Evaluating the Trade-offs in the Hardware Design of the LEDAcrypt Encryption Functions

Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi and Davide Zoni
Department of Electronics, Information and Bioengineering – DEIB
Politecnico di Milano, 20133 Milano, Italy
Email: *name.surname*@polimi.it

Abstract—**The confidentiality provided by widely employed asymmetric cryptosystems relying on the hardness of factoring large integers or computing discrete logarithms in a cyclic group will be jeopardized by the availability of quantum computers. As a consequence, the design of quantum-computing resistant cryptographic primitives has gained traction lately, especially thanks to the U.S. National Institute of Standards and Technology initiative, which aims at selecting a portfolio of primitives for standardization. A prime position in the set of asymmetric encryption primitives is occupied by the ones relying on decoding random linear error correction codes as their trapdoor. Among these primitives, the LEDAcrypt cryptosystem was recently announced to be admitted to the second round of the standardization initiative, where the investigation of the implementation performance is the main focus. In this paper we analyze the possible hardware designs to implement fast polynomial multiplications tailored to the encryption functions of the LEDAcrypt key encapsulation mechanism, and public key encryption primitives. In particular, we describe two designs for binary polynomial multiplications and analyze the advantages provided by exploiting the sparse nature of one of the operands in LEDAcrypt multiplications, validating our designs on a Xilinx Artix 100.**

Index Terms—**Post-quantum Cryptography, LEDAcrypt, Code-based Cryptosystems, FPGA Implementation**

I. Introduction

Building asymmetric cryptosystems resistant against attacks performed with quantum computers requires a change in the underlying computationally hard problems exploited to build a trapdoor. To this end, the aim is to pick a problem either belonging, or equivalent to one belonging, to the NP-Complete class [1]. Problems belonging to such a computational complexity class are widely believed not to have a polynomial-time solution algorithm, even on quantum computers. Among these problems lie the one of decoding a syndrome obtained with a random linear block code, and the one of finding a fixed weight codeword in the said code [2], [3]. As a consequence, Robert McEliece was the first one to propose a cryptosystem relying on the hardness of the decoding problem in 1978 [4], followed by the work of Niederreiter in 1986 [5]. The trapdoor function in these cryptosystems is constituted by the fact that the linear block code employed in them is not random, but instead it is picked from a family of linear block codes for which an efficient decoding algorithm is known. To prevent an attacker from deciphering, both cryptosystems provide as a public key an obfuscated representation of the efficiently decodable code, obtained multiplying either its generator matrix, or its parity check matrix, by a random non-singular conformant one.

Code based cryptosystems have a remarkably good security track, whenever the hidden code structure is a binary Goppa code (which is the one originally proposed by McEliece). However, such strength comes at the disadvantage of quite large public key sizes (in the megabyte range). Willing to reduce the key size, a significant amount of research work was aimed at employing code families admitting a space-efficient representation, such as: QC-LDPC codes and quasi-cyclic moderate-density parity-check (QC-MDPC) codes. Quasi-cyclic codes are characterized by generator and parity check matrices that are quasi-cyclic, i.e., they are composed by circulant square sub-matrices, where all the rows are obtained cyclically rotating the first one. The arithmetics of circulant matrices with size p can be proven to be isomorphic to the one of the polynomials modulo x^p+1 over the same field of the coefficients of the circulant matrices. In particular, in the case of binary linear block codes, the arithmetics of $p\times p$ circulant matrices over $\mathbb{Z}_2$ can be substituted with the arithmetics of polynomials in $\mathbb{Z}_2[x]/(x^p+1)$. This, in turn, implies a reduced size of the keypairs and faster arithmetic operations.

Contributions. We consider the case of LEDAcrypt [6], [7], a set of two cryptographic primitives based on QC-LDPC codes tackling the arithmetic operation of polynomial multiplication. We describe two scalable designs for binary polynomial multipliers for an FPGA-target, where we explore the advantage of trading off a sub-quadratic multiplication strategy relying on a Karatsuba multiplier approach with a quadratic multiplication strategy exploiting the sparsity of one of the two factors to speed up the multiplication. We validate our designs locating the tradeoff points between sparse and dense multiplications for operand sizes matching the LEDAcrypt specifications, targeting a Xilinx Artix 100 FPGA, and reporting resource utilization and performance figures for our designs.

II. LEDAcrypt Code-based Cryptoschemes

The LEDAcrypt specification [6] describes a Key Encapsulation Method (KEM) relying on the Niederreiter cryptoscheme [5], and a Public Key Encryption (PKC) system relying on the McEliece cryptoscheme [4] plus the IND-CCA Kobara-Imai γ-construction. Both systems employ a binary QC-LDPC code having a systematic public parity check matrix representation. The key-generation algorithms consider a QC-LDPC code $\mathcal{C}(n,k)$, with codeword length $n{=}pn_0$ and information word length $k{=}p(n_0{-}1)$, where $n_0{\in}\{2,3,4\}$, p is a large prime number.

For both KEM and PKC cryptosystems, the private key is composed as a pair of random binary matrices, $sk{=}\{H,Q\}$, that correspond to the quasi-cyclic $p{\times}pn_0$ parity-check matrix H of $\mathcal{C}(n,k)$ and to a $pn_0{\times}pn_0$ quasi-cyclic sparse binary matrix Q. The matrix H is structured as $1{\times}n_0$ circulant blocks, each of which with size $p{\times}p$ and with d_v non null elements per row/column: $H{=}[H_0,H_1,\ldots,H_{n_0-1}]$, while the matrix Q is a full-rank $n_0{\times}n_0$ block matrix, where each block is a $p{\times}p$ binary circulant matrix, with m non null elements per row/column [6]. The matrices H and Q are generated ensuring that each one of the full-rank n_0 circulant blocks resulting from the product $L{=}HQ{=}[L_0,L_1,\ldots,L_{n_0-1}]$ has a number of asserted bits in the first row equal to $d_v{\cdot}m \ll p$ ($d_v{\cdot}m{\approx}\sqrt{pn_0}$). Subsequently, starting from the multiplicative inverse of L_{n_0-1}, the public key pk of KEM is obtained as:

$$M = L_{n_0-1}^{-1}L = [M_0|M_1|\ldots|M_{n_0-2}|I]\,,$$

where I is a $p{\times}p$ identity matrix. The public key of the LEDAcrypt PKC is obtained by converting M into the corre-

Algorithm 1: Dense-to-sparse Multiplication

```
Input: $A = A_{p-1}x^{p-1} + \ldots + A_1x + A_0$, with $A_l \in \{0,1\}$,
       $0 \le l < p$, $B = B_{p-1}x^{p-1} + \ldots + B_1x + B_0$, with
       $B_l \in \{0,1\}$, $0 \le l < p$
Output: $C = A \cdot B \bmod (x^p + 1)$
Data: $\mathtt{B}$, array with $t$ cells, $\mathtt{B}[0], \mathtt{B}[1], \ldots, \mathtt{B}[t-1]$, each storing the
      integer $l$ of a coeff. $B_l=1$, $0 \le l < p$, $\mathtt{B}[0] < \mathtt{B}[1] < \ldots < \mathtt{B}[t-1]$.
      pu, number of operations which can be parallelized
1  begin
2      $Z \leftarrow 0$                                        // $\mathtt{deg}(Z) < 2p$
3      for $i \leftarrow 0$ to $\lfloor t/\mathtt{pu} \rfloor$ step pu do
4          $Y \leftarrow 0$                                    // $\mathtt{deg}(Y) < 2p$
5          for $j \leftarrow i$ to $i + \mathtt{pu} - 1$ do
6              $Y \leftarrow Y + A \cdot x^{\mathtt{B}[j]}$
7          $Z \leftarrow Z + Y$
8      $Y \leftarrow 0$                                        // $\mathtt{deg}(Y) < 2p$
9      for $j \leftarrow \mathtt{pu} \cdot \lfloor t/\mathtt{pu} \rfloor$ to $t - 1$ do
10         $Y \leftarrow Y + A \cdot x^{\mathtt{B}[j]}$
11     $Z \leftarrow Z + Y$
12     $C \leftarrow Z \bmod x^p + Z/x^p$
13     return $C$
```

Algorithm 2: Dense-to-dense Multiplication

```
Input: $A = A_{p-1}x^{p-1} + \ldots + A_1x + A_0$, with $A_l \in \{0,1\}$,
       $0 \le l < p$, $B = B_{p-1}x^{p-1} + \ldots + B_1x + B_0$, with
       $B_l \in \{0,1\}$, $0 \le l < p$;
       digit_size, maximum schoolbook multiplication factor size
Output: $C = A \cdot B \bmod (x^p + 1)$
1  Function Karatsuba($A$, $B$):
2      if $\mathtt{deg}(A) <$ digit_size or $\mathtt{deg}(B) <$ digit_size then
3          $C \leftarrow A \cdot B$
4      else
5          split $\leftarrow \lceil \mathrm{MIN}(\mathtt{deg}(A), \mathtt{deg}(B)) \rceil / 2$
6          $A_{\mathtt{h}} \leftarrow A / x^{\mathtt{split}}$, $A_{\mathtt{l}} \leftarrow A \bmod x^{\mathtt{split}}$
7          $B_{\mathtt{h}} \leftarrow B / x^{\mathtt{split}}$, $B_{\mathtt{l}} \leftarrow B \bmod x^{\mathtt{split}}$
8          $Y_{\mathtt{l}} \leftarrow A_{\mathtt{l}} \cdot B_{\mathtt{l}}$
9          $Y_{\mathtt{m}} \leftarrow (A_{\mathtt{h}} + A_{\mathtt{l}}) \cdot (B_{\mathtt{h}} + B_{\mathtt{low}})$
10         $Y_{\mathtt{h}} \leftarrow A_{\mathtt{h}} \cdot B_{\mathtt{h}}$
11         $C \leftarrow Y_{\mathtt{h}} \cdot x^{2 \cdot \mathtt{split}} + (Y_{\mathtt{m}} - Y_{\mathtt{h}} - Y_{\mathtt{l}}) \cdot x^{\mathtt{split}} + Y_{\mathtt{l}}$
12     return $C$
13 begin
14     $Z \leftarrow$ Karatsuba($A, B$)                         // $\mathtt{deg}(Y) < 2p$
15     $C \leftarrow Z \bmod x^p + Z/x^p$
16     return $C$
```

sponding systematic generator matrix of the same code as $pk = [D \mid [M_0 \mid \ldots \mid M_{n_0-2}]^T]$, where D is composed as a block matrix with n_0-1 replicas of I on its diagonal.

The LEDAcrypt KEM encryption function takes as input a public key $pk=[M_0 \mid \ldots \mid M_{n_0-2}|I]$ and a message composed as a randomly chosen $1 \times pn_0$ binary vector e, with exactly t asserted bits, $t \ll p$ ($t \approx \sqrt{pn_0}$), and outputs a ciphertext, called syndrome, performing n_0-1 multiplications, as follows: $s = [M_0 \mid \ldots \mid M_{n_0-2}|I]\ e^T$.

The LEDAcrypt PKC encryption function takes three input parameters: a plaintext message composed as a $1 \times p(n_0-1)$ binary vector u, an *error vector* composed as a $1 \times pn_0$ binary vector e, with exactly t asserted bits, with $t \ll p$ ($t \approx \sqrt{pn_0}$), and a public key pk. The algorithm computes an intermediate ciphertext c, performing n_0-1 multiplications as follows:

$$c = [e_0|\ldots|e_{n_0-2}|e_{n_0-1}] + \left[u_0 \mid \ldots \mid u_{n_0-2} \mid \sum_{j=0}^{n_0-2} u_j M_j\right].$$

Given the secret key $sk=\{H, Q\}$, the core steps of both the LEDAcrypt KEM and LEDAcrypt PKC decryption algorithms consider $L=HQ=[L_0, L_1, \ldots, L_{n_0-1}]$ and compute a multiplication of either L_{n_0-1} times the received syndrome or L times the received codeword, respectively, to obtain a syndrome subsequently fed to the LEDAcrypt *syndrome-decoding* algorithm with the aim of recovering the unknown error vector.

Binary Polynomial Multiplications. The encryption and decryption algorithms of the LEDAcrypt KEM require n_0-1 multiplications, each with a pair of $p \times p$ blocks that are represented as polynomials in the ring $\mathbb{Z}_2[x]/(x^p+1)$, where one of the factors is a polynomial with a small number of non null coefficients w.r.t. the p (i.e., $d_v \cdot m \approx \sqrt{pn_0}$ for the multiplications required in the encryption algorithm, and $t \approx \sqrt{pn_0}$ for the ones required in the decryption algorithm), while the other factor has a random number of non null bits (on average $p/2$). In the following, polynomial multiplications in the LEDAcrypt KEM system will be referred to as *sparse-to-dense* multiplications. The encryption algorithm of the LEDAcrypt PKC requires n_0-1 binary polynomial multiplications modulo x^p+1, where the factors have a number of non null bits on average equal to $p/2$, thus will be labeled as *dense-to-dense* multiplications.

Given the peculiar features of the modular polynomial multiplication in LEDAcrypt, their hardware implementation can be differentiated depending on the known number of asserted bits in one of the considered operands.

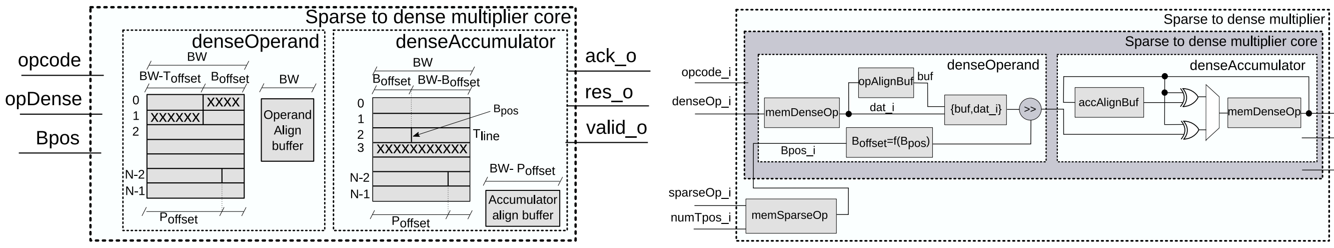
For the case of *sparse-to-dense* multiplication, we adopt the approach described in Algorithm 1, i.e., optimize a schoolbook multiplication approach only adding the partial product where the dense factor is multiplied by a non null coefficient. In this case, the entire multiplication amounts to the sum of shifted replicas of the dense factor, by amounts depending on the exponents of the non null coefficients of the second. Such an approach can be fruitfully parallelized picking a number of shifts and additions to be performed in parallel. For the sake of clarity, Algorithm 1 reports the schoolbook shift-and-add loop highlighting the parallelization factor as the number of parallel units pu which can be employed (lines 3–11). On line 12, the algorithm computes the $\bmod x^p + 1$ operation via a single p-bit wide addition due to the peculiar nature of the modulus itself.

Concerning the case of the *dense-to-dense* polynomial multiplication, we consider the sub-quadratic approach described by Karatsuba, which allows to trade off multiplications for additions. The approach, described in Algorithm 2, performs a multiplication splitting the two factors in two, equally sized, and performing three half-sized multiplications plus six additions/subtractions. The key idea is that, given that multiplications are more expensive than additions/subtractions this will save computational effort. The same operand splitting strategy can be applied further to the three half-sized multiplications, until the operands are small enough that the trade-off between addition and multiplication becomes unfavorable. Once the point where the multiplication-addition trade-off is unfavorable, a schoolbook multiplication strategy between the operands is adopted. The typical case takes place where the operand size, denoted as `digit_size` in Algorithm 2, matches the underlying arithmetic unit operand size.

III. Two polynomial multiplier designs

In this section, we describe our hardware designs implementing the sparse-to-dense and dense-to-dense multiplications.

Sparse-to-dense multiplier. Figure 1a provides a high-level view of the architecture of our Sparse-to-Dense Multiplier (SDM)-*core*, highlighting its interface and the two main components: *denseOperand* and *denseAccumulator*. The *opcode* signal is employed to drive the SDM through four stages of the computation: loading the operands, performing the sparse multiplication as described by Algorithm 1,



(a) (b)

Figure 1: Logic View (a) and Architectural View (b) of the Sparse-to-dense Multiplier

outputting the result, and resetting the accumulator memory. The dense operand is loaded through the *opDense* bus which is `BW`-bit wide, while the values of the t positions of the non null coefficients of the sparse operand B are loaded one per cycle via the *Bpos* bus. The *SDM*-core implements an asynchronous request-response communication protocol, and the acknowledge signal *ack_o* signals the end of the last requested operation. The *res_o* has a width of `BW` bits and is used to output the result of the computation while the *valid_o* signal is asserted when the result is valid. Multiple SDM-*core*s can be instantiated in our SDM, depicted in Figure 1b, acting as the parallel units indicated in Algorithm 1. The results which are progressively accumulated in the *denseAccumulator* components are combined together via a combinatorial XOR cone before they are output via the *res_o* signal of the SDM.

To perform a portion of the sparse-to-dense multiplication, the SDM-*core* employs the *denseOperand* and *denseAccumulator* submodules, both contain p-bit memories implemented as a BRAM, which are employed to store the dense operand, and to accumulate the result of the multiplication, respectively. Operatively, the SDM-*core* module receives the dense operand and stores it in the BRAM memory of the *denseOperand* submodule, which is then only read during the multiplication process. Since the length of the dense operand p will likely not be a multiple of the BRAM line width, which matches the internal bus width `BW`, the *denseOperand* module also contains a small operand alignment buffer of `BW` bits. Such a buffer allows to provide a continuous stream of data to the *denseAccumulator* module when computing, regardless of the amount of shift of the dense operand which has to be computed. A buffer, with a similar purpose, i.e., allow the writing of a continuous stream of accumulated bits is present in the *denseAccumulator* unit, and is denoted as *Accumulator align buffer* with `BW` ($p \bmod$ `BW`) bits.

Operatively, the multiplication of the dense operand by a single monomial of the sparse one starts by reading `BW` bits of the dense operand at the location dictated by the value of the exponent of the non null coefficient of B obtained via the *Bpos*. Since it is highly unlikely that all the `BW` bits are contained in a single line of the *denseOperand* BRAM, the *denseOperand* module performs two read actions, and stores the excess bits in the operand alignment buffer. All subsequent read operations by the *denseOperand* submodule will be able to compose a `BW` contiguous bit string of the dense operand combining the ones already present in the operand alignment buffer, and forward the result to the *denseAccumulator* module. Therefore, only a single cycle overhead is required to cope with the fact that read operations misaligned to the BRAM lines are made. A similar approach is employed in the *denseAccumulator* module to reduce the overhead of updating the current accumulated result in the BRAM. To this end, the accumulator align-buffer is employed to store the

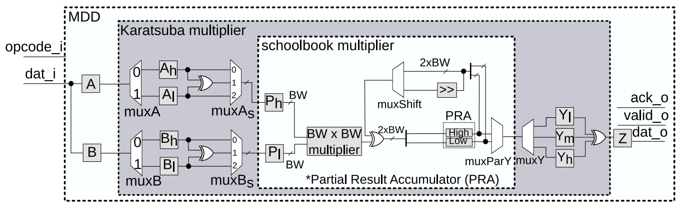


Figure 2: Dense-to-dense binary polynomial multiplier

portion of the updated BRAM line which should be stored in the subsequent clock cycle. This design allows the SDM-*core* unit to perform a monomial-by-polynomial multiplication with a number of clock cycles equal to $\lceil p/\texttt{BW} \rceil + 1$, i.e. with a single clock cycle overhead with respect to reading the dense operand.

Figure 1b depict the architectural view of the shift-and-add multiplier, aka, dense sparse multiplier (DSM), as made of two parts: the DSEM-*core* and the memory for the sparse operand (*memSparseOp*). The proposed design allows to instantiate multiple DSM-*core* units where each operates on a separate element of the sparse operand. To this extent the output of all the DSM-*core* units is bitwise eXclusive OR-ed (XORed) to deliver the final output result.

Dense-to-dense multiplier. The architecture of the Dense-to-Dense Multiplier (DDM), depicted in Figure 2, employs a single data bus to load the two operands, depending on the value of the *opcode_i* signal. The *Karatsuba multiplier* represents the-*core* unit of the DDM and implements Algorithm 2 up to a single level of recursion for the sake of code maintainability. To allow the parallel execution of the three multiplications in the Karatsuba algorithm, the operands A and B are split in two and their respective halves are memorized in two separate BRAMs for each source operand. Similarly, the double precision result Y is memorized as its three components Y_l, Y_m, and Y_h (lines 8–10 of Algorithm 2). Since the operands are polynomials with binary coefficients, both the additions at lines 9 and 11 and the subtractions at line 11 are mapped onto eXclusive-ORs (XORs). Our design exploits the low overhead on the path imposed by a single Boolean operation, computing all the XORs while moving the split operands into, or moving the result out from, the schoolbook multiplier. The architecture depicted in Figure 2 reports a single schoolbook multiplier through which all the three multiplications of split operands are performed in sequence. Note that the schoolbook multiplier and the *muxA*$_s$,*muxB*$_s$, and *muxY* multiplexers can be replaced with three schoolbook multipliers to trade-off resources for a faster computation time. The schoolbook multiplier computes the multiplication between the split operands in a simple operand-scanning fashion, processing the split operands in portions of `BW` bits, matching the bit-width of the internal Karatsuba multiplier bus. To

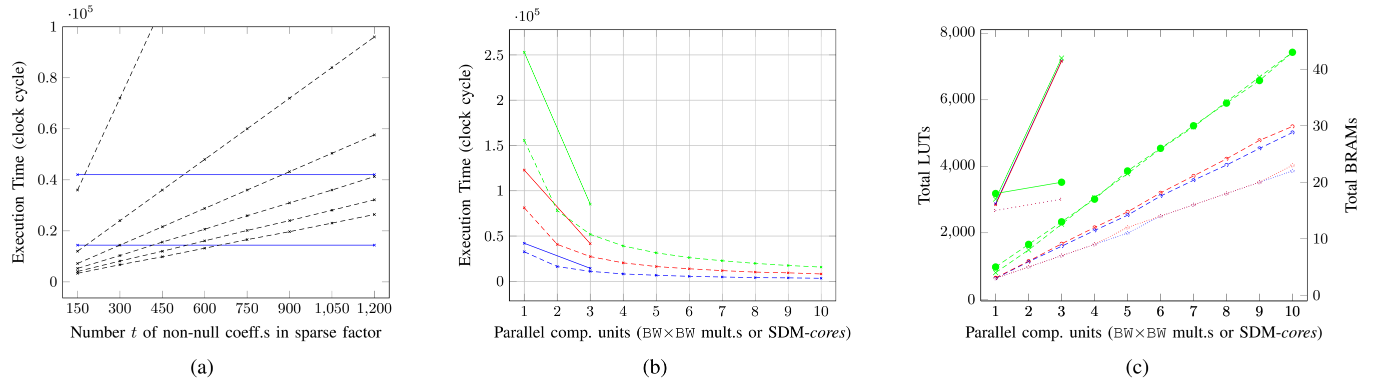


Figure 3: Resource and performance figures of the SDM and DDM implementations. Solid lines refer to the DDM, while dashed lines refer to the DDM, colors are mapped to security categories (1: blue, 3: red, 5: green). Figure (a) Compares SDM and DDM execution times while varying the number of non null coefficients; Figure (b) and (c) report execution performance and area requirements for all security categories of LEDAcrypt KEM parameters for the case of $n_0 = 2$

minimize the number of write actions to the result buffers Y_l, Y_m, and Y_h, the schoolbook multiplier schedules the BW×BW multiplications according to the strategy proposed by Comba, i.e. computing all the products contributing to a given BW sized portion of the result. To implement this strategy, a 2×BW *Partial Result Accumulator* (PRA) register is employed to store the sum of all the contribution to the said portion of the result. When the computation of the sum is completed, the least significant BW bits of the PRA are then committed to the appropriate buffer among Y_l, Y_m, and Y_h, while the most significant ones are copied over the least significant ones in the PRA. The memory units in the DDM unit (*A*, *B*, *Res*) are required to implement an input-output interface equal to the one of the DSM unit.

IV. Experimental validation

We implemented the SDM and DDM multiplier architectures in SystemVerilog and synthesized and implemented them with Vivado v2018.3. We targeted a Xilinx Artix 7 100T FPGA (`xc7a100tcsg324-1`) for the mapping, placing and routing phases, and instructed Vivado to match a 100 MHz operating frequency. Figure 3a reports a comparison in the execution time (in clock cycles) required by the DDM (in solid blue) and the DSM (in dashed black) to compute a sparse polynomial multiplication where t bits are set out of $p = 14,939$. Both the DDM and the SDM performances are shown while varying the number of parallel execution units (BW×BW multipliers for the DDM and SDM-*cores* for the SDM) in $\{1, 3\}$ and $\{1, 3, 5, 7, 9, 11\}$, respectively. As it can be seen, increasing the number of SDM-*cores* allows to push the trade-off point with respect to a DDM up to 600 asserted coefficients.

We employed as case studies the parameter recommended by the LEDAcrypt specification for the three different security levels equivalent to AES-128 (category 1), AES-192 (category 3) and AES-256 (category 5), and consider the case where the public key $pk = [M_0|I]$ has a single non-identity block, i.e., $n_0 = 2$. In this case, the LEDAcrypt KEM encryption computes a single multiplication between polynomials with p equal to $14,939$ bits, $25,693$ bits, and $36,877$ bits, respectively, and a number of non null position, t, in the sparse operand equal to 136, 199 and 267, respectively. Figure 3b and 3c report the performance and required resources in terms of total LUTs and BRAMs by the different LEDAcrypt parameter sets (blue for category 1, red for category 3 and green for category 5). As it can be seen, while the DDM with three BW×BW multipliers provides better performance than the SDM, its resource requirements are far higher. Indeed, comparing solutions with the same amount of resources used, i.e., the aforementioned DDM with an SDM with 10 SDM-*cores*, it is evident how the sparse nature of the multiplications of the LEDAcrypt KEM favour the SDM design over the proposed DDM. In particular, the SDM design with 10 parallel SDM-*cores* completes a sparse multiplication in $3,362$ cycles, $8,162$ cycles and $15,743$ cycles for security categories 1 and 3 and 5 parameters, respectively, considering (as worst-case) all the asserted t terms of the error vector in a single polynomial.

V. Concluding Remarks

We presented two binary polynomial multiplier architectures which can be employed in post quantum cryptosystems such as LEDAcrypt. We analyzed the effectiveness of exploiting the sparsity of one factor to build a sparse-to-dense multiplier which compares favourably to a Karatsuba based design in terms of area and execution time.

Acknowledgments

This work was supported in part by the EU grant: "RECIPE" (RIA), Grant agreement ID: 801137.

References

[1] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Proc. of a symposium on the Complexity of Computer Computations, March 20-22, 1972, IBM T. J. Watson Research Center, Yorktown Heights, New York, USA*. Plenum Press, New York, 1972.

[2] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Trans. Inf. Theory*, vol. 24, no. 3, 1978.

[3] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "A Finite Regime Analysis of Information Set Decoding Algorithms," *Algorithms*, vol. 12, no. 10, 2019.

[4] R. J. McEliece, "A Public-Key Cryptosystem Based on Algebraic Coding Theory," *JPL Deep Space Network*, vol. 44, no. 42, 1978.

[5] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Probl. Contr. and Inf. Theory*, vol. 15, 1986.

[6] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAkem: A post-quantum key encapsulation mechanism based on QC-LDPC codes," in *Post-Quantum Cryptography - 9th Int.'l Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proc.*, ser. LNCS, vol. 10786. Springer, 2018.

[7] ——, "LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate," in *Code-Based Cryptography - 7th International Workshop, CBC 2019, Darmstadt, Germany, May 18-19, 2019, Revised Selected Papers*, ser. LNCS, vol. 11666. Springer, 2019.