LEAP: Lightweight and Efficient Accelerator for Sparse Polynomial Multiplication of HQC

Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and Jiafeng Xie

***Abstract*— The Hamming quasi-cyclic (HQC) code-based encryption scheme is one of the fourth-round algorithms selected by the National Institute of Standards and Technology (NIST) postquantum cryptography (PQC) standardization process. However, very few hardware implementations have been reported for HQC to date. In this brief, we propose a novel Lightweight and Efficient Accelerator for sparse Polynomial multiplication (LEAP) of HQC, compatible with different parameters, on the field-programmable gate array (FPGA) platform. First, we give a mathematical derivation process for the sparse polynomial multiplication deployed in HQC. Then, we explain the proposed hardware structure in detail. Finally, we present the FPGA implementation results to confirm the efficiency of the proposed LEAP, for example, the proposed design for hqc-192 has at least 31.03% less area-delay product (ADP) than the existing design. LEAP can be extended further to construct efficient HQC cryptoprocessors.**

***Index Terms*— Hamming quasi-cyclic (HQC), hardware design, lightweight and efficient accelerator, postquantum cryptography (PQC), sparse polynomial multiplication.**

I. Introduction

It has been proven that traditional cryptosystems such as RSA and ECC are vulnerable to attacks launched from large-scale quantum computers [1]. Hence, the need for cryptosystems that are safe against quantum attacks, collectively known as postquantum cryptography (PQC), is at an all-time high [1], [2]. The National Institute of Standards and Technology (NIST) has started the PQC standardization process, and the code-based Hamming quasi-cyclic (HQC) is recently selected as one of the fourth-round candidates [3], [4].

Code-based cryptography refers to the cryptosystem that is based on error-correcting codes [5]. HQC is a code-based PQC scheme whose security relies on decoding small weight vectors of random quasi-cyclic codes [4], [6]. Following the NIST PQC standardization process, efforts are needed for efficient hardware implementation of HQC, for example, on field-programmable gate array (FPGA) [3], [7], which offers flexibility and fast development cycle to evaluate the PQC scheme's hardware complexity (a recent trend in the PQC field [2]).

On the other side, however, very few hardware implementation reports have been released for HQC: 1) the authors of HQC have given a high-level synthesis-based hardware result [4] (no subcomponents' results like polynomial multiplier) and 2) another recent hardware design was reported in [7].

A. Challenges

It is noted that the polynomial multiplier over $\mathbb{F}_2$ is a critical component of HQC and its performance largely determines the overall efficiency of the scheme implementation [4]. However, one has to be aware that the polynomial multiplier of HQC involves operations of sparse matrices or vectors with very large dimensions [4], [7], for example, polynomials of size $n = 57\,637$ with only 149 nonzero elements, which is extremely time- and space-consuming. Overall, there exist two challenges: 1) for the multiplication algorithm, the size and sparsity of the polynomial multiplication make it difficult to deploy a suitable algorithmic procedure and 2) for the hardware architecture design, the selection of a proper processing strategy is tricky, as it will affect the other components (like memories) and the total performance. Finally, we also consider the fact that very few hardware designs are released so far.

Thus, in this brief, we propose a **L**ightweight and **E**fficient **A**ccelerator for sparse **P**olynomial multiplication (LEAP) of HQC. In particular, we made three layers of contributions, as follows.

1) We give a detailed mathematical formulation to derive a new algorithm for the sparse polynomial multiplication in HQC.
2) We construct the proposed LEAP through efficient algorithm-to-architecture mapping techniques and strategies.
3) We provide sufficient implementation and comparison results to showcase the superior performance of the proposed LEAP.

The rest of this brief is organized as follows. The preliminaries are introduced in Section II. The mathematical derivation is presented in Section III. The proposed LEAP is presented in Section IV. Implementation and comparison are presented in Section V. Conclusions are given in Section VI.

II. Preliminaries

A. Notations [4]

We define $\mathbb{Z}$ as the ring of integers and $\mathbb{F}_2$ as the binary finite field. $\mathcal{V}$ is a vector space of dimension n over $\mathbb{F}_2$, where $n \in \mathbb{Z}$. Elements in $\mathcal{V}$ are vectors/polynomials in $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ (lower-case bold), while matrices are denoted by upper-case bold letters. $\omega(\cdot)$ denotes the Hamming weight of a vector (number of its nonzero coordinates). $\mathcal{C}[n, k]$ denotes a linear code with length n and dimension. Elements of $\mathcal{C}$ are referred to as codewords [8]. δ is the minimum number of errors that the decoding algorithm can correct. All computations in HQC are made in the ambient space $\mathbb{F}_2^n$.

We have also used G, D, and W to represent related polynomials for the proposed algorithm ($[G]$ denotes the corresponding matrix). $P[\cdot]$ denotes the index of nonzero elements. N_{mem} is the processing bit-length of the memory.

B. Hamming Quasi-Cyclic

HQC is an Indistinguishability under Chosen Ciphertext Attack (IND-CCA) secure encryption scheme built on the hardness of a decision version of the syndrome decoding on structured codes [4]. HQC uses a decodable code $\mathcal{C}[n, k]$ and a random double-circulant $[2n, n]$ code, which features a precise upper bound for the decryption failure probability analysis [4].

C. Algorithms

Let $\mathcal{G}(\cdot), \mathcal{H}(\cdot), \mathcal{K}(\cdot)$ denote SHAKE256-512 (B7||`G_FCT_DOMAIN`), SHAKE256-512 (B7||`H_FCT_DOMAIN`), and SHAKE256-512 (B7||`K_FCT_DOMAIN`), respectively. Algorithms 1

Manuscript received 14 October 2022; revised 1 January 2023; accepted 9 February 2023. Date of publication 13 March 2023; date of current version 23 May 2023. The work of Jiafeng Xie was supported in part by NIST-60NANB20D20 and in part by NSF under Grant SaTC-2020625. The work of Çetin Kaya Koç was supported by TUBITAK under Grant 1001-121F348. *(Corresponding author: Yazheng Tu.)*

Yazheng Tu, Pengzhou He, and Jiafeng Xie are with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA 19085 USA (e-mail: ytu1@villanova.edu; phe@villanova.edu; jiafeng.xie@villanova.edu).

Çetin Kaya Koç is with the Department of Computer Science (DEEC), University of California at Santa Barbara, Santa Barbara, CA 93106 USA, and also with Iğdır University, NUAA, Nanjing 210016, China (e-mail: cetinkoc@ucsb.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2023.3246923.

Digital Object Identifier 10.1109/TVLSI.2023.3246923

Algorithm 1 HQC.PKE [4]

```
    Setup($1^\lambda$):
1   generate and output the global parameters param =
      $(n, k, \delta, \omega, \omega_{\mathbf{r}}, \omega_{\mathbf{e}})$.
    KeyGen(param):
2   sample $\mathbf{h} \leftarrow \mathcal{R}$, 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   returns $(pk, sk)$.
    Encrypt($pk$, m):
6   generate $\mathbf{e} \leftarrow \mathcal{R}$, $\mathbf{r} = (\mathbf{r_1}, \mathbf{r_2}) \leftarrow \mathcal{R}^2$ such that
      $\omega(\mathbf{e}) = \omega_{\mathbf{e}}$ and $\omega(\mathbf{r_1}) = \omega(\mathbf{r_1 2}) = \omega_{\mathbf{r}}$
7   $\mathbf{u} = \mathbf{r_1} + \mathbf{h} \cdot \mathbf{r_2}$
8   $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r_2} + \mathbf{e}$
9   return $\mathbf{c} = (\mathbf{u}, \mathbf{v})$.
    Decrypt($sk$, c):
10  return $\mathcal{C}.\mathbf{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{v})$.
```

Algorithm 2 HQC.KEM [4]

```
    Setup($1^\lambda$):
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($pk, \mathbf{c}, \mathbf{d}$):
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 $c \neq \mathbf{c}'$, or $d \neq \mathcal{H}(\mathbf{m}')$ then
16      abort
17  else
18      derive the shared key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$
19  end
```

and 2 are the public-key encryption (PKE) and key encapsulation mechanism (KEM) versions of HQC, respectively.

D. Security

There exist three security levels of HQC: hqc-128, hqc-192, and hqc-256 [4] (LEAP is applicable to all three levels).

III. LEAP (Mathematical Derivation)

Polynomial multiplication over ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ exists in most steps of HQC and thus has a huge impact on the overall performance. Due to the large size of operand vectors but with only very few nonzero elements, this polynomial multiplication is

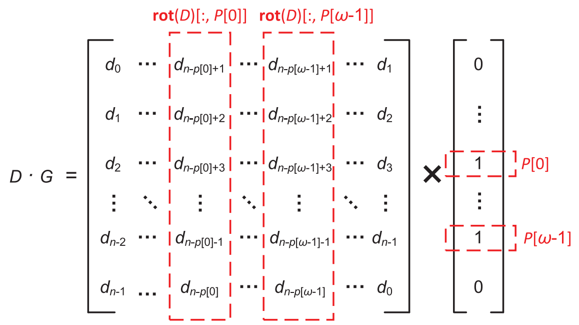

Fig. 1. Indices and related columns within $[G]$ and $[D]$.

defined as a "sparse polynomial multiplier." For instance, in hqc-128, the Hamming weight of the vector is only $\omega = 66$ or $\omega_{\mathbf{e}} = 75$ when $n = 17\,669$ [4]. Thus, traditional strategies like the schoolbook method would be inappropriate as its complexity is $\mathcal{O}(n^2)$. Based on this consideration, we decided to propose a new polynomial multiplication algorithm.

Definition 1: We define the sparse polynomial multiplication for HQC as $W = GD \bmod f(x)$, where $f(x) = x^n - 1$, $W = \sum_{i=0}^{n-1} w_i x^i$, $G = \sum_{i=0}^{n-1} g_i x^i$, and $D = \sum_{i=0}^{n-1} d_i x^i$ (g_i, d_i, and w_i are all binary values in $\mathbb{F}_2/(x^n - 1)$). Note that G is a polynomial with only ω nonzero coefficients. Then, we have

$$W = \left(g_0 + g_1 x + \cdots + g_{n-1}x^{n-1}\right) d_0 \bmod f(x) + \cdots + \left(g_0 + g_1 x + \cdots + g_{n-1}x^{n-1}\right) d_{n-1}x^{n-1} \bmod f(x) \tag{1}$$

which can be further derived as (since $x^n - 1 \equiv 0$)

$$W = \left(g_0 + g_1 x + \cdots + g_{n-1}x^{n-1}\right) d_0 + \cdots + \left(g_0 x^{n-1} + g_1 + \cdots + g_{n-1}x^{n-2}\right) d_{n-1} \tag{2}$$

where we can have $w_0 = g_0 d_0 + g_{n-1} d_1 + \cdots + g_1 d_{n-1}$, $w_1 = g_1 d_0 + \cdots - g_2 d_{N-1}, \cdots, w_{n-1} = g_{n-1} d_0 + g_{n-2} d_1 + \cdots + g_0 d_{n-1}$, where w_i $(0 \le i \le n-1)$ can be obtained as

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{bmatrix} = \begin{bmatrix} g_0 & g_{n-1} & \cdots & g_1 \\ g_1 & g_0 & \cdots & g_2 \\ \vdots & \vdots & \cdots & \vdots \\ g_{n-1} & g_{n-2} & \cdots & g_0 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{bmatrix}. \tag{3}$$

A. Proposed Computation Strategy

The direct computation of (3) will incur huge resource usage and long delay time. Even with the deploying of subquadratic complexity fast algorithms such as the Karatsuba algorithm [6], the computation complexity could still be large. Considering that G is a polynomial with only ω nonzero coefficients and $[G]^{n\times n}$ is a circulant matrix, we thus have the following definition.

Definition 2: For a polynomial $G = (g_1, g_2, \ldots, g_{n-1}) \in \mathbb{F}_2^n$, we define its circulant matrix $\mathbf{rot}(G)$ as

$$\mathbf{rot}(G) = \begin{bmatrix} g_0 & g_{n-1} & \cdots & g_1 \\ g_1 & g_0 & \cdots & g_2 \\ \vdots & \vdots & \ddots & \vdots \\ g_{n-1} & g_{n-2} & \cdots & g_0 \end{bmatrix} \in \mathbb{F}_2^{n\times n}. \tag{4}$$

We can then have

$$[W]^{n\times 1} = \mathbf{rot}(G) \times [D]^{n\times 1} = \mathbf{rot}(D) \times [G]^{n\times 1}. \tag{5}$$

From (4) and (5), one can see that after we switched the circulant matrix from $\mathbf{rot}(G)$ to $\mathbf{rot}(D)$, we can compute (3) as follows: 1) mark the nonzero indices of those coefficients within vector $[G]^{n\times 1}$ and 2) sum the corresponding columns in the circulant matrix of $\mathbf{rot}(D)$ to obtain the final results, as shown in Fig. 1. Following

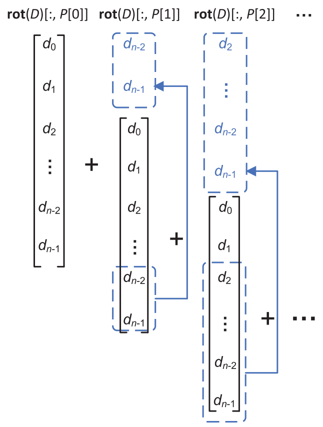

Fig. 2. Sum of column vectors.

Algorithm 3 Proposed Algorithm for Sparse Polynomial Multiplication of HQC

```
Input  : $G$ and $D$ are binary polynomials.
Output: $W = GD \bmod (x^n - 1)$.
Initialization step
1 make ready the inputs $G$ and $D$.
2 record the indices of nonzero elements in $G$ into $P$.
Main step
3 for $i = 0$ to $n - 1$ do
4     for $j = 0$ to $\omega - 1$ do
5         $w_i = w_i + \mathbf{rot}(D)[:,P[j]]$;
6     end
7 end
Final step
8 serially deliver all the coefficients of output $W$;
```

this strategy, the computation complexity of (3) is now reduced to only $\mathcal{O}(n\omega)$.

As shown in Fig. 1, the indices of nonzero elements G are denoted as $P[\cdot]$ and we can see that the polynomial multiplication is equivalent to the sum of column vectors in $\mathbf{rot}(D)$ corresponding to the nonzero elements in $[G]^{n\times 1}$ (Fig. 2). We finally summarize this process in Algorithm 3.

IV. LEAP (Proposed Hardware Structure Design)

The overall structure of the proposed accelerator is shown in Fig. 3, where it consists of two major components—the Preparatory Component and the Multiplier—as described below.

A. Preparatory Component

The Preparatory Component contains one memory and one Index Marker. As the proposed Algorithm 3 executes the multiplication process assuming that the indices of nonzero elements in polynomial G are provided as the input, an Index Marker for nonzero elements is needed to deliver the correct indices to the following Multiplier.

B. Index Marker

As is shown in Fig. 4, the Index Marker consists of a circular-shift register (ω blocks, each with $\lceil \log_2 n\rceil$ bits) and a counter. The Index Marker takes one coefficient of the polynomial as its only input each cycle and outputs the indices of the nonzero coefficients/elements in serial. The counter counts one up each cycle and delivers the count to the shift register; while the register, taking the count as an input and the incoming coefficient/element of the polynomial as its *enable* signal, records the count, that is, the index, when the element is "1." After recording the index, the shift register will shift 1 block ($\lceil \log_2 n\rceil$ bits) until all elements are examined.

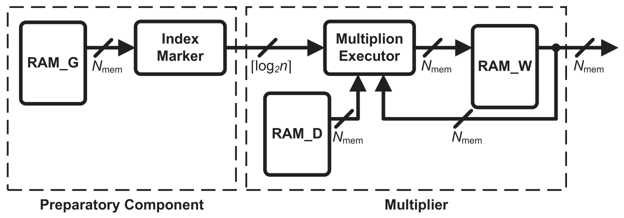

Fig. 3. Overview of the proposed accelerator: LEAP.

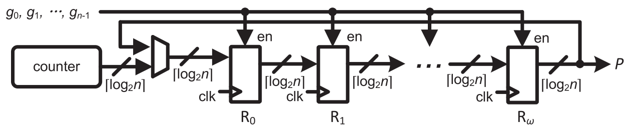

Fig. 4. Structural details of the Index Marker.

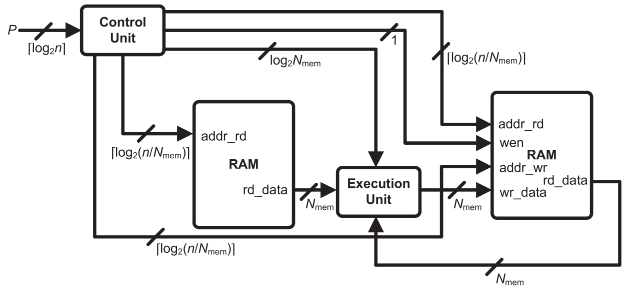

Fig. 5. Structural details of the Multiplier Component. Control signals: "addr_rd" denotes the address read; "rd_data" is the data read; "wen" denotes the write enable; "addr_wr" represents the address write; and "wr_data" is the data write.

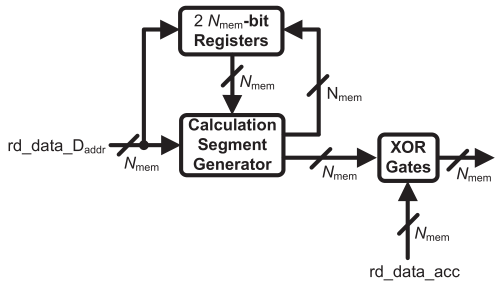

Fig. 6. Internal structure of the Execution Unit. Signal "rd_data_D$_{\text{addr}}$" denotes the data read of D-related address and "rd_data_acc" represents the data read of accumulation.

C. Multiplier Component

LEAP deals with the data reading from a whole memory block at a time based on Algorithm 3 (see Fig. 5). According to Fig. 2, the summation of the columns in $\mathbf{rot}(D)$ is bit-wise XOR operations, where one operand can be obtained by circularly shifting from another. The Component executes the multiplication by taking a whole block of memory and adding it to the sum of the previous column(s) read from the other memory (new input) and then writing the data back. Note N_{mem} is the processing bit-length of the memory.

The Control Unit of the Multiplier Component takes the indices from the Index Marker as its input and then calculates the corresponding reading addresses. Apart from that, the Control Unit also determines the position of the first bit in the column vector, that is, the index of $p[i][0]$ in the memory block of Fig. 1. Then starting from the calculated bit, the Control Unit determines if the next memory block will be used to form a N_{mem}-bit calculation segment, which is XORed with the data read from the other memory. A counter counting from 0 to $\lceil n/N_{\text{mem}}\rceil - 1$ is also involved when calculating the reading addresses for coefficients of G. Also, a $\lceil \log_2 n\rceil$-bit counter is used for calculating the writing addresses.

The Execution Unit (see Fig. 6) contains two N_{mem}-bit registers, which are used to store the calculation segment and coefficients formed/read in the previous cycle. The Calculation Segment

TABLE I
IMPLEMENTATION RESULTS FOR THE PROPOSED LEAP

*	ω	Fr.[1]	Time		Area			
			Cycle	(μs)	LUT	FF	CLB	#
hqc-128 (n = 17,669)								
32	66	203	36,565	179.97	214	152	73	2
32	75	196	41,551	211.36	213	152	78	2
64	66	182	18,349	100.72	440	278	135	2
64	75	179	20,851	116.81	458	278	150	2
128	66	196	9,241	47.14	2,453	663	678	4
128	75	201	10,501	52.23	2,443	662	689	4
hqc-192 (n = 35,581)								
32	100	202	111,301	551.50	238	156	86	4
32	114	198	126,883	640.82	234	154	91	4
64	100	188	55,701	296.16	452	280	152	4
64	114	191	63,499	333.12	439	280	152	4
128	100	210	27,901	132.50	2,370	664	657	4
128	114	201	31,807	158.08	2,371	664	665	4
hqc-256 (n = 57,637)								
32	131	195	236,194	1,208.13	211	158	84	8
32	149	195	268,648	1,376.82	206	156	76	8
64	131	182	118,163	648.36	448	291	174	8
64	149	188	134,399	713.12	440	284	164	8
128	131	207	59,213	285.47	2,247	665	648	8
128	149	211	71,869	340.44	2,330	669	684	8

*: N_{mem}. #: BRAM tiles. [1]: Unit for Fr. (frequency): MHz.

Generator generates an N_{mem}-bit calculation segment based on the count from the Control Unit, the remainder of $index/N_{\text{mem}}$, and the data segments from two registers. The generated calculation segment and the data read from memory are written to the two N_{mem}-bit registers, which can be used in the next cycle when a new calculation segment is generated.

The XOR gates perform the XOR operation of the calculation segment and the data read from the other memory, that is, N_{mem} XOR gates in parallel to perform N_{mem} bit-wise additions that N_{mem} results of sum-of-product can be calculated and updated to the memory at one cycle.

LEAP is compatible with all commonly used memories by simply adjusting the number of XORs and the register length (and Control Unit). The RAMs used for G and D are single-port, while the one storing the accumulated values is dual-port.

V. LEAP (COMPLEXITY AND COMPARISON)

A. Complexity Analysis

The complexities of the Index Marker are: one $\lceil \log_2 n \rceil \times n$ bit circular shift register and one $\lceil \log_2 n \rceil$ counter. The Index Marker needs n cycles to examine G and records the indices of the nonzero coefficients.

The complexities of the proposed LEAP are listed below: the Control Unit requires a $\lceil \log_2 n \rceil$-bit counter and a $\lceil \log_2 n/N_{\text{mem}} \rceil$-bit counter; the Execution Unit requires two N_{mem}-bit registers and N_{mem} XOR gates. The latency is $\lceil (n/N_{\text{mem}}+1) \rceil \times \omega$ (or ω_r) cycles.

B. Implementation

The experimental setup is as follows: 1) the proposed LEAP was coded in VHDL and tested using Modelsim, then implemented using Xilinx Vivado 2020.2; 2) the accelerator was coded in a generic format that different parameter sets of HQC (n, ω, and ω_r) can be set during the implementation process; 3) different memory block lengths were also chosen, that is, N_{mem} = 32, 64, and 128, respectively (these are the regular memory processing bits, the implemented results help us to have a comprehensive understanding of the complexity of the proposed design); and 4) the accelerator was implemented on the Artix-7 xc7a200t-3 FPGA. The obtained results are listed in Table I, including the number of LUTs, FFs, CLBs, and required BRAM tiles.

C. Performance Discussion

The area usage of LEAP grows with the length of the memory block (see Table I). This is because the registers storing the unemployed coefficients (during the formation of the calculation segment) possess the same length as the memory blocks and become larger as the length of memory blocks increases. Also, the involved XORs increase linearly with the length of the memory blocks. Besides, the number of memory blocks involved decreases as N_{mem} becomes larger. Finally, it is worth mentioning that the slight differences in CLB usage between two cases (same n and N_{mem}, but different ω) are due to the parameter changes in the Control Unit and related place and route efficiency on the FPGA.

The latency is inversely proportional to the length of memory blocks, while the maximum frequency for the proposed accelerator overall remains high (delay time decreases as the N_{mem} increases while proportionally changes to ω and n). Following Table I, one can choose the desired LEAP with a proper N_{mem} based on the potential application requirements.

D. Comparison

We have also listed the results of the state-of-the-art work in Table II for comparison, based on the same memory block size, the same ω, and the same FPGA device. Note that the existing report of [7] does not give the CLB usage and hence we just use the LUT to obtain the equivalent LUT (ELUT) to calculate area-delay product (ADP) (following the strategy in [9] that one BRAM (8 k) is equal to 70 CLBs and one CLB contains four LUTs). It is shown that the proposed design has better overall area-time complexities than the existing one. For the security levels of hqc-128, hqc-192, and hqc-256, the proposed LEAP has 23.46%, 31.03%, and 24.72% less ADP than the one of [7], respectively.

E. Extension to the Sparse Polynomial Multiplier in BIKE and Comparison

BIKE is another NIST fourth-round PQC standardization candidate [3], [12], which also involves a sparse polynomial multiplier with different parameter settings. We have extended the proposed design to the sparse polynomial multiplier in BIKE [12] and implemented it on the Artix-7 xc7a200t-3 FPGA and compared the performance with [10], [11] (on the same FPGA). We have followed the comparison strategy of Table III; both area reduction and time reduction are considered comprehensively to reach the final conclusion: the proposed one has better area–time complexities than the existing ones given in [10] and [11] (note that as both the proposed and the existing designs have reported the CLB usage, we just follow the strategy of [9] to convert the overall resource usage into equivalent CLB (ECLB) usage). For instance, for $n = 12,323$ and $\omega = 134$, the proposed design has 35.92% more CLBs than [10] but with 50% reduction in BRAM usage (the ECLB of the proposed design is smaller than [10]). Meanwhile, as LEAP has 14.03% less latency time than [10], the ADP of LEAP is 16.44% less than [10]. A similar situation applies to the comparison with [11]: 84.18% less ADP than [11].

F. Discussion and Future Works

The proposed accelerator is lightweight overall as it contains relatively small resource usage. Meanwhile, the proposed LEAP has a low latency time and hence is also feasible for high-performance applications.

While the major focus of this work is to develop an efficient polynomial multiplier accelerator for HQC, future works may focus on the construction of an HQC accelerator, further complexity reduction, and side-channel attacks.

TABLE II
IMPLEMENTATION RESULTS AND COMPARISON (SPARSE POLYNOMIAL MULTIPLIER FOR HQC)

Design	N_{mem}	ω	Fr.[1]	Time Complexity			Area Complexity							ADP[5]	ADPR[6]
				Cycle	(μs)	TR[2]	LUT	LUTR[2]	FF	BRAM	BR[2]	ELUT[3]	AR[2,4]		
hqc-128 (n = 17,669)															
[7]	128	75	228	18,765	82.30	-	1,834	-	573	4	-	2,954	-	243k	-
LEAP	128	75	201	10,501	52.23	36.54%	2,443	-24.93%	662	4	0%	3,563	-17.09%	186k	23.46%
hqc-192 (n = 35,581)															
[7]	128	114	215	56,969	264.97	-	1,821	-	587	6	-	3,501	-	928k	-
LEAP	128	114	201	31,807	158.08	40.34%	2,371	-23.20%	664	6	0%	4,051	-13.58%	640k	31.03%
hqc-256 (n = 57,637)															
[7]	128	149	203	119,333	587.85	-	1,837	-	606	6	-	3,517	-	2,067k	-
LEAP	128	149	211	71,869	340.44	42.09%	2,330	-21.16%	669	8	-25%	4,570	-23.04%	1,556k	24.72%

[1]: Unit for Fr. (frequency): MHz. [2]: TR (time reduction); LUTR (LUT reduction); BR (BRAM reduction); AR (area reduction).
[3]: ELUT (equivalent LUT, reported LUTs added with the transferred equivalent LUTs from BRAMs). As [7] did not report the configurable logic block (CLB) usage, we hence use LUT calculate the overall area: equivalent LUT (ELUT). 1 BRAM(8k) equals 70 CLBs and 1 CLB contains 4 LUTs (from [9]).
[4]: Refers to the overall area-complexity reduction, a comprehensive consideration including both the LUT usage and the BRAM usage reductions (which are calculated as the ELUT usage reduction). [5]: (Area-delay product) ADP=#ELUT×Time. [6]: ADPR (area-delay product reduction).

TABLE III
IMPLEMENTATION RESULTS AND COMPARISON (EXTENSION TO BIKE, WHERE $N_{\mathrm{mem}} = 128$)

Design	n	ω	Fr.[1]	Time Complexity			Area Complexity								ADP[5]	ADPR[6]
				Cycle	(μs)	TR[2]	LUT	FF	CLB	CLBR[2]	BRAM	BR[2]	ECLB[3]	AR[2,4]		
[10]	12,323	71	185	7,172	38.77	-	1,136	381	378	-	8	-	938	-	36k	-
LEAP	12,323	71	208	6,959	33.46	13.70%	2,433	660	676	-44.08%	4	50%	956	-1.88%	32k	11.11%
[10]	12,323	134	184	13,535	73.44	-	1,249	386	437	-	8	-	997	-	73k	-
LEAP	12,323	134	208	13,133	63.14	14.03%	2,467	661	682	-35.92%	4	50%	962	3.51%	61k	16.44%
[11]	10,163	71	210	51,688	246.13	-	-	-	292	-	5	-	642	-	158k	-
LEAP	10,163	71	222	5,752	25.91	89.47%	2,460	661	681	-57.12%	4	20%	961	-33.19%	25k	84.18%

[1]: Unit for Fr. (frequency): MHz. [2]: TR (time reduction); CLBR (CLB reduction); BR (BRAM reduction); AR (area reduction).
[3]: ECLB (equivalent CLB, reported CLBs added with the transferred equivalent CLBs from BRAMs). 1 BRAM(8k) equals 70 CLBs (from [9]).
[4]: Refers to the overall area-complexity reduction, a comprehensive consideration including both the CLB usage and the BRAM usage reductions (which are calculated as the ECLB usage reduction). [5]: (Area-delay product) ADP=#ECLB×Time. [6]: ADPR (area-delay product reduction).

G. Other Works

Other hardware implementations in the PQC field also include lattice-based PQC designs of [13], [14], [15], [16], [17], [18], and [19].

VI. CONCLUSION

This brief proposes a novel hardware accelerator for a sparse polynomial multiplier of HQC: LEAP. We derived a new algorithm for the targeted sparse polynomial multiplication. Then, we provided the architectural details of the proposed accelerator. Finally, we presented the complexity analysis and implementation, and a related comparison confirms the superior performance of the proposed LEAP. This work is expected to be useful for the ongoing NIST PQC standardization process.

REFERENCES

[1] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.

[2] J. Xie, K. Basu, K. Gaj, and U. Guin, “Special session: The recent advance in hardware implementation of post-quantum cryptography,” in *Proc. IEEE 38th VLSI Test Symp. (VTS)*, Apr. 2020, pp. 1–10.

[3] *PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates*. [Online]. Available: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4

[4] C. A. Melchor et al. *Hamming Quasi-Cyclic (HQC) (NIST Round 4 Submission)*. [Online]. Available: https://pqc-hqc.org/index.html

[5] R. Overbeck and N. Sendrier, “Code-based cryptography,” in *Post-Quantum Cryptography*. Cham, Switzerland: Springer, 2009, pp. 95–145.

[6] C.-Y. Lee and J. Xie, “Digit-serial versatile multiplier based on a novel block recombination of the modified overlap-free Karatsuba algorithm,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 1, pp. 203–214, Jan. 2019.

[7] S. Deshpande, M. Nawan, K. Nawaz, J. Szefer, and C. Xu, “Towards a fast and efficient hardware implementation of HQC,” *Cryptol. ePrint Arch.*, pp. 1–24, 2022.

[8] S. Lin and D. J. Costello, *Error Control Coding*, vol. 2. New York, NY, USA: Prentice-Hall, 2001.

[9] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O’Neill, “Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 10, pp. 2459–2463, Oct. 2019.

[10] R.-Brockmann et al., “Racing BIKE: Improved polynomial multiplication and inversion in hardware,” in *Proc. IACR TCHES*, 2022, pp. 557–588.

[11] J. Hu, W. Wang, R. C. C. Cheung, and H. Wang, “Optimized polynomial multiplier over commutative rings on FPGAs: A case study on BIKE,” in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Dec. 2019, pp. 231–234.

[12] N. Aragon et al. *Bit Flipping Key Encapsulation (BIKE)) (NIST Round 4 Submission)*. [Online]. Available: https://bikesuite.org/

[13] A. Basso and S. S. Roy, “Optimized polynomial multiplier architectures for post-quantum KEM saber,” in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 1285–1290.

[14] T. Bao, P. He, and J. Xie, “Systolic acceleration of polynomial multiplication for KEM saber and binary ring-LWE post-quantum cryptography,” in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Jun. 2022, pp. 157–160.

[15] K. Basu et al., “NIST post-quantum cryptography—A hardware evaluation study,” *Cryptol. ePrint Arch.*, pp. 1–16, 2019.

[16] D.-E.-S. Kundi, A. Khalid, S. Bian, C. Wang, M. O’Neill, and W. Liu, “AxRLWE: A multilevel approximate ring-LWE co-processor for lightweight IoT applications,” *IEEE Internet Things J.*, vol. 9, no. 13, pp. 10492–10501, Jul. 2022.

[17] C. P. Rentería-Mejía and J. Velasco-Medina, “High-throughput ring-LWE cryptoprocessors,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 8, pp. 2332–2345, Aug. 2017.

[18] S. Bian, M. Hiromoto, and T. Sato, “Filianore: Better multiplier architectures for LWE-based post-quantum key exchange,” in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[19] B. J. Lucas et al., “Lightweight hardware implementation of binary ring-LWE PQC accelerator,” *IEEE Comput. Archit. Lett.*, vol. 21, no. 1, pp. 17–20, Jan. 2022.