Hindawi
Security and Communication Networks
Volume 2022, Article ID 5226390, 12 pages
https://doi.org/10.1155/2022/5226390

Research Article

Crystals-Dilithium on ARMv8

Youngbeom Kim,[1] **Jingyo Song**,[1] **Taek-Young Youn**,[2] **and Seog Chung Seo**[1]

[1]*Department of Financial Information Security, Kookmin University, Seoul, Republic of Korea*
[2]*Department of Industrial Security, Dankook University, Yongin, Gyeonggi-do, Republic of Korea*

Correspondence should be addressed to Taek-Young Youn; taekyoung@dankook.ac.kr and Seog Chung Seo; seoseogchung82@gmail.com

Received 13 October 2021; Accepted 5 February 2022; Published 27 February 2022

Academic Editor: Zhe-Li Liu

Copyright © 2022 Youngbeom Kim et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Crystals-Dilithium is one of the digital-signature algorithms in NIST's ongoing post-quantum cryptography (PQC) standardization final round. Security and computational efficiency concerning software and hardware implementations are the primary criteria for PQC standardization. Many studies were conducted to efficiently apply Dilithium in various environments; however, they are focused on traditionally used PC and 32-bit Advanced RISC Machine (ARM) processors (Cortex-M4). ARMv8-based processors are more advanced embedded microcontrollers (MCUs) and have been widely used for various IoT devices, edge computing devices, and On-Board Units in autonomous driving cars. In this study, we present an efficient Crystals-Dilithium implementation on ARMv8-based MCU. To enhance Dilithium's performance, we optimize number theoretic transform (NTT)-based polynomial multiplication, the core operation of Dilithium, by leveraging ARMv8's architectural properties such as large register sets and NEON engine. We apply task parallelism to NTT-based polynomial multiplication using the NEON engine. In addition, we reduced the number of memory accesses during NTT-based polynomial multiplication with the proposed merging and register-holding techniques. Finally, we present an interleaved NTT-based multiplication simultaneously executed with ARM processor and NEON engine. This implementation can further optimize performance by eliminating the ARM processor latency with NEON overheads. Through the proposed optimization methods, for Dilithium 3, we achieved a performance improvement of about 43.83% in key pair generation, 113.25% in signing, and 41.92% in verification compared to the reference implementation submitted to the final round of the NIST PQC competition.

1. Introduction

In the communication network field, sensor nodes and devices use cryptographic protocol with digital-signature and key-exchange algorithms for integrity and confidentiality [1, 2]. With the development of technology, the number of sensor nodes has increased dramatically compared to the past, and accordingly, research on the definition of security standards and vulnerabilities for the increased nodes has been actively conducted [2–5]. In addition, various applications that use encryption systems have emerged in relation to privacy protection, such as image encryption [6, 7].

However, as Google developed a 72 q-bit quantum computer, a fatal issue arose for the existing cryptographic system. It is solved in polynomial time in a public-key cryptography security system based on the factorization and discrete logarithm using the Shor algorithm [8] within a quantum environment. Recognizing this issue, NIST held the post-quantum cryptography standardization for key encapsulation mechanism (KEM) and digital signature in 2016 to replace the international standard public-key cryptography. Candidates for the final round of the PQC standardization were recently announced. The KEM algorithms are *Crystals-Kyber, SABER, NTRU*, and *Classic McEliece*, and the digital signatures are *Rainbow, Falcon, and Crystal-Dilithium*. Except for *Classic McEliece* and *Rainbow*, all finalists use lattice-based cryptography.

The multivariable-based *Rainbow* has fast signature generation and verification and a very short signature length among the digital-signature algorithms [9]. However, due to the high key generation cost caused by the huge key size of 10 KB or more, mounting *Rainbow* on constrained

embedded devices is challenging. Furthermore, *Rainbow*'s security is being reconsidered in light of the intersection and rectangular MinRank attacks recently proposed in [10].

Falcon with NTRU lattice is compact and has an efficient operation. Compared to other digital-signature algorithms proposed in PQC competition, it has the shortest key length and the highest verification speed [11]. However, *Falcon* requires a high key generation cost because it has to solve the NTRU equation. Also, since the floating-point operation was introduced, embedded devices that do not support floating-point operation performed poorly.

Crystals-Dilithium is a lattice-based algorithm that employs the hardness of the Learning With Error (LWE) problem [12]. In addition, compared to other digital-signature algorithms, its key generation performance, signature generation, and signature verification are uniformly distributed. Specifically, in the 2nd Round of the PQC competition, a method for implementing the algorithm more efficiently was proposed, where the security analysis for QROM is well applied to the *Crystals-Dilithium*, making it the most promising candidate among the final ones [13]. However, it is necessary to consider the cost of the application layer/program for the application of PQC-DSA on secure protocols and block-chain systems. Therefore, optimization of the NTT-based polynomial multiplication algorithm in *Crystals-Dilithium* is essential.

Optimization studies for PQC in various environments have been conducted. Efforts have been made to mount KEM and digital signatures from Advanced RISC Machine (ARM)-based MCUs to CPU and GPU environments. In general, because the PQC algorithm has a longer key and signature length than ECDSA, research into mounting the PQC algorithm in constrained devices is an important issue in terms of future applicability. ARM cores, which are the most widely used in the embedded environment, are used in a variety of boards, depending on their performance. Since the ARM-Cortex-M4 using ARMv7 was chosen by NIST as one of the performance evaluation equipment of the PQC competition, various optimization studies, including the PQM4 project, have been conducted on ARMv7-based equipment [14–16]. Similarly, research on PQC implementation in the ARMv8 environment is ongoing. In ARMv8-based devices, optimization studies on algorithms such as *Newhope*, a two-round KEM algorithm, *SABER*, and *Kyber* were carried out [17–19].

ARMv8 is a key device in the Internet of Things (IoT) society, serving as a core MCU for high-end computers in addition to MCUs for mobile, tablet, and desktop computers. As a result, it is expected that the use of ARMv8-based boards will increase in the future. The Jetson Xavier with an ARMv8.2 core is our study's target device. Jetson Xavier is currently in use for a variety of IoT/Cloud platforms, including autonomous driving environments that require digital signatures. As a result, in this study, we present an optimized Crystals-Dilithium implementation in the ARMv8 environment. We propose the parallel logic of the NTT-based polynomial multiplication algorithm, which is the core operation of *Crystals-Dilithium*, by fully utilizing the ARM processor and NEON engine.

1.1. Contribution

1.1.1. First Work of Crystals-Dilithium on ARMv8. Until now, an official *Crystals-Dilithium* study has only been conducted in ARM-Cortex-M3 and ARM-Cortex-M4 [15], but optimization studies on more diverse platforms are needed before it can be used in real-world applications. The ARMv8-A series, in particular, is being developed not only as a core MCU for mobiles and tablets, but also as an MCU for autonomous driving and high-end computers. Since the ARMv8-A series is becoming more popular as a core MCU in the embedded industry, optimization studies of PQC-based digital signatures in the ARMv8-A series should be considered. For the first time, we discuss in-depth optimization of NTT multiplication, the core operation of *Crystals-Dilithium* in the ARMv8-A series. As a result of the proposed methods, our *Crystals-Dilithium* software improved its performance by 43.83% in KeyGen, 113.25% in Sign, and 41.92% in Verify when compared to previous research [13] based on *Crystals-Dilithium* level 3.

1.1.2. Proposing Memory Optimization Techniques. Memory access not only has a high-performance overhead in an embedded environment, but it is also an expensive instruction. As a result, our goal is to reduce the number of memory accesses. We present an optimal path for NTT to minimize memory access in it. Memory access was minimized from Depth 0 to 2 in the NTT using the merging method, which was able to reduce memory access instructions (LD1 and ST1) by about 32 when compared to the standard implementation. Furthermore, for Depths 2–7 in the NTT, all coefficients required for conversion are stored in NEON vector registers and held until the NTT is completed. Because all coefficients are handled by holding into vector registers, the register hold technique has the advantage of avoiding memory access. We reduce the number of memory accesses that can occur in the NTT by using the proposed memory optimization techniques.

1.1.3. Optimizing NTTs for ARMv8. We present a method for designing the NTT multiplication of *Crystals-Dilithium* considering the resources of the ARMv8-A series. Our target device has ARM processor modules and a NEON engine, and we optimize NTT multiplication leveraging these features. We present NEON-based and ARM-based butterfly methods for NTT and inverse NTTs, respectively. The NEON-based butterfly method uses an Advanced Single Instruction Multiple Data (ASIMD) instruction and a vector register to efficiently perform four coefficients in parallel. The ARM-based butterfly method employs a barrel shifter to process two coefficients. ARM processors are not as powerful as the NEON engine, but they are adequate for small tasks. Finally, we combine all of the butterfly method implementations. This software converts the latencies of ARM operations into NEON overheads, improving performance even further. The same optimization technique as described above is used in point-by-point multiplication. We achieved performance improvements of 251% in NTT, 20%

in point-wise multiplication, and 304% in inverse NTT using the proposed methods, and NTT multiplication overall achieved a 260% performance improvement compared to previous work [13].

1.2. The Necessity for PQC-DSA in the 5G Communication Network. As social networks based on the 5G industry develop, so does the importance of communication security and personal information protection. Social network websites and applications are actively used in the user's closest space through IoT devices. For the security of communications in SocialNet-oriented cyberspace, we consider three things in this article.

The first is to minimize the load on the cryptographic algorithm. Users want more rapid responses; however, the cost of using a cryptographic system is fixed. Therefore, it is important to optimize the key-exchange and digital-signature algorithms used for cryptographic protocols. The second consideration is the IoT devices used in SocialNet-oriented cyberspace. In mesh with the first consideration, we implement cryptographic algorithms to match the characteristics of IoT devices. Optimization methods that take into account the characteristics of the device further accelerate the cryptographic algorithm. Finally, we consider the security of future-oriented communications. Currently, the 5G industry is accelerating, and various countries have started to develop the 6G industry. In addition, the PQC algorithm must be mounted on the cryptographic protocol to address the threat of quantum computing.

Therefore, In this article, we propose an implementation of the PQC-DSA algorithm, *Crystals-Dilithium*, optimization on the ARMv8 platforms used in the most popular. Our research accelerates the speed of mobile Internet and social networks.

1.3. Organization. The rest of the study is summarized as follows. Section 2 introduces the *Crystals-Dilithium* and analysis profiling of the reference code, as well as a description of the ARMv8 platforms. Section 3 discusses and analyzes existing implementation research for PQC. Section 4 presents an optimized NTT implementation on ARMv8-A series. Section 5 evaluates our works. Finally, in Section 6, we conclude this study.

2. Preliminaries

2.1. Crystals-Dilithium. Crystals-Dilithium is one of the most promising digital-signature algorithm candidates for the NIST PQC conference's final algorithm. *Crystals-Dilithium* is based on the difficulty of the Module Learning with Error problem and shares basic characteristics and structure with *Crystals-Kyber* [12, 13]. *Crystals-Dilithium* employs Fiat–Shamir with an abort method and borrows Module-LWE; as a result, it provides a higher level of security than other ring-LWE-based ciphers. Furthermore, for all security levels, *Crystals-Dilithium* employs the same ring and modulus. This has an advantage in terms of implementation over other competitors.

The polynomial ring used by Dilithium is $Z_q[x]/<2^{256}+1>$, where q is $2^{23}-2^{13}+1$, and the parameters are maintained by simply changing the dimension of the public matrix $\mathbf{A}$ according to the security level. Therefore, the core process of *Crystals-Dilithium* is the operation to generate the open matrix $\mathbf{A}$ and polynomial multiplication to generate the LWE-based problem. Similar to the general digital-signature algorithm, the structure of *Crystals-Dilithium* consists of KeyGen, Sign, and Verify processes.

The KeyGen process of *Crystals-Dilithium* is depicted in Algorithm 1. Using random seeds ρ and K, this process generates the public matrix A as well as the secret information $\mathbf{s}_1$ and $\mathbf{s}_2$. Through SHAKE-based rejection sampling, the ExpandA operation extracts a very small range of numbers. In all algorithms, the SHAKE algorithm serves as the collision-resistant hash (CRH).

Algorithm 2 depicts the *Crystals-Dilithium*'s Sign Process. Because the size of the public matrix in *Crystals-Dilithium* is greater than 1 KB, it is more efficient to regenerate the public matrix via ρ during the Sign process. A masking vector for polynomial $\mathbf{y}$ is generated during the signing process, and $\mathbf{Ay}$ is calculated. In this case, a challenge is generated by hashing the message with $\mathbf{w}_1$, which is the $\mathbf{Ay}$'s high-order bit. In the Verify process, MakeHint_q and UseHint_q are in charge of reconstructing the bits. The public key can be reduced by about 2.5 times using this method, at the cost of a slight increase in signature size.

Algorithm 3 depicts the *Crystals-Dilithium*'s Verify process. The signature verifier determines whether $\mathbf{w}_1'$ is accepted and whether the signature $\mathbf{z}$ is within the acceptable range.

2.2. Core Operation

2.2.1. NTT-Based Multiplication. The number theoretic transform (NTT) is a variant of the fast Fourier transform and an algorithm used by many lattice-based cryptographic algorithms that have advanced to the PQC contest 3 Round [20]. NTT's main feature is that it reduces the complexity of polynomial multiplication from $O(n^2)$ to $O(n \cdot \text{logn})$. NTT divides polynomials to the smallest unit through n-th of unity and performs point-wise multiplication in $O(n)$ complexity using the divide and conquer algorithm. Finally, it entails transforming the result into an $O(n \cdot \text{logn})$ complexity coefficient representation. The condition that NTT can be used in a general polynomial ring using $Z_q[x]/<2^n+1>$ is that n must be a power of 2, and q must be congruent to 1 modulo 2 n. Based on the NTT, the formula for polynomial multiplication is

$f(x)\times g(x)=\text{NTT}^{-1}(\text{NTT}(f)\odot\text{NTT}(g))$, where $\odot$ is the point-wise multiplication of the coefficients.

The depth of logn of NTT varies depending on the polynomial ring used. Due to the size of q, first-order multiplication must be performed during point-wise multiplication in the case of *Crystals-Kyber*, which is the same family of *Crystals* algorithms. However, in the case of *Crystals-Dilithium*, q is $2^{23}-2^{13}+1$ and n is 256; hence,

(1) $\rho, K \leftarrow \{0,1\}^{256}$
(2) $(\mathbf{s}_1, s_2) \in S_\eta^\ell \times S_\eta^k :=$ ExpandA (K)
(3) $\mathbf{A} \in R_q^{k\times\ell} :=$ ExpandA (ρ)
(4) $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + s_2$
(5) $(t_1, t_0) := \texttt{Power2Round}(\mathbf{t}, d)$
(6) $tr \in \{0,1\}^{384} := \texttt{CRH}(\rho \parallel \mathbf{t}_1)$
(7) **return** $pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

Algorithm 1: Key generation keyGen ().

(1) $\mathbf{A} \in R_q^{k\times\ell} :=$ ExpandA (ρ)
(2) $\mu \in \{0,1\}^{384} :=$ CRH $(tr \parallel M)$
(3) $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
(4) **while** $(\mathbf{z}, \mathbf{h}) := \perp$ do
(5) $y \in S_{\gamma_1 - 1}^\ell :=$ ExpandMask $(K \parallel \mu \parallel \kappa)$
(6) $\mathbf{w} := \mathbf{A}\mathbf{y}$
(7) $\mathbf{w}_1 :=$ HighBits$_q(\mathbf{w}, 2\gamma_2)$
(8) $c \in B_{60} :=$ H$(\mu \parallel \mathbf{w}_1)$
(9) $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
(10) $(r_0, r_1) :=$ Decompose$_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$
(11) **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ **then** $(\mathbf{z}, \mathbf{h}) := \perp$
(12) else
(13) $\mathbf{h} :=$ MakeHint$_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 - c\mathbf{t}_0, 2\gamma_2)$
(14) **if** $\|c\mathbf{t}_0\| \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is greater than w **then** $(\mathbf{z}, \mathbf{h}) := \perp$
(15) $\kappa = \kappa + 1$
(16) return $\sigma = (\mathbf{z}, \mathbf{h}, c)$

Algorithm 2: Signing sign (sk, M).

(1) $\mathbf{A} \in R_q^{k\times\ell} :=$ ExpandA (ρ)
(2) $\mu \in \{0,1\}^{384} :=$ CRH $($CRH $(\rho \parallel \mathbf{t}_1) \parallel M)$
(3) $\mathbf{w}_1' :=$ UseHint$_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
(4) **return** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c :=$ H$(\mu \| \mathbf{w}_1)$ and # of 1's in $\mathbf{h}$ is $\leq w$

Algorithm 3: Verification verify $(pk, M, \sigma = (\mathbf{z}, \mathbf{h}, c))$.

NTT can be performed up to 32-bit coefficients. Finally, because *Crystals-Dilithium* can transform up to the 8-th square root, 32-bit multiplication occurs 256 times for point-wise multiplication.

There are numerous methods for computing the NTT. *Crystals-Dilithium* employs the bit-reverse-based algorithms Cooley–Tukey [21] and Gentlemen–Sande [22]. The butterfly's method is used to match the two transform methods, and because both algorithms are bit-reverse based, there is no need to invert additional bits.

2.2.2. Profiling of Crystals-Dilithium Reference Codes. In this section, we profile the final submission *Crystals-Dilithium*'s code and discuss the optimization strategy of this study. Reference code is compiled in the Jetson Xavier with ARMv8.2, our target platform. Although reference code has reference and optimization implementations, AVX2-based source code cannot be built in the ARMv8 environment; additionally, as far as we know, the *Crystals-Dilithium* development team has not implemented code in the ARMv8-A series. As a result, the code submitted to the finals is the best option.

Table 1 shows a profiling performance of *Crystals-Dilithium* reference code on Jetson Xavier. The *Crystals-Dilithium* algorithm is made up of three parts: KeyGen, Sign, and Verify. The ExpandA operation performs rejection sampling based on SHA-3, and it is a common operation in each Dilithium component. Because the Keccak algorithm is used repeatedly in rejection sampling, it requires a lot of computation resources. According to our findings, the ExpandA operation took 46.4%, 29.8%, and 45.4% of the computational load in the KeyGen, Sign, and Verify processes, respectively. As a result, an optimization method capable of efficiently parallelizing rejection sampling is required. Optimization studies for the Keccak algorithm in embedded devices exist [25], but studies on optimal implementations in ARMv8 environments do not exist, to

the best of our knowledge. In order to reduce the performance load of ExpandA in the ARMv8 environment, it is recommended to use the fully assembled XKCP library made by the Keccak development team. This study does not take this into account.

Aside from the ExpandA operation, the NTT and the point-wise multiplication process consume the most computational resources in *Crystals-Dilithium*. The NTT and point-by-point multiplication processes are responsible for 23.5%, 65.7%, and 50.4% of the KeyGen, Sign, and Verify processes, respectively. Despite the fact that NTT -based multiplication is the fastest polynomial multiplication method, the public matrix has a maximum size of (8,7) and thus incurs a performance load.

Accordingly, in Jetson Xavier, the *Crystals-Dilithium* implementation logic must be redesigned by considering registers and instruction sets. As much information as possible should be kept in a small number of registers, operations should be performed, and the memory access cycle should be as short as possible. Furthermore, because some embedded processors support parallel instruction sets, this must be considered when determining the optimal load. In Section 4, we propose optimization methods for *Crystals-Dilithium* in the ARMv8 environment.

2.3. Target Devices: ARMv8-A Processor. ARM is widely used in the embedded industry due to its low power consumption and high performance when compared to previous low-end processors, AVR and MSP. According to their performance, ARM processors are classified into M-series, R-series, and A series levels. Among them, the ARM-A series provides the best performance. Furthermore, the most recent version of an ARM processor is the ARMv8 architecture.

The ARMv8-A series includes an ARM processor as well as a NEON engine. Unlike the NEON engine, the ARM processor does not support parallel processing, but it is adequate for small tasks. Furthermore, the ARM processor includes a barrel shifter, which can hide clock cycles for shift operations in the operand, making it a very powerful technology. The ARM processor's register structure is made up of 64-bit general-purpose registers x0-x30, and an A64 instruction set architecture [23] is provided. The NEON engine is a powerful parallel processing engine that supports 128bit vector register v0-x31 and ASIMD instructions set architecture [24]. Within a 128-bit vector register, this parallel processing can be done in 64-bit, 32-bit, 16-bit, and 8-bit units.

Furthermore, the ARM processor and NEON engine are separate modules that operate independently of one another. In other words, for the sequential instruction order of an ARM/NEON processor, it is the sum of the execution times of the ARM/NEON processor, but for the interleaving approach, the pipeline stall of each instruction can be hidden and performance-optimized efficiently [26]. Table 2 describes the ARM/NEON instructions and clock cycles used in this study to optimize NTT multiplication.

3. Related Works

Since the proposal of *Crystals-Dilithium*, implementation studies on *Crystals-Dilithium* in various embedded environments have been conducted. Submissions of current quantum-resistant cryptography implementations are mostly done on a CPU. The implementations used Intel instructions or the AVX2 parallel processing instruction. In the CPU environment, quantum-resistant cryptography implementations show no significant difference or slower performance than the elliptic curve-cryptographic system.

The environment in which the actual encryption equipment is used, on the other hand, is primarily comprised of low-spec embedded equipment. Because these devices have limited flash memory, RAM, and operation speed, it is critical to investigate the optimization of quantum-resistant cryptography's core operations. There are implementation results for the ARM-Cortex-M4 targeted by the NIST software performance evaluation model. An optimization study for *Crystals-Dilithium* in ARM-Cortex M3 and M4 environments was proposed at CHES'2021 [15]. By converting the unsigned expression to the signed expression in the M4 environment, the additional operation that prevents negative representation from appearing in the positive representation is omitted. It was also implemented by combining two NTT layers and maximizing SIMD instructions to fit the M3/M4 environment. The NTT process improved performance by reducing data access to a bare minimum through the integration of two layers. Finally, [15] presented three implementation strategies based on public and secret information storage space.

Except for *Crystals-Dilithium* in the ARMv8 environment, other PQC optimization studies have been conducted on *Newhope*, *Crystals-Kyber*, and *SABER*. An optimization implementation study for *Newhope* in the ARMv8 environment was carried out in 2017 [9]. Currently, *Newhope* is a PQC alternative candidate, and an ARMv8-based parallel-based NTT multiplication implementation is relevant to our research. To reduce NEON instruction and computational division, an unsigned 16-bit integer representation is used. Furthermore, the parallel logic is newly designed so that no conversion to the Montgomery domain is required, and the existing load of Barrett-reduction was removed by suggesting a method to perform subtraction in the multiplication process by point. Through this, [17] achieved an 8.3 times performance improvement over the existing C-based reference implementation in the ARM-Cortex-A53 core.

Crystals-Kyber implementation in the ARMv8 environment has recently been proposed [18]. The NTT operation for 16-bit coefficients is optimized in the same way that *Newhope* is. Because the modulus q is different, the techniques used in *Newhope* cannot be used. Vectorization is used to optimize almost all core operations, including sampling and reduction operations. It also accelerates *Crystals-Kyber*'s core work of symmetric functions via ARMv8 cryptographic extensions. [18] achieved a performance improvement of up to 8.6 times over the reference code through this optimization study.

Table 1: Profiling performance of *Crystals-Dilithium* reference code on Jetson Xavier (ARMv8.2)·(ExpandA).

ALG	All	$M \cdot E$	NTT	Others
KeyGen	34,087 (100%)	15,816 (46.4%)	8,010 **(23.5%)**	10,260 (30.1%)
Sign	106,331 (100%)	31,687 (29.8%)	69,859 **(65.7%)**	4,785 (4.5%)
Verify	33,761 (100%)	15,327 (45.4%)	17,016 **(50.4%)**	1,418 (4.2%)

$M \cdot E$ means matrix expand (ExpandA), and **NTT** means NTT-based multiplication.

Table 2: Summary of A64 and ASIMD instruction set in the ARMv8 platform [23, 24].

Instructions	Operand	Description	Cycles
		ARM A64 instructitons	
SMULL	X_n, W_m, W_d	Signed multiplying two 32-bit registers, and writing the 64-bit destination register $X_n = W_m \times W_d$	3
SMSUBL	X_n, W_m, W_d	Signed multiplying two 32-bit registers, subtracts the product from a 64-bit register value $X_n = X_n\text{-}(W_m \times W_d)$	3
ADD, SUB	X_n, X_m, X_d	Arithmetic operations (addition, subtraction) $X_n = X_m + X_d$, $X_n = X_m\text{-}X_d$	1
LSR	X_n, #n	Logical shift right (X_n, #n) X_n	1
LDP, STP	X_n, $[X_d]$	Loading and storing the data from memory to the pair of general-purpose registers and the pair of general-purpose registers to memory, $[X_d]$ X_n	2
		NEON ASIMD instructions	
MULL	V_n, V_m, V_d	Multiplying two vector register, $V_n = V_m \times V_d$	2
SMULL	V_n, V_{mH}, V_{dH}	Signed multiplying two vector registers (upper half), $V_n = V_{mH} \times V_{dH}$	3
SMULL2	V_n, V_{mL}, V_{dL}	Signed multiplying two vector registers (lower half), $V_n = V_{mL} \times V_{dL}$	3
SMLSL	X_n, W_{mH}, W_{dH}	Signed multiplying two vector registers (upper half), subtracts the product from a 64-bit register value $X_n = X_n\text{-}(W_{mH} \times W_{dH})$	3
SMLSL2	X_n, W_{mL}, W_{dL}	Signed multiplying two vector registers (lower half), subtracts the product from a 64-bit register value $X_n = X_n\text{-}(W_{mL} \times W_{dL})$	3
XTN, XTN2	V_{nH}, V_{mH}, V_{dH}, V_{nL}, V_{mL}, V_{dL}	Narrowing half in each vector register, and writes the vector to the lower or upper half, $[V_{mH}, V_{dH}] \xrightarrow{\text{Narrow}} V_{nH}$	2
ADD, SUB	V_n, V_m, V_d	Arithmetic operations (addition, subtraction) $V_n = V_m + V_d$, $V_n = V_m\text{-}V_d$	1
LD1, ST1	$\{V_m\text{-}V_n\}$, $[X_d]$	Loading and storing the data from memory to vector registers and vector registers to memory, $[X_d]V_m\text{-}V_n$	2

Chung et al. [19] proposed a method for applying the NTT to the SABER polynomial ring of power of 2. This resulted in a performance improvement of about 60% in SABER when compared to Toom–Cook-based multiplication. Benchmarking and research of SABER using NEON in the ARM-Cortex A series was carried out in [27]. The NTT technique and the NEON instruction proposed in [19] were used, and benchmarking was performed on the Apple M1 core and the ARM-Cortex A72.

4. Optimization Strategies of NTT

In this section, we present an optimized method of NTT multiplication, the core operation of *Crystals-Dilithium*, to accelerate signature processing in the ARMv8-A series. Because NTT multiplication is divided into NTT, inverse NTT, and point-wise multiplication, we introduce detailed optimization strategies by categorizing it as NTT/InvNTT and point-wise multiplication. We present a memory optimization technique and a parallel optimization method in NTT and inverse NTT. Furthermore, we present the interleaving concept of the butterfly method, which was codesigned with the ARM/NEON processor. In the point-wise multiplication, the same optimized methods used in NTT and inverse NTTs are used except for the memory optimization.

4.1. NTT and NTT^{-1}.

The most computationally expensive parts of NTT multiplication are the NTT and inverse NTTs. Due to the limited resources of the embedded environment, the amount of memory access varies depending on how it is implemented, and memory access is an expensive instruction in the embedded environment. As a result, we present memory optimization techniques, such as merging and register-holding, to reduce these memory accesses. In addition, we present an efficient parallel implementation using the NEON engine. Finally, by utilizing the target devices' independent cores, we further optimize our parallel implementation using the butterfly method, which was codesigned with the ARM/NEON processor. Figure 1 depicts the overall structure of the proposed optimization techniques for the NTT. As a result, not only the memory access was minimized through the merging and register-holding method, but also the performance was further enhanced by processing the NTT of some coefficients by the ARM processor, concealing the latencies of some ARM operations with NEON overheads.

4.1.1. Memory Optimization: Merging and Holding.

Figure 2 shows a comparison of standard and merging implementations. At each depth, the standard implementation performs the butterfly method sequentially.

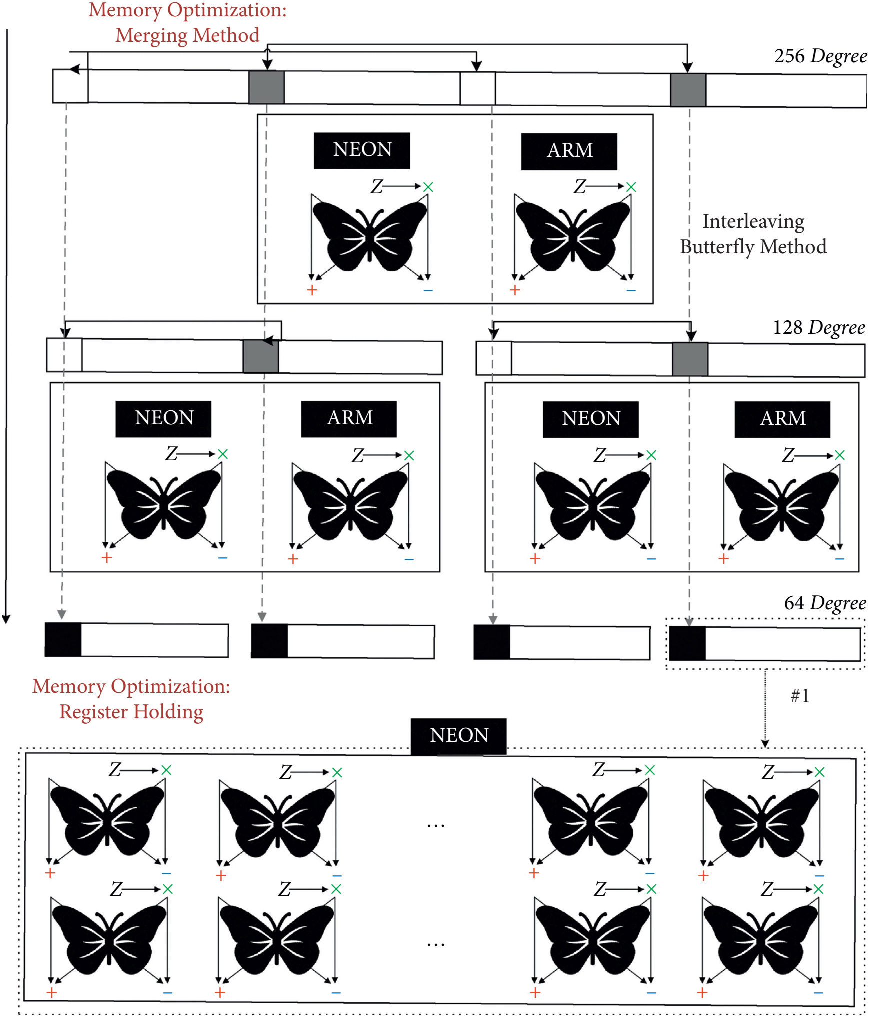

FIGURE 1: Overall structure of the proposed optimized NTT.

Given the target device's resources, this implementation necessitates multiple memory accesses. The standard implementation is distinguished by the fact that it necessitates multiple load and stores instructions for each depth. Specifically, 48 LD1 and ST1 instructions are required to convert the 256-degree polynomial to the 64-degree polynomial. Because these memory access instructions are very expensive in the embedded environment, memory accesses must be reduced to minimize costs. The primary goal of merging implementation is to reduce memory accesses. The merging implementation, on the other hand, performs NTT of 1 depth without memory access by concurrently processing coefficients of a specific order required for the butterfly method of the next depth. Many LD1 and ST1 instructions are saved as a result of this. Since the 256-degree polynomial is transformed without memory access up until the 64-degree polynomial, 16 LD1 and ST1 instructions are saved using the merging method. As a result, in an embedded environment, this merging method is an efficient way to reduce memory access overheads. Following that, the register-holding method is used to minimize memory accesses until the NTT is completed. In other words, all coefficients of the 64-degree polynomial are stored in vector registers v0-v31, and operations are performed by holding them in the register without accessing memory until the NTT is completed.

4.1.2. Butterfly Method on the ARM Processor. Barrel shifter and 64-bit general-purpose registers are supported by ARM processors. Furthermore, due to backward compatibility with the previous version, the lower part of the 64-bit can be used as a 32-bit general-purpose register. Using these features, we describe the ARM-based butterfly method for processing two coefficients. Algorithm 4 shows how the proposed ARM-based butterfly method works with two coefficients. Step 3 involves performing signed multiplication on the input and Zetas, where Zetas is the twiddle factor of NTT.

Step 4 is a signed multiplication of Zetas, the first operation of the Butterfly operation, and one input. Because a signed multiplication was performed, a Montgomery reduction is required to return it to the ring's elements. Steps 5–9 are a proposed Montgomery reduction based on an ARM processor, and we process the multiplication and

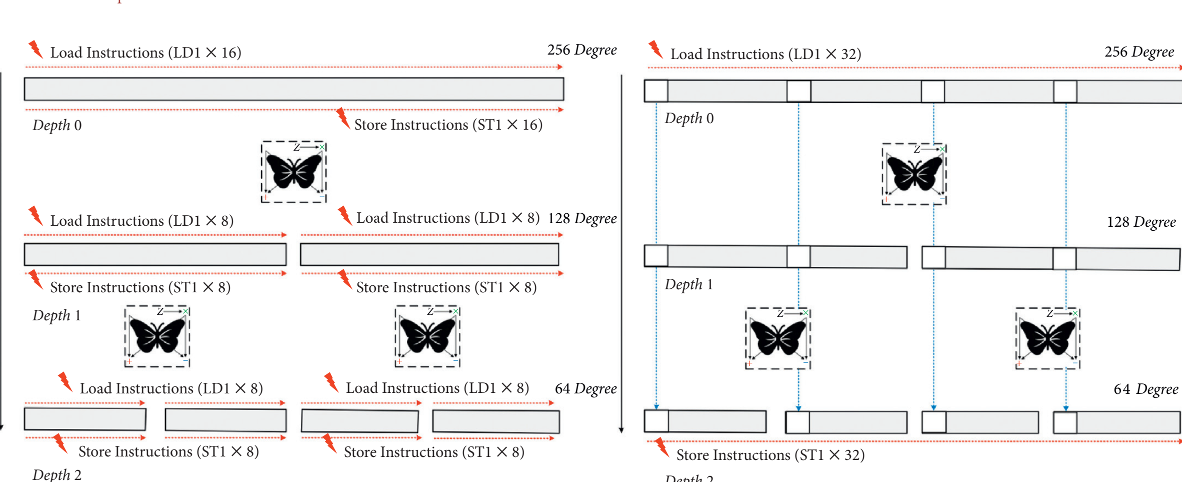

Figure 2: Comparison between standard implementation and the proposed merging method of NTT.

(1) **Input:** in1 (x8), in2 (x9) ▷ 4 (in1:2, int2:2) coefficient on ARM
(2) Output: out1, out2 ▷ 4 (out1:2, out2:2) coefficient on ARM
(3) // Butterfly 1
(4) SMULL x7, w9, w11 ▷ a_1 = in2 (w9 : 1 coefficient ▽) × zetas (x11)
(5) SMULL x12, w20, w7 ▷ $t_1 = Q^{-1}(\text{x20}) \times a_1$
(6) SMSUBL x7, w12, w21, x7 ▷ $t_1 = a_1 - t_1 \times Q(\text{x21})$
(7) LSR x7, x7, #32 ▷ $t_1 = t_1 \gg 32$
(8) ADD w10, w8, w7 ▷ Addition: w10 = in1 ▽ + t_1
(9) SUB w13, w8, w7 ▷ Subtraction: w13 = in1 ▽ − t_1
(10) // Masking
(11) AND x8, x6, x8, LSR#32 ▷ w8 (in1: 1 coefficient △) = (x8&0xFFFFFFFF) ≫ 32
(12) AND x9, x6, x9, LSR#32 ▷ w9 (in2: 1 coefficient △) = (x9 & 0xFFFFFFFF) ≫ 32
(13) // Butterfly 2
(14) SMULL x7, w9, w11 ▷ a_2 = in2 (w9 : 1 coefficient △) × zetas (x11)
(15) SMULL x12, w20, w7 ▷ $t_2 = Q^{-1}(\text{x20}) \times a_2$
(16) SMSUBL x7, w12, w21, x7 ▷ $t_2 = a_2 - t_2 \times Q(\text{x21})$
(17) LSR x7, x7, #32 ▷ $t_2 = t_2 \gg 32$
(18) ADD w12, w8, w7 ▷ Addition: w12 = in1 △ + t_2
(19) SUB w7, w8, w7 ▷ Subtraction: w7 = in1 △ − t_2
(20) // Concatenation
(21) EOR out1, x10, x12, LSL#32 ▷ out1 = w10⊕w12 (x12 ≪ 32)
(22) EOR out2, x13, x7, LSL#32 ▷ out2 = w13⊕w7 (x7 ≪ 32)

Algorithm 4: Butterfly method on ARM processor.

subtraction operations required for Montgomery reduction at the same time using the SMSUBL instruction. Steps 8–9 perform the remaining addition and subtraction operations of the butterfly operation, and the butterfly operation for each coefficient of the inputs is completed.

To prepare for the next butterfly method, the upper 32-bit is shifted to the lower 32-bit using the barrel shifter in steps 11–12. The ARM-based butterfly method described above does the same thing in the following step to process one coefficient from each remaining input. Finally, to reduce memory access, we concatenate two coefficients on which the butterfly method is performed into a 64-bit general-purpose register, and the concatenated process can be simply performed with an ARM processor's barrel shifter.

4.1.3. Butterfly Method on the NEON Engine. ARMv8-A series supports a powerful NEON engine, which is a SIMD instruction architecture. The NEON engine provides vectorization of 16 8-bit, 8 16-bit, 4 32-bit, and 2 64-bit within a 128-bit vector register. Since the modulus of *Crystals-Dilithium* is 8380417, each coefficient of the polynomial should be an element within the modulus 8380417. Considering the modulus of *Crystals-Dilithium* and the parallel unit of NEON engine, we present task parallelism for the NTT multiplication of *Crystals-Dilithium* leveraging the NEON engine. Within a 128-bit vector register, it processes four coefficients at the same time. The task parallelism for the butterfly method, which is the basic operation of NTT multiplication, is demonstrated in Algorithm 5. In steps 4-5,

```
(1) Input:in1, in2                                   ▷ 8 (in1:4, int2:4) coefficient on NEON
(2) Output: out1, out2                               ▷ 8 (out1:4, out2:4) coefficient on NEON
(3) // Zetas(twiddle factor) multiplication
(4) SMULLv26.2d, \in2\(▷).2s, v27.2s                 ▷ in2 (2 coefficient △) × zetas (v27)
(5) SMULL2v24.2d, \in2\(▷).4s, v27.4s                ▷ in2 (2 coefficient ▽) × zetas (v27)
(6) // Masking
(7) XTNv28.2s, v26.2d                                ▷ Narrow Extract(Lower)
(8) XTN2v28.4s, v24.2d                               ▷ Narrow Extract(Upper)
(9) // Montgomery reduction
(10) MULv28.4s, v28.4s, v30.4s                       ▷ t1 = Q^-1 (v30)× in2 (4 coefficient)
(11) SMLSLv26.2d, v28.2s, v29.2s                     ▷ t1 = in2 − t1 (2 coefficient △) ×Q
(12) SSHRv26.2d, v26.2d, #32                         ▷ t1 = t1 ≫ 32
(13) SMLSL2v24.2d, v28.4s, v29.4s                    ▷ t2 = in2 − t1 (2 coefficient ▽) ×Q
(14) SSHRv24.2d, v24.2d, #32                         ▷ t2 = t2 ≫ 32
(15) // Masking, Addition, and Subtraction
(16) XTNv25.2s, v26.2d                               ▷ Narrow Extract (t1 : Lower)
(17) XTN2v25.4s, v24.2d                              ▷ Narrow Extract (t2 :Upper)
(18) ADDout1\.4s, v25.4s, \in1\.4s                   ▷ Addition of Butterfly
(19) SUBout2\.4s, \in1\.4s, v25.4s                   ▷ subtraction of Butterfly
```

Algorithm 5: Butterfly method on the NEON engine.

a signed multiplication is performed between one input and zetas. The SMULL (or SMULL2) instruction performs signed multiplication between the upper (or lower) two 32-bits of two 128-bit vector registers and stores the result in two 64-bit vector registers. Following that, Montgomery reduction is used to make it a part of the ring. Because Montgomery reduction only requires the lower 32-bits of the multiplication result, steps 7–8 collect the 32-bits of the multiplication result into one vector register using XTN and XTN2 instructions. `XTN` instructions extract narrow within the vector register and are divided into XTN and XTN2 instructions based on upper and lower.

In steps 10–14, Montgomery reduction is performed in parallel for four coefficients. Step 10 is a step to perform multiplication with QINV, which is one of the Montgomery reduction steps. Through the previous masking process, we optimize it to process four multiplications at the same time. Furthermore, SMLSL and SMLSL2 instructions are similar to SMLSL and SMLSL2 instructions in that multiplication and subtraction can be performed in the same clock cycle. This allows us to improve the performance of NEON-based Montgomery reduction. The result of the Montgomery reduction is collected into a single 128-bit vector register in steps 16–17 using the proposed masking process. Finally, the NEON-based butterfly method with task parallelism is completed by performing the remaining butterfly method operations of addition and subtraction.

4.1.4. Interleaving Butterfly Method Utilizing ARM/NEON. The ARMv8-A series has two cores: an ARM processor and a NEON engine. The two cores are independent modules that compute independently of one another. The ARM processor is not as powerful as the NEON engine, but it is adequate for some minor tasks. As a result, we present the butterfly method, which was developed in collaboration with the ARM/NEON processor. This codesign aims at interleaving rather than serializing each implementation of our butterfly method of the ARM processor and NEON engine. Figure 3 depicts the processing of the butterfly method, which was codesigned with an ARM/NEON processor. By utilizing both ARM/NEON processors concurrently, the latencies of some coefficient operations in the ARM processor are effectively hidden by NEON overheads, allowing performance to be further maximized than utilizing a simple single core.

4.2. Point-Wise Multiplication on ARMv8. Point-wise multiplication is a modular multiplication process that consists of simple multiplication followed by Montgomery reduction, similar to zetas multiplication followed by Montgomery reduction in butterfly operations. Thus, modular multiplication can be implemented by performing multiplication with one coefficient instead of zetas multiplication and then performing Montgomery reduction. Except for the memory optimization in the butterfly operation, the optimization method in point-wise multiplication uses only the parallel and interleaving methods. Figure 4 depicts the optimization method proposed in the point-wise multiplication process. The point-wise multiplication process, like the interleaving butterfly method, employs both the ARM core and the NEON engine concurrently. The NEON engine processes four coefficients in parallel, whereas the ARM engine processes two coefficients and mixes them. The interleaving implementation improves performance by incorporating ARM computation latency into NEON overheads. Furthermore, we can minimize pipeline stalls in each implementation and use both cores, including parallel implementation and barrel shifter.

5. Evaluation

5.1. Jetson Xavier. The Jetson Xavier CPU has 8 ARMv8.2 cores, and the same out-of-order pipeline as ARMv8.

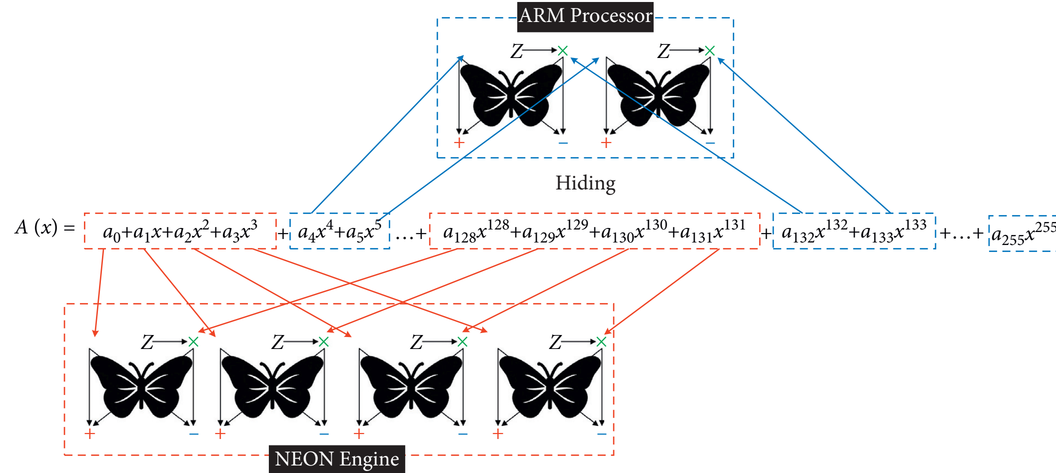

FIGURE 3: The proposed interleaving butterfly method utilizing both the ARM processor and NEON engine.

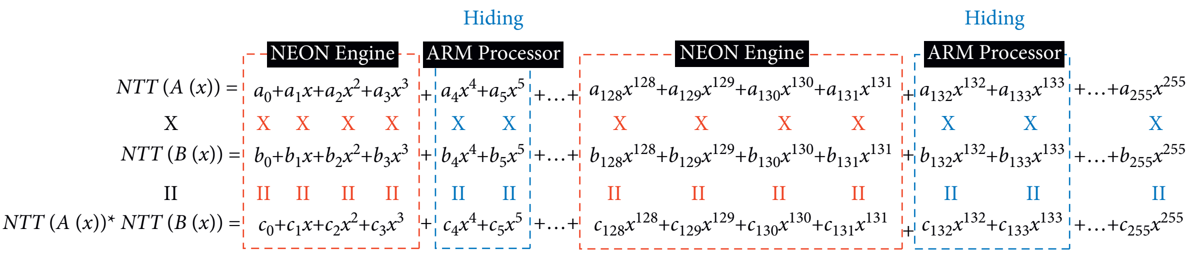

FIGURE 4: The proposed interleaving modular multiplication utilizing both the ARM processor and NEON engine.

TABLE 3: Cycle comparison of the NTT and *Crystals-Dilithium* on Jetson AGX Xavier.

Works	NTT	Point-wise multiplication	Inverse NTT	NTT-based multiplication
Reference code *Crystals-Dilithium*	3,966 (–)	264 (–)	5,677 (–)	9,907 (–)
Our work	1,128 (+**251%**)	219 (+**20%**)	1,403 (+**304%**)	2,750 (+**260%**)
Works	Security level	KeyGen	Sign	Verify
Reference code *Falcon*	Level-1	21,159,212	864,183	6,546
Reference code *Crystals-Dilithium*	Level-2	136,992 (–)	750,998 (–)	150,687 (–)
Our work		98,972 (**+38.41%**)	390,918 (**+92.11%**)	105,367 (**+42.99%**)
Reference code *Crystals-Dilithium*	Level-3	253,395 (–)	1,257,304 (–)	255,421 (–)
Our work		176,174 (**+43.83%**)	589,567 (**+113.25%**)	179,971 (**+41.92%**)
Reference code *Crystals-Dilithium*	Level-5	356,908 (–)	1,439,069 (–)	388,523 (–)
Reference code *Falcon*		64,084,086	1,724,079	13,265
Our work		286,684 (**+24.49%**)	711,549 (**+102.24%**)	307,115 (**+26.51%**)

ARMv8.2 supports half-precision floating-point processing, RAS, statistical profiling, and an improved memory model architecture when compared to ARMv8 [28]. It has a 64 KB L1 data cache, a 128 KB L1 instruction cache, and a 2 MB L2 cache, and it can run at up to 2.26 GHz. The software is compiled with GCC with the -O3 option, and as a result, the benchmarking reference code uses the NEON engine partially by the compiler for each function. For benchmarking, the reference code and our code are executed 10,000 times and the clock cycles on the registers in ARMv8.2 are measured. A Crystals-Dilithium submission serves as the reference implementation [13].

5.2. Results for NTT and NTT^{-1}. Table 3 compares the performance of NTT/InvNTT and point-wise multiplication in ARMv8.2-based Jetson Xavier between the reference implementation and the presented implementation. Except for the multiplication of the twiddle factor in the NTT/InvNTT conversion process, the reference implementation was partially executed in parallel through the NEON engine in addition and subtraction. In our work, we use the merging, register-holding, and interleaving methods to reduce memory access to the input-polynomial and compactly compare the execution cycle; as a result, we achieve performance improvements of about 251% and 304% in the

NTT/InvNTTs, respectively. All processes for the reference code of point-wise multiplication were carried out in full parallel via the NEON engine. We achieve a 20% performance improvement on point-wise multiplication by compactly using interleaving and vector registers of the NEON engine. Finally, we achieved a 260% percent performance improvement over the reference implementation in full NTT-based multiplication.

5.3. Results for Full Schemes. We achieved performance improvements of approximately 43.83%, 113.25%, and 41.92% in KegGen, Sign, and Verify based on *Crystals-Dilithium* security level 3, respectively, using our NTT-based multiplication optimization method. Furthermore, at all security levels, it outperforms the reference implementation. To the best of our knowledge, this is the first implementation of *Crystals-Dilithium* optimization in an ARMv8 environment. Additionally, we compare our results with another finalist algorithm, *Falcon*. In official reference implementations, *Crystals-Dilithium* always outperforms Falcon in the KegGen and Sign process. Our implementation further enhances the performance advantages of *Crystals-Dilithium* and minimizes the performance gap that occurred during the Verify process compared to Falcon.

6. Conclusion

We present three implementation strategies for high-speed NTT implementation in an ARMv8 environment, merging, register-holding, and interleaving, and demonstrate them in *Crystals-Dilithium*. We achieve extremely fast implementations in ARMv8 platforms as a result of this, making *Crystals-Dilithium* a very efficient candidate in the ARMv8 environment. The parallel load proposal, the use of barrel shifters, and the use of the interleaving technique, in particular, are very well-suited implementations for ARM-based platforms. We achieved 43.83%, 113.25%, and 41.92% in KegGen, Sign, and Verify, respectively, compared to the reference implementation of *Crystals-Dilithium* security level 3 in the ARMv8 environment.

More broadly, we believe that the approach of merging multiple NTT layers, register-holding for the remaining layers, and finally interleaving can be applied to the ring of PQC variables. It can be used in particular when other PQC algorithms that have selected NTT high-speed implementation, such as *Crystals-Kyber* and *Falcon*, are implemented in an ARMv8 environment by selecting a special ring. From the standpoint of implementation design, it is intriguing that the NEON engine and the ARM processor collaborate with each other via the new parallel load and reduce memory access because it can lower computation costs and define a compact routine that resynchronizes the algorithm-specific NTT layers.

Data Availability

The "source code data" and "optimization method data" used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work was partly supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea government (MSIT) (No. A2021-0270, 6G autonomous security internalization-based technology research to ensure security quality at all times, 100%) and Korea Evaluation Institute of Industrial Technology (KEIT) grant funded by the Korea government (MOTIE).

References

[1] S. Nowaczewski and W. Mazurczyk, "Securing future internet and 5g using customer edge switching using dnscrypt and dnssec," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 11, no. 3, pp. 87–106, 2020.

[2] P. Dutta, W. Susilo, D. Hoang Duong, J. Baek, and P. Sarathi Roy, "Identity-based unidirectional proxy re-encryption and re-signature in standard model: lattice-based constructions," *Journal of Internet Services and Information Security (JISIS)*, vol. 10, no. 4, pp. 1–22, 2020.

[3] B. Soelistijanto and G. Manoah, "Network size estimation in opportunistic mobile networks: the mark-recapture method," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 12, no. 3, pp. 29–46, 2021.

[4] P. Thorncharoensri, W. Susilo, and J. Baek, "Efficient controlled signature for a large network with multi security-level setting," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 10, no. 3, pp. 1–20, 2019.

[5] Q. Su, H. Rong, S. Duan, F. Kong, X. Liu, and J. Yu, "Secure computation outsourcing for inversion in finite field," *Journal of Internet Services and Information Security (JISIS)*, vol. 10, no. 2, pp. 35–48, 2020.

[6] M. Park, S. Kim, and J. Kim, "Research on note-taking apps with security features," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 11, no. 4, pp. 63–76, 2020.

[7] E. Bashier and T. Ben Jabeur, "An efficient secure image encryption algorithm based on total shuffling, integer chaotic maps and median filter," *Journal of Internet Services and Information Security (JISIS)*, vol. 11, no. 2, pp. 46–77, 2021.

[8] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, Washington, DC, USA, November 1994.

[9] J. Ding, M.-S. Chen, and P. Albrecht, "Rainbow specifications and supporting documentation," 2020, https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[10] B. Ward, "Improved cryptanalysis of uov and rainbow," Cryptology ePrint Archive, Report 2020/1343, 2020.

[11] T. Prest, P.-A. Fouque, J. Hoffstein et al., "Falcon specifications and supporting documentation," 2020, https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[12] L. Ducas, E. Kiltz, T. Lepoint et al., "Crystals-dilithium: a lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 238–268, 2018.

[13] B. Shi, D. Leo, E. Kiltz et al., "Crystals-dilithium algorithm specifications and supporting documentation," 2020, https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[14] M. J. Kannwischer, J. Rijneveld, S. Peter, and S. Ko, "pqm4: testing and benchmarking NIST PQC on ARM cortex-m4," IACR Cryptol. ePrint Arch, Report 2019/844, 2019.

[15] D. O. C. Greconici, M. J. Kannwischer, and D. Sprenkels, "Compact dilithium implementations on cortex-m3 and cortex-m4," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, pp. 1–24, 2020.

[16] A. Karmakar, J. M. Bermudo Mera, S. Sinha Roy, and I. Verbauwhede, "Saber on ARM," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 3, pp. 243–266, 2018.

[17] S. Streit and F. De Santis, "Post-quantum key exchange on armv8-a: a new hope for neon made simple," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1651–1662, 2018.

[18] P. Sanal, E. Karagoz, H. Seo, R. Azarderakhsh, and M. Mozaffari-Kermani, "Kyber on arm64: compact implementations of kyber on 64-bit arm cortex-a processors," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, Moscow, Russia, September 2021.

[19] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "NTT multiplication for NTT-unfriendly rings," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 2, pp. 159–188, 2021.

[20] N. Hamid, N. Dutt, S. Ray, and F. Regazzoni, "Post-quantum lattice-based cryptography implementations: a survey," *ACM Computing Surveys*, vol. 51, no. 6, 2019.

[21] C. James and J. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[22] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the 1966 Fall Joint Computer Conference AFIPS '66 (Fall)*, pp. 563–578, New York, NY, USA, November 1966.

[23] Arm, "A64 instructions," 2020, https://developer.arm.com/documentation/ddi0596/2020-12/Base-Instructions.

[24] Arm, "Asimd vector instructions," 2020, https://developer.arm.com/documentation/ddi0596/2020-12/SIMD-FP-Instructions.

[25] Y. B. Kim, T.-Y. Youn, and S. C. Seo, "Chaining optimization methodology: a new sha-3 implementation on low-end microcontrollers," *Sustainability*, vol. 13, no. 8, 2021.

[26] H. Seo, T. Park, S. Heo et al., "Parallel implementations of lea, revisited," *Information Security Applications*, Springer International Publishing, no. 3, , pp. 318–330, New York, NY, USA, 2017.

[27] D. Tri Nguyen and K. Gaj, "Optimized software implementations of crystals-kyber, ntru, and saber using neon-based special instructions of armv8," in *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*, New York, NY, USA, June 2021.

[28] NVIDIA Developer Support Site, "NVIDIA AGX xavier specification," 2021, https://developer.nvidia.com/embedded/jetson-modules.