Parallel Implementation of Number Theoretic Transform

Hwajeong Seo[1], Zhe Liu[2], Yasuyuki Nogami[3], Jongseok Choi[1], Taehwan Park[1], 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
{hwajeong,jschoi85,pth5804,howonkim}@pusan.ac.kr
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu}@uni.lu
[3] Okayama University,
Graduate School of Natural Science and Technology,
3-1-1, Tsushima-naka, Kita, Okayama, 700-8530, Japan
{yasuyuki.nogami}@okayama-u.ac.jp

Abstract. Number Theoretic Transform (NTT) based polynomial multiplication is the most important operation for Lattice-based cryptography. In this paper, we implement the parallel NTT computation over ARM-NEON architecture. Our contributions include the following optimizations: (1) we vectorized the Iterative Number Theoretic Transform, (2) we propose the 32-bit wise Shifting-Addition-Multiplication-Subtraction-Subtraction (SAMS2) techniques for speeding up the modular coefficient multiplication, (3) we exploit the incomplete arithmetic for representing the coefficient to ensure the constant time modular reduction. For medium-term security level, our optimized NTT implementation requires only $27,160$ clock cycles. Similarly for long-term security level, it takes $62,160$ clock cycles. These results are faster than the state-of-art sequential implementations by 31% and 34% respectively.

Keywords: Ring learning with errors (Ring-LWE), software implementation, public-key encryption, SIMD, Number Theoretic Transform (NTT), ARM-NEON

1 Introduction

Today's widely used public-key cryptosystems such as RSA and Elliptic Curve Cryptography are mainly based on integer factorization and discrete logarithm problems. However, these hard problems can be solved by using Shor's algorithm [15] with a quantum computer. Lattice-based cryptography is considered as a

⋆ Corresponding Author

promising candidate for post-quantum cryptosystems. Its security lies in the worst-case computational assumptions in lattices that remain hard even for quantum computers.

The introduction of learning with errors (LWE) problem [13] and its ring variant (ring-LWE) [10] provides an efficient way to build lattice based public key cryptosystems. The first practical evaluations of LWE and ring-LWE based encryption schemes were presented by Göttert et al. in CHES'12 [7]. In the experimental results, the ring-LWE based encryption scheme is faster by at least a factor of four and requires less memory in comparison to the encryption scheme based on the standard LWE problem. The following hardware or software implementations [11, 5, 3, 2, 12, 9] of ring-LWE based public-key encryption or digital signature schemes reduced clock cycles and memory requirements. Oder et al. in [11] presented an efficient implementation of Bimodal Lattice Signature Schemes (BLISS) on a 32-bit ARM Cortex-M4F microcontroller. The most optimal variant of their implementation cost $6M$ cycles for signing, $1M$ cycles for verification and $368M$ cycles for key generation, respectively, at a medium-term security level. In DATE'15, De Clercq et al. in [5] implemented ring-LWE encryption scheme on the identical ARM processors, their implementation required $121K$ cycles per encryption and $43.3K$ cycles per decryption at medium-term security level while $261K$ cycles per encryption and roughly $96.5K$ cycles per decryption for long-term security level. The first time when a lattice-based cryptographic scheme was implemented on an 8-bit processor belonged to Boorghany et al. in [3, 2]. The authors evaluated four lattice-based authentication protocols on both 8-bit AVR and 32-bit ARM processors. In particular, for 8-bit AVR implementation, their implementation needed $754,668$ cycles and $2,207,787$ cycles for Fast Fourier Transform (FFT) transform at medium-term and long-term security levels, respectively. For 32-bit ARM implementation, $109,306$ and $260,521$ cycles are required for medium and long-term security levels. Recently, Pöppelmann et al. [12] studied and compared implementations of Ring-LWE encryption and the Bimodal Lattice Signature Scheme (BLISS) on an 8-bit Atmel ATxmega128 microcontroller. For medium-term security level, they achieved $1,314,977$ cycles and $381,254$ for ring-LWE encryption and decryption operations, respectively. In CHES'15, Zhe et al. presents the high speed and memory optimized ring- LWE results [9]. The work introduces MOV-and-ADD technique for coefficient multiplication and Shifting-Addition-Multiplication-Subtraction-Subtraction (SMAS2) approach for reduction operation. Furthermore, they exploit the incomplete arithmetic [16] for representing the coefficients and perform the reduction operation in a lazy fashion. For medium-term security level, the former one only requires $590K$, $666K$ and $299K$ clock cycles for key- generation, encryption and decryption, respectively. Similarly for long-term security level, the key-generation, encryption and decryption take $2.3M$, $2.7M$ and $700K$ clock cycles, respectively.

1.1 Research Contributions

This paper continues the line of research on the efficient implementation of the Number Theoretic Transform (NTT) on an ARM-NEON processor. The core contributions are several optimizations to reduce the execution time in parallel fashion. More specifically, our contributions are listed as follows:

1. The efficiency of coefficient modular multiplication is a pre-requisite for high-speed NTT operation. We propose the parallel coefficient multiplication for Iterative NTT procedures. The method aims at computing multiple multiplication in a parallel manner.
2. The modular reduction is the most time consuming operation. We apply the 32-bit wise Shifting-Addition-Multiplication- Subtraction-Subtraction (SMAS2) approach for reduction operation. The approach replaced the expensive `MUL` operation into cheaper shifting and `ADD` operations.
3. We exploit the incomplete arithmetic [16, 9] for representing the coefficients and perform the reduction operation in a lazy fashion. This technique ensures constant time solution which is strong against side channel attacks.

Based on the above optimization techniques, we present high speed and constant time implementations of NTT for both medium-term and long-term security levels on ARM-NEON processors. For medium-term security level, it only requires $27,160$ clock cycles. For long-term security level, it takes $62,160$ clock cycles. The rest of this paper is organized as follows. In the next section, we review the background of NTT. In Section 3, we introduce the optimization techniques for NTT on ARM-NEON processors. In particular, we propose several optimization techniques to reduce the execution time in SIMD architecture. In Section 4, we report the implementation results and compare with the state-of-the-art NTT implementations. Finally, we draw our conclusions in Section 5.

2 Number Theoretic Transform

Our implementation adopts the Number Theoretic Transform (NTT) for performing the polynomial multiplication. An NTT can be seen as a variant of Fast Fourier Transform (FFT) but performs in a finite ring $\mathbb{Z}_q$. Instead of using the complex roots of unity, NTT evaluates a polynomial multiplication $a(x) = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Z}_q$ in the n-th roots of unity ω_n^i for $i = 0, \ldots, n-1$, where ω_n denotes a primitive n-th root of unity. Algorithm 1 shows the iterative version of NTT algorithm, which is originally from Cormen et al. in [4]. As shown in Algorithm 1, the iterative NTT algorithm consists of three nested loops. The outermost loop (i-loop, line $2 \sim 11$) starts from $i = 2$ and increases by doubling i, and the loop stops when $i = n$, thus it has only $log_2 n$ iterations. In each iteration, the value of twiddle factor ω_i are computed by executing a power operation $\omega_i = \omega_n^{n/i}$, and the value of ω is initialized by 1. Compared to i-loop, the j-loop (line $4 \sim 10$) executes more iterations, the number of iteration can be seen as

Algorithm 1 Iterative Number Theoretic Transform

Require: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$ and n-th primitive $\omega \in \mathbb{Z}_q$ of unity
Ensure: Polynomial $a(x) = NTT(a) \in \mathbb{Z}_q[x]$
1: $a = BitReverse(a)$
2: **for** i from 2 by $i = 2i$ to n **do**
3: $\omega_i = \omega_n^{n/i}$, $\omega = 1$
4: **for** j from 0 by 1 to $i/2 - 1$ **do**
5: **for** k from 0 by i to $n - 1$ **do**
6: $U = a[k + j]$
7: $V = \omega \cdot a[k + j + i/2]$
8: $a[k + j] = U + V$
9: $a[k + j + i/2] = U - V$
10: $\omega = \omega \cdot \omega_i$
11: **return** a

a sum of a geometric progression for 2^i where i starts from 0 and has a maximum value of $log_2(n-1)$, thus, the j-loop has $n-1$ iterations. In each iteration of j-loop, the twiddle factor ω is updated by performing a coefficient modular multiplication in line 10. Apparently, the innermost loop (k-loop, line $5 \sim 9$) occupies most part of the execution time of NTT algorithm since it is executed roughly $\frac{n}{2} log_2 n$ times. In each iteration of the innermost loop (line $6 \sim 9$), two coefficients $a[i+j]$ and $a[i+j+i/2]$ are loaded from memory into registers, and then $a[i + j + i/2]$ are multiplied by the twiddle factor ω, after that, the value of $a[k + j]$ and $a[k + j + i/2]$ are updated and stored in the memory.

In NTT computations, the majority of the processor's clock cycles are spent on modular operations. The straight-forward method of evaluating modular is to perform an integer division. However, constrained devices often do not have any dedicated hardware to divide the variables which generates a bulk of code containing loops, multiplications and additions. For this reason, the optimal modular implementation is the important consideration for high-speed NTT implementations. In [3, 2], Boorghany et al. introduces the technique that calculates the approximation of $\lfloor a/q \rfloor$ and then executes the $q = a - q \times \lfloor a/q \rfloor$. The results show acceptable approximation of modular operation. The following works by [12, 9] applied the approximation techniques to 8-bit AVR processor in assembly level. The technique optimizes the number of addition and shift operations by taking advantages of 8-bit word and instruction sets. Furthermore, it only utilizes the temporal registers without saving which avoids push/pop operations before/after function call. In [5], De Clercq et al. presented memory efficient polynomial multiplication. When we store single 13-bit coefficient variable into single 32-bit word in ARM processor, we cannot utilize the remaining 19-bit. The author stores two coefficients into single word. This approach can reduce the number of memory load and store instructions into half cases.

3 Optimization Techniques for NTT Computation

In this section, we describe several optimization techniques to reduce the execution time of NTT on ARM-NEON architectures. Our implementation adopts the parameter sets (n, q, σ) with $(256, 7681, 11.31/\sqrt{2\pi})$ and $(512, 12289, 12.18/\sqrt{2\pi})$ for security level of 128-bit and 256-bit, respectively. These parameter sets were also used in most of the previous hardware implementations, e.g., [7, 14] and software implementations, e.g., [3, 2, 5, 12, 9]. This also helps us to compare our work with previous works.

3.1 Vectorized Iterative Number Theoretic Transform

Sequential order of NTT operation described in Algorithm 1 executes the single coefficient multiplication with the single instruction in every round. However, proposed vectorized form of NTT operation computes the multiple number of coefficient multiplications with single SIMD instruction. We divide proposed NTT largely into two parts including sequential and parallel ways. When the number of consecutive coefficient multiplication satisfies the width of SIMD, we can execute the SIMD based vectorized computations. When the number of consecutive coefficient multiplication is smaller than width of SIMD, we simply adopt the sequential fashion. The detailed descriptions are available in Algorithm 2. The Steps from 3 to 12 conduct the sequential way of NTT, because offsets (the number of consecutive coefficient multiplication) between two coefficients ($a[k+j]$, $a[k+j+i/2]$) are only 1 and 2 for $i = 2$ and $i = 4$ cases, respectively. Since the other cases ($i > 4$) have at least four consecutive coefficient multiplication operations, we readily execute the parallel operation.

Firstly, we conduct whole twiddle factors (ω) in consecutive array form throughout the Steps 14 $\sim$ 16. Since the twiddle factors are fixed variables, we compute the values in off-line. In Steps 19 $\sim$ 24, the coefficient variables are loaded into registers in consecutive array form such as U_{array}, V_{array} and ω_{array}. We conduct the four different modular multiplications with $\omega_{array}[p : p+3] \cdot a[k+j+i/2 : k+j+3+i/2]$. After then pointer p increases by 4 (SIMD width). Lastly the multiple number of coefficient variables are added and subtracted each other.

3.2 Parallel Coefficient Multiplication

The coefficient multiplication is one of the most expensive operations of NTT computation, since each NTT computation requires $\frac{n}{2} log_2 n$ coefficient multiplications. In our implementation, the coefficient is at most 13-bit long, which can be kept in one 32-bit register. Even though we can store two coefficients into one register, we only store one coefficient in a register, because 13-bit wise multiplication outputs at most 26-bit length which introduces another post processing to extract the 13-bit out of 26-bit. The 128-bit ARM-NEON register can contain four 32-bit wise variables. We loaded four different aligned consecutive variables and then conduct the four different multiplications with single operation.

Algorithm 2 Vectorized Iterative Number Theoretic Transform

Require: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$ and n-th primitive $\omega \in \mathbb{Z}_q$ of unity
Ensure: Polynomial $a(x) = NTT(a) \in \mathbb{Z}_q[x]$

1: $a = BitReverse(a)$ {LUT based BitReverse; Section 3.6}
2: **for** i from 2 by $i = 2i$ to n **do**
3: $\omega_i = \omega_n^{n/i}$, $\omega = 1$ {LUT for twiddle factors; Section 3.5}
4: **if** $i = 2$ or $i = 4$ **then**
5: **for** j from 0 by 1 to $i/2 - 1$ **do**
6: **for** k from 0 by i to $n - 1$ **do**
7: $U = a[k + j]$ {sequential computation}
8: $V = \omega \cdot a[k + j + i/2]$
9: $a[k + j] = U + V$
10: $a[k + j + i/2] = U - V$
11: $\omega = \omega \cdot \omega_i$
12: **else**
13: $\omega_{array}[0] = \omega$
14: **for** p from 1 by 1 to $i/2 - 1$ **do**
15: $\omega = \omega \cdot \omega_i$, $\omega_{array}[p] = \omega$ {multiple computations of ω}
16: **for** k from 0 by i to $n - 1$ **do**
17: $p = 0$
18: **for** j from 0 by 4 to $i/2 - 1$ **do**
19: $U_{array} = a[k + j : k + j + 3]$ {parallel computation}
20: $V_{array} = \omega_{array}[p : p + 3] \cdot a[k + j + i/2 : k + j + 3 + i/2]$ {Section 3.2}
21: $p = p + 4$
22: $a[k + j : k + j + 3] = U_{array} + V_{array}$
23: $a[k + j + i/2 : k + j + 3 + i/2] = U_{array} - V_{array}$
24: **return** a

3.3 Fast Reduction

In the NTT computation, the majority of the execution time is spent on computing reduction operation since it is performed in the innermost k-loop. Thus, fast reduction operation is an essential for high-speed implementation of NTT algorithm. Our implementation chooses the prime modulus $q = 7681$ (i.e. `0x1e01` in hexadecimal representation) and $q = 12289$ (i.e. `0x3001` in hexadecimal representation).

We propose an optimized 32-bit wise SMAS2 reduction technique for performing the mod 7681 operation. The first SMAS2 method is introduced in [9] and the method is highly optimized in 8-bit AVR processors in terms of register utilization and the number of operations. However, ARM-NEON processor has two distinguished features over 8-bit AVR. First the processor provide 32-bit word size. We can readily compute the 13-bit wise multiplication in single instruction and up-to 31-bit shift is available within single cycle. Second multiple number of operations are conducted at once by exploiting SIMD instructions. With these features in mind, we redesign the original SMAS2 for ARM-NEON architecture.

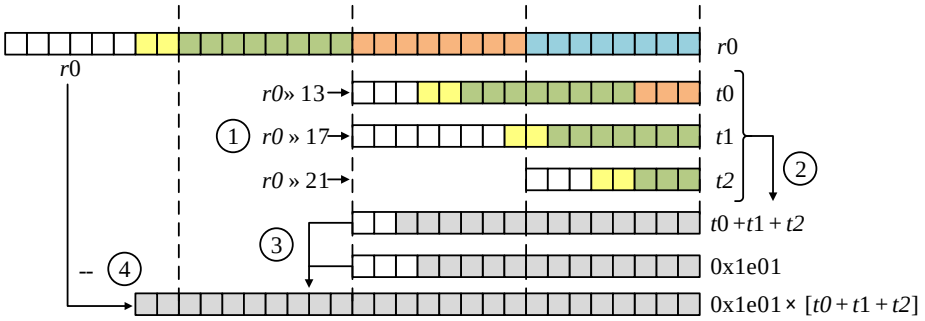

Fig. 1: Fast reduction operation with 32-bit wise SMAS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication; ④: subtraction.

This main idea of SMAS2 is to first estimate the quotient of $t = \frac{a}{q}$, and then perform the subtraction $a - t \cdot q$ where the value of t is $(a \gg 13) + (a \gg 17) + (a \gg 21)$. The reduction process consists of four different basic operations, namely, 32-bit wise Shifting → Addition → Multiplication → Subtraction → Subtraction (SAMS2). As shown in Figure 1, we keep the product in 32-bit long register ($r0$, a quarter of NEON register). The colorful parts mean that the storage has been occupied while the white part is not. The reduction with 7681 using SAMS2 approach can be performed as follows:

1. Shifting. We right shift $r0$ by 13-bit, 17-bit and 21-bit. This outputs results $t0$, $t1$ and $t2$.
2. Addition. We then perform the addition of $t0 + t1 + t2$.
3. Multiplication. The third step is to multiply the constant `0x1e01` by $(t0 + t1 + t2)$, which is a 16×13-bit multiplication.
4. Subtraction. We subtract the product obtained from Step 3 from $r0$.
5. Subtraction. However, the result we get in step 4 may still be larger than $p = 7681$, thus, we do the correction by subtracting the modulus p once.

3.4 Modular Reduction Operations in Constant Time

The coefficient multiplication needs to conduct final subtraction when the result is larger than $p = 7681$. In cases of addition $r = a + b \bmod p$ and subtraction $r = a - b \bmod p$, it also needs one or two times of subtraction and addition with the prime p. The intermediate result is kept in the range of $[0, p]$. Inspired by the incomplete modular arithmetic [16], our implementation does not perform an exact comparison between r and p, but rather tolerate an incompletely reduced coefficient $r \in [0, 2^{\lceil log_2 p \rceil}]$. Taking $p = 7681$ as an example, the incomplete coefficient addition works as follows. We first perform a normal coefficient addition, after that, we conduct the 13-bit shift to the right and perform the modular reduction by multiplying the modulus with the shifted results. For incomplete

Algorithm 3 Pseudo codes of vectorized NTT computation

Require: Eight 32-bit coefficients $A[0:3]$(q2), $B[0:3]$(q3), ω(q1), modulo(q0).
Ensure: Eight 32-bit results C(q5,q10).

```
 1: vmul.i32 q3, q3, q1                          {coefficient multiplication}
 2: vshr.u32 q4, q3, #13
 3: vshr.u32 q5, q3, #17
 4: vshr.u32 q6, q3, #21
 5: vadd.i32 q4, q4, q5
 6: vadd.i32 q4, q4, q6
 7: vmls.i32 q3, q4, d0[0]
 8: vshr.u32 q4, q3, #13
 9: vmls.i32 q3, q4, d0[0]
10: vadd.i32 q5, q2, q3                             {coefficient addition}
11: vshr.u32 q4, q5, #13
12: vmls.i32 q5, q4, d0[0]
13: vshl.i32 q1, q0, #2                          {coefficient subtraction}
14: vadd.i32 q2, q2, q1
15: vsub.i32 q10, q2, q3
16: vshr.u32 q14, q10, #13
17: vmls.i32 q10, q14, d0[0]
```

coefficient subtraction, We first perform a normal coefficient subtraction, after that, we add $4 \times p$ and then conduct the 13-bit shift to the right and perform the modular reduction by multiplying the modulus with the shifted results. This approach replaces the subtraction into addition which avoids the negative cases. In the very last outermost iteration of NTT, a correction process is performed to bring the final result back into the range $[0, p]$. Our practical results show this approach ensures constant time solution which is secure against side channel attacks.

In Algorithm 3, pseudo codes for vectorized NTT computation with constant time reduction is described. Firstly four coefficients (q3) and four twiddle factors (q1) are multiplied in Step 1. From Steps 2 $\sim$ 6, the intermediate results are shifted to right by 13, 17 and 21-bit and accumulated. In Step 7, we conduct multiplication with modulo (d0[0]) and intermediate result (q4). This process is readily available by using vmls instruction, which conducts four different multiplication and then subtract operations from the destination (q3). From Steps 8 $\sim$ 9, results over 13-bit are shifted and then reduced once again. In case of coefficient addition, two operands (q2 and q3) are added and then one time of reduction is follows in Steps 10 $\sim$ 12. For subtraction, we firstly calculate the value (4$\times$modulus) in Step 13. After then the value is added to operand (q2). Since the operand (q3) is placed within $[0, 2^{\lceil log_2 p \rceil}]$, the subtraction in Step 15 does not introduce negative values. Conveniently we can conduct one time of reduction that is same with addition case.

Algorithm 4 Look-up tabled based bit-reverse

Require: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$
Ensure: A bit-reversed polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$
1: **for** i from 0 by 1 to cnt **do**
2: $temp = a[in\_idx]$
3: $a[in\_idx] = a[out\_idx]$
4: $a[out\_idx] = temp$
5: **return** a

3.5 Look-Up Table for the Twiddle Factors

In each iteration of the i-loop, a new twiddle factor ω (line 3 of Algorithm 1) is computed by performing a modular multiplication. The total number of times a new ω is computed in an NTT operation is n. In each iteration of the j-loop, the twiddle factor ω is computed as shown in line 10 of Algorithm 1. A straightforward computation of $\omega = \omega \cdot \omega_i$ on-the-fly needs to perform $n-1$ times of coefficient modular multiplications. Both of the computations of the power of ω_n in i-loop and twiddle factor $\omega = \omega \cdot \omega_i$ in j-loop can be considered as fixed costs. We can pre-compute the all twiddle factors ω into RAM which is similar to the technique used in [9]. Fortunately, ARM-NEON process provides huge RAM size ($1 \sim 4$GB) and the storing all the intermediate twiddle factors ω into RAM is very cheap approach. We only need to transfer the twiddle factor that is required for the current iteration. For vectorized operation, whole twiddle factors are stored in aligned vector form which ensures efficient memory access pattern and vector operations as well.

3.6 Look-Up Table for the Bit Reverse

Before NTT operation (line 1 of Algorithm 1), coefficient variables are required to conduct proper shuffling. The bit-reverse converts the variables in reverse index. Since the reverse index is fixed value, we can conduct the bit-reverse with fixed index variables. The Algorithm 4 shows that bit-reverse process. By the number of loop (cnt), the coefficient a is properly mixed by following the input index (in_idx) and output index (out_idx). The detailed parameters are available in Appendix A.

4 Performance Evaluation and Comparison

4.1 Experimental Platform

The ARM Cortex-A9 is full implementations of the ARMv7 architecture including NEON engine. Register sizes are 64-bit and 128-bit for double(`d`) and quadruple(`q`) word registers, respectively. Each register provides short bit size computations such as 8-bit, 16-bit, 32-bit and 64-bit. This feature provides more precise operation and benefits to various word size computations. In particular,

the main structure of NTT and interface are written in C while the modular operations are implemented in Assembly Language. We complied our implementation with speed optimization option -O3. In order to obtain accurate timings, we ran each operation at least 1000 times and calculated the average cycle count for one operation.

Table 1: Performance comparison of software implementation of Number Theoretic Transform on different processors.

Implementations	NTT/FFT
32-bit ARM-NEON processors, e.g., Cortex-A9:	
Previous work [5, 9] (256)	39,480
This work (256)	**27,160**
Previous work [5, 9] (512)	95,200
This work (512)	**62,160**
32-bit ARM processors, e.g., Cortex-M4F, ARM7TDMI:	
Boorghany et al. [3] (256)	109,306
DeClercq et al. [5] (256)	**31,583**
Boorghany et al. [3] (512)	260,521
Oder et al. [11] (512)	122,619
DeClercq et al. [5] (512)	**71,090**
8-bit AVR processors, e.g., ATxmega64, ATxmega128:	
Boorghany et al. [2] (256)	1,216,000
Boorghany et al. [3] (256)	754,668
Pöppelmann et al. [12] (256)	334,646
Zhe et al. [9] (256)	**193,731**
Boorghany et al. [3] (512)	2,207,787
Pöppelmann et al. [12] (512)	855,595
Zhe et al. [9] (512)	**441,572**

4.2 Experimental Results

Table 1 summarizes the execution time of Number Theoretic Transform for medium-term and long-term security levels. The parallel NTT operations only require $27,910$ and $62,160$ clock cycles for medium-term and long-term security levels. Furthermore, our NTT implementation is computed within constant time, which ensures secure against side channel attacks.

Table 1 also compares software implementations of Number Theoretic Transform on different processors. For the 8-bit AVR and 32-bit platforms, the previous works [3, 2, 12, 5, 11] and our implementations adopt the same parameter sets as we mentioned in Section 3. The most suitable comparison is 32-bit ARM implementations, since the target processor shares similar ARM instructions of

ARMv7. By comparing the both sequential and parallel implementations, we can point out the 11.4 % enhancements with NEON engine. For better comparison, we implemented the state-of-art techniques, which include SMAS2 method and Negative-Wrapped Iterative Forward NTT in sequential fashion [5, 9], over identical ARM-NEON platform. The previous state-of-art implementations show that 39,480 and 95,200 cycles are required for medium-term and long-term security levels, respectively. Finally we can draw the conclusion that our proposed parallel NTT implementations outperform the previous methods by 31% and 34% for medium-term and long-term security levels, respectively.

5 Conclusion

This paper presented parallel implementations of Number Theoretic Transform on ARM-NEON platform. We proposed three optimizations to accelerate the execution time for the NTT-based polynomial multiplication. A combination of these optimizations results in a efficient NTT computation, which is faster than the previous best implementation techniques by 31% and 34% for medium-term and long-term security levels, respectively. Our future works are implementing full scales of Ring-LWE encryption scheme with our proposed NTT techniques and further researches on KY sampler over SIMD architectures.

References

1. A. Boorghany and R. Jalili. Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers. *IACR Cryptology ePrint Archive*, 2014:78, 2014.
2. A. Boorghany, S. B. Sarmadi, and R. Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):42, 2015.
3. T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.
4. R. De Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-lwe encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 339–344. EDA Consortium, 2015.
5. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 512–529. Springer, 2012.
6. Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors.
7. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43, 2013.
8. T. Oder, T. Pöppelmann, and T. Güneysu. Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.
9. T. Pöppelmann, T. Oder, and T. Güneysu. Speed records for ideal lattice-based cryptography on avr.

10. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.
11. S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 371–391. Springer, 2014.
12. P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.
13. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2):46–52, Mar. 2002.

A Index Parameters of 256-bit Bit Reverse

Table 2: Index Parameters of 256-bit Bit Reverse

256-bit Input Index
128, 64, 192, 32, 160, 96, 224, 16, 144, 80, 208, 48, 176, 112, 240, 136, 72, 200, 40, 168, 104, 232, 152, 88, 216, 56, 184, 120, 248, 132, 68, 196, 164, 100, 228, 148, 84, 212, 52, 180, 116, 244, 140, 76, 204, 172, 108, 236, 156, 92, 220, 188, 124, 252, 130, 194, 162, 98, 226, 146, 82, 210, 178, 114, 242, 138, 202, 170, 106, 234, 154, 218, 186, 122, 250, 134, 198, 166, 230, 150, 214, 182, 118, 246, 142, 206, 174, 238, 158, 222, 190, 254, 193, 161, 225, 145, 209, 177, 241, 201, 169, 233, 217, 185, 249, 197, 229, 213, 181, 245, 205, 237, 221, 253, 227, 211, 243, 235, 251, 247
256-bit Output Index
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 49, 50, 51, 53, 54, 55, 57, 58, 59, 61, 62, 63, 65, 67, 69, 70, 71, 73, 74, 75, 77, 78, 79, 81, 83, 85, 86, 87, 89, 91, 93, 94, 95, 97, 99, 101, 103, 105, 107, 109, 110, 111, 113, 115, 117, 119, 121, 123, 125, 127, 131, 133, 135, 137, 139, 141, 143, 147, 149, 151, 155, 157, 159, 163, 167, 171, 173, 175, 179, 183, 187, 191, 199, 203, 207, 215, 223, 239