HSPA: High-Throughput Sparse Polynomial Multiplication for Code-based Post-Quantum Cryptography

PENGZHOU HE, Electrical and Computer Engineering, Villanova University, Villanova, United States
YAZHENG TU, Electrical and Computer Engineering, Villanova University, Villanova, United States
TIANYOU BAO, Electrical and Computer Engineering, Villanova University, Villanova, United States
ÇETIN ÇETIN KOÇ, Computer Science, University of California Santa Barbara, Santa Barbara, United States
JIAFENG XIE, Electrical and Computer Engineering, Villanova University, Villanova, United States

Increasing attention has been paid to code-based post-quantum cryptography (PQC) schemes, e.g., HQC (Hamming Quasi-Cyclic) and BIKE (Bit Flipping Key Encapsulation), since they've been selected as the fourth-round National Institute of Standards and Technology (NIST) PQC standardization candidates. Though sparse polynomial multiplication is one of the critical components for HQC and BIKE, hardware-implemented high-performance sparse polynomial multiplier is rarely reported in the literature (due to its high-dimension and sparsity of polynomials involved in the computation). Based on this consideration, in this article, we propose two novel **H**igh-throughput **S**parse **P**olynomial multiplication **A**ccelerators (HSPA) for the mentioned two code-based PQC schemes. Specifically, we have designed the two accelerators based on two different implementation strategies targeting potential applications with different resource availability, i.e., one accelerator deploys a memory-based structure for computation while the other does not need memory usage. We have proposed three layers of coherent interdependent efforts to obtain the proposed accelerators. First, we have proposed two implementation strategies to execute the targeted sparse polynomial multiplication, i.e., a new parallel segment based accumulation (PSA) approach and a novel permutating-with-power (PWP)-based method. Then, the proposed two hardware accelerators are presented with detailed structural descriptions. Finally, field-programmable gate array (FPGA)-based implementation is presented to showcase the superior performance of the proposed accelerators. A proper comparison is also carried out to confirm the efficiency of the proposed designs. For instance, the proposed accelerator (using memory-based structure) has 56.84% and 80.25% less area-delay product (ADP) than the existing memory-based design (an extended high-speed version) on the UltraScale+ device, respectively, for $n = 17,669$ and $\omega = 75$ (HQC) and $n = 12,323$ and $\omega = 142$ (BIKE). The proposed design strategy fits well with the two targeted code-based PQC schemes, which can

Pengzhou He, Yazheng Tu, and Tianyou Bao contributed equally.
The work of J. Xie was supported by NSF Award SaTC-2020625 and NIST-60NANB20D203. Ç. K. KoÇ was supported by TUBITAK Project 1001-121F348.
Authors' Contact Information: Pengzhou He, Electrical and Computer Engineering, Villanova University, Villanova, Pennsylvania, United States; e-mail: phe@villanova.edu; Yazheng Tu, Electrical and Computer Engineering, Villanova University, Villanova, Pennsylvania, United States; e-mail: ytu1@villanova.edu; Tianyou Bao, Electrical and Computer Engineering, Villanova University, Villanova, Pennsylvania, United States; e-mail: tbao@villanova.edu; Çetin Çetin Koç, Computer Science, University of California Santa Barbara, Santa Barbara, California, United States; e-mail: cetinkoc@ucsb.edu; Jiafeng Xie, Electrical and Computer Engineering, Villanova University, Villanova, Pennsylvania, United States; e-mail: jiafeng.xie@villanova.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1539-9087/2024/12-ART16
https://doi.org/10.1145/3703837

be extended further to construct high-performance hardware cryptoprocessors. We hope the results of this work will be useful for the ongoing NIST PQC standardization process.

CCS Concepts: • **Hardware** → **Hardware accelerators**; **Application specific processors**; • **Security and privacy** → **Hardware security implementation;**

Additional Key Words and Phrases: Code-based post-quantum cryptography, column-based accumulation, hardware accelerator, high-throughput, permutating-with-power, sparse polynomial multiplication (polynomial multiplication over $\mathbb{F}_2$)

ACM Reference Format:
Pengzhou He, Yazheng Tu, Tianyou Bao, Çetin Çetin Koç, and Jiafeng Xie. 2024. HSPA: High-Throughput Sparse Polynomial Multiplication for Code-based Post-Quantum Cryptography. *ACM Trans. Embedd. Comput. Syst.* 24, 1, Article 16 (December 2024), 24 pages. https://doi.org/10.1145/3703837

1 Introduction

Security analyses have mathematically confirmed the vulnerability of traditional cryptosystems such as **RivestShamirAdleman** (**RSA**) and **elliptic curve cryptography** (**ECC**) when facing attacks launched from large-scale quantum computers executing Shor's algorithm [30]. Therefore, the need for cryptosystems that are secure against quantum attacks, which are collectively known as **post-quantum cryptography** (**PQC**), is at an all-time high [30, 35]. The **National Institute of Standards and Technology** (**NIST**) started the PQC standardization process in 2016 and has recently announced the candidates for the fourth round [2]. Among these four fourth-round candidates, two structured code-based encryption schemes, namely **Hamming Quasi-Cyclic** (**HQC**) and **Bit Flipping Key Encapsulation** (**BIKE**), have recently gained substantial attention from the research community [2, 21].

Code-based cryptography refers to the cryptosystems whose security rely on the hardness of decoding in linear error correction codes, following the seminal work of McEliece and Niederreiter [19, 29]. HQC is a public key encryption scheme based on the hardness of decoding random quasi-cyclic codes in Hamming metric [21]. While BIKE is built on another class of linear codes, namely **Quasi-Cyclic Moderate-Density Parity-Check** (**QC-MDPC**) codes [3].

Prior Research. Despite the increasing attention being paid to the mentioned two PQC schemes (especially after they advanced to the fourth round of the NIST PQC standardization process), only a limited amount of hardware implementation results have been reported so far. A high-level synthesis work for HQC was presented in the original submission of Reference [21] (but with no individual component result such as sparse polynomial multiplier). An efficient hardware implementation of HQC was presented in Reference [6], and was revised in a later-on work [7]. Another efficient hardware-implemented sparse polynomial multiplier for HQC was recently presented in Reference [32]. Optimized sparse polynomial multiplier for BIKE was presented in Reference [9]. A scalable hardware implementation of BIKE was then proposed in Reference [25], and an improved polynomial multiplication accelerator for BIKE was recently reported in Reference [24]. To the authors' best knowledge, these are the major hardware implementations for the two code-based schemes up to now.

Challenges. Sparse polynomial multiplication over $\mathbb{F}_2$ is a key arithmetic step in both HQC and BIKE that the efficiency of this polynomial multiplication determines the overall performance of the final implementation to a large extent. For efficient hardware implementation of the targeted sparse polynomial multiplication, however, there still exist two main challenges: (i) the large size of the linear codes within HQC and BIKE has resulted in high-dimension of the polynomial multiplication length n, e.g., at least 12,323 [3] and could reach up to 57,637 [21], which involves very

large computational complexity; (ii) the sparsity of the polynomial has brought huge difficulty on actual implementation as the majority of the coefficients in one of the input polynomials are all '0's (for instance, one input polynomial of the sparse polynomial multiplication used in hqc-256 for $n = 57,637$ involves only 149 random nonzero elements [21], and thus, any implementation without appropriately considering this feature will have extremely large resource usage [7, 32]). Note that the existing general binary polynomial multiplication computation methods like Karatsuba are not ideal for the mentioned sparse polynomial multiplication, which has already been approved in Reference [26]. As a result, the specific works for sparse polynomial multiplication are not many and the state-of-the-art sparse polynomial multiplication works were merely focusing on designing compact sparse polynomial multiplications at the cost of very long latency [24, 32]. Due to these reasons, there has been no high-speed sparse polynomial multiplier ever reported.

Motivation. Apart from the above-mentioned challenges, we also notice that the existing hardware sparse polynomial multipliers were mostly designed with memory-processing-based structures, and no other variant of hardware accelerators has been proposed. These existing designs' memory-based style limits their potential application to a wide range of environments, e.g., applications where the memory resources are designated for other usages, specifically that the memory resources on modern hardware platforms such as **field-programmable gate array** (**FPGA**) devices (**block RAMs** (**BRAMs**)) are considered as critical components for other computing systems integrated within the same chip. Therefore, a new type of sparse polynomial multiplication accelerator, other than memory-based structures, is also greatly needed.

To fill this research gap, in this article, we propose two novel **H**igh-throughput **S**parse **P**olynomial multiplication **A**ccelerators (HSPA) for HQC and BIKE (generic design), on the FPGA platform. In particular, we have designed one accelerator using a memory-processing-based structure targeting memory resource-abundant applications; while the other Accelerator is designed without memory usage, specifically for the applications where the memory resources are apportioned for other purposes. Overall, we have made three layers of contributions to carry out the proposed work.

- We have presented a detailed mathematical formulation to derive two new computation strategies for the targeted sparse polynomial multiplication in two code-based PQC (HQC and BIKE). The first one is based on a new parallel segment based accumulation (PSA) method, and the second one is originated from a novel **permutating-with-power** (**PWP**)-based approach.
- We have constructed the proposed HSPA through efficient algorithm-to-architecture mapping techniques and novel hardware design strategies. Detailed architectural components are described to illustrate the proposed accelerators' design processes.
- We have provided sufficient complexity analysis, implementation, and comparison to showcase the superior performance of the proposed accelerators. For instance, the proposed high-through accelerator (with memory-processing-based structure) has at least 56.84% and 80.25% less **area-delay product** (**ADP**) than the state-of-the-art design (extended high-speed version) for HQC and BIKE, respectively, for $n = 17,669$ and $n = 12,323$.

Note that the proposed PSA strategy is introduced for memory-processing-based high-throughput computation of the sparse polynomial multiplication, while the novel PWP-based method is proposed to design the targeted polynomial multiplication accelerator with no memory usage.

The rest of this article is organized as follows. The preliminaries are introduced in Section 2. The proposed two algorithms are formulated and proposed in Section 3. Two corresponding

Table 1. Notations Used Throughout This Article

General notations for HQC and BIKE	
n	Size of the polynomial (also security level of the PQC)
ω	Hamming weight (#nonzero coefficients in polynomial)
Notations for deriving the proposed algorithms	
B	Polynomial with ω nonzero coefficients
$P[\cdot]$	The index of nonzero coefficients in B (we set as 16-bits)
D	Dense polynomial
W	Sparse polynomial multiplication product
$\mathbf{rot}(D)$	Circulant matrix (based on coefficients of D)
Notations specifically used in the proposed Algorithm 3 and related Accelerator-I	
t	Number of parallel segments being processed at the same time
l	*Number of rounds to compute the sparse polynomial multiplication
z	Number of parallel segments being involved in the last round
D_{shift}	$P[\cdot]$-matched columns in $\mathbf{rot}(D)$
N_{mem}	Bit-length of the processing word (memory, data flow in the accelerator)
Notations specifically used in the proposed Algorithm 4 and related Accelerator-II	
v	Divided each chunk's bit-length (within one column)
$k = \lceil \log_v n \rceil$	Number of shifting stages
δ'	Number of positions (bits) to be circularly-shifted to obtain the targeted vector
$perm(x', \eta, p)$	Shifting operation (see Algorithm 4)
len_load	Bit-length of the data flow (in the accelerator)

high-throughput accelerators are detailedly introduced in Section 4. FPGA-based Implementation and comparison are presented in Section 5. Conclusions are given in Section 6.

2 Preliminaries

This section gives a brief introduction of HQC and BIKE, as well as the targeted sparse polynomial multiplication and the existing works' limitations.

Notations. This article uses the following notations to represent the items involved in the mathematical derivation for the proposed algorithms (implementation strategies) and accelerators (major ones are listed in Table 1). $\mathbb{F}_2$ is the binary finite field, where all computations in the encryption schemes and proposed algorithm are taking place. $\mathbb{Z}$ denotes the ring of integers. Vectors/polynomials in $\mathcal{R} = \mathbb{F}_2[X]/(X^n + 1)$ with a dimension of n and represented by lower-case bold letters, Matrices are represented by upper-case bold letters. The Hamming weight of a vector, denoted by $\omega(\cdot)$, is defined by the number of the nonzero coordinates in the vector. δ is the minimum number of errors that the decoding algorithm can correct. Also, all the additions/subtractions in index calculations are over the ring $\mathcal{R} = \mathbb{Z}_n$. Interested readers may also refer the details of these notations at [3, 21]. **Note that** the notations used in Algorithms 1 and 2 are ONLY limited to themselves, except those specifically specified in Table 1.

We have also used B, D, and W to represent related polynomials to derive the proposed algorithms, where B is the sparse polynomial while D denotes the dense polynomial and W represents the product of the two polynomials. $P[\cdot]$ denotes the index of nonzero coefficients in B. len_load is the bit-length of the data flow (as well as N_{mem}). The details of other notations can be seen in Table 1.

ALGORITHM 1: HQC.KEM [21]

Setup(1^λ):
1 generate and output the global parameters **param** = $(n, k, \delta, \omega, \omega_{\mathbf{r}}, \omega_{\mathbf{e}})$, k will be the length of the symmetric key being exchanged, typically k = 256;
KeyGen(param):
2 samples $\mathbf{h} \leftarrow \mathcal{R}$, the generator matrix $\mathbf{G} \in \mathbb{F}_2^{k\times n}$ of $\mathcal{C}$;
3 $sk = (\mathbf{x}, \mathbf{y}) \leftarrow \mathcal{R}^2$ such that $\omega = \omega(\mathbf{x}) = \omega(\mathbf{y})$;
4 $pk = (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$;
5 return (pk, sk);
Encapsulate(pk**)**:
6 generate $\mathbf{m} \leftarrow \mathbb{F}_2^k$;
7 derive the randomness $\theta = \leftarrow \mathcal{G}(\mathbf{m})$;
8 generate the cyphertext $\mathbf{c} \leftarrow (\mathbf{u}, \mathbf{v}) = \mathcal{E}.\mathbf{Encrypt}(pk, \mathbf{m}, \theta)$;
9 derive the symmetric key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$;
10 $d \leftarrow \mathcal{H}(\mathbf{m})$;
11 send **(c, d)**;
Decapsulate(pk**, c, d)**:
12 decrypt $\mathbf{m}' = \mathcal{E}.\mathbf{Decrypt}(sk, \mathbf{c})$;
13 compute $\theta' = \mathcal{G}(\mathbf{m}')$;
14 (re-)encrypt $\mathbf{m}'$ to get $\mathbf{c}' \leftarrow \mathcal{E}.\mathbf{Encrypt}(pk, \mathbf{m}', \theta')$;
15 **if** $\boldsymbol{c} \neq \mathbf{c}'$, *or* $\boldsymbol{d} \neq \mathcal{H}(\mathbf{m}')$ **then**
16 | abort;
17 **else**
18 | derive the shared key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$;
19 **end**

In July of 2022, HQC and BIKE were selected as the NIST fourth-round PQC standardization candidates [2]. We give their brief introductions below.

HQC. HQC is an efficient encryption scheme based on the hardness of a decision version of the Syndrome Decoding on structured codes. It is IND-CPA (Indistinguishability under Chosen-Plaintext Attack) secure and allows to get a hybrid encryption scheme to achieve IND-CCA2 (Indistinguishability under Adaptive Chosen Ciphertext Attack) with a precise upper bound for **Decryption Failure Rate** (**DFR**) analysis. A decodable code $\mathcal{C}[n, k]$ and a random double-circulant $[2n, n]$ code are used in HQC. Algorithm 1 represents the **key encapsulation mechanism** (**KEM**) version of HQC, where $\mathcal{G}(\cdot), \mathcal{H}(\cdot), \mathcal{K}(\cdot)$ are random oracles realized by SHAKE function. For more details of HQC and the public key version, interested readers may refer to Reference [21].

BIKE. BIKE is a KEM based on QC-MDPC code[3]. The QC-MDPC code was introduced to reduce the key size as well as the resource usage during calculation. BIKE is proven to be IND-CPA secure under assumptions of the hardness of $\text{QCSD}_{r,t}$ (Quasi-Cyclic Syndrome Decoding) and $\text{QCCF}_{r,w}$ (Quasi-Cyclic Codeword Finding), and IND-CCA secure under the assumptions above with the correctness of decoder [3]. Algorithm 2 describes the BIKE algorithm (KEM version). **H**, **K**, **L** denote random oracles instantiated by SHAKE-based PRNG (Pseudorandom Number Generator) and SHA2-384. Interested ones may read [3] for more details.

Security Levels. There are three security levels for both HQC and BIKE, respectively: hqc-128, hqc-192, and hqc-256 [21] for HQC, and level 1, level 3, level 5 for BIKE [3]. The proposed algorithms and accelerators (HSPA) are applicable to the parameter sets of all these security levels, for both code-based HQC and BIKE.

ALGORITHM 2: BIKE [3]

KeyGen(param = (n, ω, t, l)):
1 samples $(\mathbf{h}_0, \mathbf{h}_1) \leftarrow \mathcal{R}^2$, with $\omega(\mathbf{h}_0) = \omega(\mathbf{h}_1) = \omega/2$;
2 samples $\sigma \leftarrow \{0,1\}^l$ uniformly at random;
3 compute $\mathbf{h} \leftarrow \mathbf{h}_1\mathbf{h}_0^{-1}$;
4 return $sk = (\mathbf{h}_0, \mathbf{h}_1, \sigma)$ and $pk = \mathbf{h}$;
Encapsulate(pk):
5 samples $\sigma \leftarrow \{0,1\}^l$ uniformly at random;
6 compute $(\mathbf{e}_0, \mathbf{e}_1) \leftarrow \mathbf{H}(m)$;
7 compute $C = (\mathbf{c}_0, \mathbf{c}_1) \leftarrow (\mathbf{e}_0 + \mathbf{e}_1\mathbf{h}, m \bigoplus \mathbf{L}(\mathbf{c}_0, \mathbf{c}_1))$;
8 compute $K \leftarrow \mathbf{K}(m, C)$;
9 return (C, K);
Decapsulate(pk, C):
10 compute syndrome $\mathbf{s} \leftarrow \mathbf{c}_0\mathbf{h}_0$;
11 decrypt $(\mathbf{e}'_0, \mathbf{e}'_1), \perp \leftarrow$ **decoder**$(\mathbf{s}, \mathbf{h}_0, \mathbf{h}_1)$;
12 compute $m' \leftarrow \mathbf{c}_1 \bigoplus \mathbf{L}(\mathbf{e}'_0, \mathbf{e}'_1)$;
13 **if** $H(m') \neq (e'_0, e'_1)$ **then**
14 | $K \leftarrow \mathbf{K}(\sigma, C)$;
15 **else**
16 | $K \leftarrow \mathbf{K}(m', C)$;
17 **end**
18 return K;

High Dimensional Sparse-Dense Polynomial Multiplication. Multiplication of a dense polynomial and a sparse polynomial (most of the coefficients are ‘0’s) over the ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n + 1)$ is performed repeatedly in both HQC and BIKE, and thus determines the overall performance of a cryptoprocessor from this standpoint. However, the large size of operands and the sparsity of the polynomial involved, could dreadfully influence the multiplication efficiency when executing traditional strategies like the schoolbook method (or even sub-quadratic complexity approaches like Karatsuba algorithm [14] and Toeplitz Matrix-Vector Product [8]) and make them inappropriate for sparse-dense polynomial multiplication involved in code-based cryptography (HQC and BIKE). Note that since both KA and TMVP are not built on the sparsity of the sparse polynomial multiplication, the deploying of these two methods has been proved as inefficient, as shown in [26]. Briefly speaking, for a sparse polynomial multiplication of $n = 57,637$ with sparsity of $\omega = 149$ (149 nonzero values), the latest work of Reference [32] can achieve a complexity of only $\mathcal{O}(n\omega)$, which is weight better the newest subquadratic complexity presented in works such as [15, 23]. As a result, in this work, we do not deploy the subquadratic complexity approach in our proposed work.

Limitation of the Existing Hardware Designs. Almost all the existing designs, though not many, have decided to use the compact design style (i.e., memory-based processing) to obtain efficient implementation for sparse polynomial multiplication (but involves very long latency cycles) [24, 32]. For high-throughput operations, however, no such accelerator has ever been reported.

Meanwhile, apart from the memory-based structures, there have not been other types of accelerators, thus limiting the potential applications for a wide range of environments. To fill the above mentioned research gap, we hereby propose two new accelerators for the targeted sparse polynomial multiplication for high-performance operations, i.e., one based on memory-processing-based structure, while the other does not need memory usage.

3 Algorithmic Operation

This section gives the detailed process to obtain the desired high-performance (high-throughput) implementation strategies (algorithms) for the targeted sparse polynomial multiplication in both HQC and BIKE.

3.1 Definitions and Sparse Polynomial Multiplication

Definition 1. We first define the sparse-dense polynomial multiplication for the targeted HQC and BIKE as

$$W = BD \bmod f(x) = DB \bmod f(x), \tag{1}$$

where B is a polynomial with only ω nonzero coefficients while D is the dense polynomial. More specifically, $f(x) = x^n + 1$, $W = \sum_{i=0}^{n-1} w_i x^i$, $B = \sum_{i=0}^{n-1} b_i x^i$, and $D = \sum_{i=0}^{n-1} d_i x^i$. Meanwhile, b_i, d_i, and w_i are (binary) 1-bit values in the ring $\mathbb{F}_2/(x^n + 1)$, respectively.

Then, we have

$$\begin{aligned} W = &\left(b_0 + b_1 x + \cdots + b_{N-1}x^{N-1}\right)\left(d_0 + d_1 x + \cdots + d_{N-1}x^{N-1}\right) \bmod f(x) \\ &= \left(d_0 + d_1 x + \cdots + d_{n-1}x^{n-1}\right) b_0 \bmod f(x) \\ &+ \left(d_0 + d_1 x + \cdots + d_{n-1}x^{n-1}\right) b_1 x \bmod f(x) \\ &+ \cdots \cdots \cdots \\ &+ \left(d_0 + d_1 x + \cdots + d_{n-1}x^{n-1}\right) b_{n-1}x^{n-1} \bmod f(x), \end{aligned} \tag{2}$$

which can be further derived as (since $x^n + 1 \equiv 0$, thus $x^n \equiv -1$ is substituted into (2))

$$\begin{aligned} W = &\left(d_0 + d_1 x + \cdots + d_{n-1}x^{n-1}\right) b_0 \\ &+ \cdots \cdots \cdots \\ &+ \left(d_0 x^{n-1} + d_1 + \cdots + d_{n-1}x^{n-2}\right) b_{n-1}, \end{aligned} \tag{3}$$

where we can have

$$\begin{aligned} w_0 &= d_0 b_0 + d_{n-1} b_1 + \cdots + d_1 b_{n-1}, \\ &\cdots \cdots \cdots \\ w_{n-1} &= t d_{n-1} b_0 + d_{n-2} b_1 + \cdots + d_0 b_{n-1}, \end{aligned} \tag{4}$$

where each w_i ($0 \leq i \leq n-1$) can be obtained as (transferred into the matrix-vector product form)

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{bmatrix} = \begin{bmatrix} d_0 & d_{n-1} & \cdots & d_1 \\ d_1 & d_0 & \cdots & d_2 \\ \vdots & \vdots & \cdots & \vdots \\ d_{n-1} & d_{n-2} & \cdots & d_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{5}$$

Definition 2. For a polynomial $D = (d_1, d_2, \ldots, d_{n-1}) \in \mathbb{F}_2^n$, we define its circulant matrix $\mathbf{rot}(D)$ as

$$\mathbf{rot}(D) = \begin{bmatrix} d_0 & d_{n-1} & \cdots & d_1 \\ d_1 & d_0 & \cdots & d_2 \\ \vdots & \vdots & \ddots & \vdots \\ d_{n-1} & d_{n-2} & \cdots & d_0 \end{bmatrix} \in \mathbb{F}_2^{n \times n}. \tag{6}$$

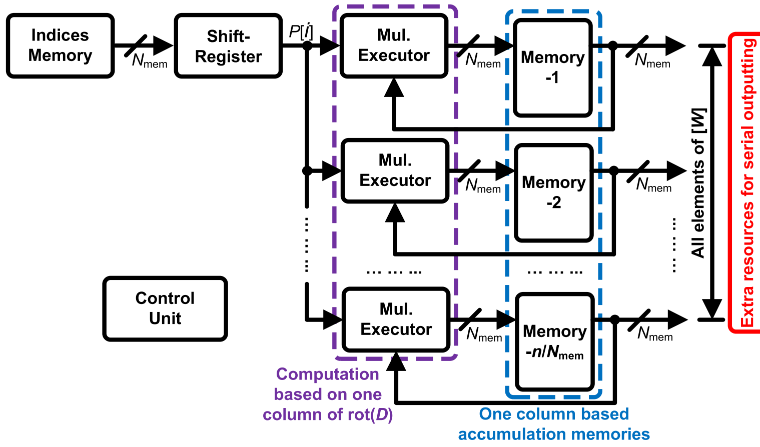

Fig. 1. The memory-processing-based high-throughput sparse polynomial multiplier (extended from Reference [32]). Mul.: multiplication. $N_{\text{mem}} = 128$, refers to the memory processing word-length. The memories and multiplication executors are paired to process one column-wise related accumulation that the output elements of $[W]$ are produced in parallel (extra resources are needed to transfer these parallel outputs in serial for outputting, as highlighted in red). A control unit is used to handle the operation of the accelerator.

3.2 Proposed Algorithm-I (Implementation Strategy-I)

PSA Method **Motivation of the Proposed Computation Strategy.** For high-throughput (high-performance) computation of sparse polynomial multiplication, it is desirable that the matrix-vector product of (6) can be obtained through column-based computation, i.e., connecting with Definition 2, we can rewrite (5) as

$$\begin{aligned} [W]^{n\times 1} &= \mathbf{rot}(B) \times [D]^{n\times 1} = \mathbf{rot}(D) \times [B]^{n\times 1} \\ &= \sum_{i=0}^{n-1} \mathbf{rot}(D)(:, i)[B] \\ &= \sum_{i=0}^{\omega-1} \mathbf{rot}(D)(:, i)P[i]. \end{aligned} \tag{7}$$

where $\mathbf{rot}(D)(:, i)$ denotes the ith column of $\mathbf{rot}(D)$ and $P[i]$ denotes the non-zero element/index within vector $[B]^{n\times 1}$ (ith position). One can easily observe that the final output $[W]$ can be obtained through the accumulation of non-zero coefficients $P[i]$-matched columns of $\mathbf{rot}(D)(:, i)$ (in total ω columns).

It is noted that the memory-processing-based high-throughput sparse polynomial multiplier has not been reported in the literature. Nevertheless, we can follow the existing compact design format to design a high-speed accelerator, e.g., following the very recent one of Reference [32]. In this case, as shown in Figure 1, $\lceil n/N_{\text{mem}} \rceil$ pairs of multiplication executors and memories are connected in a loop format to process one column-wise related accumulation based on (7), i.e., a column (n-bit) of $\mathbf{rot}(D)$ is divided into $\lceil n/N_{\text{mem}} \rceil$ segments and each segment is processed by one memory and its paired multiplication executor. Finally, all the output coefficients of $[W]$ are produced in parallel after the required number of accumulations.

This type of design (Figure 1), however, involves huge (inefficient) hardware resource usage, extra processing cycles, and complicated control setup. (i) One single column of the matrix $\mathbf{rot}(D)$ involves at least 12,323 bits (for BIKE [3], or at most 57,637 bits for HQC [21]), which requires

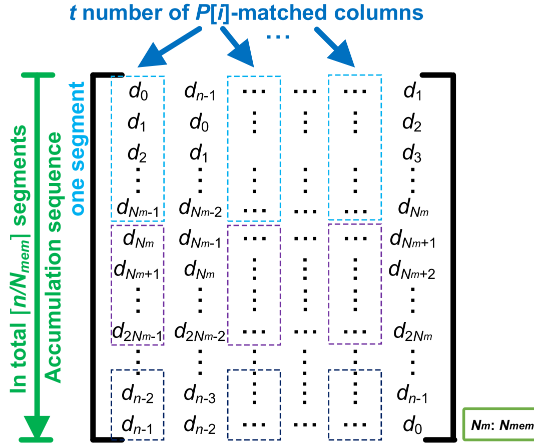

Fig. 2. Illustration of the proposed PSA method, where the positions of t number of $P[i]$-matched columns are determined by the actual situation. Each column is decomposed into $\lceil n/N_{mem} \rceil$ segments, and there will be t parallel segments being processed in the same cycle. These parallel segments will be multiplied with respective $P[i]$ to be added together (in the same colored dash boxes), and the result is accumulated with the following parallel segments related results (as demonstrated by the accumulation sequence).

hundreds of BRAMs to store/process all the elements of that certain column of **rot**(D) at the same time for high-throughput processing. Besides that, extra resources are needed to transfer the final output results in serial for practical application. (ii) The final output results (n bits) are available in parallel, which takes extra clock cycles to deliver them out in serial for practical usage (due to the size of n, it is almost impossible to process these data in parallel with other components in the further PQC processor). (iii) Due to the circularly-shifted features of the elements between different columns in **rot**(D) in (6), it is complicated to generate proper control signals for each memory to read/write the correct data for column-wise based accumulation (because of the randomness of $P[i]$, the matched column within **rot**(D) is hard to predict in advance).

Proposed Computation Strategy. Based on the above consideration, we propose to use a new strategy for memory-processing-based computation for sparse polynomial multiplication. Specifically, we notice that the typical processing word-length in the cryptoprocessor is much larger than the bit-length of one index $P[i]$ (e.g., could be 128-bit, as shown in the one of Figure 1), i.e., multiple indices can be conveyed in one typical processing value of the accelerator. Meanwhile, we also notice that the complete one-column-based accumulation in Figure 1 requires extra resources to transfer these parallel values into serial outputting. Finally, we decided to have the proposed computation strategy as follows: (i) fully utilize all the indices (t numbers) contained in one processing word of the accelerator, i.e., these indices will be multiplied with the matched columns in **rot**(D) in parallel; (ii) evenly decompose these related columns (in **rot**(D)) into $\lceil n/N_{mem} \rceil$ segments (except for the last segment, due to the prime number of n); (iii) the indices will be multiplied with these decomposed segments, respectively, in sequential order, i.e., from the first to the last; (iv) all the multiplied results (in the same level) are added together to be stored in the memory, while the following results will be carried out in an accumulation format. The overall procedure of this computation strategy is illustrated in Figure 2, and we name it as "PSA" method due to its involved arithmetic feature. Based on the illustration in Figure 2, we can have the proposed sparse polynomial multiplication implementation strategy (for HQC and BIKE) in Algorithm 3.

Some Details of Algorithm 3. In Algorithm 3, the calculation is divided into two conditions: the first l rounds corresponding to Lines 8–11, where all the parallel segments are involved; and

ALGORITHM 3: Proposed PSA Originated Implementation Strategy for Sparse Polynomial Multiplication of HQC and BIKE

```
   Input   : G and D are binary polynomials.
   Output  : W = GD mod (x^n − 1).
   Initialization step
 1 make ready the inputs G and D.
 2 record the indices of nonzero elements in G into P (P contains all P[i]).
   Main step
 3 W = 0;
 4 D_shift = [t];
 5 l = ⌊ω/t⌋;
 6 z = ω mod t;
 7 for i = 0 to l do
 8 |   if i < l then
 9 |   |   for j = 0 to t − 1 do
10 |   |   |   D_shift[j] = rot(D)[:,P[t · i + j]];
11 |   |   end
12 |   else
13 |   |   for j = 0 to t − 1 do
14 |   |   |   if j ≤ z − 1 then
15 |   |   |   |   D_shift[j] = rot(D)[:,P[t · i + j]];
16 |   |   |   else
17 |   |   |   |   D_shift[j] = 0;
18 |   |   |   end
19 |   |   end
20 |   end
21 |   W = W + D_shift[0] + D_shift[1] + · · · + D_shift[t − 1];
22 end
   Final step
23 Store the coefficients of output W;
```

the last ($(l+1)$th) round, where only some ($\omega \bmod t$) of the segments are operating. In each round of the operation, t different columns (denoted as $D_{shift}[i]$) in **rot**(D) will be loaded at the same time and then be summed together with W obtained in the previous round to calculate the answer. Note that all of t columns loaded ($D_{shift}[0], D_{shift}[1], \ldots D_{shift}[t-1]$) are used during the first l rounds, and only ($\omega \bmod t$) columns are involved in the last round (which means that the rest $t - (\omega \bmod t)$ elements in D_{shift} are set as zero). Finally, as the accumulation is done in a sequential format through memory-based processing, no extra resources are needed for final output data transferring (e.g., as that in Figure 1).

Extra Consideration. As the operational time of the proposed computation strategy is determined by ω, t, and n, and the number of cycles for each segment to load a column from memory is consistent every time, the proposed Algorithm 3 is thus constant-time operated. The following hardware accelerator design in Section 4 also reflects this feature.

3.3 Proposed Algorithm 2 (Implementation Stategy-II): Permutation-with-Power (PWP)-Based Strategy

Motivation. Although Algorithm 3 can also be implemented using a memory-less design, the area usage for the t parallel storing components (especially when t is large) for the polynomials/

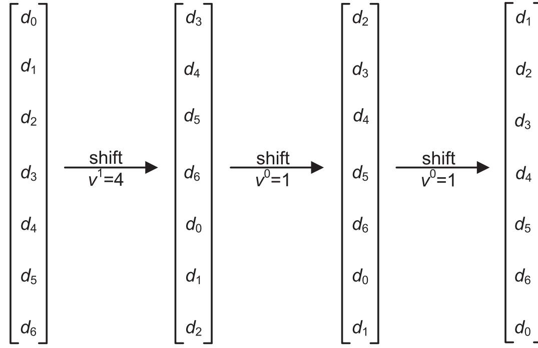

Fig. 3. Example of shifting a 7-bit long polynomial by six positions (circularly downward) using the proposed computation strategy in Definition 3, where we get $k = \lceil \log_v n \rceil = 2$, and $v^2 = 16$ (thus $v^1 = 4$ and $v^0 = 1$).

columns as well as the component where W is stored, would be humongous. Furthermore, since we need to circularly shift the polynomial (column) according to different random numbers of $P[i]$, the related logic circuits to realize the shifting would be very complicated, which would lead to a significantly long critical path in the design and thus bring the frequency down to an extremely low level. Based on these considerations, we propose a novel implementation strategy (algorithm) for memory-less based processing.

Proposed Computation Strategy. From (6), one can also observe that all the columns in $\mathbf{rot}(D)$, which are $\mathbf{rot}(D)(:,0)$, $\mathbf{rot}(D)(:,1)$, ..., $\mathbf{rot}(D)(:,n\text{-}1)$, can be derived from each other by simply circularly shifting the difference between the indices, e.g., circularly-shifting the elements of $\mathbf{rot}(D)(:,0)$ by one position (downward) to obtain $\mathbf{rot}(D)(:,1)$. Thus the calculation could be developed to a shift-addition procedure: first derive a specific column of $\mathbf{rot}(D)(:,i)$ according to the index $P[i]$, and then add it to the final result.

Definition 3. When shifting the columns, we first divide the column into chunks with a length of v bits. Let us define again $k = \lceil \log_v n \rceil$ be the number of shifting stages, where we at most operate $v - 1$ times of shifting. Then, we can represent the number-to-shift using $v^0, v^1, \ldots, v^k$, and shift the column by $v^k, v^{(k-1)}, \ldots, v^0$ bits (circularly), each with a different number of times to derive the desired column.

Example. Here, we give an example to initiate our proposed computation strategy. As shown in Figure 3, let δ' be the number we need to circular-shift obtain the vector (rightmost) from the vector at the leftmost. As the length of the shown polynomial $\mathbf{u}$ is 7 ($n = 7$), we can get δ' as 6. Following the above Definition 3, we can choose v as 4 (v can be any powers of 2 integers). In this case, we get $k = \lceil \log_v n \rceil = 2$, and $v^2 = 16$, $v^1 = 4$, and $v^0 = 1$. We thus can circularly shift six bits to obtain the vector (rightmost) from the one at the leftmost, i.e., by first shifting four bits once, and then shifting one bit twice.

Permutating-with-Power (PWP). The above example can be used to derive a novel PWP-based method to calculate the targeted sparse polynomial multiplication, i.e., for any nonzero index $P[i]$-matched column vector within matrix $\mathbf{rot}(D)$, one can obtain it through the circular-shifting of a previous column vector (matched with $P[i-1]$), where the positions to be circularly-shifted are in the format of $v^k, v^{(k-1)}, \ldots, v^0$. Due to the definition of the positions to be shifted (in the power of v format), we define this method as "PWP". In the following part of this article, we use $perm(x', \eta, p)$ to represent the shifting operation involved within the proposed PWP-based method, where x' is the polynomial to be shifted, η is the number of shifting, and p is the number of bits (or

ALGORITHM 4: Proposed PWP-Based Strategy (Algorithm) for High-Performance Computation of Sparse Polynomial Multiplication (Applicable to Both HQC and BIKE)

Input : D is a dense binary polynomials, $P[i]$ is the indices of non-zero coefficients in the sparse polynomial B, and v is the chunk size;
Output : $W = BD \bmod (x^n + 1)$;
Initialization step
1 Load input coefficients in serial.
2 $D' = D, W = 0$;
Main step
3 **for** $i = 0$ *to* $\omega - 1$ **do**
4   **if** $i = 0$ **then**
5     $\delta' \leftarrow P[i]$;
6   **else**
7     $\delta' \leftarrow P[i] - P[i-1]$;
8   **end**
9   $k \leftarrow \lceil log_v n \rceil$;
10   **for** $j = 0$ *to* $k - 1$ **do**
11     $\eta \leftarrow \delta' \mid v$;
12     $\delta' \leftarrow \delta' \bmod v$;
13     $D' = perm(D', \eta, v^{k-j-1})$; // $p = v^{k-j-1}$
14   **end**
15   $W = W + D'$;
16 **end**
Final step
17 Serially deliver all the coefficients of output W;

positions) to be shifted each time. For instance, in the example given above, the shiftings executed are $perm(u, 1, 4^1)$ and $perm(u, 2, 4^0)$. We can finally derive the proposed computation process based on these definitions, as shown in Algorithm 4.

Some Details of Algorithm 4. According to Algorithm 4, one column's shifting operation (by δ' positions) is completed from Lines 4 to 14, where δ' is the difference between the indices of two non-zero elements in $[B]$. In the example mentioned above, where $\delta' = 6$, $v = 4$, and $k = 2$, we can complete the shifting in two iterations, as shown in Lines 10–13 ($j = 0$ and $j = 1$). In the first iteration ($j = 0$), $\eta = 6 \mid 4 = 1$, so we shift the polynomial by 4-bits' positions, and meanwhile $\delta' = \delta' \bmod v = 6 \bmod 4 = 2$; then, in the second iteration, we have $\eta = 2 \mid 1 = 2$, so we just shift the polynomial by 1-bit twice to obtain the final result.

Constant-Time Consideration. The constant-time computation of the proposed PWP-based method basically determines if the proposed algorithm can be used for practical applications. We seriously take this into consideration and the following description gives a detailed introduction to how the proposed computation strategy is operated in a constant-timing manner.

Overall, we can divide Algorithm 4 into different k stages, where each stage we need to circularly shift v^i ($i = 0, 1, \ldots, k-1$) bits. It is obvious that the number of shifting times at each stage may vary (e.g., shown in the example above), and it will cause hindrance for constant-time operation. To solve this problem, for the proposed Algorithm 4, we propose to execute at most $v - 1$ times of shifting at the ith stage, i.e., we can execute the circular-shifting p times and wait for $v-1-p$ times to make the execution time at all stages equally the same. Therefore, the proposed Algorithm 4 is time-constant with a time complexity of $\omega(v-1)k$. The corresponding hardware accelerator design will also cover this aspect.

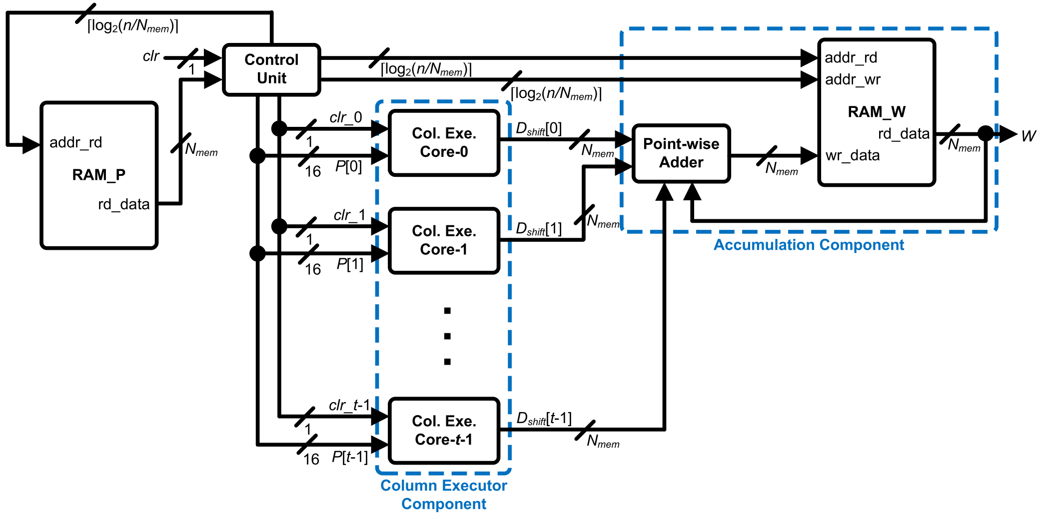

Fig. 4. Overview of the proposed Accelerator-I. Col. Exe.: Column Executor. Notions follow Algorithm 3 and Table 1. addr: address; rd: read; wr: write. N_{mem}: processing bit-length of the memory.

4 HSPA: Proposed Hardware Accelerators

We follow the proposed implementation strategies to design the hardware accelerators (HSPA). Specifically, Algorithm 3 is mapped into an efficient high-throughput memory-processing-based accelerator, while Algorithm 4 is designed into a novel memory-less high-throughput accelerator.

4.1 Proposed Accelerator-I: High-Throughput Memory-Processing-Based Accelerator

As shown in Figure 4, the proposed accelerator contains three major parts, namely the **Column Executor** (**CE**) Component, Accumulation Component, and **Control Unit** (**CU**). The inputs to the Accelerator-Include: the coefficients of D (already stored in the memories inside of the CE Component); the indices of non-zero elements in B ($P[i]$, which is assumed to be generated by a uniformly distributed sampler and then stored in the memory); and the speed choice t. While the outputs of the accelerator are mostly the product polynomial W and an indicator signal "*done*". The structural details of the proposed accelerator are described below.

The Column Executor (CE) Component. The CE Component is responsible for loading coefficients of D from memory and the formation of the segments based on the read coefficients and the non-zero indices $P[i]$ coming from the CU. It takes the indices $P[i]$ and the clear signal "clr_i" as its inputs and delivers out the columns $D_{shift}[i]$, corresponding to Lines 7–20 in Algorithm 3. The CE Component consists of t parallel CE Cores, where each core corresponds to one column ($D_{shift}[0], D_{shift}[1], \ldots, D_{shift}[t-1]$) and the details of the CE Core are shown in Figure 5.

As shown in Figure 5, three sub-components, one sub-control cell, one memory (where one copy of coefficients of D are stored), and a Column Former (which reads the coefficients from memory and then forms the segments), are involved in the CE Core. The sub-control cell determines the address and the position of the starting coefficient (of the column to be read) according to the incoming index $P[i]$. Then, the Column Former will take the data read from memory and put the coefficients (those should be involved) in the segment, while storing the rest for future use. When a segment (with the length of N_{mem}) is formed, it will output the segment to the Accumulation Component. Note the CE Core is deactivated when the clear signal ("clr") is '1' so that only a correct

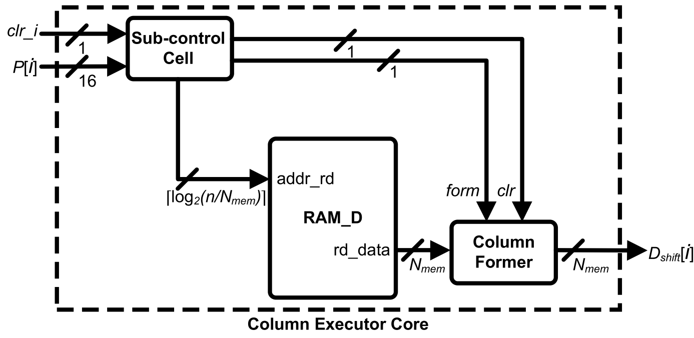

Fig. 5. Internal structure of the CE Core.

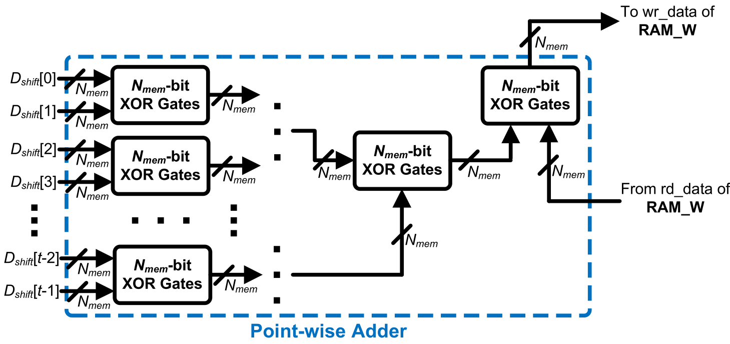

Fig. 6. Internal structure of the Point-wise Adder.

number of CE Cores are working during the last round of the sparse polynomial multiplication process (with some of the CE Cores shut down, following Algorithm 3).

The Accumulation Component. The Accumulation Component is in charge of the accumulation of columns $D_{shift}[i]$ with the result W from the previous round during the multiplication process, as well as outputting and storing the final result in the memory, as is shown in Line 21 of Algorithm 3. After receiving the segments of different columns from the CE Component, the Accumulation Component sums up all the segments together with W read from memory in the Point-wise Adder and then writes the result back to the memory to store the newly obtained W for future calculation/usage. The CU decides the reading/writing address of the resulting memory during the accumulation. Note here we realize the Point-wise Adder by implementing a simple segment-length XOR-tree-based combinational logic circuit (with t number of N_{mem} inputs, see Figure 6) since all the additions are bit-wise XOR operations under $GF(2)$. When the polynomial multiplication is done, the final result W will be stored in the memory (RAM_W) and can be read out serially with the word-length of N_{mem} (no extra resources are needed in this case).

The Control Unit (CU). The CU is responsible for controlling the operations of the other two components throughout the polynomial multiplication process. A series of control signals are generated, such as "*clr_i*" (which activates/deactivates the CE Cores), the write-enable signal for the accumulation memory, and calculating signals for the reading/writing addresses of all the memories. Also, the CU takes in the data from memory (RAM_P) stored with all the non-zero indices, and then distributes them to corresponding CE Cores. Overall, CU is realized by a **finite state machine (FSM)** with five different states as shown in Figure 7, namely *Reset*, *Rd_idx* (read indices), *Full* (the first l rounds according to Algorithm 3), *Last round*, and *Done*, respectively, each

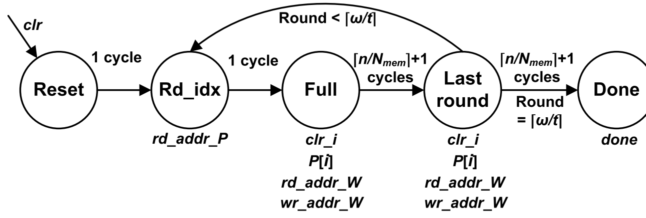

Fig. 7. Major states of the FSM inside the CU, where *Rd_idx* refers to the stage of reading indices; *Full* refers to the first l rounds according to Algorithm 3. addr: address; rd: read; wr: write.

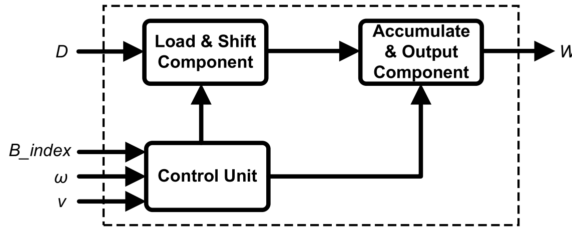

Fig. 8. Overview of the proposed Accelerator-II.

with different control signals generated to navigate the operations of the accelerator. The FSM switches between different stages multiple times to execute the whole polynomial multiplication process smoothly.

Work Flow of the Accelerator. When a "*clr*" signal is received from outside, the CU enters the *Reset* state where all the registers and components are reset. Then, it takes the accelerator $\lceil t/(N_{mem}/16)\rceil$ cycle(s) to read non-zero indices from the memory and distribute the indices to the parallel CE Cores (note in the implementation we use $t = 8$ and $N_{mem} = 128$ that this step needs only 1 cycle). After that, the accelerator enters the *Full* stage where all parallel CE Cores are activated for $\lceil n/N_{mem}\rceil + 1$ cycles to load whole columns from memory while the Accumulation Component executes the accumulation. For the first $\lfloor \omega/t \rfloor$ rounds, the accelerator goes back to state *Rd_idx* to accumulate the first $\lfloor \omega/t \rfloor \cdot t$ non-zero columns of $\mathbf{rot}(D)$. For the last round, the accelerator enters state *Last round* to load and accumulate the last $\omega \bmod t$ columns of $\mathbf{rot}(D)$, which also costs $\lceil n/N_{mem}\rceil + 1$ cycles and the final result of W is stored in the memory by the time the accelerator completes the accumulation. The proposed accelerator finally moves to state *Done* where an indicator "*done*" is generated to indicate that the polynomial multiplication is completed.

4.2 Proposed Accelerator-II: Novel High-Throughput Memory-Less Hardware Accelerator

Following the procedure of the proposed Implementation Strategy-II (Algorithm 4), we further present the hardware structure of the proposed Accelerator-II in this subsection. As shown in Figure 8, the proposed accelerator contains three major components, namely the **Load and Shift** (**LS**) Component, **Accumulation and Output** (**AO**) Component, and CU. The inputs are the coefficients of D, the indices of non-zero elements in B, and the speed choice/chunk size v; while the outputs are the product polynomial W and an indicator signal "*done*". The structural details of Accelerator-II are provided below.

The LS Component. The LS Component is responsible for loading the coefficients of D to the accelerator and shifting coefficients (executing *perm*() function in Algorithm 4) to obtain the desired column in $\mathbf{rot}(D)$ while executing the multiplication, corresponding to Lines 10–13 in

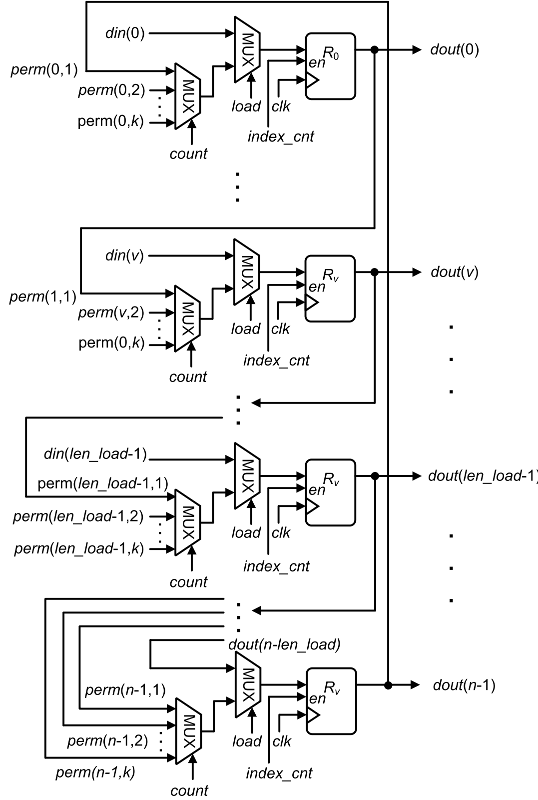

Fig. 9. The internal structure of the LS Component for $v = 2$. The $perm(\cdot)$ function here is a simplified form for the one in Algorithm 4. din: data-in; dout: data-out.

Algorithm 4. Figure 9 shows the internal structure of the LS Component, where $perm(i, j)$ denotes the ith element in the polynomial obtained by shifting j bits. The inputs of the first MUX are the corresponding bits in the polynomials after being shifted by v^0, v^1, …, v^k bits, which are also outputs of other registers, and are selected by a signal "*count*" which will count up by 1 every $sel_cnt = 2^{\lceil \log_2 k \rceil}$ cycles. The second MUX is responsible for determining if the LS Component is taking in or shifting the coefficients of D, which is controlled by the signal "*load*". When the LS Component is loading in the inputs, the first *len_load* registers will take the input while the others will take the output of the register *len_load* in front of it. During the shifting operation, the registers will take the bit from the proper position according to *perm* function. The "*en*" signal (when shifting the coefficients) is connected to the control signal "*index_cnt*", which is used to determine if the shifting should be executed according to the calculated η. The output of the LS Component is the shifted polynomial D' and will be delivered to the AO Component directly.

The AO Component. The AO Component is responsible for accumulating the shifted polynomial D' with the result W during calculation and outputting the final result, corresponding to Lines 15–17 in Algorithm 4 (shown in Figure 10). The accumulation is realized by the addition of the coefficients coming out of the IS Component and the coefficients stored in the registers. Here, the addition is depicted by XOR gates since it is a bit-wise addition under $GF(2)$. When executing the accumulation, all the registers will take the output from the XOR gates. When outputting, the registers will take the output from the register of *len_load* position before it as its input (e.g.,

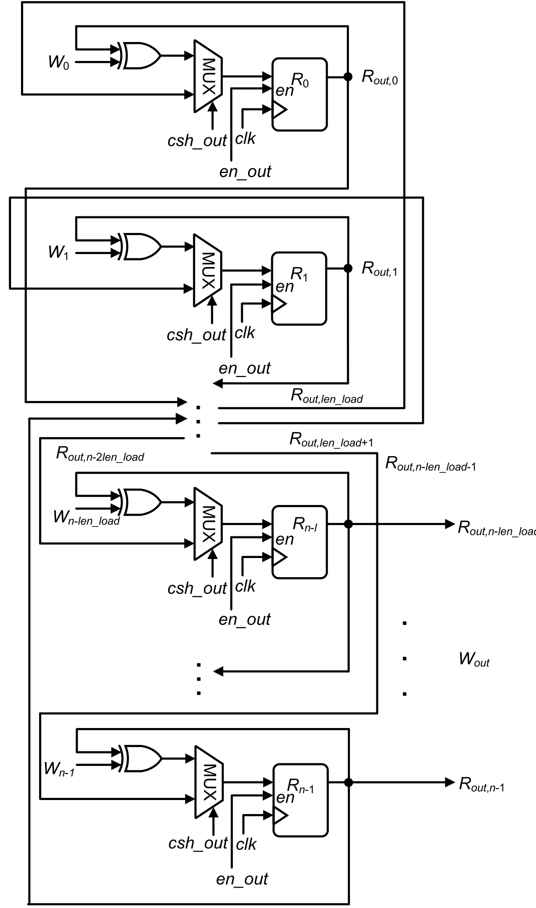

Fig. 10. The internal structure of the AO Component.

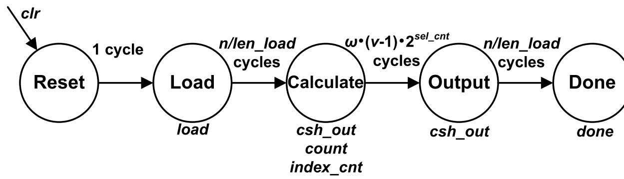

Fig. 11. Major stages of the FSM, where related control signals produced during each stage can be found in Figs. 9 and 10, respectively.

R_{n-1} will take in R_{n-len_load}'s output). And the output of the final *len_load*-length registers will be the overall output of the accelerator to be delivered out. Here *len_load* is set as 128 in the final implementation, which is the same as N_{mem} in Accelerator-I (Figure 4). Finally, whether the AO Component is accumulating or outputting is controlled by the signal "*csh_out*".

The Control Unit (CU). The CU is responsible for controlling the operations of the other two components throughout the multiplication process by generating control signals such as "*csh_out*", "*count*", "*load*", and "*index_cnt*". This unit is also realized by an FSM with five different states, as shown in Figure 11. Different control signals are generated during those states to determine

related operations such as input loading, shifting, and output delivering. CU is also responsible for calculating parameters for the *perm* function like k, η, and δ' (it takes the indices as the input), i.e., CU decides how the IS Component should shift (e.g., v^0, v^1, ..., or v^k) for each input according to the calculated parameters.

Overall Operation of the Accelerator. The workflow of the proposed Accelerator-II can be depicted through five consecutive stages, namely *Reset*, *Load*, *Calculate*, *Output*, and *Done*, respectively. After receiving a "1" from the "clr" port, the accelerator enters the *Reset* state, where all the components are cleared and reset to prepare for the coming calculation. Then, in the next clock cycle, the proposed accelerator moves into the *Load* state that all the coefficients of polynomial D are loaded into the IS Component, which lasts for $\lceil n/len_load \rceil$ clock cycles. Having completed the loading process, the proposed accelerator enters the *Calculate* state and stays for $\omega \cdot (v-1) \cdot k$ cycles to operate the circular shiftings and the accumulation of ω columns of B in the IS as well as the AO Component. In order to deliver the obtained polynomial, it takes the proposed design another $\lceil n/len_load \rceil$ cycles to output all the coefficients serially in a data length of len_load bits. The proposed Accelerator-II will finally move to the *Done* state, which indicates the completion of the whole multiplication process and outputs the indicator signal "*done*".

Constant-time Operation. During the process of circular shifting of the column, the number of execution of shiftings may vary at each stage. For example, in the example given above, to shift 6 bits, the proposed Accelerator-II needs to shift 4 bits once and 1 bit twice. The inconsistency in the number of execution makes the design operate in a non-constant time. To solve this problem, we set the time for each stage to $v-1$ cycles, controlling the IS component to shift p times and not to work $v-1-p$ cycles. In this way, we can avoid the variation of the clock cycles required for each stage and thus have a constant calculation time of $\omega \cdot (v-1) \cdot k$ cycles.

5 Implementation and Comparison

Complexity Analysis. The proposed Accelerator-I contains $t+1$ copies of memories (with a depth of $\lceil n/N_{mem} \rceil$), $2 \cdot t$ N_{mem}-bit long registers, t 128-bit XOR gates, and some logic circuits for the CU and logic conditional statements. For each round of calculation, it takes $\lceil t/8 \rceil$ cycle(s) to load the non-zero indices and $\lceil n/N_{mem} \rceil + 1$ cycles to load the whole columns and execute the point-wise addition. The whole accelerator has a computation latency of $\omega \cdot (\lceil n/N_{mem} \rceil + 2)$ cycles, and $\lceil n/N_{mem} \rceil$ cycles for outputting (reading stored W out of the memory). Both timing complexity and area usage of the proposed accelerator are determined mostly by t, n, and ω (constant-time operation). Additionally, t sub-control cells are needed for the proposed accelerator.

Meanwhile, the proposed Accelerator-II contains $2n$ registers, $2n$ 2-to-1 MUXes, n k-to-1 MUXes, n XOR gates, and some logic circuits. When loading and outputting, both the LS Component and AO Component work in a serial fashion with the data flow of len_load bits. The whole accelerator has a computation latency of $\omega \cdot (v-1) \cdot k$ cycles, and $\lceil n/len_load \rceil$ cycles for input/output. The complexities of the proposed accelerator under different parameter settings are determined mostly by the specific n and v values. Additionally, a control unit is needed for the proposed accelerator.

Implementation on the FPGA Platform. We have implemented the proposed two accelerators for different security levels of HQC and BIKE, on the FPGA platform. The experimental setup is as follows: (i) the proposed designs were described in VHDL[1] and (ii) the correctness of the proposed accelerators with all security levels of HQC and BIKE was tested using Modelsim; (iii) the tested designs were implemented on a high-performance UltraScale+ XCZU9EG-2FFVB FPGA (with enough resources) using Vivado 2020.2 (after place and route) since the proposed accelerators are designed for high-speed applications; (iv) performance of the implemented

[1]We have published the source code of this work at https://github.com/nobsessive/HSPA

Table 2. Implementation Results of the Proposed HSPA on AMD-Xilinx Device

Design	Device	LUT	FF	CLB	Fmax	BRAM	Latency[1]	Delay	Thru.[2]
HQC-128 (n = 17,669, ω = 75)									
Accelerator-I	UltraScale+	20,334	4,942	4,172	403	18	1,410	4	5.05
Accelerator-II (v=16)	UltraScale+	53,041	53,457	8,943	471	0	4,500	10	1.85
Accelerator-II (v=8)	UltraScale+	53,041	53,522	10,147	445	0	2,625	6	3.00
Accelerator-II (v=4)	UltraScale+	70,710	53,320	12,173	441	0	1,800	4	4.33
Accelerator-II (v=2)	UltraScale+	121,188	53,324	20,219	200	0	1,125	6	3.14
HQC-192 (n = 35,581, ω = 114)									
Accelerator-I	UltraScale+	18,905	4,181	3,923	376	18	4,200	11	3.19
Accelerator-II (v=16)	UltraScale+	106,763	107,146	18,194	458	0	6,840	15	2.38
Accelerator-II (v=8)	UltraScale+	128,415	107,222	21,698	400	0	4,788	12	2.97
HQC-256 (n = 57,637, ω = 149)									
Accelerator-I	UltraScale+	18,466	4,675	3,920	378	18	9,508	23	2.53
Accelerator-II (v=16)	UltraScale+	172,896	173,363	28,407	442	0	8,940	20	2.85
Accelerator-II (v=8)	UltraScale+	202,004	173,241	35,154	275	0	6,258	23	2.53
BIKE Level 1 (n = 12,323, ω = 142)									
Accelerator-I	UltraScale+	18,901	5,317	3,950	404	18	891	2	5.59
Accelerator-II (v=16)	UltraScale+	37,006	37,334	6,453	467	0	4,260	9	1.35
Accelerator-II (v=8)	UltraScale+	37,006	37,187	6,991	444	0	2,485	6	2.20
BIKE Level 2 (n = 24,659, ω = 206)									
Accelerator-I	UltraScale+	18,219	5,087	3,747	393	18	2,535	6	3.82
Accelerator-II (v=16)	UltraScale+	74,014	74,407	12,097	439	0	6,180	14	1.71
Accelerator-II (v=8)	UltraScale+	74,014	74,465	14,337	417	0	3,605	9	2.85
BIKE Level 3 (n = 40,973, ω = 274)									
Accelerator-I	UltraScale+	19,515	5,334	3,606	373	18	5,653	15	2.70
Accelerator-II (v=16)	UltraScale+	122,955	123,294	19,743	413	0	8,220	20	2.06
Accelerator-II (v=8)	UltraScale+	147,847	123,171	26,183	333	0	5,754	17	2.37

Unit for Fmax: MHz.
Unit for delay (calculated by Latency/Fmax): μs.
[1]: Latency refers to the computation time.
[2]: Thru. (Throughput)=n/delay ($\times 10^3$).

accelerators for three security level parameter sets of both HQC and BIKE (hqc-128, hqc-192, hqc-256, BIKE-Level 1, BIKE-Level 3, BIKE-Level 5), including the number of resource usage (LUTs, FFs, Slices, BRAMs), maximum frequency (Fmax, MHz), latency, delay time, and the throughput, are listed in Table 2. Note that due to the large size of n for HQC-192, HQC-256, BIKE Level 2, and BIKE Level 3, we just implemented the proposed Accelerator-II with v = 16 and v = 8.

Performance Discussion. The proposed two accelerators overall obtain superior performance on the targeted FPGA device, as evidenced by their low-latency computation, high operational frequency, and high throughput rate. Proposed Accelerator-I, because of the use of BRAMs for accumulation-related computation, involves low resource usage comparatively. While Accelerator-II has to process all the computations within the non-memory based combinational circuits, it involves a relatively large logic resource usage, especially when v becomes smaller. Nevertheless, Accelerator-II involves higher frequency (in most cases) and better flexibility than Accelerator-II. Meanwhile, we want to mention that the proposed Accelerator-II is probably the first try in the field to design large-scale sparse polynomial multiplication with no memory usage (or similar),

Table 3. The Comparison of Implementation Results between Proposed Designs and the Related Work on AMD-Xilinx UltraScale+ Device

Design	LUT	FF	CLB	Fmax[1]	BRAM	WCLB[2]	Latency[3]	Delay[4]	Thru.[5]	ADP[6]
HQC-128 (n = 17,669, ω = 75)										
Extended from [32]	215,508	45,923	38,575	287	278	58,035	225	0.8	22.09	45.50
Accelerator-I	20,334	4,942	4,172	403	18	5,612	1,410	3.5	5.05	19.64
Accelerator-II (v=4)	70,710	53,320	12,173	441	0	12,173	1,800	4	4.33	49.69
BIKE Level 1 (n = 12,323, ω = 142)										
Extended from [32]	148,229	32,702	25,610	287	194	39,190	426	1.5	8.22	58.17
Accelerator-I	18,901	5,317	3,950	404	18	5,210	891	2.2	5.59	11.49
Accelerator-II (v=8)	37,006	37,187	8,092	444	0	8,092	2,485	5.6	2.20	45.29

[1]: Unit for Fmax: MHz.
[2]: WCLB (Weighted CLB) = #CLB+#BRAMs×70, where we have adopted the method presented in [17] (also used in [32]) that 1 BRAM(8k) is equivalent to 70 CLBs.
[3]: Latency refers to the computation time (input loading and output delivery are not included).
[4]: Calculated by Latency/Fmax. Unit: μs.
[5]: Thru. (Throughput)=n/delay ($\times 10^3$).
[6]: ADP=(WCLB)×Delay/1000, which is a normalized metric to measure the overall area-time complexities of a design.

further efforts can be made to improve its performance better. Finally, as the two accelerators are originated from two completely different design concepts with their own unique features, they shall be evaluated in a balanced format.

Note that from Table 2, we can see that the latency of Accelerator-II decreases linearly, which aligns the theoretical timing complexity analysis. However, the delay, being calculated by Latency/Fmax, is affected by the dropping frequency, which decreases from 441 to 200 Mhz when it comes to $v = 2$. This is mainly because the number of signals being fed into the MUXes before every register (see Figure 9) increases as v decreases, which increases the size of the MUXes and the number of wires connecting different components in the accelerator, and thus prolongs the critical path when the design is being implemented. In summary, the theoretical computation time (latency) of Accelerator-II still drops linearly, and the tradeoff between v and computation time is still clear, but the actual delay increases due to the nearly cut-half frequency.

Comparison Consideration. As mentioned in Sections I and II, the existing designs for sparse polynomial multiplications of HQC and BIKE are all memory-based compact designs with very long latency cycles [6, 24, 26, 32]. As the proposed accelerators are targeting high-throughput applications, we do not directly compare the proposed designs with them (different design style originated accelerators have different advantages over each other). Nevertheless, we consider the fact that: (i) the design of Reference [32] is the most recent compact structure for sparse polynomial multiplication in the literature and has shown its advantages over the other ones like [6, 9, 24, 26] (the one in Reference [7] did not consider the input memory usage, and we thus do not include it here, see Figure 1 of Reference [7]), (ii) Figure 1 is the extended high-speed version of the most recent work of [32], we thus decided to use the high-speed structure of Figure 1 (extended from Reference [32]) as our comparison counterpart.

Again, we have faithfully coded the design of Figure 1 (extended from Reference [32]) with VHDL (with function verified through ModelSim) and have implemented it on the same UltraScale+ XCZU9EG-2FFVB FPGA through the same Vivado 2020.2. We have obtained the corresponding area and time complexities, such as the number of LUTs, FFs, and CLBs, maximum frequency, BRAM usage, latency, and related ADPs in Table 3. Meanwhile, we have listed the complexities of the proposed Accelerator-I and Accelerator-II ($v = 4/v = 8$) in the same table. Note that the

extended architecture of Reference [32] has very large resource usage, and we just obtained its performance following the parameter sets of HQC-128 and BIKE Level 1.

Comparison and Discussion. As seen from Table 3, the proposed two accelerators have more balanced area-time complexities than the high-speed structure extended from Reference [32] (see Figure 1). Though the extended high-speed version of Reference [32] involves very low latency, its resource usage is huge (e.g., 278 BRAMs for HQC-128) and thus is not ideal for general-purpose high-performance applications. To have a balanced comparison, we have also adopted the strategy proposed in [17] (used in Reference [32] as well) that 1 BRAM equals to 70 CLBs to calculate the overall weighted ADP. One can see that the proposed memory-based Accelerator-I involves 56.84% and 80.25% smaller ADP than the extended high-speed version of Reference [32], for HQC-128 and BIKE Level 1, respectively. While comparing with the proposed Accelerator-II, the extended high-speed structure of [32] has 0.84% less ADP for HQC-128 but involves 22.14% larger ADP for BIKE Level 1, i.e., the proposed Accelerator-II has better balanced area-time complexities than the extended version of Reference [32].

Apart from the normalized ADP comparison, we also want to emphasize the fact that the existing design (extended of [32], see Figure 1) probably is not practical for actual applications due to its large resource usage. The proposed Accelerator-I, built on the proposed PSA algorithm, is desirable for quite a number of high-performance applications because of its small area occupation and high throughput capability. In some memory-demanding applications, however, we recommend using the proposed Accelerator-II since it does not require any memory usage (benefited from the proposed PWP-based method). Note that the proposed Accelerator-II processes all the computation-related operations such as loading, shifting, accumulation, and even outputting in the corresponding structure (which potentially increases the resource usage). Nevertheless, its resource usage is still considered as decent and its throughput is very high (as revealed in Tables 2 and 3), and can be extended to build high-speed code-based cryptoprocessors with small memory usage.

Further Discussion and Future Research. The proposed two accelerators have constant operational time (connecting with Sections 3 and 4), and hence are considered as secure against regular timing attacks [28]. However, for related side-channel power and fault attack resistances, we recommend this research as one of our future exploration directions as these attacks generally require sophisticated attack setups and dedicated countermeasure designs.

Other future research can also be: (i) extending the proposed sparse polynomial multiplication accelerators to the actual cryptographic processor designs for HQC/BIKE (which will be desirable when the NIST fourth-round PQC standardization process is completed [1]); (ii) developing new design techniques to obtain a unified accelerator (a more parameterized design) for different v and/or security levels; (iii) propose novel implementation strategies to deploy other resources like DSPs (dedicated designs to be designed to conduct parallel operations in each DSP to compensate the overhead) to obtain further efficiency.

Side-Channel Considerations. The parallel processing featured Accelerator-I, and the sequential shifting operations in Accelerator-II's LS Component could potentially expose power consumption patterns, which may be vulnerable to differential power analysis. Future implementations could incorporate countermeasures such as masking techniques for intermediate value randomization and power consumption balancing circuits. However, these protective measures commonly introduce overhead in area utilization and reduce performance. Evaluate the security-performance tradeoff by validating it through differential power analysis, which would be valuable for security-critical applications. We believe comprehensive side-channel security analysis and implementation of corresponding countermeasures are important directions for future research.

Related Works. Traditional way of implementing polynomial multiplication over $\mathbb{F}_2$ (binary field) can be seen in quite a number of literature works [10, 13, 14, 16, 20, 22, 23, 27, 34, 36]. Most

notably, two sub-quadratic methods, namely Karatsuba algorithm and TMVP, are widely used to reduce the computational complexity [8, 14]. These methods, however, cannot be deployed for implementing the targeted very large size sparse polynomial multiplier (see the explanation in the Paragraph of High Dimensional Sparse-Dense Polynomial Multiplication of Section 2).

Besides that, multiple polynomial multiplication accelerators for lattice-based PQC have been proposed recently. A systolic multiplication accelerator for KEM and Ring-LWE based PQC was reported in Reference [4]. A high-performance polynomial multiplication accelerator was reported in Reference [33]; Optimized high-performance and lightweight polynomial multiplication architectures were presented in [5]. Another High-speed polynomial multiplication architecture was proposed in Reference [31]. An efficient implementation for Inverted Binary Ring-LWE Based PQC was reported in Reference [11] and a lightweight hardware accelerator for Binary Ring-LWE PQC was presented in Reference [18]. Area-optimized polynomial multiplication for NTRU was proposed in Reference [12]. There may also exist other works, which we do not list them here (but they are all considered as related works in the PQC field).

Note that both above-mentioned types of polynomial multiplications are not suitable for direct qualitative comparison with the proposed HSPA. (i) The conventional polynomial multiplications like [10, 13, 14, 16, 20, 22, 23, 27, 34, 36] are specific works for small-size binary polynomial multiplications ranging from $n = 163, 233, 283, 409$, and 571, which is too small to be compared with the size we targeted here like $n = 57,637$. Besides that, these polynomial multiplications did not consider the sparsity of the coefficients (i,e., coefficients can be "0"/"1"), and hence, these approaches cannot be directly compared with the work presented here. (ii) For the polynomial multiplication used in lattice-based PQC, we want to mention that these polynomial multiplications are integer polynomials (not binary values), and again, the related size is also relatively small, e.g., $n = 256/512$ in [18, 31]. Hence, comparing these designs with the proposed works qualitatively is also inappropriate.

6 Conclusion

This article, for the first time, presented two novel high-throughput hardware accelerators for the sparse polynomial multiplier found in two code-based PQC schemes (HQC and BIKE). First, we have proposed two new algorithms for the targeted sparse polynomial multiplication for high-throughput operation. Then, the architectural details of the proposed two accelerators (according to the proposed algorithms) are demonstrated. Finally, the complexity analyses and implementation results are provided, along with the comparison with the state-of-the-art design, confirming the superior performance of the proposed accelerators. We hope the proposed work will benefit the efficient implementation of code-based PQC and the ongoing NIST PQC standardization process.

References

[1] 2024. Are we there yet? An Update on the NIST PQC Standardization Project. NIST 5th PQC Standardization Conference. Retrieved from https://csrc.nist.gov/Presentations/2024/update-on-the-nist-pqc-standardization-project

[2] 2022. PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates. Retrieved from https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4. (2022).

[3] Nicolas Aragon et al. 2022. BIKE: Bit Flipping Key Encapsulation (Round 4 Submission). Retrieved from https://bikesuite.org/#content. (2022).

[4] Tianyou Bao, Pengzhou He, and J. Xie. 2022. Systolic acceleration of polynomial multiplication for KEM saber and binary ring-LWE post-quantum cryptography. In *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 157–160. DOI:https://doi.org/10.1109/HOST54066.2022.9839980

[5] Andrea Basso and Sujoy Sinha Roy. 2021. Optimized polynomial multiplier architectures for post-quantum KEM saber. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1285–1290.

[6] Sanjay Deshpande, Mamuri Nawan, Kashif Nawaz, Jakub Szefer, and Chuanqi Xu. 2022. Towards a Fast and Efficient Hardware Implementation of HQC. Cryptology ePrint Archive, Paper 2022/1183. (2022). Retrieved from https://eprint.iacr.org/2022/1183

[7] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. 2022. Fast and efficient hardware implementation of HQC. *Cryptology ePrint Archive* (2022), 1–30.
[8] Haining Fan and M Anwar Hasan. 2007. A new approach to subquadratic space complexity parallel multipliers for extended binary fields. *IEEE Trans. Comput.* 56, 2 (2007), 224–233.
[9] Jingwei Hu, Wen Wang, Ray CC Cheung, and Huaxiong Wang. 2019. Optimized polynomial multiplier over commutative rings on FPGAs: A case study on BIKE. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 231–234.
[10] José L Imaña. 2015. High-speed polynomial basis multipliers over $GF(2^m)$ for special pentanomials. *IEEE Transactions on Circuits and Systems I: Regular Papers* 63, 1 (2015), 58–69.
[11] José L. Imaña, Pengzhou He, Tianyou Bao, Yazheng Tu, and J. Xie. 2022. Efficient hardware arithmetic for inverted binary ring-LWE based post-quantum cryptography. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 8 (2022), 3297–3307. DOI:https://doi.org/10.1109/TCSI.2022.3169471
[12] Safiullah Khan, Wai-Kong Lee, Ayesha Khalid, Abdul Majeed, and Seong Oun Hwang. 2022. Area-optimized constant-time hardware implementation for polynomial multiplication. *IEEE Embedded Systems Letters* 15, 1 (2022), 5–8.
[13] Chang Hoon Kim, Chun Pyo Hong, and Soonhak Kwon. 2005. A digit-serial multiplier for finite field $GF(2^m)$. *IEEE Transactions on Very Large Scale Integration (Vlsi) Systems* 13, 4 (2005), 476–483.
[14] Chiou-Yng Lee and J. Xie. 2018. Digit-serial versatile multiplier based on a novel block recombination of the modified overlap-free karatsuba algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers* 66, 1 (2018), 203–214.
[15] Chiou-Yng Lee and Jiafeng Xie. 2018. Digit-serial versatile multiplier based on a novel block recombination of the modified overlap-free karatsuba algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers* 66, 1 (2018), 203–214.
[16] Chiou-Yng Lee and J. Xie. 2019. High capability and low-complexity: Novel fault detection scheme for finite field multipliers over $GF(2^m)$ based on MSPB. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 21–30.
[17] Weiqiang Liu, Sailong Fan, Ayesha Khalid, Ciara Rafferty, and Máire O'Neill. 2019. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA. *IEEE TVLSI Systems* 27, 10 (2019), 2459–2463.
[18] Benjamin J. Lucas, Ali Alwan, Marion Murzello, Yazheng Tu, Pengzhou He, Andrew J. Schwartz, David Guevara, Ujjwal Guin, Kyle Juretus, and J. Xie. 2022. Lightweight hardware implementation of binary ring-LWE PQC accelerator. *IEEE Computer Architecture Letters* 21, 1 (2022), 17–20. DOI:https://doi.org/10.1109/LCA.2022.3160394
[19] Robert J. McEliece. 1978. A public-key cryptosystem based on algebraic. *Coding Thv* 4244 (1978), 114–116.
[20] Pramod Kumar Meher. 2008. Systolic and super-systolic multipliers for finite field $GF(2^m)$) based on irreducible trinomials. *IEEE Transactions on Circuits and Systems I: Regular Papers* 55, 4 (2008), 1031–1040.
[21] Carlos Aguilar Melchor et al. Hamming Quasi-Cyclic (HQC) (NIST Round 3 Submission). Retrieved from https://pqc-hqc.org/index.html. (n.d.).
[22] Ashkan Hosseinzadeh Namin, Huapeng Wu, and Majid Ahmadi. 2012. An efficient finite field multiplier using redundant representation. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 2 (2012), 1–14.
[23] Jeng-Shyang Pan, Chiou-Yng Lee, Anissa Sghaier, Medien Zeghid, and J. Xie. 2019. Novel systolization of subquadratic space complexity multipliers based on toeplitz matrix–vector product approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 7 (2019), 1614–1622.
[24] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. 2021. Racing BIKE: Improved polynomial multiplication and inversion in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022, 1 (Nov. 2021), 557–588. DOI:https://doi.org/10.46586/tches.v2022.i1.557-588
[25] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. 2021. Folding BIKE: Scalable hardware implementation for reconfigurable devices. *IEEE Trans. Comput.* 71, 5 (2021), 1204–1215.
[26] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. 2020. Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices. Cryptology ePrint Archive, Paper 2020/897. (2020). Retrieved from https://doi.org/10.1109/TC.2021.3078294 https://eprint.iacr.org/2020/897.
[27] Erkay Savaš, Alexandre F Tenca, and Cetin K Koç. 2000. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded SystemsCHES 2000: Second International Workshop Worcester, MA, USA, August 17–18, 2000 Proceedings 2*. Springer, 277–292.
[28] Tobias Schneider, Amir Moradi, and Tim Güneysu. 2016. ParTI–towards combined hardware countermeasures against side-channel and fault-injection attacks. In *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, August 14-18, 2016, Proceedings, Part II 36*. Springer, 302–332.
[29] Nicolas Sendrier. 2017. Code-based cryptography: State of the art and perspectives. *IEEE Security & Privacy* 15, 4 (2017), 44–50. DOI:https://doi.org/10.1109/MSP.2017.3151345
[30] Peter W Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Ieee, 124–134.

[31] Weihang Tan, Antian Wang, Xinmiao Zhang, Yingjie Lao, and Keshab K. Parhi. 2023. High-speed VLSI architectures for modular polynomial multiplication via fast filtering and applications to lattice-based cryptography. *IEEE Trans. Comput.* 72, 9 (2023), 2454–2466. DOI:https://doi.org/10.1109/TC.2023.3251847

[32] Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and J. Xie. 2023. LEAP: Lightweight and efficient accelerator for sparse polynomial multiplication of HQC. *IEEE Trans. VLSI Systems* 31, 6 (2023), 892–896. DOI:https://doi.org/10.1109/TVLSI.2023.3246923

[33] Yazheng Tu, Pengzhou He, Chiou-Yng Lee, Danai Chasaki, and J. Xie. 2022. Hardware implementation of high-performance polynomial multiplication for KEM saber. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS).* 1160–1164. DOI:https://doi.org/10.1109/ISCAS48785.2022.9937606

[34] Huapeng Wu, M. Anwarul Hasan, Ian F. Blake, and Shuhong Gao. 2002. Finite field multiplier using redundant representation. *IEEE Trans. Comput.* 51, 11 (2002), 1306–1316.

[35] J. Xie, K. Basu, K. Gaj, and U. Guin. 2020. Special session: The recent advance in hardware implementation of post-quantum cryptography. In *IEEE VTS*. 1–10.

[36] J. Xie, Chiou-Yng Lee, Pramod Kumar Meher, and Zhi-Hong Mao. 2019. Novel bit-parallel and digit-serial systolic finite field multipliers over $GF(2^m)$ based on reordered normal basis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 9 (2019), 2119–2130.

Received 11 May 2023; revised 4 May 2024; accepted 17 October 2024