilities of the `smull` and `smull2` instruction, which provide high and low returns of extended multiplication. Regrettably, the NEON instruction set available in ARMv7 don't offer the same latitude as their ARMv8 counterparts. Considering these limitations, we aim to introduce a method tailored for Montgomery multiplication and Barrett reduction that aligns with the specifications of the NEON extension in ARMv7 environment.

3 Vectorized NTT using NEON extension

3.1 NEON extension of ARMv7-based Cortex-A series

The ARMv7 NEON, an advanced SIMD extension for ARM Cortex-A processors, is integral for enhancing performance in media-intensive tasks and signal processing. It comprises 32 64-bit registers, which can be paired as 16 128-bit wide units. Supporting diverse integer types and single-precision floating-point, it is adept for a broad range of operations, from basic arithmetic to complex element-wise multiplication. Designed for seamless operation alongside the ARMv7-A main processor, ARMv7 NEON sets a benchmark in versatility, paving the way for subsequent improvements in later architectures like ARMv8-A.

However, when compared to the ARMv8-A NEON, ARMv7 lacks advanced multiplication operations such as polynomial multiplication, complex number multiplication, and rounding versions of multiply-add/subtract instructions. Therefore, it is imperative to utilize the provided vector instructions as efficiently as possible. In this section, we will briefly introduce the instructions that will be leveraged in this paper. `vmov.s32` moves a 32-bit scalar to a single lane of the destination register. `vmlal.s16` performs a long multiply accumulate. `vuzp.s16` unzips 16-bit elements from the source registers. `vmls.s16` multiplies 16-bit elements from the first source register by 16-bit elements from the second source register and subtracts the product from the corresponding 16-bit elements in the destination register. `vqdmulh.s16` performs a saturated doubling multiply high half, returning the high half of the 2x product. `vmul.s16` multiplies 16-bit elements from the first source register by 16-bit elements from the second source register and writes the product to the destination register. `vhsub.s16` subtracts 16-bit elements in the second source register from corresponding elements in the first source register, halves the results.

3.2 Modified Modular reduction

Listing 3 presents a variation of Barrett reduction specifically designed for the Cortex-A series. The goal is to enhance performance while retaining 8 16-bit coefficients within a 128-bit vector register. To achieve this, compared to reference approach(cf. Listing 1), we adjust the parameter of Barrett constant to 16, enabling the utilization of an implicit shift. The finalized assembly code is provided in Listing 5. The assembly code for Barrett reduction employs six NEON instructions. Reductions are executed concurrently for eight coefficients. Although adjusting $R = 2^{26}$ to $R = 2^{16}$ slightly alters the range of the reduction results, it does not materially impact the overall functioning of Kyber.

Listing 4 shows a modification of Montgomery reduction tailored for the Cortex-A series. Unlike the ARMv8-based neon engine, the 32-bit neon engine lacks upper word multiplication or lower word multiplication capabilities (`smull` and `smull2`). Instead, it only supports `vqdmulh` (vector saturating doubling multiply returning high-half). Therefore, we adapt the Montgomery reduction formula to leverage this specific instruction. Failing to utilize the `vqdmulh` instruction would necessitate an additional instruction for long word multiplication. The assembly code incorporating the `vqdmulh` command and `vhsub` (vector halving subtract) is shown in Listing 6.

Unlike Barrett reduction, Montgomery reduction retains the same parameters and proceeds with vectorization. In contrast to the reference implementation where multiplication is followed by reduction, we implement it in a combined manner, that is, as Montgomery multiplication. This approach boasts the advantage of allowing concurrent multiplication and reduction for eight coefficients. Given that accumulation result by multiply-long instruction is not required, it conservatively uses registers. The capability to handle concurrent multiplication for eight coefficients stems from the transformation of Montgomery multiplication for using `vqdmulh` and `vhsub` instruction (refer to Listing 5 and 6).

Listing 1: Ref Barrett C code

```
int16_t barrett_reduce(int16_t a) {
 int16_t t
 const int16_t v = ((1<< 26) + q/2)/q
 t =(int32_t)v*a + (1<< 25) >> 26
 t *= q
 return a-t}
```

Listing 2: Ref Montgomery C code

```
int16_t montgomery_reduce(int32_t a) {
 int16_t t
 t = (int16_t) a * qinv
 t = (a - (int32_t)t * q) >>  16
 return t
 }
```

Listing 3: Our Barrett C code

```
 int16_t our_barrett(int16_t a) {
 int16_t t
 const int16_t v = ((1<< 16) + q/2)/q
 t =(int32_t)v*a + (1<< 15) >> 16
 t *= q
 return a-t}
```

Listing 4: Our Montgomery C code

```
int16_t our_montgomery(int32_t a) {
 int16_t t
 t = (int16_t) a * qinv
 t = {(2*a-(int32_t)t*q*2) / 2} >> 16
 return t
 }
```

Listing 5: Our Barrett asm code

```
.macro mc_barrett_reduce
  vmov.s32 t, #32768
  vmov.s32 t, #32768
  vmlal.s16 t, a0, bar
  vmlal.s16 v, a1, bar
  vuzp.s16 t, v
  vmls.s16 r, v, q
.endm
```

Listing 6: Our Montgomery asm code

```
.macro mc_mont_mul
  vqdmulh.s16 r, a, b
  vmul.s16  r, a, b
  vmul.s16 t, t, qinv
  vqdmulh.s16 t, t, q
  vhsub.s16 r, r, t
.endm
```

3.3 NTT Implementation

We implement NTT using CT butterflies and NTT^{-1} with GS butterflies. To minimize memory access costs, we adopt a merging strategy, extensively researched in [6, 7]. This strategy harnesses the powerful load and store instructions inherent to the ARM architecture. For example, the Cortex-M4 allows us to load/store four coefficients into the register simultaneously, enabling us to perform butterfly operations across multiple layers at once. The number of registers available determines the maximum layers we can merge. Considering the need to retain constants for modular arithmetic in the register, we have found merging up to four layers on the Cortex-M4 to be efficient, as detailed in (3-3-1 or 4-3) by [7]. In the ARMv7 environment, we merge three layers, followed by two, then another two, out of the seven layers in NTT. Similarly, for NTT^{-1}, we merge the seven layers in reverse order, combining them as two, then another two, and finally (3+1). During the last phase of NTT^{-1}, we integrate the conversion process to the integer domain into the final layer, and pre-multiply the Montgomery multiplication of half the coefficients by the twiddle factor. This strategy eliminates the need for 128 Montgomery multiplications. As we employ the GS Butterfly for NTT^{-1}, we invoke Barrett-reduction exclusively in the 3rd and 5th layers. For optimal performance, we unfold one outer and one inner loop.

Table 1: Cycle counts (cc) for NTT on ODROID-XU4

Implementation	NTT (cc)	Point multiplication (cc)	NTT^{-1} (cc)
This work	1,718 (+62%)	1,200 (+50%)	2038 (+56%)
PQM4 [16] (-)	4,474	2,422	4,684
Ref [1]	11,692	8,713	21,456

4 Evaluation

4.1 Benchmarking Setup

We have selected the ODROID-XU4 device, which features an ARM Cortex-A7 MPU, as our target board. For compilation, we employ `gcc 11.3.0` and the following compilation options: `-O3 -mcpu=native -mfpu=neon -ftree-vectorize`. For comparative evaluation, we utilize vectorized versions of the most recent PQM4 code and the reference code. These versions incorporate the Better-accumulation and Asymmetric-multiplication techniques from PQM4. In a pure PQM4 scenario, we use a non-vectorized implementation of the keccak algorithm, ensuring a benchmark comparison without altering the codebase. Both the reference and the proposal presented in this paper undergo comparison using the keccak implementation, optimized with neon-vectorized assembly.

4.2 Performance of Number Theoretic Transform

Although both the Reference and state-of-the-art PQM4 [16] codes automatically enable neon during compilation, PQM4, implemented using ARMv7 assembly, exhibits a slower NTT-based multiplication speed compared to the Reference code. Consequently, we select the Reference code as the comparative baseline for NTT-based multiplication. For ARMv7-based Cortex A environment, Our implementation achieves a performance improvement of 62 %, 50 %, and 56 % in NTT, Point multiplication, and NTT^{-1}, respectively (cf Table 1).

4.3 Performance of Kyber

Similar to the comparison in NTT-based multiplication, we conduct a performance comparison of Kyber against the vectorized Reference code rather than the PQM4 code (Please see Table 2). Our Kyber512 implementation achieves a performance improvement of 51 %, 50 %, and 47 % in KeyGen, EnCapsulation, and DeCapsulation, respectively. For Kyber768, Our work achieves a performance improvement of 48 %, 43 %, and 41 % in KeyGen, EnCapsulation, and DeCapsulation, respectively. Finally, Our Kyber1024 implementation achieves a performance improvement of 57 %, 52 %, and 48 % in KeyGen, EnCapsulation, and DeCapsulation, respectively. The primary reason for the performance enhancement in our implementation lies in the vectorized (fully parallelized) NTT-based multiplication.

Table 2: Cycle counts (cc) for KeyGen, EnCaps, and Decaps of Kyber on ODROID-XU4

work		Kyber512 (cc)	Kyber768 (cc)	Kyber1024 (cc)
This work	K	110k (+51%)	185k (+50%)	294k (+47%)
	E	139k (+48%)	230k (+43%)	348k (+41%)
	D	133k (+57%)	221k (+52%)	339k (+48%)
PQM4 [16]	K	430k	702k	1,119k
	E	526k	861k	1,314k
	D	472k	780k	1,211k
Ref [1] (-)	K	223k	369k	552k
	E	265k	405k	590k
	D	309k	459k	655k

5 Conclusion

In this paper, we have proposed a parallel processing solution for NTT-based polynomial multiplication in the Kyber, leveraging the NEON extension set in an ARMv7 environment. To our knowledge, our work stands as the first parallelized implementation on the ARMv7-based Cortex-A series. Specifically, we introduce a modified parameter set and operation method to maintain consistency in the parallel logic during modular multiplication and single coefficient subtraction. As a result, Barrett reduction becomes faster during parallelization due to an implicit shift, and Montgomery multiplication can undergo parallel processing throughout the multiplication and subtraction process without the need for additional registers and instructions. Additionally, by adopting the latest optimization strategies, we perform merging in each NTT and NTT$^{-}1$ and omit 128 multiplications during the integer domain conversion process in the final layer of InvNTT. Looking forward, we are keen to explore optimizations for the Dilithium digital signature algorithm, another member of the Crystals series.

6 Acknowledgments

This work was partly supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) Grant by the Korean Government through Ministry of Science and ICT (MSIT) (A study on PQC optimization and security protocol migration to neutralize advanced quantum attacks in Beyond 5G-based next-generation IoT computing environments, 50%) under Grant 2022-00207416, and partly supported by IITP grant funded by Korean Government through Ministry of Science and ICT (MSIT) (No. 2021-0-00796, Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security, 50%)

References

[1] 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. *NIST PQC Round*, 2(4):1–43, 2019.

[2] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. Crystals-dilithium. *Algorithm Specifications and Supporting Documentation*, 2020.

[3] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. *Post-Quantum Cryptography Project of NIST*, 2020.

[4] Daniel J Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2129–2146, 2019.

[5] Gregor Seiler. Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018. https://eprint.iacr.org/2018/039.

[6] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for {R, M} lwe schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 336–357, 2020.

[7] Amin Abdulrahman, Vincent Hwang, Matthias J Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-m4. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, pages 853–871. Springer, 2022.

[8] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022.

[9] Hanno Becker, Vincent Hwang, Matthias J Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: faster dilithium, kyber, and saber on cortex-a72 and apple m1. *Cryptology ePrint Archive*, 2021.

[10] Youngbeom Kim, Jingyo Song, Taek-Young Youn, Seog Chung Seo, et al. Crystals-dilithium on armv8. *Security and Communication Networks*, 2022, 2022.

[11] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[12] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

[13] 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.

[14] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[15] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology—CRYPTO'86: Proceedings*, pages 311–323. Springer, 1986.

[16] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, 2021.

[17] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. Ntt multiplication for ntt-unfriendly rings: New speed records for saber and ntru on cortex-m4 and avx2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 159–188, 2021.

A Kyber.CPAPKE Algorithm

Algorithm 4 Kyber.CPAPKE Keygen [1, 8]

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

Algorithm 5 Kyber.CPAPKE Enc [1, 8]

```
Input: $pk = (\hat{\mathbf{b}}, \rho)$, message $\mu$ in $R_q$, seed $coin \in \{0, \cdots, 255\}^{32}$
Output: Ciphertext $(\mathbf{u}', h)$
1: $\hat{\mathbf{A}} \leftarrow$ GenMatrixA$(\rho)$
2: $\mathbf{s}' \leftarrow$ SampleVec$(coin, 0)$
3: $\mathbf{e}' \leftarrow$ SampleVec$(coin, 1)$
4: $\mathbf{e}'' \leftarrow$ SampleVec$(coin, 2)$
5: $\hat{\mathbf{t}} \leftarrow$ NTT$(\mathbf{s}')$
6: $\mathbf{u} \leftarrow$ NTT$^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{t}}) + \mathbf{e}'$
7: $\hat{\mathbf{v}} \leftarrow$ NTT$^{-1}(\hat{\mathbf{v}}^T \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mu$
8: return $(\mathbf{u}' =$ Compress$(\mathbf{u})$, $h =$ Compress$(\mathbf{v}'))$
```

Algorithm 6 Kyber.CPAPKE Dec [1, 8]

```
Input: Ciphertext $c = (\mathbf{u}', h)$, secret key $sk = \hat{\mathbf{s}}$
Output: Message $\mu \in R_q$
1: $\mathbf{u} \leftarrow$ Decompress$(\mathbf{u}')$
2: $\mathbf{v}' \leftarrow$ Decompress$(h)$
3: return $\mu = \mathbf{v}' -$NTT$^{-1}(\hat{\mathbf{s}}^T \circ$ NTT$(\mathbf{u}))$
```

B Kyber.CCAKEM Algorithm

Algorithm 7 Kyber.CCAKEM Keygen [1]

```
Output: Public key $pk$
Output: Secret key $sk$
 1: $z \leftarrow \mathcal{B}^{32}$
 2: $(pk, sk') :=$ Kyber.CPAPKE keygen()
 3: $sk := (sk'||pk||\mathrm{H}(pk)||z)$
 4: return $(pk, sk)$
```

Algorithm 8 Kyber.CCAKEM Encaps [1]

```
Input: Public key $pk$
Output: Ciphertext $c$
Output: Shared key $K$
 1: $m \leftarrow \mathcal{B}^{32}$
 2: $m \leftarrow \mathrm{H}(m)$
 3: $(\overline{K}, r) := \mathrm{G}(m||\mathrm{H}(pk))$
 4: $c :=$ Kyber.CPAPKE Enc$(pk, m, r)$
 5: $K := \mathrm{KDF}(\overline{K}$——$\mathrm{H}(c))$
 6: return $(c, K)$
```

Algorithm 9 Kyber.CCAKEM Decaps [1]

```
Input: Ciphertext $c$
Input: Secret key $sk$
Output: Shared key $K$
 1: $pk := sk + 12 \cdot k \cdot n/8$
 2: $h := sk + 24 \cdot k \cdot n/8 + 32$
 3: $z := sk + 24 \cdot k \cdot n/8 + 64$
 4: $m' :=$ Kyber.CPAPKE Dec$(c, sk)$
 5: $(\overline{K}', r) := \mathrm{G}(m'||h)$
 6: $c' :=$ Kyber.CPAPKE Enc$(pk, m', r')$
 7: if $c = c'$ then
 8:     return $(K := \mathrm{KDF}(\overline{K}'||\mathrm{H}(c))$
 9: else
10:     return $(K := \mathrm{KDF}(z$——$\mathrm{H}(c))$
11: return $K$
```