computers

Article

Optimizing Hardware Resource Utilization for Accelerating the NTRU-KEM Algorithm [†]

Yongseok Lee [1,‡], Jonghee Youn [2,‡], Kevin Nam [1], Hyunyoung Oh [3,*] and Yunheung Paek [1,*]

[1] Department of Electrical and Computer Engineering and Inter-University Semiconductor Research Center, Seoul National University, Seoul 08826, Republic of Korea; yslee@sor.snu.ac.kr (Y.L.); kvnam@sor.snu.ac.kr (K.N.)
[2] Department of Computer Engineering, Yeungnam University, Gyeongsan-si 38541, Republic of Korea; youn@yu.ac.kr
[3] Department of AI·Software, Gachon University, Seongnam-si 13120, Republic of Korea
[*] Correspondence: hyoh@gachon.ac.kr (H.O.); ypaek@snu.ac.kr (Y.P.)

[†] This paper is an extended version of our paper published in ICCSA 2023: Proceedings of the 24th International Conference on Computational Science and Its Applications, 1–4 July 2023.
[‡] These authors contributed equally to this work.

Abstract: This paper focuses on enhancing the performance of the *N*th-degree truncated-polynomial ring units key encapsulation mechanism (NTRU-KEM) algorithm, which ensures post-quantum resistance in the field of key establishment cryptography. The NTRU-KEM, while robust, suffers from increased storage and computational demands compared to classical cryptography, leading to significant memory and performance overheads. In environments with limited resources, the negative impacts of these overheads are more noticeable, leading researchers to investigate ways to speed up processes while also ensuring they are efficient in terms of area utilization. To address this, our research carefully examines the detailed functions of the NTRU-KEM algorithm, adopting a software/hardware co-design approach. This approach allows for customized computation, adapting to the varying requirements of operational timings and iterations. The key contribution is the development of a novel hardware acceleration technique focused on optimizing bus utilization. This technique enables parallel processing of multiple sub-functions, enhancing the overall efficiency of the system. Furthermore, we introduce a unique integrated register array that significantly reduces the spatial footprint of the design by merging multiple registers within the accelerator. In experiments conducted, the results of our work were found to be remarkable, with a time-area efficiency achieved that surpasses previous work by an average of 25.37 times. This achievement underscores the effectiveness of our optimization in accelerating the NTRU-KEM algorithm.

Keywords: post-quantum security; NTRU; key encapsulation mechanism; hardware architecture; ASIC

check for updates

Citation: Lee, Y.; Youn, J.; Nam, K.; Oh, H.; Paek, Y. Optimizing Hardware Resource Utilization for Accelerating the NTRU-KEM Algorithm. *Computers* **2023**, *12*, 259. https://doi.org/10.3390/computers12120259

Academic Editors: Osvaldo Gervasi and Damiano Perri

Received: 15 November 2023
Revised: 5 December 2023
Accepted: 7 December 2023
Published: 13 December 2023

1. Introduction

The advent of quantum computing marks a revolutionary leap in computational technology, characterized by its ability to perform complex calculations at speeds unattainable by classical computers. This breakthrough is driven by the principles of quantum mechanics, enabling quantum computers to process vast amounts of data simultaneously and solve problems that are currently intractable for traditional computing systems [1]. Despite its promising advancements, quantum computing poses significant security risks, particularly to traditional cryptographic systems used in classical computing, as illustrated by past research revealing susceptibilities in classical key establishment methods when faced with quantum computing [2]. In response to these challenges, a new wave of algorithms, known as post-quantum cryptography (PQC), has been developed as a countermeasure to withstand cryptoanalytic attacks facilitated by quantum computing [3–5]. These algorithms are designed to provide security in the age of quantum computing. Among the array of PQC algorithms tailored for key establishments [6], the *N*th-degree truncated-polynomial ring units (NTRU) [7] approach has emerged as an especially promising cryptosystem. Impressively, since its inception in 1996, NTRU has demonstrated resilience and remains uncompromised. Diverging from classical key establishment algorithms [8,9], NTRU employs a key encapsulation mechanism (KEM). This mechanism utilizes an asymmetric key to encrypt a symmetric key, such as a session key, establishing a secure communication channel. This strategy aligns with the overarching objective of ensuring security in the realm of quantum computing, positioning NTRU as a robust solution within the domain of post-quantum cryptography.

However, like other PQC algorithms, NTRU-KEM requires increased storage space and computation resources compared to classical cryptosystems, resulting in substantial memory and performance overheads [10–12]. To address these overheads, researchers have focused on enhancing the efficiency of the NTRU-KEM algorithm, particularly in encapsulation and decapsulation functions [13–15]. Previous studies exclude the third function, *key generation* (KEYGEN), proposing that keys can be generated in advance and stored for repeated use. While this approach limits NTRU-KEM to a long-term key scenario, previous research relying on software for KEYGEN in a session-key context has led to a significant increase in latency.

Our approach differs, aiming to implement an efficient NTRU-KEM algorithm with full functionality, including KEYGEN. We adopt a hardware and software co-design methodology, a well-established platform for PQC algorithm accelerators [16,17], proven effective in addressing challenges associated with NTRU-KEM. This approach combines the strengths of both hardware and software components. Our implementation strategy aims to accelerate repetitive and parallelizable operations through hardware, focusing on increasing the utilization of limited hardware resources. Functions with high inherent parallelism and significant execution time are implemented in hardware, utilizing available resources for substantial performance gains. In contrast, functions with no repetitions or minor execution time are implemented in software to avoid unnecessary hardware resource allocation.

In our previous work [18], we concentrated on operating the NTRU-KEM functions using as few hardware resources as possible. However, this approach led to relatively high latencies, limiting the performance efficiency in terms of area. We also observed that during approximately 55% of the NTRU-KEM function's processing time, hardware components like the bus interface were idle. To improve this, we explored in a more in-depth way the data dependencies between the various sub-functions, identifying opportunities for greater parallelism. This led to the development of a new scheduling technique that maximizes the use of hardware resources, including bus interfaces and ALUs. Our approach allows for different ALU operations, such as addition and multiplication, to be executed concurrently. However, running these operations simultaneously necessitates an increase in the number of registers to prevent data overwriting. To address this without significantly increasing the hardware size, we designed an integrated register array that can be efficiently reused throughout NTRU-KEM processes. The specifics of this implementation are detailed in Section 4.

Our implementation is designed to run on 45 nm process technology at a 1 GHz clock frequency for the hardware components, and a CPU (i5-8550) operating at 2.5 GHz for the software components. In our experimental comparisons, our implementation demonstrates higher efficiency in hardware resource usage compared to previous studies that implemented only the encapsulation and decapsulation functions, leaving out KEYGEN. We achieved this by optimizing the utilization of a smaller area, resulting in an improved performance ranging from 1.82 to 51.38 times greater per area.

A list of the contributions of this work is provided below:

- **Efficient full-functionality implementation:** Unlike previous studies that focused only on encapsulation and decapsulation functions, this work aims to efficiently implement the entire NTRU-KEM algorithm, including the key generation (KEYGEN) function, using a hardware and software co-design methodology.

- **Optimized use of hardware resources:** Our work addresses the challenge of high latencies and underutilized hardware resources (like bus interfaces) observed in previous work [18]. Our approach achieves this by developing a new scheduling technique that maximizes hardware resource usage, particularly for cross-functional operations that can be executed in parallel.
- **Minimized area for registers:** To run concurrent operations without data overwriting, we design an integrated register array. This design allows for efficient reuse of registers throughout NTRU-KEM processes, minimizing hardware size while maintaining performance.
- **Enhanced performance efficiency:** Our implementation demonstrates a significant improvement in hardware resource efficiency, showing a performance increase of 1.82 to 51.38 times per area compared to previous work.

2. Background

This section provides a brief introduction to the fundamentals of the 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. It is 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 the lattice are bound to hardness based on problems such as the shortest vector problem or the 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 the underlying problems, even for quantum computers, which makes it superior to other public-key cryptosystems. Furthermore, the 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 [7]. The 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 [7] 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. The NTRU cryptosystem samples four polynomials, f, g, r, and m from Γ, which are used as will be described in Section 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 public key 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 precomputed polynomial $h_q = h^{-1} \ mod \ (q, \Phi_n)$. Using the polynomials above, encapsulation and decapsulation are computed as depicted in Figure 1.

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

Figure 1. The three functions of NTRU-KEM.

Table 1 depicts the four parameter sets considered as standards for NTRU. The HPS and HRSS parameters show small difference in the preprocessing of the algorithm (e.g., sorting the seed data before the start of the KEM function itself). All parameters have distinct polynomial degree N and modulus prime q. Accordingly, they have different security strengths and computational complexity, for which reason users should choose parameters with care, according to their need.

Table 1. Parameter sets of NTRU-KEM.

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

3. Design Overview

Our work focuses on designing a single ASIC accelerator for a generic purpose—to be able to compute all four parameters in Table 1. We focused on an area of the ASIC design and maximizing the utilization. This approach aligns with the goal stated in our paper, which is to maximize the area efficiency. This design allows for efficient implementation without the need to aggregate separate hardware modules designated for each parameter set to support each function, such as `KEYGEN`, encapsulation and decapsulation in Figure 1. Our hardware design methodology focuses on area efficiency and hardware utilization, including the memory bus and ALU.

Figure 2 illustrates the design overview of our hardware implementation. At the center of our hardware setup, we have two ALUs responsible for executing arithmetic operations, and finite state machine (FSM) modules that manage and orchestrate the ALUs to perform the NTRU-KEM functions. Communication between these components is facilitated through a 64-bit bus interface, which transmits control and data signals. Specifically, each ALU is equipped with 64 instances of both multipliers and adders, handling 16-bit integers for input and output data types. The bus interface serves as a crucial path for all data that needs to be read or written. In addition to this, our design includes a general purpose register array with 56 entries, used for storing intermediate values during the execution of various sub-functions. These register entries are directly and in parallel connected to the ALUs to enable simultaneous computing. A key aspect of our design is the balanced allocation of resources across each module, ensuring an optimized trade-off between the area and performance. The specifics of this resource allocation are detailed in subsequent sections.

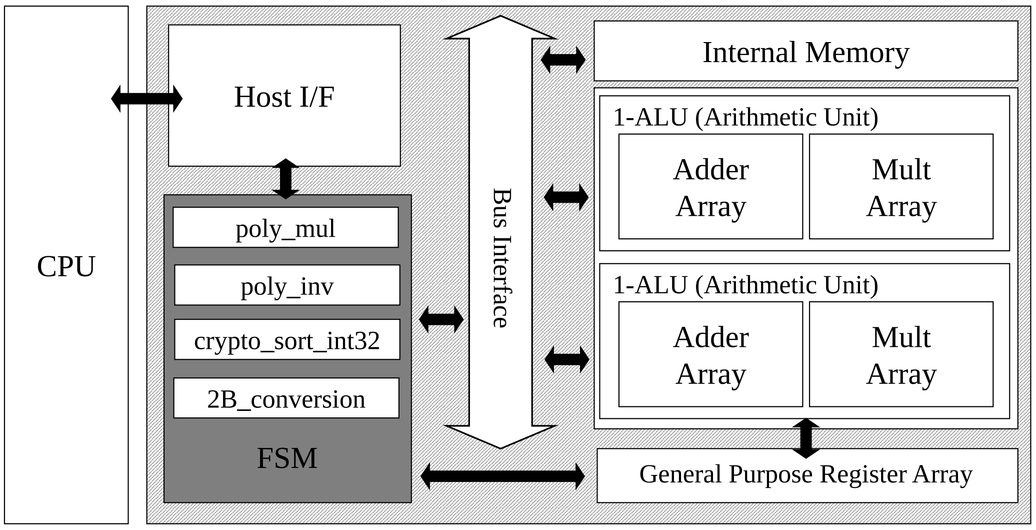

Figure 2. Overview of our hardware implementation.

4. Implementation

This section presents the details of our accelerator and the design of the modules. Our architecture targets a hardware and software co-design, emphasizing area-efficient hardware design. First, we profiled the sub-functions to select the portions to be implemented as dedicated hardware modules. In other words, we selected heavy tasks to be implemented as dedicated hardware and let small tasks be implemented as the software part. As such, we designed four separate FSM modules, each dedicated to a specific sub-function: polynomial multiplication (`poly_*_mul`), polynomial inversion (`poly_*_inv`), integer sorting (`crypto_sort_int32`), and binary conversion (`2B_conversion`). These FSMs were designed based on their respective common arithmetic patterns. `poly_*_mul` includes poly_R_q_mul, poly_S_3_mul, poly_S_q_mul, and poly_R_2_inv_to_R_q_inv sub-functions for polynomial multiplication with specific ring modulus. And `poly_*_inv` includes poly_R_2_inv and poly_S_3_inv sub-functions for polynomial inverse operation. This approach is similar to our previous work [18]. However, we observed that while our previous work was efficient in terms of area, its use of resources was not optimal. This was due to uneven allocation of resources, leading to time-related bottlenecks. For example, the bus bandwidth was underutilized, using less than half of its capacity when the ALU was busy, which negatively affected the overall performance and efficiency. To address such issues, we adopt a different approach that aims to maximize resource utilization while still maintaining the design's compact area efficiency.

4.1. FSM-Oriented Hardware Design

As implied by its name, poly_R_q_mul includes multiplications between polynomials on a ring R_q. To ease the complexity of the multiplications, conventional methods employ fast Fourier transform (FFT) or number theoretic transform (NTT) [19–25]. However, NTRU does not use such operations for two major reasons. First, the quotient rings used in NTRU exhibit properties that are not directly supportive of FFT/NTT, as such approaches demand polynomials to have degrees of power of two [26,27]. However, NTRU variants use polynomials with degrees that are prime numbers, thus not powers of two. Previous research [28] extends the polynomials to higher degrees of powers of two but results in more computation than the naive multiplications. The second reason is that the predefined modulus used in NTRU is rather simple and can be efficiently computed using specialized operation sequences. For instance, NTRU frequently uses the modulo 3, which only uses 1, 0, and -1 as coefficients, so any computation between such values can be easily managed by zeroization and bit-flip operations.

We, therefore, choose to naively compute polynomial operations, instead of employing additional techniques as prior work did, by extending the polynomials and using FFT/NTT. The obvious but crucial bottleneck of our approach is the increase in conditional branches within the algorithm, which are needed in finding the inverse polynomials and coefficient swappings. This is the reason why we select FSM-oriented designs. To efficiently handle these conditional branches, our hardware implementation of NTRU leverages finite state machines (FSMs) for the computation of sub-functions. Each FSM manages a pool of basic arithmetic units (i.e., multipliers and adders) to process internal operations in parallel. We carefully select the number of basic arithmetic units by a cycle-accurate 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 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 the latency. Our FSM-oriented design enabled us to achieve a balance between performance and the hardware area, resulting in an efficient implementation.

4.2. Latency Profiling 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 needs to be performed first. To identify the specific sub-functions that have the potential to significantly reduce the overall execution time of NTRU-KEM, a comprehensive latency 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, intending to determine which ones may be optimized for improved efficiency. We used the NIST submission version of NTRU-KEM with an i5-8500 CPU at 2.5 GHz clock frequency.

Table 2 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 reside within them. For en/decapsulation, poly_R_q_mul, poly_S_3_mul, and poly_S_q_mul account for between 83 and 97% of the total runtime. Accordingly, the aforementioned existing NTRU-KEM accelerators that only accelerated these two functions [13–15] aimed to boost the performance of the polynomial multiplication. However, in the case of KEYGEN, other sub-functions, poly_R_q_inv and poly_S_3_inv, account for the biggest portion, over 86%, while poly_R_q_mul only accounts for 12%.

Therefore, to accelerate the entire NTRU-KEM algorithm, poly_R_q_inv and poly_S_3_inv also need to be taken into consideration when designing the hardware. It is worth 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 performed independently of the others. In addition, crypto_sort_int32 in encapsulation and KEYGEN is also selected to implement in the hardware since it can be processed in a parallel manner. Conversely, the sub-function named randombytes within the 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. The randombytes sub-function is used or the seed generation before the KEYGEN and encapsulation function. And the SHA3-hash sub-function is also left to the software part because of the small execution time in the CPU. To efficiently process the above-selected two kinds of polynomial operations, the optimizations that we have devised will be presented in the following subsections.

Table 2. NTRU-KEM latency profiling results in CPU (i5-8500).

Function	**NTRU-KEM Variant**							
	HPS2048509		**HPS2048677**		**HPS4096821**		**HRSS701**	
Key Generation	43.8088 ms	100%	77.0217 ms	100%	114.3290 ms	100%	82.5875 ms	100%
poly_Rq_inv	20.7199	47.30	36.4755	47.36	53.7846	47.04	39.3617	47.66
poly_R2_inv	.	.	.	.	.	.	.	.
poly_R2_inv_to_Rq_inv	.	.	.	.	.	.	.	.
poly_S3_inv	17.3077	39.51	30.4974	39.60	44.8878	39.26	32.7177	39.61
poly_Rq_mul	5.4949	12.54	9.7996	12.72	14.2652	12.48	10.3971	12.59
crypto_sort_int32	0.1359	0.31	0.1999	0.26	0.2534	0.22	-	-
randombytes	0.0407	0.09	0.0493	0.06	0.0605	0.05	0.0236	0.03
Encapsulation	1.3186 ms	100%	2.2319 ms	100%	3.2032 ms	100%	2.1644 ms	100%
poly_Rq_mul	1.0974	83.23	1.9365	86.76	2.8308	88.37	2.0700	95.64
crypto_sort_int32	0.1333	10.11	0.1985	8.89	0.2520	7.87	-	-
randombytes	0.0374	2.84	0.0483	2.16	0.0586	1.83	0.0217	1.00
SHA3-hash	0.0076	0.57	0.092	0.41	0.0098	0.30	0.0094	0.43
etc.	0.0430	3.26	0.0395	1.77	0.0520	1.62	0.0633	2.92
Decapsulation	3.3900 ms	100%	5.9193 ms	100%	8.6410 ms	100%	6.3532 ms	100%
poly_Rq_mul	1.0936	32.26	1.9678	33.24	2.8410	32.88	2.0849	32.82
poly_S3_mul	1.1082	32.69	1.9434	32.83	2.8354	32.81	2.0797	32.73
poly_Sq_mul	1.1032	32.54	1.9404	32.78	2.8310	32.76	2.0727	32.62
SHA3-hash	0.0207	0.64	0.0269	0.45	0.0282	0.33	0.0294	0.46
etc.	0.0644	1.90	0.0408	0.69	0.1054	1.22	0.0866	1.36

4.3. Optimizing Bus Utilization

In our previous work [18], we had allocated a computation array and a sufficient number of registers to prevent delays in the operations, focusing on operational efficiency. This setup allowed continuous computation in each clock cycle, especially during polynomial multiplication, involving both multiplication and addition operations. However, upon shifting our focus to the bus interface in this study, we discovered that its utilization was quite low. For instance, Figure 3 shows that the bus utilization ratio was only about 25% during the `poly_`R_q`_mul` sub-function using 1-ALU.

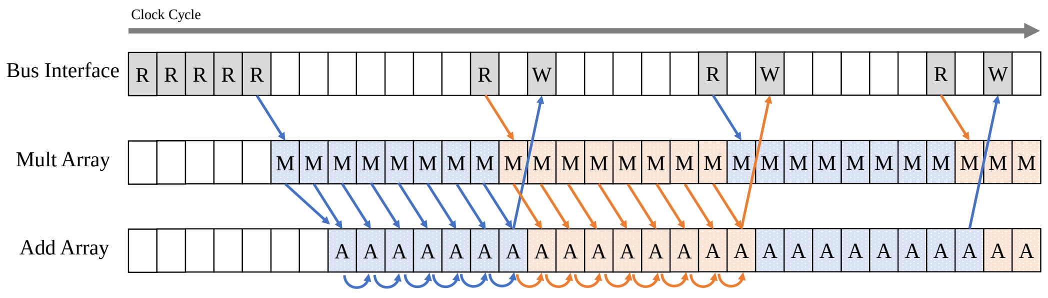

Figure 3. Example of bus utilization in `poly_`R_q`_mul` sub-function with 1-ALU.

Given that polynomial multiplication is a significant part of the NTRU-KEM functions, requiring around 80,000 cycles for a single sub-function operation, this low bus utilization was a concern. To address this issue, we decided to make constructive use of the underutilized bus interface during these computational periods. Our strategy involved increasing both the computation array and the number of registers to expedite the operational time. In the 1×16 computation array of 1-ALU, the upper and lower 16-bit operations were processed sequentially. By doubling this array to a 2-ALU array, we could simultaneously process the upper and lower bits, allowing sequential operation of the corresponding addition array. As a result, utilizing the 1×16 computation array in this manner reduced the operational cycles to half. Not only did this modification reduce the cycles needed for operations by half, but it also increased the frequency of bus interface usage during the shorter operational cycle. Consequently, we achieved a 50% bus utilization rate and a 50% reduced clock cycle. To further maximize bus utilization, we applied the additional techniques detailed in Section 4.4.

4.4. Parallel Sub-Function Scheduling

To minimize the overall processing time with limited resources in the NTRU-KEM function, we developed a parallel sub-function scheduling technique. This technique allows two different sub-functions to be executed simultaneously. For example, it is possible to perform polynomial multiplication while also carrying out polynomial inversion or sorting tasks concurrently. This not only enables parallel execution but also makes efficient use of previously underutilized bus interfaces during polynomial multiplication tasks. To achieve this, we conducted an in-depth data flow analysis of each sub-function, synchronizing the timing of the bus interface usage across different sub-functions. The goal was to optimize resource utilization, particularly during the execution of the polynomial multiplication sub-function. The next three subsections provide a comprehensive explanation of the techniques.

4.4.1. Key Generation Scheduling

The `KEYGEN` function in NTRU-KEM includes a series of `poly_inv` and `poly_mul` sub-functions. NTRU utilizes an efficient almost inverse algorithm [29], which calculates almost an inverse of a polynomial that yields an exact result across all NTRU-KEM functions. This inverse calculation primarily consists of iterating a swap operation based on the highest degree of a non-zero coefficient in each polynomial. Notably, the `poly_inv` sub-function does not require the use of multiplication or addition arrays. This is because it operates on coefficients that are only 1 or 2 bits in size associated with the polynomial inverse sub-functions within the R_2 or S_3 ring modulus. For the polynomial inverse in the R_q ring modulus, the poly_R_q_inv result is achieved by combining the poly_R_2_inv and poly_R_2_inv_to_R_q_inv sub-functions. In these processes, we primarily use bit operations, `reg_array`, and the swap operation within the FSM.

4.4.2. Encapsulation Scheduling

In the encapsulation function of our design, we implemented a parallel scheduling strategy for both the `poly_mul` and `crypto_sort_int32` sub-functions. Initially, the function receives seed data (r, m) and public key data pk. The `poly_mul` sub-function works with r and pk, while the `crypto_sort_int32` processes m, making it feasible for these functions to operate simultaneously. However, efficient parallelization is challenging due to the limitations of the bus interface. Both sub-functions have idle time in the bus interface. The bus utilization rates are 50% for `poly_mul` and 80% for `crypto_sort_int32`. To manage their scheduling, the intervals of bus usage for `poly_mul` are spaced out in two-cycle units. Although the ALU operates every cycle, the output is stored in memory every four cycles, with one cycle for loading new data and another for storing results.

As depicted in Figure 4, we maximized bus utilization by aligning the scheduling of the `poly_mul` and `crypto_sort_int32` sub-functions. This was performed by creating two-cycle intervals in which the bus interface is idle, during which the `crypto_sort_int32` sub-function operates. The bus interface for `crypto_sort_int32` was designed to match these two-cycle intervals of the `poly_mul` sub-function for optimal scheduling. By thoroughly analyzing the bus interface requirements of both sub-functions, we achieved efficient parallel scheduling. This led to a remarkable bus utilization rate of approximately 97% for the encapsulation function using the HPS4096821 parameter. This optimization significantly improved our design's performance, eliminating the need for extra resources to increase the bus interface's bit-width.

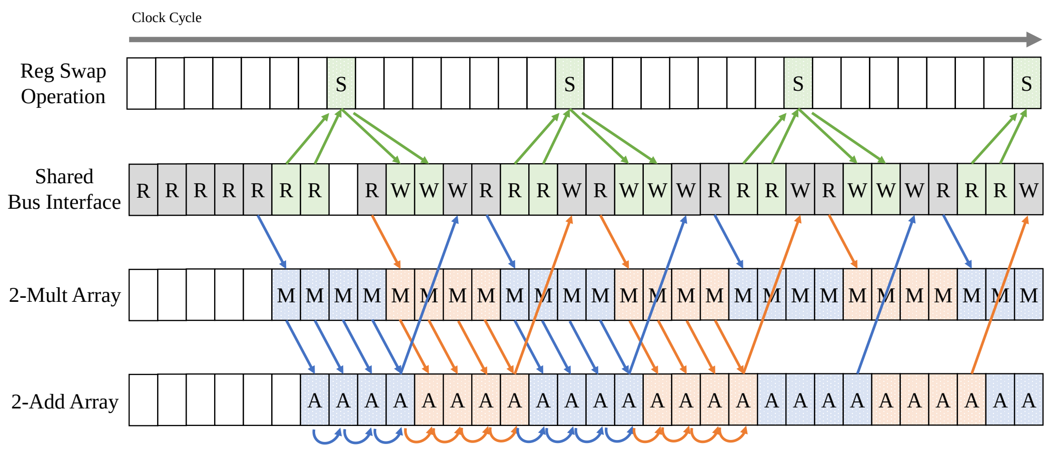

Figure 4. Parallel sub-function scheduling between `poly_`R_q`_mul` and `crypto_sort_int32` with 2-ALU.

4.4.3. Decapsulation Scheduling

In the decapsulation function, `poly_`R_q`_mul`, `poly_`S_3`_mul`, and `poly_`S_q`_mul` sub-functions are scheduled in a parallel and pipelined manner. Each of these sub-functions, which perform polynomial multiplication, has a bus interface utilization rate of about 50%. A key challenge is simultaneously executing two polynomial multiplication sub-functions, as this not only requires bus utilization but also additional ALU resources.

On analyzing these sub-functions, we found that they operate with different ring moduli. The R_q and S_q rings require a 16-bit integer multiplier and adder, while the S_3 ring works with 2-bit data and coefficients in the range [0,1,2]. This allows the `poly_`S_3`_mul` sub-function to be executed using bitwise operators, eliminating the need for an ALU. Using this knowledge, we parallelized the `poly_`S_3`_mul` operation with the textttpoly_R_q_mul and `poly_`S_q`_mul` sub-functions. Despite sharing the same bus utilization rate, the unique characteristics of the `poly_`S_3`_mul` allowed for efficient parallel scheduling.

Moreover, the data for the `poly_`S_3`_mul` sub-function become available from the timing of completing half of the `poly_`R_q`_mul` sub-function, allowing for immediate input to the subsequent `poly_`S_3`_mul` sub-function. Similarly, starting from the timing of completing half of the `poly_`S_3`_mul` sub-function, the input data for the `poly_`S_q`_mul` sub-function could be obtained. We applied a pipelined approach to coordinate these three sub-functions effectively.

4.5. Optimizing `Reg_Array` Utilization with Combined Structure

In this section, we aim to explain our techniques that reduce the design area while preserving the performance enhancements achieved in the earlier sections. Our goal is to develop an area-efficient design that balances both performance improvement and area conservation. We implemented an integrated management approach for controlling the registers in FSMs. This involved allocating registers to arrays that were not in use at a particular time. This strategy, linked to the parallel sub-function scheduling technique described previously, effectively reduces the number of control registers needed. For instance, in the `KEYGEN` and encapsulation functions, the polynomial inversion and sorting sub-functions are never executed simultaneously as they are both scheduled alongside polynomial multiplication sub-functions. Thus, combining the control registers for these sub-functions does not hinder performance.

In the decapsulation function, we initially allocated a register array of 56 for storing intermediate data during the parallel and pipelined scheduling of polynomial multiplication sub-functions. However, we found that in various scenarios, such as in `KEYGEN` for polynomial inversion and multiplication, in encapsulation for sorting and polynomial multiplication, and in decapsulation for two parallel polynomial multiplication sub-functions, fewer registers were used (44, 40, and 56, respectively). Consequently, we were able to combine the control registers for two sub-functions into these 12 unused register arrays. This approach not only reduced the total number of registers required in the entire accelerator but also improved the utilization rate of the existing registers. Through this integrated management, we achieve a decrease in the design area without compromising performance, aligning with the goal of an area-efficient design.

5. Evaluation

To evaluate our design, we integrated the hardware part as an ASIC operation at 1 GHz over 45 nm processing technology. We used HDL language and the Design Compiler 2017.09-SP2 tool to design and obtain the synthesized results. For the software part, we evaluated over an Intel core i5-8500 CPU @2.5 GHz with 32 GB main memory, using a single core only.

5.1. Impact of Our Optimization Methods

Table 3 shows a gradual representation of our optimization process for our hardware implementation starting from our previous work's design (v1 [18]) (i.e., how we reached our optimal design). Note that all but the three functions performed by our hardware part are executed by our software part with consistent performance, no matter which entry of Table 3 is used. Each ALU consists of an array of 16 multipliers and 16 adders. Each multiplier can perform 1 × 16-bit multiplications, while the adder can handle 16-bit additions. Note that the number of registers mentioned in the table *only* refers to the general purpose registers—some extra registers are used in the FSMs.

Table 3. Performance comparison with variable optimization options in our design.

#	Description	Parameters	Register (16-bit)	Area (10^3 μm^2)	t_{Keygen} (ms)	t_{Enc} (ms)	t_{Dec} (ms)	Time × Area
v1 [18]	Reference (1-ALU)	HPS2048509	28	29.82	0.6671	0.0507	0.0993	4.47
		HPS2048677			1.1994	0.0845	0.1768	7.79
		HPS4096821			1.7418	0.1187	0.2587	11.25
		HRSS701			1.2598	0.0624	0.1873	7.45
v2	2-ALU	HPS2048509	56	45.15	0.4555	0.0341	0.0497	3.78
		HPS2048677			0.8228	0.0551	0.0884	6.48
		HPS4096821			1.1908	0.0755	0.1294	9.25
		HRSS701			0.8609	0.0312	0.0936	5.64
v3	4-ALU	HPS 048509	112	60.48	0.3497	0.0258	0.0248	3.07
		HPS2048677			0.6346	0.0403	0.0442	5.11
		HPS4096821			0.9152	0.0540	0.0647	7.18
		HRSS701			0.6615	0.0156	0.0468	3.77
v4	2-ALU + Parallel Sub-Function Scheduling	HPS2048509	56	45.15	0.4452	0.0238	0.0331	2.57
		HPS2048677			0.8044	0.0367	0.0589	4.32
		HPS4096821			1.1638	0.0486	0.0862	6.09
		HRSS701			0.8414	0.0312	0.0624	4.23
v5	2-ALU + Parallel Sub-Function Scheduling + Combined Register	HPS2048509	56	39.60	0.4452	0.0238	0.0331	**2.25**
		HPS2048677			0.8044	0.0367	0.0589	**3.79**
		HPS4096821			1.1638	0.0486	0.0862	**5.34**
		HRSS701			0.8414	0.0312	0.0624	**3.71**

ALU: One ALU includes 16-multiplication and 16-addition array modules used in this design. *Time* × *Area* = (*Encap time* + *Decap time*) × *Area*.

Increasing the number of ALUs to four enables 1.33× better efficiency on average, in terms of the time × area metric compared to the single ALU reference. The number of registers is increased accordingly to meet the maximum needs of the overall computation (i.e., it cannot be reduced). However, note that such an approach induces a linear increase in the area usage, while the latency of each function is reduced with a smaller scale. This is due to the bottleneck induced at the bus interface, as its bandwidth is fixed at 64 bits. Increasing the number of ALUs requires more data movement—having four ALUs demands almost full bandwidth usage to compute the multiplications during `poly_mul`, making it difficult to perform other tasks since all tasks need data to pass the bus interface. Consequently, parallelizing the sub-function scheduling becomes infeasible and leaves several registers unused, which, in turn, leads to reduced resource utilization.

We, therefore, select a design with 2 ALUs (v4) and apply parallel sub-function scheduling, maintaining the same design area as v2 but with enhanced bus interface utilization. As a result, we obtained even better performance compared to the 4-ALU design (v3) for all sub-functions. As a final stage, we also added our `reg_Array` technique and reached our optimized design (v5). While the overall latency remains similar, the area is reduced, thus resulting in better performance in terms of time $\times$ area compared to all the other variants.

5.2. Performance Comparison with Prior Work

When comparing with prior work [15], we assumed that the pre-/post-processing steps, such as generating seed and hashing, are executed in the software part, which also has the small execution time discussed in Section 4.2. Prior work considered only the seed generation as a software part. In our approach, hashing is considered as a pre-/post-processing step, because it is performed either before encapsulation or after decapsulation, rather than during the hardware operation. Therefore, for a fair comparison, we included the time taken for hashing in our latency calculations, as shown in Table 4.

Table 4. Performance and area comparisons with hardware for various parameters.

	Parameters	**Latency (ms)**		**Area (10^3 μm^2)**	**Time $\times$ Area**	
		Encap	**Decap**			
Ours (SW-SHA3 + HW-Ours-v5)	HPS2048509	0.0313	0.0538	39.60	3.37	-
	HPS2048677	0.0458	0.0858		5.21	-
	HPS4096821	0.0583	0.1145		6.84	-
	HRSS701	0.0443	0.0918		5.39	-
x-net [15]	HPS2048509	0.0062	0.0071	460.25	6.12	1.82$\times$
	HPS2048677	0.0084	0.0094	580.47	10.33	1.98$\times$
	HPS4096821	0.0102	0.0115	728.85	15.82	2.31$\times$
	HRSS701	0.0041	0.0117	762.17	12.04	2.24$\times$
comba [15]	HPS2048509	0.3492	1.0471	102.97	143.78	42.65$\times$
	HPS2048677	0.6161	1.8476	101.04	245.62	47.75$\times$
	HPS4096821	0.9047	2.7135	99.99	361.79	52.87$\times$
	HRSS701	0.6604	1.9834	104.69	276.78	51.38$\times$

Time $\times$ Area = (Encap time + Decap time) $\times$ Area.

Table 4 shows the performance of our design and existing ASIC accelerators, with ours outperforming the others by 1.82$\times$–52.87$\times$ in terms of the time $\times$ area metric. Antognazza et al. [15] focused on accelerating the encapsulation and decapsulation functions of NTRU-KEM. They proposed two sets of designs: `x-net`, which is a performance-centric design set, and `comba`, one that focuses on area efficiency. For each set, these authors designed separate accelerators for each of the four parameters; thus, single designs cannot be used for a generic purpose (i.e., supporting all parameters). Their design areas for each parameter of NTRU-KEM ranged from 99.99 $\times$ 10^3 μm^2 to 762.17 $\times$ 10^3 μm^2 when combining the encapsulation module and the decapsulation module together.

In contrast, our optimized design (v5) can perform NTRU computation over all four parameters, offering a more generic purpose usage. Our design also supports not only encapsulation and decapsulation but also `KEYGEN`. All the above extra features are supported in a single design with an efficient area usage of 39.60 $\times$ 10^3 μm^2. Note that even when compared to the area efficient version of `comba`, our design uses at least 2.87$\times$ less area, while demonstrating, on average, 17.66$\times$ faster performance in terms of latency.

Compared to the `x-net` version, our accelerator exhibited an average performance that was 7.69 times slower but achieved an average design area that was 15.98 times smaller. In terms of the time $\times$ area, our accelerator exhibits superior efficiency, surpassing that of both the `x-net` and `comba` accelerator versions by average factors of 2.09 and 48.66 across all parameters, respectively. It is worth noting that the mitigation of side-channel attacks (SCA) is beyond the scope of this work. However, future research could explore applying and optimizing the methods mentioned in [30] to mitigate SCA. Given our design's efficiency in terms of the time $\times$ area metric compared to previous works, we believe that our design will outperform others even after incorporating methods to mitigate SCA.

6. Conclusions

This paper proposes an area-efficient accelerator for the NTRU-KEM algorithm based on a hardware and software co-design. Unlike prior work that requires separate designs for each of the parameters, our solution supports all four standard parameters in a single design. Our optimizing methodology focuses on maintaining minimal area usage while maximizing hardware resource utilization by implementing parallel sub-function execution and the combined use of registers. Our approach achieves 52$\times$ better efficiency in terms of the time $\times$ area metric compared to previous works. Overall, our design offers a generic NTRU-KEM accelerator supporting all four standard parameters using a small hardware area, while achieving competitive performance.

Author Contributions: Conceptualization, H.O. and Y.P.; methodology, J.Y.; software, K.N.; validation, Y.L., J.Y. and K.N.; formal analysis, H.O.; investigation, Y.L.; resources, J.Y.; data curation, Y.L.; writing—original draft preparation, Y.L.; writing—review and editing, H.O. and Y.P.; visualization, K.N.; supervision, Y.P.; project administration, H.O.; funding acquisition, H.O. and Y.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially 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 and Communications Technology Planning and Evaluation (IITP), grant funded by the Korean Government (MSIT) (No. 2020-0-01840 Analysis on technique of accessing and acquiring user data in smartphone), the Institute of Information and Communications Technology Planning and Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081), grant funded by the Korean Government (MSIT), the National Research Foundation of Korea (NRF), grant funded by the Korean Government (MSIT) (No. RS-2022-00166529), the Gachon University research fund of 2022 (GCU-202208860001), and the National Research Foundation of Korea (NRF), grant funded by the Korean Government (MSIT) (RS-2023-00277326). The EDA tool was supported by the IC Design Education Center (IDEC), Republic of Korea.

Data Availability Statement: Research data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nielsen, M.A.; Chuang, I.L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*; Cambridge University Press: Cambridge, UK, 2011.
2. Shor, P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.* **1999**, *41*, 303–332. [CrossRef]
3. Kumar, M.; Pattnaik, P. Post quantum cryptography (PQC)-An overview. In Proceedings of the 2020 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 22–24 September 2020; pp. 1–9.
4. Raheman, F. The future of cybersecurity in the age of quantum computers. *Future Internet* **2022**, *14*, 335. [CrossRef]
5. Shinohara, N.; Moriai, S. Trends in Post-Quantum Cryptography: Cryptosystems for the Quantum Computing Era. *The Magazine of New Breeze*, 2019; pp. 9–11. Available online: https://www.ituaj.jp/wp-content/uploads/2019/01/nb31-1_web-05-Special-TrendsPostQuantum.pdf (accessed on 14 November 2023).
6. Yaman, F.; Mert, A.C.; Öztürk, E.; Savaş, E. A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 1020–1025.

7. Hoffstein, J.; Pipher, J.; Silverman, J.H. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory, Proceedings of the Third International Symposiun, ANTS-III, Portland, OR, USA, 21–25 June 1998*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 267–288.
8. Rivest, R.L.; Shamir, A.; Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [CrossRef]
9. Diffie, W.; Hellman, M.E. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*; Morgan & Claypool: San Rafael, CA, USA , 2022; pp. 365–390.
10. Dang, V.B.; Farahmand, F.; Andrzejczak, M.; Gaj, K. Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign. In Proceedings of the 2019 International Conference on Field-Programmable Technology (ICFPT), Tianjin, China, 9–13 December 2019; pp. 206–214.
11. Kannwischer, M.J.; Rijneveld, J.; Schwabe, P. Faster multiplication in on Cortex-M4 to speed up NIST PQC candidates. In Proceedings of the International Conference on Applied Cryptography and Network Security, Bogota, Colombia, 5–7 June 2019; pp. 281–301.
12. He, P.; Tu, Y.; Khalid, A.; O'Neill, M.; Xie, J. HPMA-NTRU: High-Performance Polynomial Multiplication Accelerator for NTRU. In Proceedings of the 2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Austin, TX, USA, 19–21 October 2022; pp. 1–6.
13. Qin, Z.; Tong, R.; Wu, X.; Bai, G.; Wu, L.; Su, L. A compact full hardware implementation of PQC algorithm NTRU. In Proceedings of the 2021 International Conference on Communications, Information System and Computer Engineering (CISCE), Beijing, China, 14–16 May 2021; pp. 792–797.
14. 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 Proceedings of the Post-Quantum Cryptography: 10th International Conference, PQCrypto 2019, Chongqing, China, 8–10 May 2019; pp. 23–43.
15. Antognazza, F.; Barenghi, A.; Pelosi, G.; Susella, R. A Flexible ASIC-oriented Design for a Full NTRU Accelerator. In Proceedings of the 28th Asia and South Pacific Design Automation Conference, Tokyo, Japan, 16–19 January 2023; pp. 591–597.
16. 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 Proceedings of the 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), Dresden, Germany, 30 August–3 September 2021; pp. 52–59.
17. Schöffel, M.; Feldmann, J.; Wehn, N. Code-based Cryptography in IoT: A HW/SW Co-Design of HQC. *arXiv* **2023**, arXiv:2301.04888.
18. Lee, Y.; Nam, K.; Joo, Y.; Kim, J.; Oh, H.; Paek, Y. Area-Efficient Accelerator for the Full NTRU-KEM Algorithm. In Proceedings of the International Conference on Computational Science and Its Applications, Athens, Greece, 3–6 July 2023; pp. 186–201.
19. Riazi, M.; Laine, K.; Pelton, B.; Dai, W. 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, Lausanne, Switzerland, 16–20 March 2020. [CrossRef]
20. 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, San Diego, CA, USA, 30 October–3 November 2022; pp. 1–9.
21. Cheon, J.H.; Kim, A.; Kim, M.; Song, Y. Homomorphic Encryption for Arithmetic of Approximate Numbers. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Hong Kong, China, 3–7 December 2017; pp. 409–437.
22. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachène, M. TFHE: Fast Fully Homomorphic Encryption over the Torus. *J. Cryptol.* **2020**, *33*, 34–91. [CrossRef]
23. Prest, T.; Fouque, P.A.; Hoffstein, J.; Kirchner, P.; Lyubashevsky, V.; Pornin, T.; Ricosset, T.; Seiler, G.; Whyte, W.; Zhang, Z. Falcon. Post-Quantum Cryptography Project of NIST. 2020 . Available online: https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022 (accessed on 14 November 2023).
24. 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. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*; IACR: Santa Barbara, CA, USA, 2020 ; pp. 49–72. Available online: https://ches.iacr.org/2020/index.php (accessed on 14 November 2023).
25. Bisheh-Niasar, M.; Azarderakhsh, R.; Mozaffari-Kermani, M. High-speed NTT-based polynomial multiplication accelerator for post-quantum cryptography. In Proceedings of the 2021 IEEE 28th Symposium on Computer Arithmetic (ARITH), Lyngby, Denmark, 14–16 June 2021; pp. 94–101.
26. Cooley, J.W.; Lewis, P.A.W.; Welch, P. The fast Fourier transform algorithm: Programming considerations in the calculation of sine, cosine and Laplace transforms. *J. Sound Vib.* **1970**, *12*, 315–337. [CrossRef]
27. Becoulet, A.; Verguet, A. A depth-first iterative algorithm for the conjugate pair fast fourier transform. *IEEE Trans. Signal Process.* **2021**, *69*, 1537–1547. [CrossRef]
28. 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*; ICAR: Santa Barbara, CA, USA, 2021 ; pp. 159–188. Available online: https://ches.iacr.org/2021/index.php (accessed on 14 November 2023).

29. Schroeppel, R.; Orman, H.; o'Malley, S.; Spatscheck, O. Fast key exchange with elliptic curve systems. In Proceedings of the Advances in Cryptology—CRYPT0'95: 15th Annual International Cryptology Conference, Santa Barbara, CA, USA, 27–31 August 1995; Proceedings; Springer: Berlin/Heidelberg, Germany, 2001; pp. 43–56.
30. Standaert, F.X. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 27–42.