

Area-Efficient Accelerator for the Full NTRU-KEM Algorithm

Yongseok Lee[1], Kevin Nam[1], Youyeon Joo[1], Jeehwan Kim[1], Hyunyoung Oh[2(✉)], and Yunheung Paek[1(✉)]

[1] Seoul National University, Seoul, Republic of Korea
{yslee,kvnam,yyjoo,jhkim}@sor.snu.ac.kr, ypaek@snu.ac.kr
[2] Gachon University, Seongnam-si, Republic of Korea
hyoh@gachon.ac.kr

Abstract. Among Post Quantum Cryptography (PQC) algorithms, Nth-degree Truncated-polynomial Ring Units Key Encapsulation Mechanism (NTRU-KEM) emerged as a promising cryptosystem for key establishment. However, the algorithm requires more storage space and computation resources compared to classical cryptosystems, resulting in substantial memory and performance overheads. To mitigate these overheads, researchers have focused on enhancing the efficiency of the NTRU-KEM algorithm with dedicated hardware implementation, but have excluded the key generation function, resulting in a tenfold increase in latency when generating new keys frequently. In this paper, we aim to implement an efficient NTRU-KEM algorithm with full functionality by incorporating all functions, including key generation, using a hardware and software co-design approach. We strategically allocate functions based on their inherent parallelism and execution time to hardware or software. Our approach entails implementing hardware modules to be shared across multiple sub-functions as much as possible to achieve optimal performance improvement over hardware resources. Our implementation demonstrated a significant speedup compared to pure software implementation in the execution time of all three functions of NTRU-KEM, with a remarkable performance improvement in key generation. Our approach resulted in more than three times the area reduction effect compared to prior work focused only on encapsulation and decapsulation functions, and showed similar or better results of performance per area depending on the function.

Keywords: NTRU · Post Quantum Cryptography · Hardware Accelerator

1 Introduction

Recently, quantum computing has emerged as a prime interest due to its potential to providing significant benefits to society in virtue of its capability to accelerate workloads compared to classical computing. As lots of research on quantum computing show great development, however, security issues were raised

H. Oh and Y. Paek—Co-corresponding authors.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023 Workshops, LNCS 14106, pp. 186–201, 2023.
https://doi.org/10.1007/978-3-031-37111-0_14

since the conventional cryptosystems used in classical computing are not resistant to quantum computing. In particular, previous research has demonstrated that classical methods of key establishment [9,17] are susceptible to compromise through quantum computing [20]. As a counter measurement, new algorithms have been developed to withstand cryptoanalytic attacks facilitated by quantum computing. These algorithms are collectively referred to as Post Quantum Cryptography (PQC). Out of several PQC algorithms that facilitate key establishment, Nth-degree Truncated-polynomial Ring Units (NTRU) [11] has emerged as a particularly promising cryptosystem. Notably, since its introduction in 1996, NTRU has yet to be compromised [1]. In lieu of classical key establishment algorithms, NTRU employs a Key Encapsulation Mechanism (KEM), which utilizes an asymmetric key to encrypt a symmetric key (e.g., session key) to facilitate its transmission for key establishment.

Sadly, like other PQC algorithms, NTRU-KEM necessitates increased storage space and computation resources relative to classical cryptosystems. Consequently, the adoption of NTRU-KEM in current systems results in substantial memory and performance overheads. To mitigate these overheads, researchers have conducted studies [2,10,15] focused on enhancing the efficiency of the NTRU-KEM algorithm with regard to storage and performance through dedicated hardware implementation. These studies generally concentrate on the encapsulation and decapsulation functions, which represent two of the three functions comprising the NTRU-KEM algorithm. They exclude the third function, key generation (`KEYGEN`), claiming the key can be generated in advance and stored in memory for repeated uses. This approach, however, confines the implementation of NTRU-KEM to a long-term key scenario. To enable the utilization of NTRU-KEM in a session-key context, wherein new keys must be generated frequently, prior research has been compelled to rely on software to execute `KEYGEN`, which results in a tenfold increase in latency relative to encapsulation and decapsulation combined.

Based on our observation, we have opted to incorporate all functions, including `KEYGEN`, in our endeavor to implement an efficient NTRU-KEM algorithm with full functionality. To achieve this goal, we have adopted a hardware and software co-design methodology in this paper, a well-established platform utilized by a multitude of PQC algorithm accelerators [12,18]. Our decision to utilize this approach is based on its proven effectiveness in addressing the challenges encountered in implementing the NTRU-KEM algorithm, including the significant performance overheads associated with conventional software-based `KEYGEN` implementation. Through the adoption of a hardware and software co-design approach, we seek to enhance the efficiency of NTRU-KEM by simultaneously leveraging the strengths of both hardware and software components.

Our implementation aimed to achieve optimal performance improvement over hardware resources by strategically allocating functions based on both their inherent parallelism and execution time to hardware or software. Specifically, functions that exhibit high inherent parallelism and consume a large portion

of execution time were implemented into hardware to harness the full potential of available hardware resources and achieve significant performance gains. Conversely, functions that either predominantly involve sequential operations or consume a small portion of execution time were implemented into the software. This approach allows us to avoid expending unnecessary hardware resources that could have otherwise been allocated to other functions exhibiting the potential for achieving markedly enhanced levels of performance. We describe our implementation in detail in the following Sects. 3, 4.

To demonstrate the efficiency of our implementation, we targeted the hardware part of our design onto a 45 nm process technology with a clock frequency of 1 GHz. Meanwhile, the software part was targeted to a Cortex M4(STM32F407VG), operating at 168 MHz of clock frequency. In our experiment, our implementation demonstrated a speedup of 579× in the average execution time of all three functions of NTRU-KEM, as compared to a pure software implementation. In particular, our implementation of `KEYGEN` has demonstrated a remarkable performance improvement over software-only implementation, as evidenced by a factor of 1,393 times faster. This finding underscores the significant practical value of our approach, especially in situations where frequent key generation is required. Compared to prior work [2] that solely focused on encapsulation and decapsulation functions excluding `KEYGEN`, our implementation stands out as more efficient in terms of hardware resources. Specifically, ours shows an area reduction of 29,821 μm^2, while offering a performance per area that exceeds the prior by a factor of 1.5.

2 Background

This section provides a brief introduction to the fundamentals of NTRU-KEM algorithm, as well as lattice-based cryptography as the theoretical basis of NTRU.

2.1 Lattice-Based Cryptography

Lattice-based Cryptography refers to a cryptosystem constructed over mathematical grounds that involves lattices in the security proof. They are currently widely used for PQC and other cryptosystems, including NTRU. A lattice $\mathcal{L}(\mathbb{B})$ in an n-th dimension refers to the end-points of vectors that can be formed by linear combinations of a given basis $\mathbb{B} = (b_1, b_2, \ldots, b_n)$. The security guarantees of cryptosystems over lattice are bound to hardness from problems such as the Shortest Vector Problem or Closest Vector Problem. In several cryptosystems, such n-th dimension vectors are interpreted as polynomials bound in integer rings. Lattice-based encryption schemes are constructed by adding a random lattice point to a plaintext vector, and the resulting ciphertext is sent to the receiver, who decrypts it using their private key to find the lattice point closest to the ciphertext. Lattice-based cryptography is resistant to quantum attacks due to the hardness of underlying problems, even for quantum computers, which makes it advantageous over other public-key cryptosystems. Furthermore, the

KeyGen(*seed*)	Encapsulate(**h**)	Decapsulate((**f**, $\mathbf{f}_p$, $\mathbf{h}_q$, s), **h**)
1. ((**f**, $\mathbf{f}_p$, $\mathbf{h}_q$), **h**) ← KeyGen′(*seed*)	1. *coins* ←$_{\$}$ $\{0, 1\}^{256}$	1. (**r**, **m**, *fail*) ← Decrypt((**f**, $\mathbf{f}_p$, $\mathbf{h}_q$), **c**)
2. s ←$_{\$}$ $\{0, 1\}^{256}$	2. (**r**, **m**) ← Sample_rm(*coins*)	2. $k_1 \leftarrow H_1(\mathbf{r}, \mathbf{m})$
3. return ((**f**, $\mathbf{f}_p$, $\mathbf{h}_q$, s), **c**)	3. **c** ← Encrypt(**h**, (**r**, **m**))	3. $k_2 \leftarrow H_2(s, \mathbf{c})$
	4. $k \leftarrow H_1(\mathbf{r}, \mathbf{m})$	4. if $fail = 0$ return k_1
	5. return (**c**, k)	5. else return k_2

Fig. 1. The Three Functions of NTRU-KEM

efficiency properties of lattice-based cryptographic schemes make them suitable for resource-constrained environments like the Internet of Things (IoT).

2.2 NTRU Cryptosystem

NTRU is a latticed-based public-key cryptosystem that was first introduced in 1996 [11]. NTRU cryptosystem is based on operations between polynomials over integer quotient rings $R_q = Z_q[x]/(x^n - 1)$, $S_q = Z_q[x]/\Phi_n$, and $S_p = Z_p[x]/\Phi_n$, where n and p are prime integers and q is coprime with both p and n while $p << q$, $\Phi_1 = x - 1 \in \mathbb{Z}[x]$ and $\Phi_n = x^{n-1} + x^{n-2} + \cdots + x + 1 \in \mathbb{Z}[x]$. The coefficients of R_q and S_q are encoded in two's complement modulo q. The existing variants of NTRU [11] use $p = 3$, which leads rings and polynomials to have coefficients using 2 bits, -1, 0, or 1. Let Γ be the set of ternary polynomials in $\mathbb{Z}_3[x]/(x^n - 1)$ with at most $n - 2$ degrees of order. NTRU cryptosystem samples 4 polynomials, f, g, r, and m from Γ, which are used as will be described in Sect. 2.3.

2.3 NTRU-KEM Algorithm

NTRU-KEM consists of three functions, `KEYGEN`, Encapsulation, and Decapsulation. `KEYGEN` generates a key pair $K_{priv} = (f, f_p, h_q)$ and $K_{pub} = h$. The process starts by sampling the four polynomials and then computing $f_p = f^{-1}\ mod(3, \Phi_n)$ and $f_q = f^{-1}\ mod\ (q, \Phi_n)$. The publickey h is a polynomial in R_q calculated from $h = (3 \cdot g \cdot f_q)\ mod\ (q, x^n - 1)$. The last component of the private key, h_q, is a pre-computed polynomial $h_q = h^{-1}\ mod\ (q, \Phi_n)$. Using the polynomials above, encapsulation and decapsulation are computed as depicted in Fig. 1.

3 Design Choices

In this section, we present the design choices we have made to optimize the performance of the three main functions of NTRU-KEM, namely encapsulation, decapsulation, and `KEYGEN`. We have carefully considered various approaches to improve the efficiency of these functions, and we will elaborate on these design choices in detail.

3.1 Selecting Sub-functions for Acceleration

In the context of hardware acceleration, the process of choosing a specific sub-function to be optimized and sped up through hardware is an essential task that

Table 1. NTRU-KEM Profiling Results

Function	NTRU Variant							
	HPS2048509		HPS2048677		HPS4096821		HRSS701	
Key Generation	3489.32 ms	100%	6094.56 ms	100%	9164.47 ms	100%	6702.06 ms	100%
poly_Rq_inv	1663.96	45.92	2891.57	46.77	4317.10	46.71	3152.91	46.87
poly_R2_inv								
poly_R2_inv_to_Rq_inv								
poly_S3_inv	1564.79	43.19	2633.54	42.59	3961.83	42.86	2881.90	42.84
poly_Rq_mul	382.57	10.55	642.18	10.38	944.20	10.21	688.49	10.23
crypto_sort_int32	8.07	0.22	11.49	0.18	14.59	0.15	-	-
randombytes	3.56	0.09	3.63	0.05	4.03	0.04	3.36	0.04
Encapsulation	92.10 ms	100%	157.91 ms	100%	211.15 ms	100%	160.61 ms	100%
poly_Rq_mul	79.97	85.94	139.72	88.48	198.80	89.89	157.98	97.84
crypto_sort_int32	9.44	10.15	13.87	8.78	17.61	7.96	-	-
randombytes	3.34	3.59	3.84	2.43	4.15	1.88	3.08	1.91
etc	0.28	0.30	0.46	0.29	0.57	0.26	0.39	0.24
Decapsulation	233.21 ms	100%	418.95 ms	100%	603.89 ms	100%	452.73 ms	100%
poly_Rq_mul	76.03	32.48	136.59	32.53	200.52	33.21	146.80	32.52
poly_S3_mul	78.54	33.55	141.09	33.60	201.20	33.32	151.64	33.59
poly_Sq_mul	78.54	33.55	141.09	33.60	200.68	33.24	151.65	33.58
hash	0.92	0.39	1.06	0.25	1.30	0.21	1.31	0.29
etc	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00

needs to be done first. In order to identify the specific sub-functions that have the potential to significantly reduce the overall execution time of NTRU-KEM, a comprehensive profiling analysis was conducted on four distinct variants of NTRU-KEM with different parameter configurations. This process involved a detailed examination of the performance characteristics of each sub-function, with the aim of determining which ones may be optimized for improved efficiency. We used the NIST submission version of NTRU-KEM with a Cortex M4(STM32F407VG) CPU at 168MHz clock frequency.

Table 1 describes the result of the profiling. For each variant, we analyzed the runtime latency of each of the three main functions and all of their sub-functions that resides within them. For en/decapsulation, `poly_`R_q`_mul`, `poly_`S_3`_mul` and `poly_`S_q`_mul` accounts for between 85–99% of the total runtime. Accordingly, the aforementioned existing NTRU-KEM accelerators that only accelerated these two functions [2,10,15] aimed to boost the performance of polynomial multiplication. However, in the case of `KEYGEN`, another sub-function, `poly_`R_q`_inv` and `poly_`S_3`_inv`, accounts for the biggest portion over 90%, while `poly_`R_q`_mul` only accounts for 10%. Therefore, in order to accelerate the entire NTRU-KEM algorithm, `poly_`R_q`_inv` and `poly_`S_3`_inv` also needs to be taken into consideration when designing the hardware. It is worth to noting that all the `poly_*_inv` and `poly_*_mul` sub-functions inherently involve parallelism since the computation of each term in the resulting polynomial can be done independently of the others. In addition, `crypto_sort_int32` in encapsulation and `KEYGEN` is also

selected to implement in hardware since it can be processed in a parallel manner. Conversely, the sub-function named `randombytes` within encapsulation and `KEYGEN` function is predominantly comprised of sequential operations. Also, due to its relatively minor contribution to the overall execution time, it has not been implemented as hardware. To efficiently process the above selected two kinds of polynomial operations, the optimizations that we have devised will be presented in the following subsection.

3.2 FSM Oriented Hardware Design

As implied by its name, `poly_`R_q`_mul` includes multiplications between polynomials. The efficient computation of polynomial multiplications is achieved by employing the widely used Fast Fourier Transform (FFT) or Number Theoretic Transform (NTT) technique. These techniques are used in most cryptosystems based on polynomial rings and their hardware implementations [5,6,13,14,16,21]. However, NTRU does not use such operations for two reasons.

Firstly, the rings used in NTRU exhibit properties that are not directly supportive of FFT/NTT. The famous techniques to compute FFT/NTT [3,4] demand that the polynomials be of degrees that are a power of 2. However, NTRU variants are made up of polynomials of degrees that are prime numbers, denoted by n, thereby deviating from the power of two forms. In previous research [7], the polynomial was extended to enable the computation of FFT/NTT, albeit with an increase in the number of operations required by the technique. Secondly, the multiplication of polynomials in NTRU employs a predefined modulo of 3 and q, which can be efficiently computed using a sequence of operations. For instance, a polynomial with modulo 3 assumes values of only -1, 0, and 1 as coefficients. The computation between those coefficients can be processed via such as zeronization and change of sign bit rather than multiplication, which does not incur significant overhead as compared to operations between larger integers.

Based on our observation, we do not employ FFT/NTT technique to multiply NTRU polynomials. However, one drawback of FTT/NTT-less polynomial multiplications is the increase of conditional branches within the algorithm. Especially, the algorithmic structure of `poly_inv` includes a notable presence of conditional branches. This is attributed to the nature of finding an inverse polynomial, which entails a search for the correct match from a pool of candidates, necessitating multiple invoking of `swap` sub-functions.

In order to efficiently handle these conditional branches, our hardware implementation of NTRU leverages Finite State Machines (FSMs) for the computation of sub-functions. FSMs dedicated to each sub-function manage a pool of basic arithmetic units (i.e., multipliers and adders) to process internal operations as much as possible in parallel. The number of basic arithmetic units is carefully determined by our cycle-by-cycle analysis of a group of internal operations within sub-functions. The group of internal operations is based on the conditional branches within the sub-function, resulting in a more fine-grained

Table 2. Parameter Sets of NTRU-KEM

Parameter	NTRU Variant			
	HPS2048509	HPS2048677	HPS4096821	HRSS701
N	509	677	821	701
q	2048	2048	4096	8192

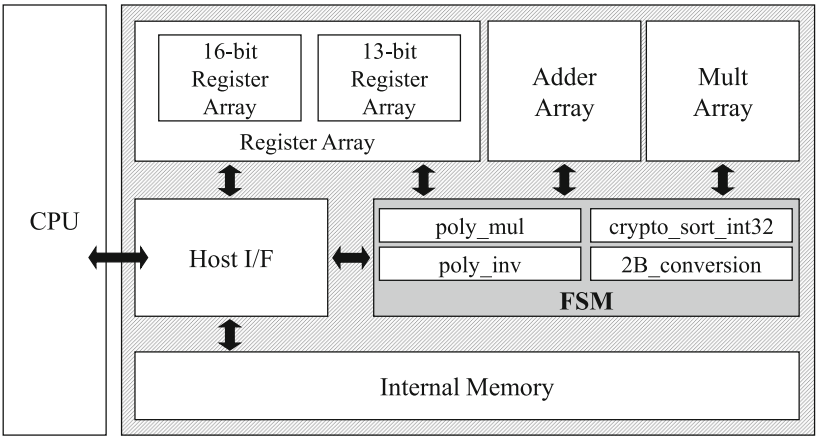

Fig. 2. Overview of Our Hardware Implementation

analysis. In cases where the inherent parallelism of the internal operations is less than the number of available units, all the needed arithmetic units are allocated to maximize parallel processing. Otherwise, although the sub-function cannot be processed as a full parallel, all the units in the pool are pipelined so that multiple operations can be processed in each cycle, thereby minimizing latency. Our FSM-oriented design has enabled us to achieve a balance between performance and hardware area, resulting in an efficient implementation.

4 Implementation

In this section, our implementation details of NTRU acceleration are described. Figure 2 overviews our hardware implementation. We use 16 instances for multipliers and adders with 13-bit and 16-bit integers as both inputs and outputs. In order to optimize the allocation of hardware resources, we decide to employ 13-bit and 16-bit units in a selective manner. The specifics of this implementation approach are explained in the following subsections. The intermediate values are stored in 28 internal registers, allowing for efficient handling of the sub-functions. Additionally, we implement 8 FSMs, with each FSM dedicated to performing one of the 8 sub-functions (`poly_`R_q`_mul`, `poly_`S_3`_mul`, `poly_`S_q`_mul`, `poly_`R_2`_inv_to_`R_q`_inv`, `crypto_sort_int32`, `poly_`R_2`_inv`, `poly_`S_3`_inv`, and `2B_conversion`).

4.1 Supporting All the NTRU Parameter Sets

As shown in Table 2, NTRU has four sets of parameters, each with different polynomial degrees n and modulus prime q. Each parameter set has different security strengths and complexities, so it can be selected and used as needed for various applications. Our implementation entails an accelerator that facilitates the concurrent support of all different parameter sets as a single instance, thereby enabling its versatile utilization for general purposes.

Initially, we determine the bit precision of each intermediate data, including the input and output of arithmetic units and internal registers. Since the prime bit q are different in each parameter set, the number of output bits in modular operations also varies, requiring rearrangement. For the HPS2048509 and HPS2048677 parameters, the q has a valid value in the lower 11 bits (2^{11}). For the HPS4096821 parameters, q has a valid value in 12 bits (2^{12}), and for the HRSS701 parameters, q has the largest valid value of 13 bits (2^{13}). Thus, it has been determined that a bit precision of 13 bits would suffice to represent the polynomial coefficients and its corresponding intermediate outcomes. However, during the execution of the encapsulation process, the sub-function `crypto_sort_int32` necessitates sorting 32-bit data. As this is the only sub-function that demands such a bit precision, we have selectively designed a subset of arithmetic units (2 out of 16 adders) and flexibly grouped a few 16-bit registers (16 out of 28) together to support the processing of 32-bit data, with the purpose of minimizing the hardware resources. Furthermore, it should be noted that `crypto_sort_int32` sub-function is required only in the remaining three of the four parameter sets apart from HRSS701. So, the selective execution of `crypto_sort_int32` is contingent on the parameter set in the HOST I/F module configured by CPU.

Additionally, the `2B_conversion` sub-function also requires 32-bit data input and shares the 16-bit registers used by the `crypto_sort_int32` sub-function. The `2B_conversion` sub-function converts and packs 32-bit data into 1–2 bit data in the R_2 or S_3 ring domain.

4.2 `poly_*_mul` and `poly_*_inv` Sub-Functions

In this subsection and §4.3, the implement details of sub-functions to be accelerated in hardware are explained. As listed up in Table 1, there are four different sub-functions associated with `poly` as a prefix in NTRU: `poly_`R_q`_mul`, `poly_`S_3`_mul`, `poly_`S_q`_mul`, and `poly_`R_2`_inv_to_`R_q`_inv`.

As represented by the notation inserted middle of the sub-function name, each is defined on the different ring domains. In principle, it is ideal for implementing different sub-functions as separate hardware modules since each represents distinct operations. However, upon analyzing low-level sub-operations, it can be found that polynomial multiplication is commonly invoked. In each sub-function, after polynomial multiplication, each newly computed coefficient of polynomials is converted to be compatible with the defined ring domain through modulo operation. The polynomial inverse sub-function additionally involves

Algorithm 1: Polynomial Ring Domain Conversion

Input: polynomial p; inverse polynomial p_inv
Output: Converted polynomial p';

```
Function poly_Rq-to-poly_S3(p):
    for i = 0 to n − 1 do
        if p[i] ≤ 1/2 then
            p[i] =MOD3(p'[i] + 2 ∗ p[n − 1] − 1);
        else
            p[i] =MOD3(p'[i] + 2 ∗ p[n − 1]);
    return p'

Function poly_Rq-to-poly_Sq(p):
    for i = 0 to n − 1 do
        p'[i] =MODQ(p[i] − p[n − 1]);
    return p'

Function poly_R2_inv-to-poly_Rq_inv(p,p_inv):
    for i = 0 to n − 1 do
        b[i] =MODQ(q − p[i]);
    for i = 0 to 1 do
        c = poly_Rq_mul(b, p_inv);
        c[0] = c[0] + 2;
        p_inv = poly_Rq_mul(c, p_inv);
    p' = p_inv;
    return p'
```

comparison and swapping operations. In light of these observations, we implement a common single module to perform polynomial multiplication for use in several sub-functions. Subsequently, the necessary ring domain conversions or remaining operations are carried out in the FSM performing each sub-function to complete the sub-function operation. This approach is feasible since only one of these sub-functions is executed at a time while encapsulation, decapsulation and `KEYGEN` function of NTRU-KEM are performed. As a result, `poly_*_mul`, `poly_*_inv` sub-functions are implemented using hardware resources efficiently.

To elaborate on the implementation detail, multiplication on the R_q ring domain, denoted as R_q`_mul` was selected as the common polynomial multiplication. This is because converting R_q to multiple required ring domains or vice versa was the most efficient. The ring domain conversion carried out within each FSM of `poly_*` sub-function is depicted in Algorithm 1. Algorithm 1 shows the three conversion processes that occur after R_q`_mul`. To convert from R_q domain to S_3 domain, we first need to convert the data with the range from $[0, q-1)$ to $[-q/2, q/2]$. We determine the criterion for performing modulus by checking if the polynomial coefficient is greater than $q/2$, and then perform `modulo` 3 operation to convert the data range to $[0, 2]$. To convert from R_q to S_q domain,

Algorithm 2: Almost-Inverse Algorithm (AIA) [19]

Input: polynomial $A(X)$, $M(X)$, where $A(X)$ is odd and coprime to $M(X)$;
Output: *almost-inverse* of $A(X)$ modulo $M(X)$;degree k

```
1  Initialize F = A(X), B = 1, C = 0, G = M(X), k = 0
2  while true do
3      while F is even do
4          F /= X;
5          C *= X;
6          k++;
7      if F == 1 then
8          return B, k // B is an almost-inverse of A(X) modulo M(X)
9      if deg(F) < deg(G) then
10         swap(F, G);
11         swap(B, C);
12     F += G;
13     B += C;
```

we subtract the polynomial coefficient of the highest degree and perform `modulo` q operation. To convert R_2`_inv` polynomial to R_q`_inv` polynomial, we need to convert data with the range of $[0, 1]$ to $[0, q-1)$. This process involves subtracting the R_2 domain polynomial from q and performing `modulo` q operation, and adding 2 a to polynomial coefficient of the lowest degree after R_q`_mul` sub-function. Using these conversion processes, we can efficiently reuse the R_q`_mul` sub-function in four sub-functions.

As mentioned earlier, the polynomial inverse sub-function involves many conditional statements and swap operations as well as polynomial multiplication. As described in Algorithm 2, NTRU adopts the almost inverse algorithm for efficiency, computing the almost inverse of a polynomial satisfying exact inverse result in all NTRU functions. The inverse process involves iterating through a swap operation, the highest degree with a non-zero coefficient in each polynomial. In NTRU key generation function, there are three required inverse sub-functions: R_2`_inv`, S_3`_inv`, and R_q`_inv`. As can be seen in Algorithm 1, R_q`_inv` can be performed through conversion from R_2`_inv`. So, we only need to implement two sub-functions, R_2`_inv` and S_3`_inv` to evaluate polynomial inverse sub-functions. We found that these two sub-functions exhibit almost the same data flow except for the type of internal computations. Thus, we implemented these two sub-functions into one `poly_inv` sub-function. This unified sub-function is configured by FSMs to operate differently for each case. When `poly_inv` is configured to operate as R_2`_inv`, bitwise OR operations for data with the range of $[0, 1]$ are executed. When `poly_inv` is configured to operate as S_3`_inv`, `modulo` 3 operations for data with the range of $[0, 2]$ are executed. Despite such differences in operations, sub-functions share a common data flow, and they could be effectively integrated into one module.

Table 3. Area of Our Implemented Hardware

	Area ($10^3\mu m^2$)	Percentage (%)
NTRU-KEM	**29.82**	**100**
REGISTER & INTERFACE	4.43	14.90
host I/F	1.37	4.60
register_array	2.65	8.90
mem I/F	0.41	1.40
FSMs	18.94	63.40
poly_mul	8.53	28.50
poly_inv	7.46	25.00
crypto_sort_int32	2.31	7.80
2B_conversion	0.64	2.10
ARITHMETIC UNITS	6.05	20.40
adder_array	1.51	5.10
mult_array	4.54	15.30

4.3 `crypto_sort_int32` Sub-Function

The sub-function `crypto_sort_int32` performs an operation that sorts the sampling data generated with random numbers at the beginning of the encapsulation function in ascending order. The operation is performed on $(n-1)$ data points. In contrast to other polynomial sub-functions, `crypto_sort_int32` sub-function operates on input and output data in units of 16-bit. It is worth mentioning that this sub-function is only required for three of the four NTRU parameter sets. Although the HRSS701 parameter set does not necessitate the execution of `crypto_sort_int32`, it is still imperative to accelerate the function. The rationale behind the acceleration of the `crypto_sort_int32` sub-function lies in its non-negligible contribution to the overall runtime during the encapsulation of the other three sets of parameters. Specifically, this sub-function accounts for approximately 7.6% of the total runtime, thereby emphasizing its significance in the encapsulation process. So, we schedule two *add* units in parallel. This enables the simultaneous comparison of two sampling data points using 16-bit precision. Notably, this approach is compatible with our memory bus width, which is set at 64-bit. By doing so, we can ensure that the processing of the bus data is completed within a single cycle, thereby improving the overall efficiency of the system.

5 Evaluation

To evaluate our design, we targeted our implementation to ASIC using the 45 nm process technology with 1 GHz of clock frequency. We implemented all components of our hardware in verilog-HDL, and synthesized those to the gatelist by

Design Compiler 2017.09-SP2 tool. The rest of the sub-functions that are not implemented in hardware are computed as software by a host processor, Cortex M4(STM32F407VG) CPU at 168MHz of clock frequency.

Table 4. Performance Comparisons with Pure Software in Various Parameters

	Parameters	Latency (ms)		
		KeyGen	Encap	Decap
Software-only [8]	HPS2048509	3489.32	92.10	233.21
	HPS2048677	6094.56	157.91	418.95
	HPS4096821	9164.47	211.15	603.89
	HRSS701	6702.06	160.61	452.73
Ours	HPS2048509	4.02	3.68	1.03
	HPS2048677	4.46	4.39	1.26
	HPS4096821	5.24	4.86	1.59
	HRSS701	4.20	3.55	1.52

5.1 Synthesized Area Result

Table 3 describes the area consumption of our hardware implementation. Out of the overall 29,821μm^2 area used, FSMs accounts for 63.4% while the registers accounts for 8.9%. In the FSM modules, the `poly_mul` sub-function occupied the largest area with 28.50%, followed by the `poly_inv` sub-function, which accounted for 25.00% of the area. These two sub-functions integrate the `poly_*_mul` and `poly_*_inv` sub-functions, respectively, and are designed to allow for the execution of multiple sub-functions as an option. The multiplication and addition arrays, each consisting of 16 elements, are composed of arithmetic units and occupy approximately 20% of the area. Lastly, the `crypto_sort_int32` sub-function was reported to occupy 7.8%.

5.2 Speedup Compared to Pure Software Implementation

In the first two rows of Table 4, we compare the performance of three NTRU functions between ours and the software-baseline in which all the functions are processed on the host CPU. As our software-baseline, we use the official version of NTRU-KEM code submitted in NIST [8]. As reported in the table, `KEYGEN` is the most time-consuming function. This can be attributed to the considerable amount of swapping and condition checking operations required by `KEYGEN`, which are significantly accelerated by our bit-wise FSM controls as well as a pool of arithmetic units. `KEYGEN` function showed an average improvement of 1,393× over four parameters. In an encapsulation function, ours exhibited an average performance improvement of 37× compared to the software implementation.

However, in our implementation, it is noteworthy that about 3% of the operations remained dependent on software, which led to a relatively lower degree of acceleration by hardware. On the other hand, the decapsulation function showed a significant average performance improvement of 307× over the software implementation. Notably, all functions demonstrated the most substantial performance improvements in the parameter set HPS4096821.

Table 5. Performance and Area Comparisons with Hardware in Various Parameters

	Parameters	Latency(ms)		Area	Time*Area	
		Encap	Decap	($10^3\mu m^2$)		
Ours	HPS2048509	3.6868	1.0378	29.82	140.8900	-
	HPS2048677	4.3999	1.2636		168.8845	-
	HPS4096821	4.8623	1.5964		192.5971	-
	HRSS701	3.5529	1.5265		151.4682	-
x-net [2]	HPS2048509	0.0062	0.0071	460.25	6.1213	0.04×
	HPS2048677	0.0084	0.0094	580.47	10.3324	0.06×
	HPS4096821	0.0102	0.0115	728.85	15.8161	0.08×
	HRSS701	0.0041	0.0117	762.17	12.0423	0.08×
comba [2]	HPS2048509	0.3492	1.0471	102.97	143.7770	1.02×
	HPS2048677	0.6161	1.8476	101.04	245.6141	1.45×
	HPS4096821	0.9047	2.7135	99.99	361.7838	1.88×
	HRSS701	0.6604	1.9834	104.69	276.7794	1.83×

† $Time * Area = (Encap\ time + Decap\ time) * Area$

5.3 Performance and Area Comparison with Prior Work

The prior work [2] presents a dedicated hardware design for NTRU encapsulation and decapsulation. They do not provide a single accelerator instance to support all NTRU parameters, thus mandating the need for separate accelerator implementations catering to different parameters. Moreover, the `KEYGEN` process is not implemented in hardware, necessitating a substantial amount of software-hardware communication to deliver the key to the hardware module, particularly when the key undergoes frequent updates throughout the session. Our proposed design, which incorporates `KEYGEN` process in hardware, offers significant acceleration of `KEYGEN`, while simultaneously eliminating unnecessary communication overhead. Along with `KEYGEN`, we have accelerated all three functions involved in NTRU and facilitated acceleration for all parameters with a single accelerator design.

The prior work [2] provides two versions. The first version comba is a design focused on compactness in a similar vein as the goal pursued in our efficient implementation. The other version, x-net is an accelerator designed to focus

on performance, regardless of chip size. The second version significantly maximizes the parallelisms within NTRU functions by instantiating multiple hardware instances. As a result, the second version necessitates an allocation of up to 7 times larger amount of hardware resources as compared to the first version.

For fairness in comparison, we mainly compared the performance and area of our design with the prior work of version comba. It is important to highlight that while x-net may surpass all other implementations in terms of the time*area metric, its area considerably exceeds that of our proposed design by two orders of magnitude. Thus, it is evident that our design is a more efficient and practical solution compared to x-net in terms of area utilization. As compared in Table 5, ours show 1.5 times higher efficiency on average in terms of performance per area, regarding the two function execution times, encapsulation and decapsulation implemented by the prior work. The version x-net is listed together just for reference. It is noteworthy that our design incorporates an additional KEYGEN function and enables support for all parameter sets, which is not a feature present in the prior research. Even when compared to parameters with high complexity, such as HPS4096821, our proposed design employs more than three times less hardware resources, while also exhibiting up to 1.7 times faster performance than the prior work.

6 Conclusion

In this paper, we implement an efficient NTRU-KEM acceleration with full functionality based on a hardware and software co-design methodology. The approach is proven effective in addressing the challenges encountered in implementing the NTRU-KEM, including the significant performance overheads associated with conventional software-based KEYGEN implementation. After the identification of commonly executed operations within sub-functions, we proceeded to develop shareable arithmetic units that could execute sub-functions in parallel. FSMs that are specifically allocated to each sub-function are responsible for managing the processing of sub-functions as an optimal schedule. This scheduling mechanism ensures a balance between performance and hardware resource utilization. The implementation demonstrated a remarkable performance improvement over software-only implementation, as evidenced by a factor of 579× faster. This finding underscores the significant practical value of the approach, especially in situations where frequent key generation is required. Compared to prior work, the implementation stands out as more efficient in terms of hardware resources, offering a performance per area that is exceeds the prior one by up to a factor of 1.8 to the maximum.

Acknowledgements. This work was supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2023, the Inter-University Semiconductor Research Center (ISRC), the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone), the National Research Foundation of

Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2022-00166529) and the Gachon University research fund of 2022 (GCU-202208860001). The EDA tool was supported by the IC Design Education Center(IDEC), Korea.

This work was supported by Samsung Electronics Co., Ltd(IO201208-07839-01).

References

1. Alagic, G., et al.: Status report on the third round of the Nist post-quantum cryptography standardization process. US Department of Commerce, NIST (2022)
2. Antognazza, F., Barenghi, A., Pelosi, G., Susella, R.: A flexible basic-oriented design for a full NTRU accelerator. In: Proceedings of the 28th Asia and South Pacific Design Automation Conference, pp. 591–597 (2023)
3. Becoulet, A., Verguet, A.: A depth-first iterative algorithm for the conjugate pair fast Fourier transform. IEEE Trans. Signal Process. **69**, 1537–1547 (2021)
4. Bekele, A.: Cooley-Tukey FFT algorithms. Advanced algorithms (2016)
5. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 409–437. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_15
6. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. J. Cryptol. **33**(1), 34–91 (2020)
7. Chung, C.M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on cortex-m4 and avx2. In: IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 159–188 (2021)
8. Cong, C., et al.: Ntru-software. https://ntru.org/software.shtml
9. Diffie, W., Hellman, M.E.: New directions in cryptography. In: Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman, pp. 365–390 (2022)
10. Farahmand, F., Dang, V.B., Nguyen, D.T., Gaj, K.: Evaluating the potential for hardware acceleration of four NTRU-based key encapsulation mechanisms using software/hardware codesign. In: Ding, J., Steinwandt, R. (eds.) PQCrypto 2019. LNCS, vol. 11505, pp. 23–43. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25510-7_2
11. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: a ring-based public key cryptosystem. In: Buhler, J.P. (eds.) Algorithmic Number Theory. ANTS 1998. LNCS, vol. 1423, pp. 267–288. Springer, Heidelberg (2006). https://doi.org/10.1007/BFb0054868
12. Kostalabros, V., Ribes-González, J., Farràs, O., Moretó, M., Hernandez, C.: HLS-based HW/SW co-design of the post-quantum classic Mceliece cryptosystem. In: 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pp. 52–59. IEEE (2021)
13. Nam, K., Oh, H., Moon, H., Paek, Y.: Accelerating n-bit operations over TFHE on commodity CPU-FPGA. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, pp. 1–9 (2022)
14. Prest, T., et al.: Falcon. Post-Quantum Cryptography Project of NIST (2020)
15. Qin, Z., Tong, R., Wu, X., Bai, G., Wu, L., Su, L.: A compact full hardware implementation of PQC algorithm NTRU. In: 2021 International Conference on Communications, Information System and Computer Engineering (CISCE), pp. 792–797. IEEE (2021)

16. Riazi, M., et al.: HEAX: an architecture for computing on encrypted data. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (2020). https://doi.org/10.1145/3373376.3378523
17. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM. **21**(2), 120–126 (1978). https://doi.org/10.1145/359340.359342
18. Schöffel, M., Feldmann, J., Wehn, N.: Code-based cryptography in IoT: a HW/SW co-design of HQC. arXiv preprint arXiv:2301.04888 (2023)
19. Schroeppel, R., Orman, H., O'Malley, S., Spatscheck, O.: Fast key exchange with elliptic curve systems. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 43–56. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-44750-4_4
20. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. **41**(2), 303–332 (1999)
21. Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., Liu, L.: Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 49–72 (2020)