Hindawi
Security and Communication Networks
Volume 2018, Article ID 7012056, 10 pages
https://doi.org/10.1155/2018/7012056

Research Article

Efficient Parallel Implementation of Matrix Multiplication for Lattice-Based Cryptography on Modern ARM Processor

Taehwan Park,[1] **Hwajeong Seo**,[2] **Junsub Kim**,[3] **Haeryong Park**,[3] **and Howon Kim**[1]

[1]*Pusan National University, School of Computer Science and Engineering, San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609-735, Republic of Korea*
[2]*Hansung University, IT Engineering, 116 Samseong-Yoro-16-Gil Seongbuk-gu Seoul 136-792, Republic of Korea*
[3]*Cryptographic Technical Team, Security Industry Division, Korea Internet Security Agency, 6F, 9, Jinheung-gil, Naju, Jeollanam-do, 58324, Republic of Korea*

Correspondence should be addressed to Howon Kim; howonkim@pusan.ac.kr

Received 6 April 2018; Accepted 5 September 2018; Published 24 September 2018

Guest Editor: Chong Hee Kim

Recently, various types of postquantum cryptography algorithms have been proposed for the National Institute of Standards and Technology's Postquantum Cryptography Standardization competition. Lattice-based cryptography, which is based on Learning with Errors, is based on matrix multiplication. A large-size matrix multiplication requires a long execution time for key generation, encryption, and decryption. In this paper, we propose an efficient parallel implementation of matrix multiplication and vector addition with matrix transpose using ARM NEON instructions on ARM Cortex-A platforms. The proposed method achieves performance enhancements of 36.93%, 6.95%, 32.92%, and 7.66%. The optimized method is applied to the Lizard. CCA key generation step enhances the performance by 7.04%, 3.66%, 7.57%, and 9.32% over previous state-of-the-art implementations.

1. Introduction

In these days, with the development of quantum computing technologies, there are security threats to the existing block cipher due to the Grover's algorithm [1] and public key cryptographic algorithms such as RSA, which is based on the integer factorization problem, the discrete logarithm problem, and ECC, which is based on elliptic curve discrete logarithm problem according to Shor's algorithm [2]. For this reason, many cryptographers are designing new cryptographic algorithms, such as lattice-based cryptography, multivariate-based cryptography, Hash-based cryptography, code-based cryptography, and supersingular elliptic curve isogeny-based cryptography, which are safe in a quantum computing environment. In PQCrypto 2016, the National Institute of Standards and Technology (NIST) announced the Postquantum Cryptography Standardization competition. The submission deadline was November 30, 2017 and the first standardization workshop date was 11 April 2018. Many Postquantum cryptographic algorithms have been proposed. Lattice-based cryptography, which is based on Learning with Errors (LWE) problems, used matrix multiplication and vector addition operations for key generation, encryption, and decryption. However, matrix multiplication and vector addition for a large matrix take much time for key generation, encryption, and decryption. For efficient implementation of lattice-based cryptography, speed optimized implementation on matrix multiplication and vector addition is needed. In this paper, we propose efficient parallel implementation of matrix multiplication and vector addition for lattice-based cryptography based LWE problems using ARM NEON SIMD intrinsic functions.

The remainder of this paper is organized as follows. Section 2 discusses the literature related to the LWE problems, NIST PQC Standardization, Lizard lattice-based cryptography, ARM NEON SIMD, and related studies on efficient implementation of lattice-based cryptography. We propose efficient ARM NEON optimized matrix multiplication and vector addition implementation methods in Section 3. Section 4 gives experimental and evaluation results on proposed ARM NEON optimized matrix multiplication and vector addition implementation and Lizard CCA key generation

with the proposed method. Section 5 provides some final conclusions.

2. Related Studies

In this section, we describe related studies on LWE problems and NIST PQC standardization.

2.1. Learning with Errors (LWE) Problems. Regev introduced the Learning with Errors (LWE) problem [4]. For example, for an n-dimensional vector $s \in \mathbb{Z}_q^n$ and an error distribution χ over $\mathbb{Z}$, the LWE distribution $A_{n,q,x}^{LWE}(s)$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is obtained by choosing a vector uniformly and randomly from $\mathbb{Z}_q^n$ and an error e from χ and using

$$(\boldsymbol{a}, b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q \quad (1)$$

The search LWE problem finds $s \in \mathbb{Z}_q^n$ for given arbitrarily many independent samples (a_i, b_i) from $A_{n,q,x}^{LWE}(s)$. The hardness of the decision LWE problem is guaranteed by the worst case hardness of the standard lattice problems, such as the decision version of the shortest vector problem (GapSVP) and the shortest independent vectors problem (SIVP). Peikert et al. [5, 6] improved the reduction of the classical version. Brakerski et al. [6] proved that the LWE problem with a binary secret is at least as hard as the original LWE problem, and Cheon et al. [7] proved the hardness of the LWE problem with a sparse secret. According to these research results, in these days, the LWE problem has been used as a hardness assumption for lattice-based postquantum cryptography. In lattice-based cryptography, errors (E) can be used during encryption and decryption procedures, and they are generated by random samplers, such as the Gaussian sampler. During encryption and decryption procedures, they used matrix multiplication between matrix A and secret matrix S and then vector addition with errors vector E. For example, Peikert [5] proposed a cryptosystem based on the LWE problem, which is secure against any chosen-ciphertext attack, and Lin et al. [8] proposed a key exchange scheme based on the LWE problem. Many lattice-based cryptography systems provide security in a quantum computing environment based on LWE problems.

2.2. NIST PQC Standardization. The United States National Institute of Standards and Technology (NIST) has initiated postquantum cryptography standardization since 2016. The submission deadline was November 30, 2017. A total of 69 postquantum cryptographic algorithms were submitted on NIST PQC standardization Round 1: 26 lattice-based cryptographic algorithms (5 signatures, 21 KEM (key encapsulation mechanism)/encryption), 19 code-based cryptographic algorithms (3 signatures, 16 KEM/encryption), 9 multivariate-based (7 signatures, 2 KEM/encryption), 3 hash-based signature schemes, and 8 others (2 signatures, 6 KEM/encryption) were submitted. Four algorithms have been withdrawn. The lattice-based cryptography is the most proposed type of postquantum cryptography for NIST PQC standardization according to NIST PQC standardization Round 1 submission.

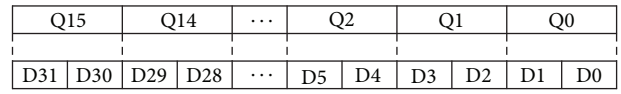

Figure 1: ARM NEON register bank.

Most lattice-based cryptographic algorithms are based on the LWE problem for providing security in a quantum computing environment and efficiency of implementation. The first NIST PQC standardization conference was scheduled to take place on April 11-13, 2018. After the first NIST PQC standardization, it will take about five to six years until the final decision for NIST PQC standardization is made. During PQC standardization, efficient implementation of submitted postquantum cryptographic algorithms is an important issue.

2.3. Lizard. Lizard [3] is a family of postquantum public key encryption (PKE) schemes and key encapsulation mechanisms (KEMs), which was submitted to NIST PQC standardization round 1. The security of Lizard is based on sparse, a small secret version of Learning with Errors (LWE), and learning with rounding (LWR). A sparse signed binary secret LWE problem is at least as hard as the original LWE problem. The public key for Lizard was chosen to be a set of LWE samples with signed binary secret information. Lizard supports IND-CPA PKE, IND-CCA2 KEM, and IND-CCA2 PKE, and there are two types of Lizard, namely, Lizard and Rlizard, which are based on Ring-LWE and Ring-LWR problems. In the key generation step of Lizard, it first samples a secret vector $s \in \{-1, 0, 1\}^n$, a random matrix $A \in \mathbb{Z}^{m \times n}$, and an error vector $e \longleftarrow DG_\sigma^m$ of which the components are expected to be small. The secret key is written as sk $\longleftarrow$ s, and the public key is written as pk $\longleftarrow$ (A,b),where $b = AS + e \in \mathbb{Z}_q^m$. Hence, the public key q is an instance of LWE with the secret vector s. There are five types of parameter sets of Lizard.CCA: CCA_CATEGORY1_N536, CCA_CATEGORY1_N663, CCA_CATEGORY3_N816, CCA_CATEGORY3_N952, CCA_CATEGORY5_N1088, and CCA_CATEGORY5_N1300. The parameter sets of Lizard.KEM are similar to the parameter sets of Lizard.CCA. However, RLizard.CCA and RLizard.KEM have four types of parameter sets: RING_CATEGORY1, RING_CATEGORY3_N1024, RING_CATEGORY3_N2048, and RING_CATEGORY5. In this study, we used the proposed method for efficient matrix multiplication and vector addition using ARM NEON SIMD on the Lizard.CCA key generation step and evaluated the performance of proposed method on the proposed methods application aspect.

2.4. ARM NEON. ARM NEON is an advanced single instruction multiple data (SIMD) engine for the ARM Cortex-A series and Cortex-R52 processor [9]. It was introduced to the ARMv7-A and ARMv7-R profiles, and it is also now as an extension to the ARMv8-A and ARMv8-R profiles. ARM NEON supports 128-bit size Q registers (Q0-Q15). Q registers can be written as 4 32-bit size data, 8 16-bit size data, and 16 8-bit size data. Each Q register can be separated into 2 D registers (64-bit size) as in Figure 1.

The ARM Cortex-A series is used for smartphones and some IoT devices, such as the Raspberry Pi series. For this reason, ARM NEON SIMD is used for high-performance multimedia processing and big-data processing in the Cortex-A series environment.

There are two methods to use ARM NEON. The first one uses ARM NEON intrinsic functions that can be mapped to the ARM NEON assembly instruction by 1-1. The other uses ARM NEON assembly code. In this study, we used ARM NEON intrinsic functions for efficient development of the proposed method.

In 2012, Bernstein introduced implementation of a cryptographic algorithm using ARM NEON [10]. Since then, there have been many research studies on efficient implementation of cryptographic algorithms. The Streit method [11] proposed efficient implementation of a NewHope postquantum key exchange scheme using NEON in an ARMv8-A environment. Seo [12] proposed a high-performance implementation of SGCM in an ARM environment using NEON. Liu Zhe et al. [13] proposed efficient Number Theoretic Transform (NTT) implementation using NEON for efficient Ring-LWE software implementation in a Cortex-A series environment. Seo et al. [14] proposed a compact GCM implementation in a 32-bit ARMv7-A processor environment using NEON.

2.5. Related Studies on Efficient Implementation of Lattice-Based Cryptography. There are many research results on efficient implementation of lattice-based cryptography. Pöppelmann [15] proposed an efficient implementation of Ring-LWE encryption in a reconfigurable hardware 8 bit microcontroller environment and software implementation of GLP on Intel/AMD CPUs and BLISS in the Cortex-M4F environment. Nejatollahi et al. [16] introduced trends and challenges for lattice-based cryptography software implementation. In this paper, the time complexity of matrix-to-matrix/vector multiplication is $O(n^2)$ and it is needed to implement matrix multiplication efficiently. The Liu Zhe method [17] surveyed implementation of lattice-based cryptography on IoT devices and suggested that the Ring-LWE-based cryptosystem would play an essential role in postquantum edge computing and the postquantum IoT environment. Lie Zhe et al. [18] proposed high-performance ideal lattice-based cryptography on an 8-bit AVR microcontroller. They proposed an efficient and secure implementation of Ring-LWE encryption in an 8-bit AVR environment against timing side-channel attack. Bos, Joppe, et al. [19] proposed CRYSTALS-Kyber, which is module-lattice-based KEM, which provides CCA-secure. In their paper, they proposed AVX2 implementation and performance of CRYSTALS-Kyber. The McCarthy method [20] proposed a practical implementation of identity-based encryption over NTRU lattice-based cryptography on an Intel Core i7-6700 CPU. They optimized the DLP-IBE and Gaussian sampler for efficient implementation. Yuan, Ye, et al. [21] proposed memory-constrained implementation of lattice-based encryption in a standard Java card environment. For efficiency, they optimized Montgomery Modular Multiplication (MMM) and Fast Fourier Transform (FFT) for NTT. Oder, Tobias, et al. [22] proposed practical CCA2-secure and masking Ring-LWE implementation in an ARM Cortex-M4F environment. They implemented masked PRNG (SHAKE-128) for a countermeasure of a side-channel attack. The O'Sullivan method [23] reviewed the state-of-the-art in efficient designs for lattice-based cryptography hardware and software implementation.

3. Proposed Method

In this section, we describe our proposed method for efficient matrix multiplication and vector addition using ARM NEON SIMD.

3.1. Problem on Matrix Multiplication and Vector Addition Implementation. First, we describe the problem on matrix multiplication and vector addition for lattice-based cryptography based on the LWE problem. For example, there are Matrix A $(a_{i,j}, 0 \le i \le M, 0 \le j \le N)$, Matrix S $(s_{j,k}, 0 \le j \le N, 0 \le k \le L)$, and Matrix E $(E_{i,k}, 0 \le i \le M, 0 \le k \le L)$ as in Figure 2. If we want to implement matrix multiplication and vector addition, we have to multiply each element on the row of Matrix A and the column of Matrix S. After matrix multiplication, we add the element of the matrix multiplication result and the element of Matrix E. These procedures have a problem, multiplying and addition between each element of the matrix, so computing takes a long time.

For solving and efficient implementation of matrix multiplication and vector addition, we propose efficient matrix multiplication and vector addition using NEON in an ARM Cortex-A environment.

3.2. Proposed Efficient Matrix Multiplication and Vector Addition. For efficient matrix multiplication and vector addition, we used ARM NEON intrinsic functions as shown in Table 1. Using ARM NEON SIMD, we could compute 128-bit size data at each instruction. ARM NEON supports the vector interleave function, vector multiplying accumulation, lane broadcast, and extracting lanes from a vector into a register. For this reason, we proposed matrix multiplication after the matrix transpose for NEON SIMD implementation using ARM NEON intrinsic functions as in Table 1. For an efficient matrix transpose, we used the vector interleave NEON function for efficient implementation. We used vector multiplying accumulation and extracting lanes from a vector into a register and NEON lane broadcast for efficient matrix multiplication.

The NEON data load operation intrinsic function can load data (128-bit) from an 8/16/32-bit data array with a size of 16, 8, or 4. Figure 3 describes a 128-bit size NEON data load from a 16-bit×8 size data array using only the NEON data load intrinsic function.

The NEON data store operation intrinsic function can store data (128-bit) into an 8/16/32-bit data array with a size of 16, 8, or 4. Figure 4 describes a 128-bit size NEON data store into a 16-bit×8 size data array using only the NEON data store intrinsic function.

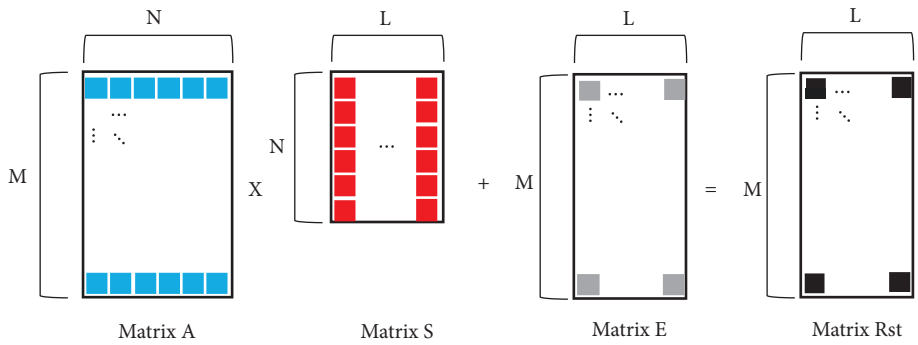

Figure 2: Matrix multiplication and vector addition (existing method).

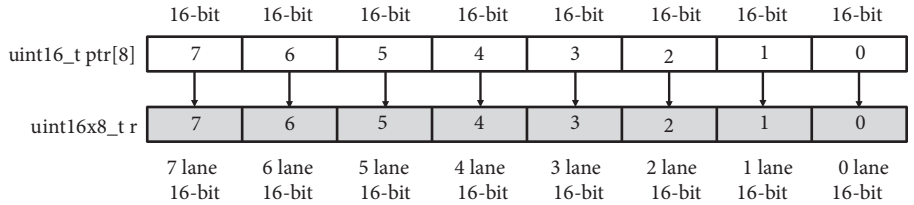

Figure 3: NEON data load operation.

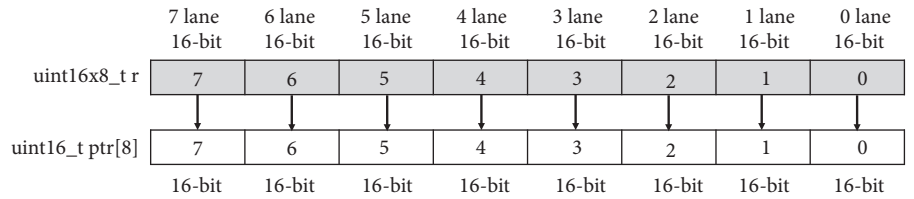

Figure 4: NEON data store operation.

The NEON extracting lane from a NEON vector to a register extracts data according to the lane number value. Figure 5 describes the NEON extracting lane number 2 data from NEON vector *a* (16-bit×8 size) to an unsigned short 16-bit size data register *r*. The NEON extracting lane from a NEON vector to a register operation can also extract data such as 8/16/32-bit data from the NEON vector. This NEON intrinsic function will be used at data accumulate and store into register during matrix multiplication procedure. The details of NEON extracting intrinsic function usage are described in Algorithm 2.

The NEON lane broadcast intrinsic function sets all the lane data in the NEON vector at the same value as in Figure 6. This NEON intrinsic function is used for initializing the accumulation NEON vector as zero during the matrix multiplication procedure. The details of the NEON lane broadcast intrinsic function usage are described in Algorithm 2.

The NEON vector interleave function supports the vector interleave between 2 NEON registers as in Figure 7. After the vector interleave, the result of the vector interleave is to store at the NEON register array (with a size of 2, 2 128-bit data). If we implemented matrix transpose using C language, we have to exchange between elements on the matrix. However, if we use NEON vector interleave, we can exchange 128-bit size data at each instruction. This NEON intrinsic function is used for matrix element transpose during the matrix transpose procedure in Algorithm 1.

Algorithm 1 describes the matrix transpose method using NEON for efficient matrix multiplication. In Algorithm 1, from lines No. 2 to No. 5, it computes the matrix index which is located at outbound of the matrix as index which is located at inbound for NEON SIMD matrix transpose. At that time, the matrix row index can be set as the matrix row index (*BLOCK_TRANSPOSE-N % BLOCK_TRANSPOSE*) and the matrix column index can be set as the matrix column index (*BLOCK_TRANSPOSE–L % BLOCK_TRANSPOSE*).

After calculating the matrix index, it repeats the data load on NEON registers and the vector interleave between NEON registers until the matrix transpose is done for each *BLOCK_TRANSPOSE* from lines No. 7 to No. 56. In Algorithm 1, we assume that each data element of the matrix has 16-bit size data so *BLOCK_TRANSPOSE* means 8 because each NEON register size is 128-bit (16-bit×8 data). After matrix transpose at each *BLOCK_TRANSPOSE*, it stores NEON register data to the transposed matrix array.

For matrix multiplication and vector addition, if we use C language, we have to multiply element by element which are on each matrix and, after matrix multiplication, we have to add each element in the matrix and vector, which takes a long execution time according to the increasing

Table 1: ARM NEON intrinsic functions for the proposed method.

Operations	ARM NEON Intrinsic functions
Load	*uint16x8_t vld1q_u16(__transfersize(8) uint16_t const * ptr);*
Store	*void vst1q_u16(__transfersize(8) uint16_t* ptr, uint16x8_t val);*
Extracting lanes from a vector into a register	*uint16_t vgetq_lane_u16(uint16x8_t vec, __constrange(0, 7) int lane);*
Lane Broadcast	*uint16x8_t vdupq_n_u16(uint16_t value);*
Vector Interleave	*uint16x8x2_t vzipq_u16(uint16x8_t a, uint16x8_t b);*
Vector Multiply Accumulate	*uint16x8_t vmlaq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c);*

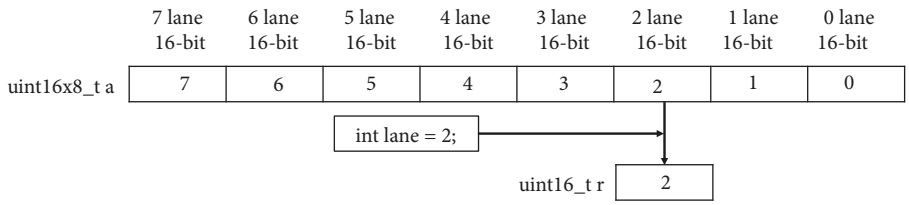

Figure 5: NEON extracting lane from a vector to a register.

matrix size. However, if we use NEON vector multiplication and accumulation as in Figure 8, we can implement matrix multiplication and vector addition by 128-bit size data at each NEON instruction, which accelerates the performance of the matrix multiplication and vector addition.

We propose an efficient matrix multiplication and accumulation method as in Algorithm 2 based on ARM NEON SIMD. Algorithm 2 is conducted after the matrix transpose. In Algorithm 2, *LANE_SHORT_NUM* has the same value as *BLOCK_TRANSPOSE* in Algorithm 1. Line No. 3 in Algorithm 2 describes setting the NEON register *sum_vect* value as 16-bit data 0 using the NEON Intrinsic function (*vdupq*) for lane broadcasting as the same value. From lines No. 4 to No. 7, it loads data from matrix *A* and matrix *S* to the NEON register according to each matrix index. Then it multiplies and accumulates NEON registers for matrix multiplication and vector addition within *N/LANES_SHORT_NUM*. For lines No. 8 and 9, it stores the NEON register value on the array (16-bit data and array size: 8) and accumulates the values on matrix *E* according to the matrix index. Then, it stores the NEON vector into the register and accumulates element values in the register and stores the result on Matrix *E* according to the Matrix *E* index. From lines No. 10 to No. 12, it calculates matrix multiplication and vector addition between matrix elements, which are located at outbound of the matrix size % NEON register lane size. In this part, if the row and column size of the matrix is even, then it does not operate. Using Algorithm 2, we calculate the matrix multiplication and vector addition using NEON, and if the position of matrix element is greater than the NEON register lane size, we used normal matrix multiplication and vector addition using C.

As previously described, we propose an efficient matrix transpose, matrix multiplication, and vector addition. Now, we propose an efficient matrix transpose, multiplication, and vector addition for LWE in lattice-based cryptography as in Algorithm 3. In Algorithm 3, we transpose matrix S using Algorithm 1 and calculate matrix multiplication and vector (matrix *E*) addition using Algorithm 2.

Figure 9 describes Algorithm 3 as a block diagram. In Figure 9, dark blue and dark red parts are calculated using NEON SIMD for matrix multiplication and vector addition based on NEON multiplication and accumulation. At that time, Matrix *S* is transposed by the NEON based matrix transpose operation in Algorithm 1. Positions of light blue and light red parts are greater than matrix row value/NEON lane size or columns value/NEON lane size. These parts are calculated using C and the normal method for matrix multiplication and vector addition.

If we re-used the ARM NEON SIMD data register, which was the result data right before the operation as operand data at the next operation during NEON SIMD programming, it has data dependency, and data dependency causes a Read After Write (RAW) data hazard (aka, stall) which takes some clock cycles to load data that was result data right before operation again. To avoid the data hazard and enhance performance, we scheduled order of NEON register used. For efficient NEON SIMD implementation, we used fully NEON Q registers (Q0-Q15).

4. Experiment & Evaluation

In this section, we describe the experimental environment, the performance measurement, and the evaluation of the proposed method. For objective evaluation, we applied the proposed method on the Lizard.CCA key generation step, which used the LWE problem for key generation.

4.1. Experiment. Our experimental environment was Raspberry Pi 3 Model B. Raspberry Pi 3 Model B has a Broadcom BCM2387 chipset (1.2GHz Quad-Core ARM Cortex-A53) and 1GB LPDDR2 memory. The operating system is Raspbian GNU/Linux 8.0 (Jessie). We used GCC compiler version 4.9.2 and the compile options *-O3 -mcpu=cortex-a53 -mfloat-abi=hard -mfpu=neon-fp-armv8 -mneon-for-64bits -mtune=cortex-a53 -std=c99* for using ARM NEON and compiling for the Cortex-A53 environment. For C version codes,

Require: Matrix S (N × L matrix, $s_{i,j}$, 0 ≤ i ≤ N, 0 ≤ j ≤ L)
Ensure: Matrix S '(L × N matrix, $s'_{i,j}$, 0 ≤ j ≤ N, 0 ≤ i ≤ L)

```
for i from 0 to N, i+= BLOCK_TRANSPOSE do
    if i+BLOCK_TRANSPOSE > N
        let i-= BLOCK_TRANSPOSE-N%BLOCK_TRANSPOSE;
    for j from 0 to L, j+=BLOCK_TRANSPOSE do
        if j+BLOCK_TRANSPOSE > L
            let j-= BLOCK_TRANSPOSE-L%BLOCK_TRANSPOSE;
        vec1_l=NEON_Vector_Load(S+i*L+j);
        vec1_h=NEON_Vector_Load(S+i*L+j+8);
        vec2_l=NEON_Vector_Load(S+(i+8)*L+j);
        vec2_h=NEON_Vector_Load(S+(i+8)*L+j+8);
        t2 = NEON_Vector_Interleave(vec1_l, vec2_l);
        t3 = NEON_Vector_Interleave(vec1_h, vec2_h);
        vec1_l = NEON_Vector_Load(S + (i + 2) * LWE_L + j);
        vec1_h = NEON_Vector_Load (S + (i + 2) * LWE_L + j + 8);
        vec2_l = NEON_Vector_Load (S + (i + 10) * LWE_L + j);
        vec2_h = NEON_Vector_Load (S + (i + 10) * LWE_L + j + 8);
        t4 = NEON_Vector_Interleave(vec1_l, vec2_l);
        t5 = NEON_Vector_Interleave(vec1_h, vec2_h);
        vec1_l = NEON_Vector_Load (S + (i + 3) * LWE_L + j);
        vec1_h = NEON_Vector_Load (S + (i + 3) * LWE_L + j + 8);
        vec2_l = NEON_Vector_Load (S + (i + 11) * LWE_L + j);
        vec2_h = NEON_Vector_Load (S + (i + 11) * LWE_L + j + 8);
        t6 = NEON_Vector_Interleave(vec1_l, vec2_l);
        t7 = NEON_Vector_Interleave(vec1_h, vec2_h);
        m0 = NEON_Vector_Interleave(t0.val[0], t4.val[0]);
        m1 = NEON_Vector_Interleave(t0.val[1], t4.val[1]);
        m2 = NEON_Vector_Interleave(t1.val[0], t5.val[0]);
        m3 = NEON_Vector_Interleave(t1.val[1], t5.val[1]);//
        m4 = NEON_Vector_Interleave(t2.val[0], t6.val[0]);
        m5 = NEON_Vector_Interleave(t2.val[1], t6.val[1]);
        m6 = NEON_Vector_Interleave(t3.val[0], t7.val[0]);
        m7 = NEON_Vector_Interleave(t3.val[1], t7.val[1]);
        t0 = NEON_Vector_Interleave(m0.val[0], m4.val[0]);
        t1 = NEON_Vector_Interleave(m0.val[1], m4.val[1]);
        t2 = NEON_Vector_Interleave(m1.val[0], m5.val[0]);
        t3 = NEON_Vector_Interleave(m1.val[1], m5.val[1]);
        t4 = NEON_Vector_Interleave(m2.val[0], m6.val[0]);
        t5 = NEON_Vector_Interleave(m2.val[1], m6.val[1]);
        t6 = NEON_Vector_Interleave(m3.val[0], m7.val[0]);
        t7 = NEON_Vector_Interleave(m3.val[1], m7.val[1]);
        NEON_Vector_Store (S' + j * LWE_N + i, t0.val[0]);
        NEON_Vector_Store (S' + j * LWE_N + i + 8, t0.val[1]);
        NEON_Vector_Store (S' + (j + 1) * LWE_N + i, t1.val[0]);
        NEON_Vector_Store (S' + (j + 1) * LWE_N + i + 8, t1.val[1]);
        NEON_Vector_Store (S' + (j + 2) * LWE_N + i, t2.val[0]);
        NEON_Vector_Store (S' + (j + 2) * LWE_N + i + 8, t2.val[1]);
        NEON_Vector_Store (S' + (j + 3) * LWE_N + i, t3.val[0]);
        NEON_Vector_Store (S' + (j + 3) * LWE_N + i + 8, t3.val[1]);
        NEON_Vector_Store (S' + (j + 4) * LWE_N + i, t4.val[0]);
        NEON_Vector_Store (S' + (j + 4) * LWE_N + i + 8, t4.val[1]);
        NEON_Vector_Store (S' + (j + 5) * LWE_N + i, t5.val[0]);
        NEON_Vector_Store (S' + (j + 5) * LWE_N + i + 8, t5.val[1]);
        NEON_Vector_Store (S' + (j + 6) * LWE_N + i, t6.val[0]);
        NEON_Vector_Store (S' + (j + 6) * LWE_N + i + 8, t6.val[1]);
        NEON_Vector_Store (S' + (j + 7) * LWE_N + i, t7.val[0]);
        NEON_Vector_Store (S' + (j + 7) * LWE_N + i + 8, t7.val[1]);
        Return S'
```

Algorithm 1: Efficient matrix transpose.

Require: Matrix A (M × N matrix, $a_{i,j}, 0 \le i \le M, 0 \le j \le N$), Matrix S (N × L matrix, $s_{j,k}, 0 \le j \le N, 0 \le k \le L$), Matrix E (M × L matrix, $e_{i,k}, 0 \le i \le M, 0 \le$ k $\le$ L)
Ensure: Matrix E (M × L matrix, $r_{i,k}, 0 \le i \le M, 0 \le$ k $\le$ L)

```
for i from 0 to M do
    for j from 0 to L do
        sum_vect = NEON_Lane_Broadcast(0);
        for k from 0 to iter_k do
            a_vec = NEON_Vector_Load (A + i * N + k * LANES_SHORT_NUM);
            s_vec = NEON_Vector_Load (S + j * N + k * LANES_SHORT_NUM);
            sum_vect = NEON_Multiply_Accumulate(sum_vect, a_vec, s_vec);
        NEON_Vector_Store (sum, sum_vect);
        E[i * L + j] += sum[0]+sum[1]+sum[2] + sum[3] +sum[4]+sum[5]+sum[6]+sum[7];
        if (k == N/LANES_SHORT_NUM) && (N%LANES_SHORT_NUM)
            for k from N-(N%LANES_SHORT_NUM) to N do
                E[i * L + j] += A[i*N+k]*B[k*N+j];
Return E;
```

Algorithm 2: Efficient matrix multiplication and accumulation.

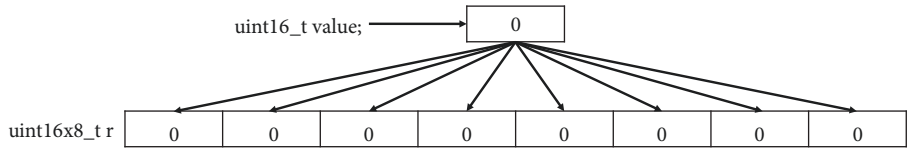

Figure 6: NEON lane broadcast operation.

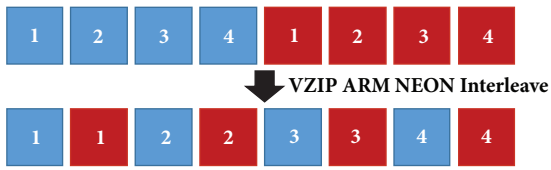

Figure 7: VZIP ARM NEON interleave operation.

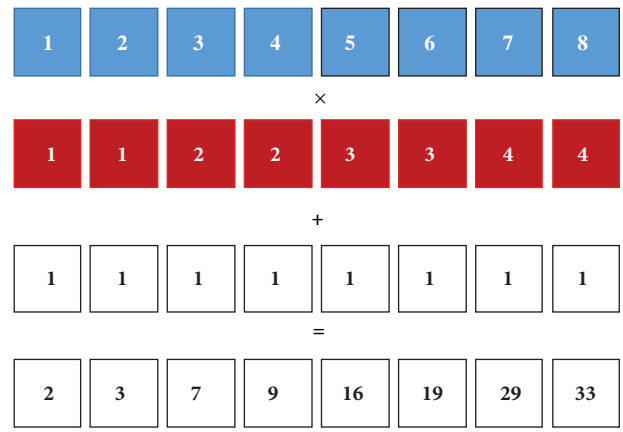

Figure 8: VMLA ARM NEON multiply accumulation operation.

we used the compile option for NEON autovectorization as *-O3 -mcpu=cortex-a53 –ftree-vectorize -mfloat-abi=hard -mfpu=neon-fp-armv8 -mneon-for-64bits -mtune=cortex-a53 -std=c99*. The GCC vectorization was enabled using the flag *–ftree-vectorize and –O3. To enable NEON, we used flags, namely, -mfloat-abi=hard -mfpu=neon-fp-armv8 -mneon-for-64bits -mtune=cortex-a53*. If we used GCC autovectorization for NEON, the GCC compiler made the C source code as NEON code by autovectorization.

4.2. Evaluation. To evaluate our method, we measured the average execution time for 1,000 periods of operation according to Lizard.CCA parameters. For Lizard.CCA CATEGORY5_N1088 and Lizard.CCA CATEGORY5_N1088 parameters, we could not measure the execution time. First, we measured and compared the performance of the proposed matrix transpose method and normal C version as in Table 2. Our proposed matrix transpose method performed better than C version (with GCC autovectorization). The C version (with GCC autovectorization) had a low performance because it had some conditional branches, such as 'while' and 'if' statements.

After we measured the proposed matrix transpose method, we measured the proposed matrix multiplication and vector addition. For an objective evaluation, we compared the performance of the proposed method with the C version from the matrix multiplication and vector addition part in the Lizard.CCA key generation step [3] according to the Lizard.CCA parameters. The C version from Lizard.CCA [3] was submitted to NIST PQC Standardization round 1, and it was normal C version matrix multiplication using C pointer. The proposed method for matrix multiplication and vector addition included the matrix transpose part. Table 3 describes the comparison results between the C version [3] (with GCC autovectorization) and the proposed

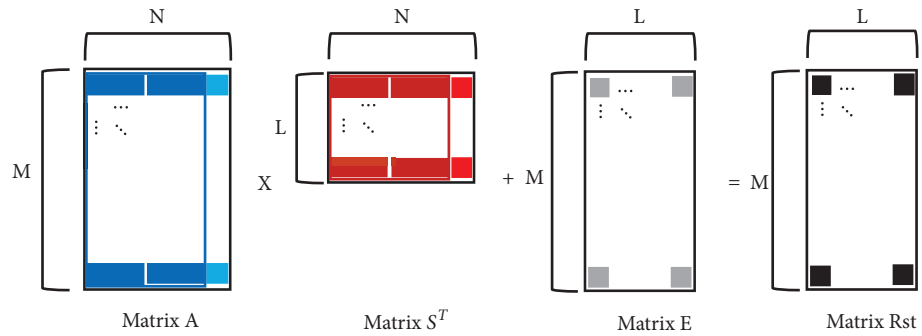

Figure 9: Proposed matrix multiplication and vector addition.

Table 2: Matrix transpose performance (Unit: ms).

N	M	L	C version (Auto-Vectorization)	Proposed (NEON)
536	1024	256	364.2304	0.446443
663	1024	256	630.0066	0.707373
816	1024	384	970.4782	1.78282
952	1024	384	1172.607	2.078113

Table 3: Matrix multiplication performance (unit: ms).

N	M	L	C version [3] (Auto-Vectorization)	Proposed (NEON)
536	1024	256	148.8991	93.91285
663	1024	256	171.0976	159.2069
816	1024	384	334.7499	224.5633
952	1024	384	391.7564	361.7326

Table 4: Lizard.CCA key generation performance (unit: ms).

N	M	L	Cheon et al. [3] (Auto-Vectorization)	Proposed (NEON)
536	1024	256	622.920	579.071
663	1024	256	736.950	709.942
816	1024	384	1075.164	993.760
952	1024	384	1239.633	1124.144

method. The proposed method improved the performance at the parameters by 36.93%, 6.95%, 32.92%, and 7.66%, respectively.

Our proposed methods performed better. Next we applied the proposed methods on the Lizard.CCA key generation step [3] for objective evaluation. Table 4 describes the performance comparison results between the Lizard.CCA key generation step [3] and the proposed method. The proposed methods with the Lizard.CCA key generation steps had improved performance at the parameters by 7.04%, 3.66%, 7.57%, and 9.32%, respectively, over the original Lizard.CCA key generation step [3].

According to Tables 3 and 4, the proposed methods for efficient matrix multiplication had improved performance. However, in the case of the Lizard.CCA CATEGORY3_N663 parameter, the rate of increase in performance was lower than the others because parameter N was 663 and it had a remainder as 7 ($663 = 8 \times 82 + 7$) so it was necessary to do matrix multiplication for matrix elements that were located from 656 to 663 using normal method.

5. Conclusions

Nowadays, many postquantum cryptography systems are being developed to deal with quantum computing technologies and security threats to the existing cryptosystem. NIST is working on postquantum cryptography standardization. A large part of the submissions to NIST's PQC Standardization competition is lattice-based cryptography, and many lattice-based cryptographic algorithms are based on the LWE

Require: Matrix A (M × N matrix, $a_{i,j}, 0 \le i \le M, 0 \le j \le N$), Matrix S (N × L matrix, $s_{j,k}, 0 \le j \le N, 0 \le k \le L$), Matrix E (M × L matrix, $e_{i,k}, 0 \le i \le M, 0 \le$ k $\le$ L)
Ensure: Matrix Rst (M × L matrix, $r_{i,k}, 0 \le i \le M, 0 \le$ k $\le$ L)
1: *NEON_Matrix_Transpose* (Matrix S) (Algorithm 1.)
2: *NEON_Matrix_Multiply_Accumulate* (Matrix A, Matrix S, Matrix E) (Algorithm 2.)
3: *Return E*

Algorithm 3: Efficient matrix transpose, multiplication, and accumulation for LWE.

problem. The LWE problem-based procedures need matrix multiplication between huge size matrices. However, normal matrix multiplication calculates element by element on the matrix. For efficient matrix multiplication, we proposed matrix multiplication and vector addition with a matrix transpose using ARM NEON SIMD techniques for efficiency. The proposed matrix multiplication and vector addition with matrix transpose method improved performance at each parameter by 36.93%, 6.95%, 32.92%, and 7.66%, respectively, and the proposed method with Lizard.CCA key generation steps have improved performance at each parameter by 7.04%, 3.66%, 7.57%, and 9.32%, respectively, over the original Lizard.CCA key generation step [3]. In the future, research on efficient matrix multiplication on matrix elements that are located at outbound of NEON register lane size is needed for further improved efficiency and using a fully NEON method. We will research on efficient implementation of matrix multiplication and vector addition for lattice-based cryptography using full NEON SIMD for any parameters, mixing ARM NEON/ARM assembly instruction, and AVX2 SIMD in an Intel x64 environment.

Data Availability

Proposed matrix transpose, multiplication, and vector addition implementation source codes are uploaded to Github repository (https://github.com/pth5804/MatTrans_Mul_NEON_PQC).

Conflicts of Interest

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

Acknowledgments

This work of Taehwan Park and Howon Kim was supported by the Ministry of Trade, Industry, & Energy (MOTIE, Korea) under the Industrial Technology Innovation Program (no. 10073236). This work of Hwajeong Seo was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (no. NRF-2017R1C1B5075742). This work of Junsub Kim and Haeryong Park was supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (no. 2017-0-00616, development for lattice-based postquantum public key cryptographic scheme).

References

[1] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pp. 212–219, ACM, 1996.

[2] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (SFCS '94)*, pp. 124–134, IEEE, 1994.

[3] J. H. Cheon, S. Park, J. Lee et al., "Post-Quantum Cryptography," National Institute of Standards and Technology, Tech. rep, 2017, https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[4] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM*, vol. 56, no. 6, article 34, 2009.

[5] C. Peikert, "Public-key cryptosystems from the worst-case shortest vector problem," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, ACM, 2009.

[6] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé, "Classical hardness of learning with errors," in *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC '13)*, pp. 575–584, ACM, June 2013.

[7] H. C. Jung, H. Kyoohyung, K. Jinsu, L. Changmin, and S. Yongha, "Practical post- quantum public key cryptosystem based on LWE," in *Proceedings of the 19th Annual international Conference on Information Security and Cryptology*, 2016.

[8] D. Jintai, X. Xiang, and L. Xiaodong, "A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem," in *Cryptology EPrint Archive*, p. 688, 688, 2012.

[9] *ARM, NEON Programmer's Guide version 1.0, (2013)*.

[10] D. J. Bernstein and P. Schwabe, "NEON Crypto," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, vol. 7428 of *Lecture Notes in Computer Science*, pp. 320–339, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] S. Streit and F. De Santis, "Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple," *IEEE Transactions on Computers*, 2017.

[12] H. Seo, "High performance implementation of SGCM on high-end IoT devices," *Journal of Information and Communication Convergence Engineering*, vol. 15, no. 4, pp. 212–216, 2017.

[13] Z. Liu, R. Azarderakhsh, H. Kim, and H. Seo, "Efficient Software Implementation of Ring-LWE Encryption on IoT Processors," *IEEE Transactions on Computers*, 2017.

[14] H. Seo, G. Lee, T. Park, and H. Kim, "Compact GCM implementations on 32-bit ARMv7-A processors," in *Proceedings of the 2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 704–707, Jeju, October 2017.

[15] T. Pöppelmann, "Efficient implementation of ideal lattice-based cryptography," *it - Information Technology*, vol. 59, no. 6, 2017.

[16] H. Nejatollahi, N. Dutt, and R. Cammarota, "Special session: Trends, challenges and needs for lattice-based cryptography implementations," in *Proceedings of the 12th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2017*, Republic of Korea, October 2017.

[17] Z. Liu, K.-K. R. Choo, and J. Grossschadl, "Securing Edge Devices in the Post-Quantum Internet of Things Using Lattice-Based Cryptography," *IEEE Communications Magazine*, vol. 56, no. 2, pp. 158–162, 2018.

[18] Z. Liu, T. Pöppelmann, T. Oder et al., "High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 4, 2017.

[19] J. Bos, L. Ducas, E. Kiltz et al., "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM," in *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 353–367, London, April 2018.

[20] S. McCarthy, N. Smyth, and E. O'Sullivan, "A Practical Implementation of Identity-Based Encryption Over NTRU Lattices," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 10655, pp. 227–246, 2017.

[21] Y. Yuan, K. Fukushima, S. Kiyomoto, and T. Takagi, "Memory-constrained implementation of lattice-based encryption scheme on standard Java Card," in *Proceedings of the 10th IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017*, pp. 47–50, USA, May 2017.

[22] O. Tobias, T. Schneider, T. Pöppelmann, and T. Güneysu, "Practical cca2-secure and masked ring-lwe implementation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 1, pp. 142–174, 2018.

[23] E. O'Sullivan and F. Regazzoni, "Special session paper: Efficient arithmetic for lattice-based cryptography," in *Proceedings of the 12th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2017*, Republic of Korea, October 2017.