ELectronics EXpress

LETTER

A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse

Yiming Huang[1], **Miaoqing Huang**[2], **Zhongkui Lei**[1, a)], **and Jiaxuan Wu**[3]

Abstract This paper presents a pure hardware implementation of CRYSTALS-KYBER algorithm on Xilinx FPGAs. CRYSTALS-KYBER is one of 26 candidate algorithms in Round 2 of NIST Post-Quantum Cryptography (PQC) standardization process. The proposed design focuses on maximizing resource utilization by reusing most of the functional modules in the encapsulation and decapsulation processes of the algorithm. For instance, the hash module integrates several different hash functions in one module. Efficient parallel and pipelined computations are applied in the NTT module. Through the analysis of simulation and synthesis results, it is found that the proposed work has the advantages of higher frequencies and lower execution times. The scheme operates at 155 MHz and 192 MHz frequencies on Xilinx Artix-7 and Virtex-7 FPGAs, respectively. Compared with the performance of an embedded Cortex-M4 processor, the hardware implementation can achieve a maximum speedup of 129 times for encryption/decryption.

Keywords: CRYSTALS-KYBER, cryptography, field-programmable gate arrays (FPGAs), PQC
Classification: Integrated circuits (memory, logic, analog, RF, sensor)

1. Introduction

It has been reported [1] that Post-Quantum Cryptography (PQC) algorithm are going to replace the classic public-key cryptography algorithms such as RSA, which will become vulnerable once quantum computers become mature. Therefore, NIST started the process to standardize PQC algorithms. In December 2017, 69 algorithms were released in Round One of the process. Among these 69 algorithms, 26 algorithms (including 17 encryption/key-encapsulation (PKE/KEM) and 9 signature schemes) advanced to Round Two in January 2019. CRYSTALS-KYBER is one of 17 PKE/KEM schemes in Round Two.

Most Post-Quantum Cryptography algorithms in Round Two of the NIST standardization process contain a large amount of complicated calculations and repeated math operations. Pure software implementations are not good at parallel computing. An encryption or decryption period on microprocessors could take a massive number of clock cycles. Therefore, it is critical to use hardware designs to accelerate the calculation processes and assess diverse PQC algorithms on hardware platforms such as FPGAs.

Recently, several PQC schemes have been implemented in pure hardware. Among these implementations, [2] presented the hardware implementation of NewHope-Simple algorithm on Xilinx Artix-7 FPGAs. In [3] the Rainbow digital signature algorithm was implemented on FPGAs. The implementation reduced about half of the number of multiplications, and it can reconfigure different security levels. Besides, a software and hardware co-design on Xilinx Zynq FPGAs was presented in [4], including three Lattice-based PQC algorithms: FrodoKEM, Round5, and Saber. It used hardware logic to accelerate most of the computation. The hardware logic is connected to the ARM processor on the Zynq platform through the AXI bus. As for the PQC CRYSTALS-KYBER (Kyber) [5] algorithm version 2.0, the work in [6] presented an implementation on ARM Cortex-M4 embedded processor. It was able to achieve 18% performance speedup while using a tiny memory footprint.

The main difficulty of a pure hardware implementation of the Kyber PQC algorithm is the large amount of math operations, including Number Theoretic Transforms (NTT), division, and shifting computation through several matrices. We mainly used two strategies in our hardware implementation. (1) We identified the operations that contribute to the most computation in the whole process and tried to accelerate them. (2) We used BRAM (block random access memory) on FPGAs to reduce the overall cost. One challenge was to coordinate dozens of bottom-level modules (including encryption/decryption modules, pre-encryption/pre-decryption modules, etc.) to realize the key encapsulation/decapsulation mechanism of the Kyber algorithm.

This paper focuses on the hardware implementations of Kyber algorithm on two different Xilinx FPGAs, Artix-7 XC7A200T on AC701 board, and Virtex-7 XC7VX485T on VC707 board. We cover three different cryptography security levels (i.e., 512, 768, and 1024) in our implementations. AC701 is the primary target board if it contains enough hardware resources. Otherwise the target board would be VC707. Using the Verilog hardware description language, we designed the major operations of the Kyber algorithm.

The remainder of this paper is organized as follows. We analyze the mathematical logic of the Kyber algorithm and make an appropriate direction of implementations in Section 2. In Section 3 we demonstrate the specific scheme of overall architecture and the design of key modules. In Section 4, we illustrate main performance results and comparison with other implementation. Lastly, we give the concluding

[1] Nanjing University of Aeronautics and Astronautics, China
[2] University of Arkansas, USA
[3] ShanghaiTech University, China
a) leizhongkui@nuaa.edu.cn

DOI: 10.1587/elex.17.20200234
Received July 7, 2020
Accepted July 16, 2020
Publicized August 14, 2020
Copyedited September 10, 2020

remarks in Section 5.

2. Implementation analysis of Kyber algorithm

Considering Kyber is an IND-CCA2-secure KEM whose mathematic basis focuses on the learning-with-errors problem in module lattices (MLWE problem [7]) presented in [8]. The whole Kyber encryption or decryption is packed by regularization data and IND-CPA cryptography with a slightly tweaked Fujisaki-Okamoto (FO) transform [9, 10] in IND-CCA2 KEM cryptography. The mathematical fundamental is Ring-LWE introduced in [11, 12], and the mathematical carrier is the polynomial rings. Recent work [13] implemented arithmetic in the polynomial ring with algorithms of Karatsuba [14] and ToomCook [15, 16]. For Kyber, the original ring $Z[X]/(X^n+1)$ is denoted by R where $n = 2^{n'-1}$ such that X^n+1 is the $2^{n'-1}$-th cyclotomic polynomial. The Kyber algorithm picks the polynomial ring $R_q = Z_q[X]/(X^n + 1)$ where q is the modulo parameter [17]. Specifically, Kyber cryptography packs $n = 256$ and $q = 3,329$ among different security levels. All parameter sets for Kyber related FPGA implementations are demonstrated in Table I.

Table I Parameter sets for Kyber implementation.

Algorithm	Level	Parameters (n/k/q)	Public key/Secret key/Ciphertext size p/s/c (in Bytes)
Kyber512	1	256/2/3,329	800/1,632/736
Kyber768	3	256/3/3,329	1,184/2,400/1,088
Kyber1024	5	256/4/3,329	1,568/3,168/1,568

Before performing encryption or decryption, the Kyber algorithm preconditions data to regular form, including compression, decompression, sampling, rejection, encoding, and decoding. For compression and decompression function, the former transfers an element data $x \in Z_q$ to an integer in $(d < log_2(q))$ and vice versa. The superimposed error matrix in Kyber is sampled from a centered binomial distribution (CBD) [18] so that each of coefficient from polynomial $f \in R_q$ is sampled. After that, they are moduloed by rejection function. The encoding works in serializing polynomials to byte arrays and decoding translates byte stream to polynomials vectors. We did not include key generation operation in our implementation. This work implements the encryption and decryption processes of the Kyber algorithm. We slightly modify the Kyber algorithm so that the demonstration looks more clearly and it becomes easier for FPGA implementation. Figure 1 and Figure 2 show the Kyber encryption KEM and decryption KEM approaches, respectively.

Specifically, the IND-CPA Kyber encryption in encryption KEM Flowchart Step 5 involves several modules including SHAKE-128, SHAKE-256, centered binomial distribution, NTT, pointwise-multiplied accumulation (PACC), inverse NTT, modulo polynomials, compress, decompress, encode, and decode. The IND-CPA Kyber decryption in Figure 2 Step 3 includes compress, decompress, NTT, inverse NTT, encode, and decode modules. From above, it can be found that the encryption and decryption share a lot of modules together. In other words, we can reuse a lot of modules when we implement both encryption and decryption of

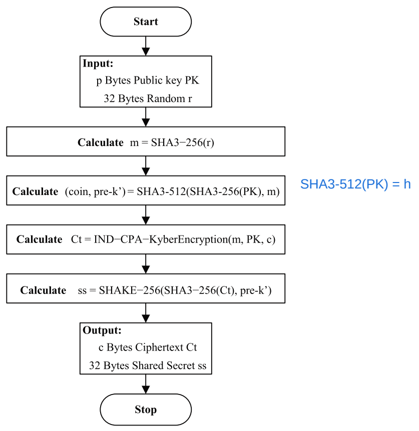
Start
Input:
p Bytes Public key PK
32 Bytes Random r
Calculate m = SHA3−256(r)
Calculate (coin, pre-k') = SHA3-512(SHA3-256(PK), m)
SHA3-512(PK) = h
Calculate Ct = IND−CPA−KyberEncryption(m, PK, c)
Calculate ss = SHAKE−256(SHA3−256(Ct), pre-k')
Output:
c Bytes Ciphertext Ct
32 Bytes Shared Secret ss
Stop

Fig. 1 The flowchart of Kyber encryption KEM.

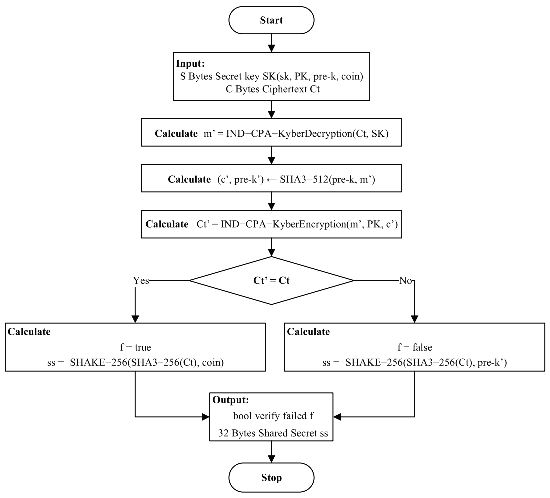
Start
Input:
S Bytes Secret key SK(sk, PK, pre-k, coin)
C Bytes Ciphertext Ct
Calculate m' = IND−CPA−KyberDecryption(Ct, SK)
Calculate (c', pre-k') ← SHA3−512(pre-k, m')
Calculate Ct' = IND−CPA−KyberEncryption(m', PK, c')
Ct' = Ct
Yes
No
Calculate
f = true
ss = SHAKE−256(SHA3−256(Ct), coin)
Calculate
f = false
ss = SHAKE−256(SHA3−256(Ct), pre-k')
Output:
bool verify failed f
32 Bytes Shared Secret ss
Stop

Fig. 2 The flowchart of Kyber decryption KEM.

the Kyber algorithm.

The reference software implementation of Kyber algorithm defines two projects, one for encryption and the other for decryption. A lot of subroutines are called in both projects. In software, calling a subroutine does not cost much physical resources. However, if we create a new instance of a module each time we call it in hardware implementation, the final implementation will occupy a massive amount of resource. For Kyber algorithm, the encryption operation and decryption operation typically do not run at the same time. Therefore, we decided to share those modules that are used in both operations. For instance, various hash functions are used widely in the Kyber algorithm. If we can design a hardware module that can realize all hash functions used in the algorithm, we will significantly reduce the hardware resource by reusing the hash module. In this work, we tried to reuse as many hardware modules as possible to maximize the resource utilization. This approach also increases the reconfigurability of the design.

FPGAs are considered as one of the most popular hardware platforms due to their easy access and abundant resources on them. Typical strategies on hardware for performance improvement include parallel execution of multiple modules and pipelining. In Kyber algorithm, the decoding and decompressing, coding and compressing are practically not rely on each other. An parallel execution of these modules would improve the performance. For other complicated functions like NTT and inverse NTT, using pipelines can improve the throughput. Overall, a good trade-off between resource utilization and performance needs to be carefully considered on FPGA devices.

3. The FPGA design scheme

3.1 Overall architecture

The overall architecture of the pure hardware implementation of Kyber cryptography algorithm is illustrated in Figure 3. The diagram contains almost all critical components while omitting the Xilinx BRAM IPs (simple dual-port BRAMs). These BRAMs are inserted between functional modules for communication purpose. In Figure 3 solid lines and dashed lines are used to present IND-CPA Kyber encryption and IND-CPA Kyber decryption data flow, respectively. The Pre/Post Encryption module implements some functions such as message creation, shared key generation, etc., which are used before and after encryption. Similarly, there is a Pre/Post Decryption module for the similar purpose. The Controller module connects to all modules and sends commands such as "enable" and "finish" to direct data flow. It also controls encryption or decryption processes through two MUXes.

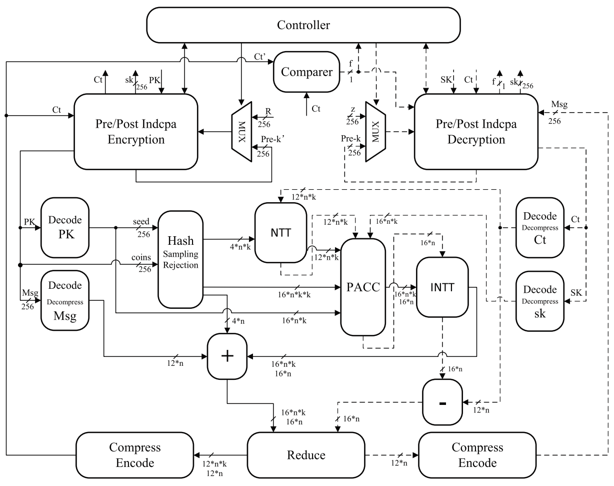

Fig. 3 Overall Kyber encapsulation and decapsulation architecture.

The Kyber encryption part and decryption part share several modules including NTT, inverse NTT, PACC, and Reduce. This sharing significantly reduce the overall hardware cost. For some other modules such as decoding and decompressing, encoding and compressing, there are multiple instances in the design for parallel execution. The hash module presented in Section 3.2 not only does integrate all hash functions, but also pipelines the rejection operation such that multiple rejections can be processed at the same time while at different stages. The NTT module uses a pipelined design and is introduced in details in Section 3.3.

There are a lot of intermediate data produced by modules in both the encryption and decryption processes. The amount of data would increase dramatically as the security level grows. For example, the number of hash polynomials is 9 with security level 1. It would increase to 25 with security level 5. On FPGA, we use BRAMs to store these intermediate data. More precisely, we insert BRAMs between two modules, one will produce the data and the other will consume the data. When the security level grows, we could increase the data depth of BRAMs without changing the whole design. Using BRAMs can also reduce the use of FPGA slices. We only use BRAMs for intermediate data. Input data, such as public key, secret key, plaintext, and ciphtertext, are saved in off-chip SRAMs. In brief, we want to maximize the resource utilization on FPGAs and put as many modules as possible on the devices.

3.2 Hash module

In the Kyber algorithm, several hash functions including SHAKE-128, SHAKE-256, SHA3-256, and SHA3-512, are involved. All hash functions above follow the specification defined in FIPS-202 [19, 20, 21]. The base of these hash functions is the Keccak algorithm [22], which needs 12 rounds of permutation operations with different round constants. The data produced by the Permutation stage are fed into Sampling module and Rejection module in parallel. Once a completed polynomial is saved into the BRAM, the controller would send the next group of input data to the Permutation stage. The structure of the hash module is demonstrated in Figure 4.

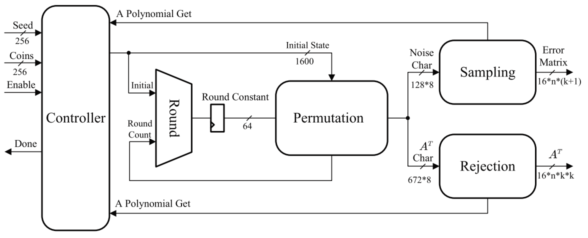

Fig. 4 Hash module structure.

The hash implementation shown above uses a permutation calculation as the bottom module, which is fed with different round constants every clock cycle. The initial input for noise character is mainly produced by the same coins. And the A^T character is mainly generates by the same seed. Due to one-way hash function, different permutation initial-state inputs have different permutation outputs. The differences of the initial state are small discrepancies of the dimension number of A^T and the succession number of noises. When permutation is done, the noise character and A^T character are fed into the Sampling module and the Rejection module in parallel. The former module samples the coefficient to be CBD satisfied. The latter module makes data within the boundary of q, in addition to transferring character array to a matrix.

In this hash design, we use combinational logic to implement Keccak permutation process. On microprocessor, the Keccak permutation process may take hundreds of clock cycles [23]. Our hardware implementation only takes one clock cycle. Then different hash functions can be built on top of the Keccak permutation process. For example, the SHAKE-256 based noise polynomial only needs one round of Keccak permutation. The SHAKE-128 based A^T generation needs four rounds to Keccak permutation. Given different hash functions and input permutation states, the controller will schedule different rounds of Keccak permutation accordingly.

3.3 NTT module

Many PQC algorithms consist of a lot of multiplications. Number Theoretic Transform (NTT) is typically used to improve the multiplication capacity [24, 25, 26]. Recent research on NTT applications involves with the Intel processors by Seiler [27], Lyubashevsky [28], and ARM Cortex-M4 in [29]. Kyber algorithm also applies NTT to accelerate multiplication performance. A typical NTT polynomial $f = \sum_{i=0}^{n-1} f_i X^i \in R_q$ with negacyclic is denoted as Equation 1.

$$\begin{aligned} NTT(f) = \hat{f} = \sum_{i=0}^{n-1} \hat{f}_i X^i, \\ \hat{f}_i = \sum_{j=0}^{n-1} \psi^j f_j \omega^{ij} \mod q \end{aligned} \tag{1}$$

By using this equation, the multiplication between two polynomials $f, g \in R_q$ can be simplified as $f \bullet g = NTT^{-1}(NTT(f) \circ NTT(g))$. The negacyclic NTT recursion with the changing of ψ transfers the coefficient of polynomial from real number domain to NTT domain. After the point-wise multiplication of accumulation in NTT domain is done, the inverse NTT would recover the data then. According to the latest Kyber algorithm specification, one of the biggest changes in version 2.0 [5] is the NTT refinement compared with version 1.0 [8]. In version 2.0 of Kyber, the q is fixed with $\mathbb{Z}_q$, which includes 256^{th} roots of unity, not 512^{th}. Thus, the NTT of a polynomial $f \in R_q$ is a vector of 128 polynomials as Equation 2. It is used to handle the CBD-sampled error matrices and public key through encryption and decryption.

$$NTT(f) = \hat{f} = \sum_{i=0}^{127} (\hat{f}_{2i} + \hat{f}_{2i+1} X) \tag{2}$$

Meanwhile, according to Montgomery Reduction [30], the solution $a \bullet \omega^{-1}(\text{mod q})$ is suitable for NTT transform with standard order by inputs and bitreversed order by outputs. A pre-multiplication for $\psi^j \bullet f_j$ and a Montgomery Reduction computation in Kyber have three multiplications and one subtraction, which can be pipelined in the hardware implementation. At the first clock cycle, the $(j+len)^{th}$ coefficient is pushed into Montgomery Reduction and the previous $(j-3)^{th}$ coefficient needs to be written back to BRAM. Then, the address of coefficient changes to j, whose original data would be captured by registers marked as 1-bit pp_state. In the meantime, the $(j + len - 3)^{th}$ coefficient, which is subtracted by the original j^{th} coefficient, is written back to BRAM. At the third and fourth clock cycles, the coefficient with new address would be pushed into Montgomery Reduction again. The result of previous 2 computation cycles would be written back to BRAM. During the NTT computation cycles, four terms of coefficients are isolated. This isolation assures the success of the pipelining design. The structure of the NTT module is shown in Figure 5. Major operations of the NTT module are as follows.

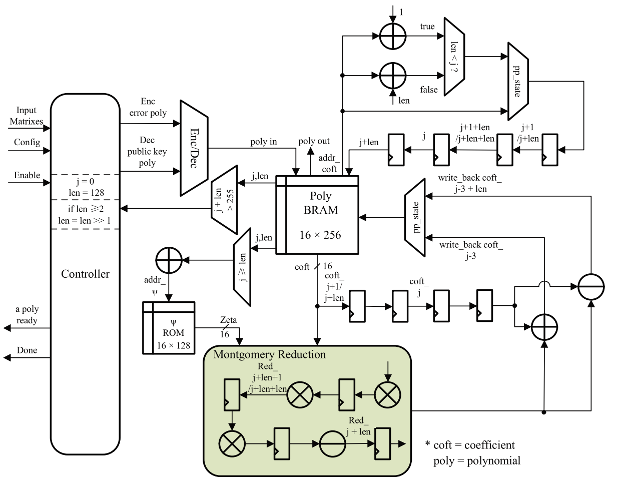

Fig. 5 NTT module structure.

- When it is enabled, the Poly BRAM would store the pending polynomial from the enc/dec mux. The initial values of $addr\_\psi$, j, len would be 1, 0, 128, respectively. The first coefficient_address in cycle is denoted by $j + len$.
- Push current coefficient and ψ into Montgomery Reduction module.
- Update coefficient_address. Following the current Montgomery Reduce coefficient address, the pending addition/subtraction coefficient address would be subtracted by len to form an address pair. Then, the new Montgomery Reduction coefficient address would increase 1 (if $len < j$) or len (otherwise). In the meantime, we update the values of $addr\_\psi$ and len as follows.
  - If $j \geqslant len$, $addr\_\psi = addr\_\psi + 1$; otherwise, $addr\_\psi = addr\_\psi$.
  - If $j + len > 255$, len does a bit right shift until len equal 2; otherwise, $len = len$.
- When addition/subtraction coefficient is read, a temporary register stores the data marked as pp_state. Meanwhile, the previous coefficient, which adds or subtracts the Montgomery result would be written back to BRAM. For example, after initial four clock cycles, $(j-3)^{th}$, $(j-3+len)^{th}$, $(j-2)^{th}$ and $(j-2+len)^{th}$ coefficients have all been written back to BRAM.
- The Montgomery Reduction recursion would stop when the last coefficient is pushed into the BRAM. The j and len would be 253 and 2, respectively, so that $j + len > 255$. After the final reduction is done, a polynomial NTT process is finished.

The polynomial would be handled one by one with the ready flag is set true. When all polynomials are processed, the whole NTT process is completed. With this NTT implementation design, for improving performance, we pipeline the Montgomery Reduction process. It would increase the efficiency more than three times compared with the non-pipelined implementation. The non-pipelined NTT design would take 6,550 clock cycles while the pipelined NTT module only takes 1,834 clock cycles. This implementation also solves the storage issue of polynomials by using BRAM inside the module.

Table II Preference comparison (clock cycles) in three implementations.

Algorithm	Implementation Platform	Encapsulation [cycles]	Decapsulation [cycles]	Encapsulation Time Reduction Percentage [%]	Decapsulation Time Reduction Percentage [%]
Kyber512	This Work	49,015	68,815	—	—
	Haswell [5]	161,440	190,206	69.6	63.8
	Cortex-M4 [6]	634,000	597,000	92.3	88.5
Kyber768	This Work	77,481	102,113	—	—
	Haswell [5]	272,254	315,976	71.5	67.7
	Cortex-M4 [6]	946,000	1,167,000	91.8	91.2
Kyber1024	This Work	107,054	135,553	—	—
	Haswell [5]	396,928	451,096	73.0	70.0
	Cortex-M4 [6]	1,525,000	1,732,000	93.0	92.2

Table III Resource utilization and timing consumption.

Algorithm	Process	FPGA	Clock Freq. [MHz]	LUTs	Slices	DSPs	BRAMs	Div.IPs	Cortex-M4 Time [ms]	This Work Time [ms]	Speedup
Kyber512	Enc.	Aritix7 AC701	155	80,322	141,825	54	200.5	2	26.417	0.316	83.6
	Dec.	Aritix7 AC701	155	88,901	152,875	354	202	3	24.875	0.444	56.0
Kyber768	Enc.	Aritix7 AC701	155	97,085	153,867	36	200.5	2	46.375	0.500	92.75
	Dec.	Aritix7 AC701	155	110,260	167,293	292	202	3	44.125	0.659	67.0
Kyber1024	Enc.	Virtex7 VC707	192	119,189	162,636	36	200.5	2	72.167	0.558	129.3
	Dec.	Virtex7 VC707	192	132,918	172,489	548	202	3	68.876	0.706	97.6

4. Result

Table II compares the performance of this work with the performances of the original reference implementations on Intel Core i7-4770K (Haswell) by C language [5] and on Cortex-M4 in 24MHz [6].

With a variety of performance optimizations in hardware implementations, the amount of total clock cycles for both encryption and decryption of this propsed design reduces notably compared with Cortex-M4 implementation as well as Haswell implementation. Besides the typical techniques such as parallel execution and pipelining in hardware, innovative design techniques, such as the integration of multiple hash functions in a single module, reusing most of the functional modules during encryption and decryption, and using variable input/output widths of BRAM IPs, lead to this remarkable performance improvement.

The target platforms of hardware designs are Xilinx AC701 and VC707 FPGA boards. The detailed results, including resource utilization, single encapsulation, and decapsulation process timing and performance comparison with [5], are presented in Table III.

The results above show that the maximum clock frequencies can reach 155 MHz and 192 MHz, respectively, on AC701 and VC707 after synthesis and implementation. As expected, for both encryption and decryption, the time consumption on hardware drops significantly compared with the software implementation on Coretex-M4. The highest speedup could reach 129.3 times and the average speedup could be 87.7 times. Also, due to functional modularization design in the top module and maximum BRAM utilization through the whole design, we are about to accomodate both encryption and decryption on the same device. Meanwhile, the use of DSPs has a certain contribution to performance speedup for operations such as additions and multiplications to accelerate the entire scheme.

5. Conclusion

In this paper, a pure hardware implementation scheme on FPGA for the CRYSTALS-KYBER Post-Quantum Cryptography algorithm is presented. This scheme implements both encryption and decryption, and uses top-down modular design approach, in which BRAM is adopted to interface communicating components. In the whole design, pipelining and parallel execution are extensively used in all modules for improving the performance. At the same time, we tried to save hardware resources by reusing bottom modules in designing upper-level modules. For example, the hash module can support multiple hash functions sharing a set of basic functional components. Compared with the software implementations running on desktop and embedded processors, the hardware implementation can achieve a speedup as high as 129 times while fitting on a single FPGA device.

References

[1] NIST: Post-Quantum Cryptography Standardization https://csrc.nist.gov/Projects/post-quantum-cryptography.

[2] T. Oder and T. Güneysu: "Implementing the NewHope-Simple key exchange on low-cost FPGAs," International Conference on Cryptology and Information Security in Latin America **11368** (2017) 128 (DOI: 10.1007/978-3-030-25283-0_7).

[3] A. Ferozpuri and K. Gaj: "High-speed FPGA implementation of the NIST round 1 rainbow signature scheme," 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig) (2018) (DOI: 10.1109/RECONFIG.2018.8641734).

[4] V.B. Dang, *et al.*: "Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign," 2019 International Conference on Field-Programmable Technology (ICFPT) (2019) 206 (DOI: 10.1109/ICFPT47387.2019.00032).

[5] P. Schwabe, *et al.*: "CRYSTALS-Kyber–algorithm specifications and supporting documentation," NIST Technical Report (2019).
[6] L. Botros, *et al.*: "Memory-efficient high-speed implementation of Kyber on Cortex-M4," International Conference on Cryptology in Africa **11627** (2019) 209 (DOI: 10.1007/978-3-030-23696-0_11).
[7] A. Langlois and D. Stehlé: "Worst-case to average-case reductions for module lattices," Designs, Codes and Cryptography **75** (2015) 565 (DOI: 10.1007/s10623-014-9938-4).
[8] J. Bos, *et al.*: "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," 2018 IEEE European Symposium on Security and Privacy (EuroS&P) (2018) 353 (DOI: 10.1109/EuroSP.2018.00032).
[9] E. Fujisaki and T. Okamoto: "Secure integration of asymmetric and symmetric encryption schemes," Annual International Cryptology Conference **26** (2013) 80 (DOI: 10.1007/s00145-011-9114-1).
[10] D. Hofheinz, *et al.*: "A modular analysis of the Fujisaki-Okamoto transformation," TCC 2017 **10677** (2017) 341 (DOI: 10.1007/978-3-319-70500-2_12).
[11] O. Regev: "On lattices, learning with errors, random linear codes, and cryptography," Journal of the ACM **56** (2009) 1 (DOI: 10.1145/1568318.1568324).
[12] V. Lyubashevsky, *et al.*: "On ideal lattices and learning with errors over rings," Journal of the ACM **60** (2013) 1 (DOI: 10.1145/2535925).
[13] M.J. Kannwischer, *et al.*: "Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates," ACNS 2019, Lecture Notes in Computer Science **11464** (2019) 281 (DOI: 10.1007/978-3-030-21568-2_14).
[14] Karatsuba: "Multiplication of multidight numbers on automata," Doklady Akad Nauk Sssr **145** (1963) 595.
[15] A. Cook Stephen, *et al.*: "On the minimum computation time of functions," Ph.D Dissertation, Harvard University, Boston (1966).
[16] A.L. Toom: "The complexity of a scheme of functional elements realizing the multiplication of integers," Doklady Akademii Nauk Sssr **3** (1963) 496 (DOI: 10.1016/j.actao.2009.04.001).
[17] C. Peikert: "Public-key cryptosystems from the worst-case shortest vector problem," ACM on Theory of Computing (2009) 333 (DOI: 10.1145/1536414.1536461).
[18] Z. Brakerski, *et al.*: "Classical hardness of learning with errors," Proceedings of the Annual ACM Symposium on Theory of Computing (2013) 575 (DOI: 10.1145/2488608.2488680).
[19] K. John, *et al.*: "SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash," NIST Special Publications (2016) 800-185.
[20] M.J. Dworkin: SHA-3 Standard: permutation-based hash and extendable-output functions," NIST FIPS (2015) 202.
[21] D.J. Bernstein, *et al.*: Tweetable FIPS 202 (2015) https://keccak.team.
[22] G. Bertoni, *et al.*: "Keccak specifications," submission to the NIST SHA-3 competition (2011).
[23] A. Langley: "Maybe skip SHA-3" (2017) https://www.imperialviolet.org/2017/05/31/skipsha3.html.
[24] L. Vadim, *et al.*: "SWIFFT: a modest proposal for FFT hashing," International Workshop on Fast Software Encryption **5086** (2008) 54 (DOI: 10.1007/978-3-540-71039-4_4).
[25] T. Pöppelmann and T. Güneysu: "Towards practical lattice-based public-key encryption on reconfigurable hardware," International Conference on Selected Areas in Cryptography **8282** (2013) 68 (DOI: 10.1007/978-3-662-43414-7_4).
[26] S.S. Roy, *et al.*: "Compact ring-LWE cryptoprocessor," International Workshop on Cryptographic Hardware and Embedded Systems **8731** (2014) 371 (DOI: 10.1007/978-3-662-44709-3_21).
[27] G. Seiler: "Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography," Cryptology ePrint Archive, Report (2018) 39.
[28] V. Lyubashevsky and G. Seiler: "NTTRU: truly fast NTRU using NTT," Transactions on Cryptographic Hardware and Embedded Systems **3** (2019) 180 (DOI: 10.13154/tches.v2019.i3.180-201).
[29] E. Alkim, *et al.*: "A new hope on ARM Cortex-M," 6th Security, Privacy, and Advanced Cryptography Engineering **10076** (2016) 332 (DOI: 10.1007/978-3-319-49445-6_19).
[30] P.L. Montgomery: "Modular multiplication without trial division," Mathematics of Computation **44** (1985) 519 (DOI: 10.2307/2007970).