ARMed Frodo
FrodoKEM on 64-bit ARMv8 Processors

No Author Given

No Institute Given

Abstract. FrodoKEM is one of Post-quantum Cryptography, which is selected Round 3 alternate candidates of Public-key Encryption and Key-establishment Algorithms at NIST Post-quantum Cryptography Standardization. FrodoKEM uses AES-128 algorithm for generate pseudo-random matrix, and also uses matrix-multiplication. At that time, a huge computational load occurred to the pseudo-random number generation and matrix-multiplication operation, reducing the overall performance of FrodoKEM. In this paper, we propose parallel matrix-multiplication and built-in AES accelerator for AES-128 on ARMv8 processors, and applied these techniques to FrodoKEM-640. To implement parallel matrix-multiplication, the vector registers and vector instructions are used. Proposed parallel matrix-multiplication can be generated 80 element of output matrix at once. As a result, the matrix-multiplication has 43.8× faster than normal matrix-multiplication in best-case, the implementation FrodoKEM-640 with all of the proposed techniques has 10.22× better performance in maximum than previous C only implementation.

Keywords: Post-quantum Cryptography · FrodoKEM · 64-bit ARMv8 Processors · Software Implementation.

1 Introduction

As the technology of quantum computers improved, classic cryptography algorithms are threatened. Cryptography algorithms based on mathematical hard problems can be easily broken by quantum algorithms. In order to resolve these issues, the National Institute of Standards and Technology (NIST) in the United States of America has held a contest to standardize post-quantum cryptography in preparation for the quantum computer era. Among them, FrodoKEM was selected as a Round 3 alternate candidate. FrodoKEM is a Public Key Encryption (PKE) algorithm based on Learning With Errors (LWE). Since the algorithm is based on very conservative mathematical problems, the execution timing is slower than other lattice based cryptography. In this paper, we challenge to the optimal implementation of FrodoKEM on 64-bit ARMv8 processors. The target processor is widely used in modern smartphone, tablet, and laptop computers. Detailed contributions are follow.

1.1 Contributions

- **Matrix-multiplication with parallel operations.** Matrix-multiplication is used in Key generation, Encapsulation, and Decapsulation of FrodoKEM-640 schemes. The size of matrix seed A is 640∗640, and error matrix s is 640∗8 or 8∗640. At this point, the matrix-multiplication occurs large computational loads, because the matrix size is huge. We propose the implementation by using vector registers and vector instructions of ARMv8 architectures, for the parallel matrix-multiplication. The proposed implementation does not transpose matrices, but it loads values used for each output generation in a vector registers. The regularity of the value loaded for all of rounds exist in the matrix multiplication and we made the value loading, accordingly. Therefore the proposed parallel matrix-multiplication can calculate 80 number of output values at once. The existing matrix-multiplication $A * s$, and $s * A$ takes average 2,228.5ms, and 2,557.7ms for 50 repetitions (each repetition has 200 iterations), respectively on the Apple A10X Fusion processor (@2.38 GHz). However the proposed parallel matrix-multiplication takes only 80.0ms, and 58.4ms, respectively. This is about a 27.9×, and 43.8× performance improvements for $A * s$ and $s * A$, respectively.
- **High-speed random number generation with AES accelerator based AES encryption.** To generate random numbers, FrodoKEM-640 scheme utilizes AES-128 encryption algorithm. We pursued higher efficiency by replacing software based AES implementation with high-speed AES accelerator supported by ARMv8 processors. This approach improved the performance significantly. Finally we combined aforementioned methods to achieve the highest performance of FrodoKEM on ARMv8 processors.
- **First FrodoKEM implementation on 64-bit ARM processors** 64-bit ARM processors are widely used in smartphone, tablet, and laptop. For this reason, benchmarking on the target processor should be regarded. To the best of our knowledge, this is the first FrodoKEM implementation on 64-bit ARM processors. We believe that this can be beneficial for following researchers to evaluate the performance of FrodoKEM.

2 Backgrounds

2.1 FrodoKEM

The FrodoKEM algorithm was first announced 2016 ACM SIGSAC conference [1]. This is one of lattice-based cryptography. It is mainly based on Learning With Errors (LWE) that can resist to quantum adversaries. In 2017, FrodoKEM was submitted to the NIST post quauntum cryptography standardization conference. In 2021, it is selected as alternate candidates in public-key encryption field in Round 3 of competition, with other 4 kinds of algorithms including BIKE [2], HQC [3], NTRU Prime [4], and SIKE [5].

FrodoKEM designed for IND-CCA security at 3-levels, FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344, where these algorithms target Level 1, 3,

and 5 in the NIST call for proposals, respectively. FrodoKEM consists of 3-steps, including Key generation, Encapsulation, and Decapsulation, each step is shown in Algorithm 1, 2, and 3.

- **Key generation.** The key generation algorithm makes public key pk and secret key sk.
- **Encapsulation.** The encapsulation algorithm encrypt a message M to ciphertext c using public key pk.
- **Decapsulation.** The decapsulation algorithm decrypt a ciphertext c to message M with secret key sk.

Algorithm 1 FrodoKEM Key Generation

Input: NONE.
Output: Key pair (PublicKey pk, SecretKey sk) $\in (\{0,1\}^{length_{seed_A}} \times \mathbb{Z}_q^{n\times\overline{n}}) \times \mathbb{Z}_q^{\overline{n}\times n}$.
1: Uniformly random seed chosen $\mathtt{seed}_A \leftarrow sU(\{0,1\}^{length_{seed_A}})$
2: Matrix $A \in \mathbb{Z}_q^{n\times n}$ generated
3: Uniformly random seed chosen $\mathtt{seed}_{SE} \leftarrow sU(\{0,1\}^{length_{seed_{SE}}})$
4: Pseudo-random bit string generated $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2n\overline{n}-1)})$ with SHAKE
5: Error matrix $\mathtt{S}^T$ sampled
6: Error matrix $\mathtt{E}$ sampled
7: $\mathtt{B} = \mathtt{AS} + \mathtt{E}$ computed
8: return PublicKey $pk \leftarrow (\mathtt{Seed}_A, \mathtt{B})$ and SecretKey $sk \leftarrow \mathtt{S}^T$

Algorithm 2 FrodoKEM Encapsulation

Input: Message $\mu \in M$, PublicKey $pk = (\mathtt{Seed}_A, \mathtt{B}) \in \{0,1\}^{length}_{seed_A} \times \mathbb{Z}_q^{n\times\overline{n}}$.
Output: Ciphertext $c = (\mathtt{C}_1, \mathtt{C}_2) \in \mathbb{Z}_q^{\overline{m}\times n} \times \mathbb{Z}_q^{\overline{m}\times\overline{n}}$.
1: Pseudo-random matrix generated with $\mathtt{Seed}_A$ $\mathtt{A} \leftarrow$
2: Uniformly random seed chosen $\mathtt{seed}_{SE} \leftarrow sU(\{0,1\}^{length_{seed_{SE}}})$
3: Pseudo-random bit string generated $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{m}\overline{n}-1)})$ with SHAKE
4: Error matrix $\mathtt{S}'$ sampled
5: Error matrix $\mathtt{E}'$ sampled
6: Error matrix $\mathtt{E}''$ sampled
7: $\mathtt{B}' = \mathtt{S}'\mathtt{A} + \mathtt{E}$, $\mathtt{V} = \mathtt{S}'\mathtt{B} + \mathtt{E}''$ computed
8: return Ciphertext $c \leftarrow (\mathtt{C}_1, \mathtt{C}_2) = (\mathtt{B}', \mathtt{V} + \mu)$

FrodoKEM uses AES-128 or SHAKE-128 algorithm to generate pseudo-random numbers, and the generated pseudo-random numbers are used to the public matrix. Since FrodoKEM stores pseudo-random numbers as a matrix, the matrix-multiplication is used internally. At this time, since the size of the matrix

Algorithm 3 FrodoKEM Decapsulation

Input: Ciphertext $c = (\mathtt{C}_1, \mathtt{C}_2) \in \mathbb{Z}_q^{\overline{m} \times n} \times \mathbb{Z}_q^{\overline{m} \times \overline{n}}$, SecretKey $sk = \mathtt{S}^T \in \mathbb{Z}_q^{\overline{n} \times n}$.
Output: Message $\mu' \in M$
1: $M = \mathtt{C}_2 - \mathtt{C}_1\mathtt{S}$ computed
2: return message $\mu' \leftarrow M$

Table 1. Type of register packing of ARMv8 processors.

Type	Unit	Data Quantity	Specifier
Byte	8-bit	16 or 8	16B or 8B
Half-word	16-bit	8 or 4	8H or 4H
Single-word	32-bit	4 or 2	4S or 2S
Double-word	64-bit	2 or 1	2D or 1D

is very large, $640 * 640$, $976 * 976$, and $1344 * 1344$, for level 1, 3, and 5, respectively. This shows that huge computational overheads occurs during the matrix-multiplication. In order to improve the performance of matrix-multiplication, we proposed parallel matrix-multiplication by using ARMv8 vector registers and instructions.

2.2 ARMv8 Processor

ARMv8 processor has 31 of 64-bit general purpose registers (i.e. scalar registers), and 32 of 128-bit vector registers [6]. Before using vector registers, the data packing process is required. At this point, the arrangement specifier is used for the packing unit. Table 1 shows supported data packing in the target architecture. The matrix-multiplication of FrodoKEM uses 2 byte-wise operation. The arrangement specifier (8H), which indicates 8 half words, is used.

2.3 Related Works

In this section, we introduce the optimized implementation of another Post Quantum Cryptography on ARMv8 processors, which is the target processor of this paper.

Sanal et al. [7] implemented the CRYSTALS-Kyber on the 64-bit ARM Cortex-A processor, which is one of ARMv8 family. They presented optimized Number Theoretic Transform (NTT), noise sampling implementation, and AES accelerator based implementation. Performance enhancements of proposed implementation are 1.72×, 1.88×, and 2.29× for key generation, encapsulation, and decapsulation than reference implementations, respectively.

Nguyen et al. [8] shows optimized implementations of CRYSTALS-Kyber, NTRU, and Saber on Apple M1 microcontroller and Cortex-A72 processor. The proposed implementation uses NEON instruction (vector instruction) to implement parallel polynomial-multiplication. The results show that on the Apple M1

Table 2. List of instructions for parallel-implementation of Matrix-Multiplication; `Xd, Vd`: destination register(scalar, vector), `Xn, Vn, Vm`: source register(scalar, vector, vector), `Vt`: transferred vector register, `T`: Arrangement specifier, `i`: Index.

asm	Operands	Description	Operation
ADD	Vd.T, Vn.T, Vm.T	Add vector	Vd ← Vn + Vm
CBNZ	Xt, (Label)	Compare and Branch on Nonzero	Go to Label
LD1	Vt.T, [Xn]	Load multiple single-element structures	Vt ← [Xn]
LD1	Vt.T[i], [Xn]	Load one single-element structures	Vt[i] ← [Xn][i]
LD1R	Vt.T, [Xn]	Load one single-element structures and Replicate	Vt ← [Xn][i]
LDR	Xt, [Xn]	Load register immediate	Xt ← [Xn]
MOV	Xd, #imm	Move immediate (scalar)	Xd ← #imm
MOVI	Vt.T, #imm	Move immediate (vector)	Vt ← #imm
MUL	Vd.T, Vn.T, Vm.T	Multiply	Vd ← Vn * Vm
RET	{Xn}	Return from subroutine	Return
ST1	Vt.T, [Xn]	Store multiple single-element structures	[Xn] ← Vt
STR	Xt, [Xn]	Store register immediate	[Xn] ← Xt
SUB	Xd, Xn, #imm	Subtract immediate	Xd ← Xn - #imm

environment, the encapsulation has 1.37-1.60×, 2.33-2.45×, 3.05-3.24×, 6.68× better performance than C language implementation for Saber, Kyber, NTRU-HPS, and NTRU-HRSS, respectively. The decapsulation shows faster than C implementation that 1.55-1.74×, 2.96-3.04×, 7.89-8.49×, and 7.24×, for Saber, Kyber, NTRU-HPS, and NTRU-HRSS, respectively

Jalali et al. [9] proposes the SIKE algorithm on the ARMv8 processor. The main contribution is mixed implementation of field arithmetic design with the parallel implementation technique. Consequently, the proposed implementation achieved overall 10% higher performance than previous work.

3 Parallel Matrix-Multiplication

In this section, we show the parallel matrix-multiplication method on the 64-bit ARMv8 processor. For the optimal implementation, efficient register scheduling and instructions are utilized.

3.1 Instruction set

64-bit ARMv8 processor has various instructions. These instructions can be classified into two types: scalar instructions and vector instructions. Vector instructions provide the parallel operation. Table 2 summarizes instructions which are used for the proposed implementation.

3.2 Register scheduling

To efficiently utilize limited registers, optimal register scheduling is required. Table 3 shows the register scheduling according to the matrix-multiplication implementation. In register notation, `X` is a scalar register, `V` is a vector register, and the number (`n`) is register number, respectively.

Table 3. Register scheduling for matrix-multiplication. `X`: Scalar registers, `V`: Vector registers, `n`: Register number.

Type	Registers	Usage
A * s	V0-V9	8 of $A[i]$ values and intermediate results
	V10-V19	Output of matrix-multiplication
	V31	$s[i * 640 * k]$ values
	X0	Output address
	X1	Matrix A address
	X2	Matrix s address
	X4-5	Loop index
s * A	V0-V9	$A[i * 640 + 80 * k]$ values and intermediate results
	V10-V19	Output of matrix-multiplication
	V31	8 of $s[i]$ values
	X0	Output address
	X1	Matrix A address
	X2	Matrix s address
	X3-5	Loop index

3.3 Matrix Multiplication: $A * s$

Upper part of Figure 1 shows the matrix-multiplication of A and s of FrodoKEM-640. A is a 640*640 matrix and s is a 640*8 matrix. For this reason, the *output* is a 640*8 matrix. To calculate the value of one *output*, 640 operations between matrix A and matrix s are required.

In the $A * s$ matrix-multiplication operation, the value of the column component of the A matrix is used and the value of the row component of the s matrix is used. Each value is multiplied, and one output value is generated by adding them together.

For instance, each *output* value can be generated following equation.

$$\texttt{Output[0]} = \texttt{A[0]} * \texttt{s[0]} + \texttt{A[1]} * \texttt{s[1]} + ... + \texttt{A[639]} * \texttt{s[639]}$$
$$\texttt{Output[1]} = \texttt{A[640]} * \texttt{s[640]} + \texttt{A[641]} * \texttt{s[641]} + ... + \texttt{A[1279]} * \texttt{s[1279]}$$
$$...$$
$$\texttt{Output}[i * 8 + k] = \texttt{A}[i * 640 + 0] * \texttt{s}[k * 640 + 0] + \texttt{A}[i * 640 + 1] * \texttt{s}[k * 640 + 1] + ... + \texttt{A}[i * 640 + 639] * \texttt{s}[k * 640 + 639]$$

Since each value is 16-bit wise, 8 values can be stored in one vector register. Therefore, 80 vector registers are required to generate single *output* value in ideal cases. However, only 10 registers can be used at once due to limit of vector

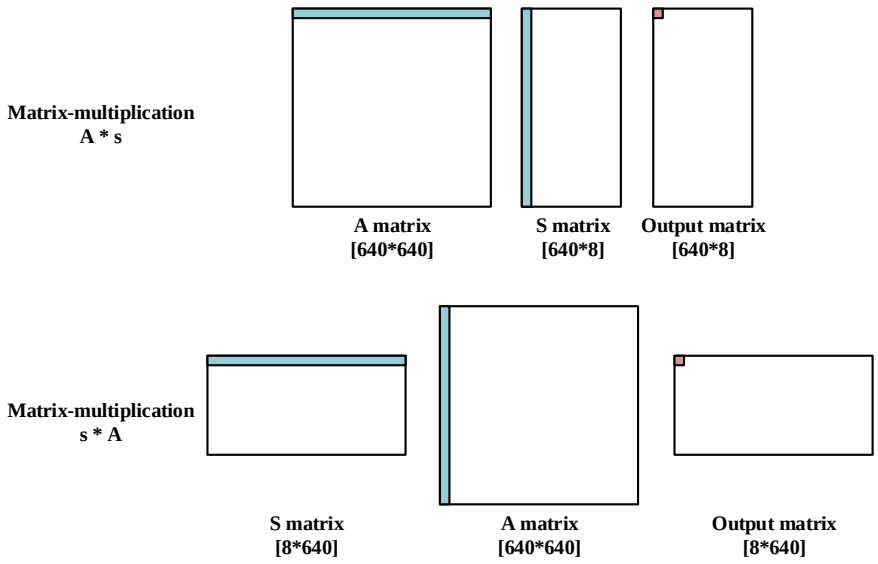

Fig. 1. (Upper) matrix-multiplication of $A * s$; (lower) matrix-multiplication of $s * A$.

registers. Also, an operation causes inefficiency when adding each element. Some vector instructions provide a pair-wise operation that adds values inner values from single vector register. Since there are 8 values to be added, not only the number of additions increases, but also the register utilization decreases.

To solve this problem, we changed the operation order. Table 4 summarizes the value used for each index to calculate the matrix-multiplication $A * s$. Every $A[0]$ element is multiplied by $s[n * 640 + 0]$ (n is maximum value of k). All of $A[1]$ elements are operated with $s[n * 640 + 1]$. As a result, all $A[m]$ values are computed with $s[n * 640 + m]$. The *output* can be calculated by adding up all column elements of Table 4. Instead of assigning values to vector registers linearly, we assign values to follow this rule.

We used 10 vector registers to store A values (`V0-V9`) and 1 vector register (`V31`) to store s value. A registers are multiplied with the s register. These results are saved to A registers. Afterward, multiplied results are added to intermediate result registers (`V10-V19`). At the final round, the *output* is obtained from intermediate result registers. We call this step as *innerloop*. Algorithm 4 shows the implementation of matrix-multiplication $A * s$ *innerloop*. One *innerloop* can be calculated 80 *output* values. In algorithm 4, line 1-16 loads s values to `V31` vector register. $LD1$ instruction can be loaded 1 value to specific place on the vector register. ADD instruction is using for moving pointer address. At this time, 1280 pointer is moved to load a value that is apart by 640 index, because the value of s is 16-bit. For this reason, `V31` has $s[0]$, $s[640]$, $s[1280]$, $s[1920]$,

Table 4. Elements of matrix-multiplication $A * s$, where $i = 0 \sim 639, k = 0 \sim 7, j = 0 \sim 649$.

	i = 0			i = 1		
	k = 0	k = 1	k = n	k = 0	k = 1	k = n
j = 0	A[0]*s[0]	A[0]*s[640]	A[0]*s[n*640+0]	A[640]*s[0]	A[640]*s[640]	A[640]*s[n*640+0]
j = 1	A[1]*s[1]	A[1]*s[641]	A[1]*s[n*640+1]	A[641]*s[1]	A[641]*s[641]	A[641]*s[n*640+1]
j = m	A[m]*s[m]	A[m]*s[640+m]	A[m]*s[n*640+m]	A[640+m]*s[m]	A[640+m]*s[640+m]	A[640+m]*s[n*640+m]

	i = 2			i = o		
	k = 0	k = 1	k = n	k = 0	k = 1	k = n
j = 0	A[1280]*s[0]	A[1280]*s[640]	A[1280]*s[n*640+0]	A[o*640]*s[0]	A[o*640]*s[640]	A[o*640]*s[n*640+0]
j = 1	A[1281]*s[1]	A[1281]*s[641]	A[1281]*s[n*640+1]	A[o*640+1]*s[1]	A[o*640+1]*s[641]	A[o*640+1]*s[n*640+1]
j = m	A[1280+m]*s[m]	A[1280+m]*s[640+m]	A[1280+m]*s[n*640+m]	A[o*640+m]*s[m]	A[o*640+m]*s[640+m]	A[o*640+m]*s[n*640+m]

s[2560], s[3200], s[3840], and s[4480] values at the first round. Line 17-36 loads A values to `V0-V9` vector registers. $LD1R$ instruction can be loaded single value and copies it through the register. For example, `V0` has 8 of A[0] values at the first round. ADD is used for same objective in line 1-17. Line 37-46 performs multiplication $A * s$, and stored results to `V0-V9`. Lastly, line 47-56 adds intermediate results to `V10-19`. In line 57-65, the address pointers are reordered and iterated by the *innerloop* through the condition. *Innerloop* iterate 640 times. afterward, it moves to *outerloop*.

Algorithm 4 *Innerloop* of Matrix-multiplication $A * s$.

Input: A address = [x1], s address = [x2], Loop counter = [x5]
Output: 80 *output* values

```
 1: LD1 V31.h[0], [x2]
 2: ADD X2, X2, #1280
 3: LD1 V31.h[1], [x2]
 4: ADD X2, X2, #1280
 5: LD1 V31.h[2], [x2]
 6: ADD X2, X2, #1280
 7: LD1 V31.h[3], [x2]
 8: ADD X2, X2, #1280
 9: LD1 V31.h[4], [x2]
10: ADD X2, X2, #1280
11: LD1 V31.h[5], [x2]
12: ADD X2, X2, #1280
13: LD1 V31.h[6], [x2]
14: ADD X2, X2, #1280
15: LD1 V31.h[7], [x2]
16: ADD X2, X2, #1280

17: LD1R V0.8h, [x1]
18: ADD X1, X1, #1280
19: LD1R V1.8h, [x1]
20: ADD X1, X1, #1280
21: LD1R V2.8h, [x1]
22: ADD X1, X1, #1280
23: LD1R V3.8h, [x1]
24: ADD X1, X1, #1280
25: LD1R V4.8h, [x1]
26: ADD X1, X1, #1280
27: LD1R V5.8h, [x1]
28: ADD X1, X1, #1280
29: LD1R V6.8h, [x1]
30: ADD X1, X1, #1280
31: LD1R V7.8h, [x1]
32: ADD X1, X1, #1280
33: LD1R V8.8h, [x1]
34: ADD X1, X1, #1280
35: LD1R V9.8h, [x1]
36: ADD X1, X1, #1280

37: MUL.8h V0, V31, V0
38: MUL.8h V1, V31, V1
39: MUL.8h V2, V31, V2
40: MUL.8h V3, V31, V3
41: MUL.8h V4, V31, V4
42: MUL.8h V5, V31, V5
43: MUL.8h V6, V31, V6
44: MUL.8h V7, V31, V7
45: MUL.8h V8, V31, V8
46: MUL.8h V9, V31, V9

47: ADD.8h V10, V10, V0
48: ADD.8h V11, V11, V1
49: ADD.8h V12, V12, V2
50: ADD.8h V13, V13, V3
51: ADD.8h V14, V14, V4
52: ADD.8h V15, V15, V5
53: ADD.8h V16, V16, V6
54: ADD.8h V17, V17, V7
55: ADD.8h V18, V18, V8
56: ADD.8h V19, V19, V9

57: SUB X2, X2, #4095
58: SUB X2, X2, #4095
59: SUB X2, X2, #2048

60: SUB X1, X1, #4095
61: SUB X1, X1, #4095
62: SUB X1, X1, #4095
63: SUB X1, X1, #513

64: SUB X5, X5, #1
65: CBNZ X5, innerloop
```

The *Outerloop* returns *output* values and resets parameters for next 80 *output* values. Algorithm 5 shows the implementation of matrix-multiplication $A * s$ *outerloop*. In line 1-2, the first address pointer from `X2` on the stack is stored.

Algorithm 5 *Outerloop* of Matrix-multiplication $A * s$.

Input: *Output* address = [x0] A address = [x1], s address = [x2], Loop counter = [x4]
Output: 5120 *output* values

```
 1: STR, X2, [sp]
 2: MOV X4, #64
 3: MOV X5, #640
 4: MOVI V10.16b, #0
 5: MOVI V11.16b, #0
 6: MOVI V12.16b, #0
 7: MOVI V13.16b, #0
 8: MOVI V14.16b, #0
 9: MOVI V15.16b, #0
10: MOVI V16.16b, #0
11: MOVI V17.16b, #0
12: MOVI V18.16b, #0
13: MOVI V19.16b, #0
14: innerloop algorithm
15: LDR X2, [sp]
16: ADD X1, X1, #4095
17: ADD X1, X1, #4095
18: ADD X1, X1, #3330
19: ST1.8h V10-V13, [x0], #64
20: ST1.8h V14-V17, [x0], #64
21: ST1.8h V18-V19, [x0], #32
22: SUB X4, X4, #1
23: CBNZ X4, outerloop
```

And then the *outerloop* counter is set to 64, because single *innerloop* can be generated by 80 *output* values. In order to get 5,120 *output* values, 64 *innerloop* iterations are needed. In line 1-2, *outerloop* is not included. In line 3 *innerloop* counter is set to 640. In line 4-13, intermediate result registers are initialized to 0. In line 14, *innerloop* algorithm 4 is performed. In line 15, s value pointer is restored. In the end of round *innerloop*, next *innerloop* needs the $s[0]$ value at the first. The first s address is stored on the stack in line 1 of Algorithm 5. It is restored by calling it again from stack. In line 16-18, address pointers are reordered. In line 19-21, 80 *output* values are returned. In line 22-23, the loop counter is checked and the iteration is continued.

3.4 Matrix Multiplication: s * A

The matrix-multiplication of $s*A$ is similar to $A*s$, except the matrix form. The result of $s*A$ takes transpose of $A*s$ matrix form. It is shown in Figure 1 Below. The overall operation structure is the same as $A * s$. Every s value is required for the calculation and the *output* is identical to s values used in $A * s$. Table 5 shows each round of element, the each *output* value can be calculated by adding up all the multiplied results of Table 5 column. We can find the rule that $A[0]$, $A[1]$, ... $A[639]$ is multiplied by $s[0]$. $A[640]$, $A[641]$, ... $A[1279]$ is multiplied by $s[1]$. Thus, $A[i]$, $A[i+1]$, ... $A[i+639]$ is multiplied by $s[i/640]$ (i is a multiple of 640, maximum 408,960). This is different from $A*s$ where variables are linear in memory. The register scheduling is equivalent to $A * s$, but can be implemented more simply to load values to be used in operations. Algorithm 6 shows the implementation of *innerloop* for the matrix-multiplication $s * A$.

$$\texttt{Output[0]} = \texttt{A[0]} * \texttt{s[0]} + \texttt{A[640]} * \texttt{s[1]} + ... + \texttt{A[408960]} * \texttt{s[639]}$$
$$\texttt{Output[1]} = \texttt{A[1]} * \texttt{s[0]} + \texttt{A[641]} * \texttt{s[1]} + ... + \texttt{A[408961]} * \texttt{s[639]}$$
$$...$$
$$\texttt{Output}[k*640+i] = \texttt{A}[1*640+i] * \texttt{s}[k*640+0] + \texttt{A}[2*640+i] * \texttt{s}[k*640+1] + ... + \texttt{A}[639*640+639] * \texttt{s}[k*640+639]$$

The *innerloop* of matrix-multiplication $s * A$ can be generated 80 *output* values. In line 1-3, A values are loaded to `V0-V9`. In line 4, s values are loaded

Table 5. Elements of matrix-multiplication $s * A$, where $i = 0 \sim 639, k = 0 \sim 7, j = 0 \sim 649$.

	i = 0			i = 1		
	k = 0	k = 1	k = n	k = 0	k = 1	k = n
j = 0	A[0]*s[0]	A[0]*s[640]	A[0]*s[n*640+0]	A[1]*s[0]	A[1]*s[640]	A[1]*s[n*640+0]
j = 1	A[640]*s[1]	A[640]*s[641]	A[640]*s[n*640+1]	A[641]*s[1]	A[641]*s[641]	A[641]*s[n*640+1]
j = m	A[m*640]*s[m]	A[m*640]*s[640+m]	A[m*640]*s[n*640+m]	A[m*640+1]*s[m]	A[m*640+1]*s[640+m]	A[640+m]*s[n*640+m]

	i = 2			i = o		
	k = 0	k = 1	k = n	k = 0	k = 1	k = n
j = 0	A[2]*s[0]	A[2]*s[640]	A[2]*s[n*640+0]	A[o]*s[0]	A[o]*s[640]	A[o]*s[n*640]
j = 1	A[642]*s[1]	A[642]*s[641]	A[642]*s[n*640+1]	A[640+o]*s[1]	A[640+o]*s[641]	A[640+o]*s[n*640+1]
j = m	A[m*640+2]*s[m]	A[m*640+2]*s[640+m]	A[m*640+2]*s[n*640+m]	A[m*640+o]*s[m]	A[m*640+o]*s[640+m]	A[m*640+o]*s[n*640+m]

Algorithm 6 *Innerloop* of Matrix-multiplication $s * A$.

Input: A address = [x1], s address = [x2], Loop counter = [x5]
Output: 80 *output* values

```
1: LD1.8h V0-V3, [x1], #64
2: LD1.8h V4-V7, [x1], #64
3: LD1.8h V8-V9, [x1], #64
4: LD1R V31.8h, [x2], #2
5: MUL.8h V0, V31, V0
6: MUL.8h V1, V31, V1
7: MUL.8h V2, V31, V2
8: MUL.8h V3, V31, V3
9: MUL.8h V4, V31, V4
10: MUL.8h V5, V31, V5
11: MUL.8h V6, V31, V6
12: MUL.8h V7, V31, V7
13: MUL.8h V8, V31, V8
14: MUL.8h V9, V31, V9
15: ADD.8h V10, V10, V0
16: ADD.8h V11, V11, V1
17: ADD.8h V12, V12, V2
18: ADD.8h V13, V13, V3
19: ADD.8h V14, V14, V4
20: ADD.8h V15, V15, V5
21: ADD.8h V16, V16, V6
22: ADD.8h V17, V17, V7
23: ADD.8h V18, V18, V8
24: ADD.8h V19, V19, V9
25: ADD X1, X1, #1120
26: SUB X5, X5, #1
27: CBNZ X5, innerloop
```

to `V31`. For example, $A[0]$, $A[1]$, $A[2]$, $A[3]$, $A[4]$, $A[5]$, $A[6]$, and $A[7]$ values are loaded to `V0` vector register, and $s[0]$ value is stored to `V31` vector register then copied by 8 times. In line 5-24, it performs the matrix-multiplication and intermediate results are stored to `V10-V19`. These steps are identical to the matrix-multiplication $A * s$. In line 25, the A address pointer is moved. In line 26-27, the condition is checked and the *innerloop* is iterated depending on the condition.

Algorithm 7 shows *outerloop* of the matrix-multiplication $s * A$, it has 2 kind of loop point (A and B). In line 1-2, *output* is stored and A address pointer is stored to stack. In line 3, the number of A loop is defined. In line 4-5, the loop label A is set and loop count B is initialized. The loop label setting does not take the operating time. In line 6-7, loop label B is set and the number of *innerloop* iteration is initialized. In line 8-17, intermediate result registers are initialized to 0 value. In line 18, *innerloop* of algorithm 6 is performed. In line 19, A address pointer is restored. In line 20-22, *output* results are stored. In line 23, *output* address pointer is adjusted, because next loop generates $output[i + 640]$. The A pointer address is already increased by 80*2 in a post-increment mode at line 20-22. In line 23, the pointer increased to 620*2. In line 24-25, the loop-counter B is checked and it returned to the $LOOP\ POINT\ B$. In line 26-28, the s pointer is moved to $s[0]$. In line 29-31, the *output* address pointer is reordered and stored to the stack pointer. In line 32-33, the condition is checked and $LOOP\ POINT\ A$ is iterated.

Algorithm 7 *Outerloop* of Matrix-multiplication $s * A$.

Input: *Output* address = [x0] A address = [x1], s address = [x2], Loop counter A = [x3], Loop counter B = [x4]
Output: 5120 *output* values

```
 1: STR, X1, [sp], #16
 2: STR, X0, [sp], #-16
 3: MOV X3, #8

 4: SET LOOP POINT: A
 5: MOV X4, #8

 6: SET LOOP POINT: B
 7: MOV X5, #640

 8: MOVI V10.16b, #0
 9: MOVI V11.16b, #0
10: MOVI V12.16b, #0
11: MOVI V13.16b, #0
12: MOVI V14.16b, #0
13: MOVI V15.16b, #0
14: MOVI V16.16b, #0
15: MOVI V17.16b, #0
16: MOVI V18.16b, #0
17: MOVI V19.16b, #0

18: innerloop algorithm

19: LDR X1, [sp]

20: ST1.8h V10-V13, [x0],
    #64
21: ST1.8h V14-V17, [x0],
    #64
22: ST1.8h V18-V19, [x0],
    #32
23: ADD X0, X0, #1120

24: SUB X4, X4, #1
25: CBNZ X4, Move to LOOP
    POINT B

26: SUB X2, X2, #4095
27: SUB X2, X2, #4095
28: SUB X2, X2, #2050

29: LDR X0, [sp, #16]!
30: ADD X0, X0, #160
31: STR X0, [sp], #-16]!

32: SUB X3, X3, #1
33: CBNZ X3, Move to LOOP
    POINT A
```

3.5 Using high-speed AES accelerator

To generate pseudo-random matrix, FrodoKEM-640 scheme utilized AES-128 or SHAKE-128 algorithm. In particular, 64-bit ARMv8 processors support the AES accelerator. We utilized the built-in AES accelerator for AES-128 encryption. This replaced inefficient software based implementation to hardware-aided implementation. Finally, we achieved high-speed implementation of pseudo-random number generation based on AES-128 algorithm.

4 Evaluation

In this section, we evaluate the proposed implementation and previous reference C implementation (PQClean; representative post-quantum cryptography library in C language)[1], because to the best of our knowledge this is the first implementation of FrodoKEM on 64-bit ARMv8 processors. The implementation was developed through the Xcode 12.5 framework. Performance measurements were performed on Apple A10X Fusion processor (@2.38GHz) that is one of ARMv8 family. In addition, the performance measurement uses the time taken for 200 iterations of each algorithm with ms unit, and average of 50 iterations is used to reduce the deviation. Therefore, the total number of operation of the algorithm is 10,000 times.

The first performance comparison is the execution timing of matrix-multiplication. Performance results are shown in Table 6. Results show that matrix-multiplicatio-n operations ($A * s$ and $s * A$) in a parallel-way are 27.9× and 43.8× faster performance than previous implementation, respectively. The performance gap is the difference in number of outputs generated by 640 iterations. The previous implementation can be generated single output at once. On the

[1] `https://github.com/PQClean/PQClean`

Table 6. Comparison of matrix-multiplication in terms of execution timing (unit: ms).

Type	A * s	s * A
Reference C implementation [1]	2228.5	**80.0**
This work	2557.7	**58.4**

other hand, the proposed parallel matrix-multiplication can be calculated with 80 outputs at once. Therefore, the parallel matrix-multiplication goes through fewer iterations compared to the previous implementation. In addition, vector instructions have a better operation performance than scalar instructions.

Second, after applying the proposed method parallel matrix-multiplication and AES accelerator to FrodoKEM, the performance comparison is performed with the C only implementation FrodoKEM. Figure 2 shows the performance measurement and comparison results between C only implementation, parallel matrix-multiplication based implementation, AES accelerator based implementation, and implementation with all proposed options. Basically, the performance difference between each step (Key generation, Encapsulation, and Decapsulation) is not very noticeable. Therefore, only the performance comparison between the applied techniques is checked. As a result of comparing C only implementation and matrix-multiplication in a parallel-way, a performance difference of 1.25× occurs for Key generation, and 1.29× both for Encapsulation, Decapsulation. The performance difference between C only implementation and implementation with AES accelerator is 3.33×, 3.07×, and 3.07× for Key generation, Encapsulation, and Decapsulation, respectively. This is because the computational load occupied by pseudo-random generation in lattice-based cryptography is enormous. Consequently, the implementation applying both techniques has 10.22×, 9.88×, and 9.82× faster performance for Key generation, Encapsulation, and Decapsulation compared to the C only implementation, respectively.

5 Conclusion

In this paper, we introduced a implementation technique of parallel matrix-multiplication for FrodoKEM-640. The proposed method uses vector registers and vector instructions of ARMv8 processors. This calculated the matrix-multiplication in a parallel-way. As a result, the proposed parallel matrix-multiplication operations ($A*s$ and $s*A$) are overall 27.9× and 43.8× faster than C only implementation on A10x Fusion processor, respectively. When applied to FrodoKEM-640, the performance of FrodoKEM is improved by 1.25× for Key generation and 1.299× for Encapsulation and Decapsulation, than C only implementation on same processor. In addition, FrodoKEM with AES accelerator has 3.33× performance compared to C only implementation in best case. When both techniques were applied, the performance enhancement is up to 10.22× that of the C only implementation.

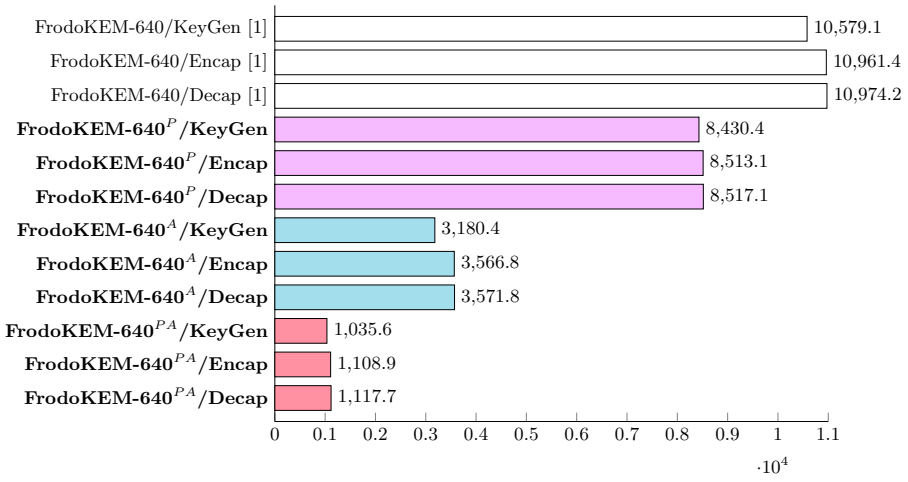

Fig. 2. Performance comparison on ARMv8 processors (unit: ms).

5.1 Future Works

This paper shows only result for FrodoKEM-640. In the future, we will apply proposed techniques to FrodoKEM-976 and FrodoKEM-1344, and measure the performance improvement. We will try a matrix-multiplication in parallel-way using a different processor (i.e. RISC-V) as well.

References

1. J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "Frodo: Take off the ring! practical, quantum-secure key exchange from LWE," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1006–1018, 2016.
2. N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C. A. Melchor, *et al.*, "BIKE: bit flipping key encapsulation," 2017.
3. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "Hamming quasi-cyclic (HQC)," *NIST PQC Round*, vol. 2, pp. 4–13, 2018.
4. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal, "NTRU prime: reducing attack surface at low cost," in *International Conference on Selected Areas in Cryptography*, pp. 235–260, Springer, 2017.
5. R. Azarderakhsh, M. Campagna, C. Costello, L. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, *et al.*, "Supersingular isogeny key encapsulation," *Submission to the NIST Post-Quantum Standardization project*, 2017.
6. C. P. Gouvêa and J. López, "Implementing GCM on ARMv8," in *Cryptographers' Track at the RSA Conference*, pp. 167–180, Springer, 2015.
7. 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,"

8. D. T. Nguyen and K. Gaj, "Optimized software implementations of crystals-kyber, ntru, and saber using neon-based special instructions of armv8," 2021.
9. A. Jalali, R. Azarderakhsh, M. M. Kermani, M. Campagna, and D. Jao, "ARMv8 SIKE: Optimized supersingular isogeny key encapsulation on ARMv8 processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 11, pp. 4209–4218, 2019.