POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Flexible NTT-based multiplier for Post-Quantum Cryptography

Original
A Flexible NTT-based multiplier for Post-Quantum Cryptography / Koleci, K., Mazzetti, P., Martina, M., Masera, G.. - In: IEEE ACCESS. - ISSN 2169-3536. - ELETTRONICO. - 11:(2023), pp. 3338-3351. [10.1109/ACCESS.2023.3234816]

Availability:
This version is available at: 11583/2974490 since: 2023-01-10T16:17:18Z

Publisher:
IEEE Access

Published
DOI:10.1109/ACCESS.2023.3234816

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

10 October 2026

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 13 November 2022, accepted 29 December 2022, date of publication 5 January 2023, date of current version 11 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3234816

RESEARCH ARTICLE

A Flexible NTT-Based Multiplier for Post-Quantum Cryptography

KRISTJANE KOLECI, (Graduate Student Member, IEEE), PAOLO MAZZETTI, MAURIZIO MARTINA, (Senior Member, IEEE), AND GUIDO MASERA, (Senior Member, IEEE)
Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Torino, Italy
Corresponding author: Kristjane Koleci (kristjane.koleci@polito.it)

ABSTRACT In this work an NTT-based (Number Theoretic Transform) multiplier for code-based Post-Quantum Cryptography (PQC) is presented, supporting Quasi Cyclic Low/Moderate-Density Parity-Check (QC LDPC/MDPC) codes. The cyclic matrix product, which is the fundamental operation required in this application, is treated as a polynomial product and adapted to the specific case of QC-MDPC codes proposed for Round 3 and 4 in the National Institute of Standards and Technology (NIST) competition for PQC. The multiplier is a fundamental component in both encryption and decryption, and the proposed solution leads to a flexible NTT-based multiplier, which can efficiently handle all types of required products, where the vectors have a length $\approx 10^4$ and can be moderately sparse. The proposed architecture is implemented using both Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) technologies and, when compared with the best published results, it features a 10 times reduction of the encryption times with the area increased by 3 times. The proposed multiplier, incorporated in the encryption and decryption stages of a code-based PQC cryptosystem, leads to an improvement over the best published results between 3 to 10 times in terms of *LC* product (LUT times latency).

INDEX TERMS Number theoretic transform, accelerator, convolution, polynomial product, applied cryptography, post-quantum cryptography, QC-MDPC codes, hardware design, ASIC, FPGA.

I. INTRODUCTION

Post-Quantum Cryptography (PQC) is nowadays a promising research field that aims at improving the security of currently adopted cryptosystems. Indeed, a new approach in Public Key Cryptography is required to handle the threats posed by Quantum Computers [1].

The advancements in the field of computation capabilities of modern computers and quantum technologies impose to reconsider the way communication and data are encrypted and keys are exchanged. As pointed out in [1], primitives currently employed in Asymmetric Cryptography, which are fundamental to establish a secure channel, are proved to be broken by a quantum computer executing the algorithms presented in [2] and [3]. Asymmetric Cryptography is also referred to as Public Key Cryptography, and it uses a pair of keys, the Secret Key (**SK**) and the Public Key (**PK**). The Secret Key is generated and then the Public key is easily obtained, but the reverse of the function is hard to obtain by using conventional computing. However, this is not the case if a quantum computer is available. As an example, the algorithm presented in [2] and a quantum computer would make it possible to quickly solve the prime factorization problem in the RSA (Rivest–Shamir–Adleman) cryptosystem, so compromising its security. Additionally, in a symmetric system, like the widely used AES-128 system, the Grover quantum algorithm halves its security level, thus requiring to double the size of the keys [4].

PQC addresses the study and implementation of new asymmetric cryptosystems able to overcome these security issues in the next future. In this scenario, the National Institute for Standardization (NIST), in 2016, launched a competition

The associate editor coordinating the review of this manuscript and approving it for publication was Stavros Souravlas.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

to find a new standard for Public Key Cryptography. The best proposals rely on functions that are proven to be hard to reverse, NP-hard problems, and admit efficient hardware and software implementations.

The first cryptosystem based on error correcting codes was proposed by McEliece, and uses Goppa codes [5], while recent advancements employ Quasi-Cyclic Low/Moderate-Density Parity-Check (QC LDPC/MDPC) codes, in order to reduce the complexity and the memory requirements of the whole system [6], [7]. The NIST competition has now reached round 4, and three code-based cryptosystems are still competing [8]: Classic McEliece [9], BIKE [10] and HQC [11].

The polynomial product is one of the operations intensively used in the encryption and decryption primitives of these cryptosystems, including one of the winners of the selection process in Public Key Encryption [12], [13]. Important aspects that affect both computational complexity and latency in polynomial multiplication are: the length of polynomials, the number of non zero elements, and the type of coefficients. In McEliece cryptosystems, based on LDPC/MDPC codes, the encryption involves a dense variable, while the decryption works on a sparse variable.

The main methods known in the literature for polynomial product calculation are the Schoolbook, the Karatsuba [14] and the Schönhage-Strassen [15] algorithms, which represent the state of the art in this domain:

- The Schoolbook algorithm is widely adopted in systems based on QC-LDPC/MDPC codes [16], [17], [18]. The implementation efficiency of this approach strongly depends on the density of the involved matrices: while the efficiency is very good for a low density matrix, it becomes rather poor when the density of the matrix is large (as at the encoding side).
- The Karatsuba algorithm has a reduced time complexity, if used to compute the polynomial products required in the encryption [19], but its efficiency is drastically reduced when applied in the decryption.
- The Schönhage-Strassen algorithm [20] can be implemented to efficiently handle the sparse and medium density vectors as well as the dense vectors. Therefore, it is preferable as a solution applied to both encryption and decryption.

This work presents a novel architecture supporting the polynomial products involved both in the encryption and in the decryption. The proposed multiplier is based on the Number Theoretic Transform (NTT) and it has been optimized for binary polynomial multiplications, where input operands are binary and results can be either binary or integer.

The same datapath architecture operates with both sparse and dense matrices. The proposed design has been synthesized for both FPGA and ASIC technologies. Moreover, the achieved results have been compared with state-of-the-art architectures that were previously published for code-based PQC and NTT multipliers able to support similar data sizes.

The work is organized as follows: in Section II, code-based Cryptosystems are presented, while in Sections III and IV the cyclic matrix product and the NTT-based multiplier are described. The details of the proposed architecture are given in Section V, and finally the results and the comparisons against the state of the art are provided in Section VI.

II. CODE-BASED CRYPTOSYSTEMS

Code-based cryptosystems adopt error correcting codes to encrypt a secret message. In communication systems, error correcting codes are commonly employed to remove the errors caused by a noisy channel; on the contrary, in code-based cryptography, errors are intentionally inserted at the encoding (encryption) side and then corrected at the decoding (decryption) side.

The security is mainly guaranteed by the complexity of the decoding of a codeword: with t errors and without knowing the decoding matrix, the problem is proven to be NP-hard [21].

In this approach, the encryption **PK** and decryption **SK** keys are the encoding and decoding matrices of an LDPC code: $\mathbf{G}$, the generator matrix and $\mathbf{H}$, the parity check matrix.

The structure of the $\mathbf{H}$ matrix is $\mathbf{H} = [\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_{n_0-1}]$, where each block (0 to $n_0 - 1$) is a binary square cyclic matrix with dimension N and d_v is the weight of each column; therefore, the matrix $\mathbf{H}$ has size $(N, N \cdot n_0)$. $\mathbf{h} = [\mathbf{h_0}, \ldots, \mathbf{h_{n_0-1}}]$ indicates the first row in $\mathbf{H}$. The $\mathbf{G}$ matrix is obtained as $\mathbf{G} = \mathbf{H}_{n_0-1}^{-1} \cdot \mathbf{H} = [\mathbf{I}_N, \mathbf{G}_l]$, where $\mathbf{I}_N$ is the identity matrix of size N, $l = n_0 - 1$, $\mathbf{G}_l$ is a quasi cyclic matrix with density at most $d_v \times d_v$, and it is the result of $\mathbf{H}_{n_0-1}^{-1} \cdot [\mathbf{H}_1, \ldots, \mathbf{H}_{n_0-1}]$. We also use $\mathbf{g_l}$ to refer to the first row of $\mathbf{G}$. The encoding adds redundancy to the input vector $\mathbf{m}$, of length $N \cdot (n_0 - 1)$, such that the encoded vector $\mathbf{c}$ (the codeword), with length $N \cdot n_0$, satisfies the set of parity equations, defined by the matrix $\mathbf{H}$.

Since **PK** and **SK** are large, cyclic matrices are adopted to reduce the memory requirement. The structure of a cyclic matrix is:

$$\mathbf{A} = \begin{bmatrix} a_0 & a_{n-1} & a_{n-2} & \ldots & a_1 \\ a_1 & a_0 & a_{n-1} & \ldots & a_2 \\ a_2 & a_1 & a_0 & \ldots & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & a_{n-3} & \ldots & a_0 \end{bmatrix} \quad (1)$$

The first column of $\mathbf{A}$, denoted as a (the first row is a^T), is sufficient to describe the whole matrix. Moreover, since the codes are sparse/moderate sparse, only few elements of the row are asserted. Hence, only the positions of the ones in a, instead of the complete column, can be stored to reduce the memory footprint. Therefore, to describe matrices $\mathbf{H}$ and $\mathbf{G}$, we only store the asserted positions of the h and g_l vectors.

A. ENCRYPTION

The encryption starts from the secret message, m, encoded with $\mathbf{G}$ into c. The error e is a binary vector with length $N \cdot (n_0)$

and t positions asserted. Thus, the encryption can be written as:

$$x = m \cdot \mathbf{G} \oplus e. \tag{2}$$

It is worth noting that the $\oplus$ operations is the sum in the Galois field $\mathbb{GF}(2)$, thus, the error flips t bits of the codeword, $c = m \cdot \mathbf{G}$.

The codeword c satisfies the parity equations defined by the code, and the errors introduced in c make some parity equations unsatisfied.

B. DECRYPTION

The decryption removes the errors in x, in order to retrieve the original message m. In LEDAcrypt and BIKE, the iterative decoding is based on a modified Bit Flipping (BF) algorithm [22]; in both schemes it has been designed such that it is suitable for PQC. The decoder receives x and, by means of $\mathbf{H}$, it evaluates the Syndrome (s) and the Unsatisfied Parity Checks (*upc*), which are respectively the response to the parity equations with the input x and the number of wrong (*unsatisfied*) parity check equations, where each bit of x is involved.

The bits with a value of *upc* over a specific threshold (b) are flipped. The decoder iteratively evaluates s, *upc* and flips specific bits until s becomes equal to $\mathbf{0}$ (vector with all elements set to 0).

The LEDAcrypt and BIKE bit flipping based decoder is reported in Algorithm 1, which incorporates two alternative forms.

Algorithm 1 Bit Flipping Based Decoder

Input: ciphertext $x^0 \in \mathbb{F}_2^N$, QC matrix $\mathbf{H} \in \mathbb{F}_2^{N \times n_0 N}$, maximum number of iterations $It_{max} \in \mathbb{N}$
Output: secret message m or decoder failure
Initialization: $It = 0$, $e^0 = \mathbf{0} \in \mathbb{F}_2^n$

1: $s^{It} = \mathbf{H} \cdot (x^{It})^T$ ▷ Syndrome Evaluation
2: **while** $It < It_{max}$ *or* $s^{It} = \mathbf{0}$ **do**
3: $upc^{It} = s^{It} \cdot \mathbf{H}$ ▷ UPC Evaluation
4: $b^{It} = f(s^{It})$ ▷ Flip Condition
5: **for** $i \in [1, n]$ **do**
6: $e_i^{It} = e_i^{It} \oplus 1$
7: **end for** ▷ Error Position Selection
8: $x^{It+1} = x^{It} \oplus e^{It}$ ▷ Error Correction
$\langle s_e^{It+1} = \mathbf{H} \cdot (e^{It})^T \rangle$ ▷ *Syndrome Correction*
9: $s^{It+1} = \mathbf{H} \cdot (x^{It+1})^T$ ▷ Syndrome Evaluation
$\langle s^{It+1} = s^{It} \oplus s_e^{It+1} \rangle$ ▷ *Syndrome Update*
10: $It = It + 1$ ▷ Increase iteration counter
11: **end while**
12: **if** $s^{It} = \mathbf{0}$ **then**
13: **return** m ▷ Decoding is successful
14: **else**
15: decoding failure
16: **end if**

Both LEDAcrypt and BIKE algorithms use the function $f(\cdot)$ to evaluate the threshold b^{It} (line 4). Alternative options to implement this function have been proposed in [10] and [23].

As said, the algorithm can be written in two variants: in the first one, the processing tasks at lines (8) and (9) are executed (Error Correction and Syndrome Evaluation). Alternatively, the correction step can be applied to the syndrome, by first computing the *Syndrome Correction*, $s_e^{It+1} = \mathbf{H} \cdot (e^{It})^T$, and then the *Syndrome Update*, $s^{It+1} = s^{It} \oplus s_e^{It+1}$. This alternative form is shown in Algorithm 1 using the italic font and angle brackets. At the end of the algorithm, if the syndrome is zero or the maximum number of iterations is reached, then the algorithm stops.

As it can be observed, both encryption and decryption need multiplications: in particular, $\mathbf{c}$ at the encryption side, and $\mathbf{s}$ and $\mathbf{upc}$ in the decryption procedure are obtained as the results of products involving a QC matrix. Given the cyclic structure showed in (1), this product operation is referred to as cyclic matrix product.

In order to obtain an efficient bit flipping decoding, an effective implementation of the cyclic matrix product is required. The proposed multiplier assumes the parameters listed in Table 1, which are derived from the LEDAcrypt [24] and BIKE [10] proposals.

TABLE 1. QC-MDPC McEliece crytopsystems parameters, with $n_0 = 2$.

	LEDAcrypt[24]			BIKE[10]		
NIST Cat.	N	d_v	t	N	d_v	t
1	21,701	71	130	12,323	119	134
3	37,813	103	196	24,659	206	199
5	58,171	137	262	40,973	274	264

III. CYCLIC MATRIX PRODUCT

The generic cyclic matrix multiplication is intended as the product $r = v \cdot \mathbf{A}$ between a $1 \times N$ vector $v = [v_0, v_1, \ldots, v_{N-1}]$ and a $N \times N$ cyclic matrix $\mathbf{A}$.

The result of the multiplication is computed as

$$\begin{array}{rcccccc} r_0 & = & v_0 \cdot a_0 & v_1 \cdot a_1 & \ldots & v_{N-1} \cdot a_{N-1} \\ r_1 & = & v_0 \cdot a_1 & v_1 \cdot a_0 & \ldots & v_{N-1} \cdot a_2 \\ r_2 & = & v_0 \cdot a_2 & v_1 \cdot a_1 & \ldots & v_{N-1} \cdot a_{N-3} \\ \vdots & = & \vdots & \vdots & \ddots & \vdots \\ r_{N-1} & = & v_0 \cdot a_{N-1} & v_1 \cdot a_{N-2} & \ldots & v_{N-1} \cdot a_0 \end{array} \tag{3}$$

A. POLYNOMIAL PRODUCT AND INTEGER MULTIPLICATION

The vectors a and v can be seen as polynomials with coefficients a_i and v_i. The same description holds for the product $\mathbf{A} \cdot v$, and the result in Equation (3) is equivalent to $r = a^T \cdot \mathbf{V}$. Thus, the techniques applied in polynomial and integer multiplication can be applied to the cyclic matrix product in LEDAcrypt and BIKE.

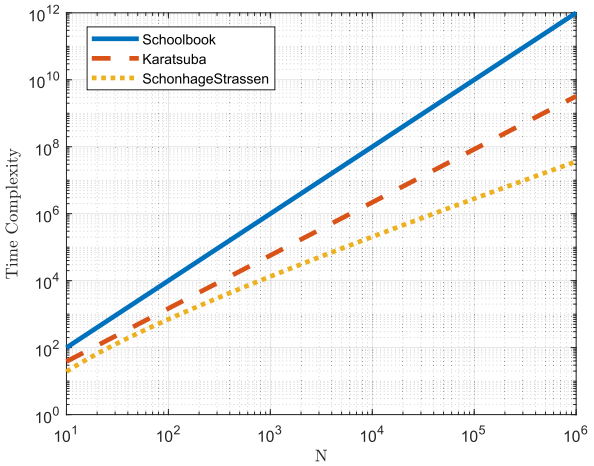


FIGURE 1. Complexity comparison among Schoolbook, Karatsuba and Schönhage–Strassen algorithms.

The choice among the best known polynomial product algorithms mentioned in Section I can be made based on time complexity and implementation complexity. The metric one can use to evaluate the time complexity is the number of element-wise multiplications, which depends on the length of the polynomials (N). By using the big O notation, the time complexity of the Schoolbook, Karatsuba and Schönhage–Strassen algorithms can be expressed as $O(N^2)$, $O(N^{\log_2(3)})$ [14], and $O(N \cdot \log(N) \cdot \log(\log(N)))$ [25] respectively (Figure 1).

LEDAcrypt and BIKE algorithms include cyclic products with a binary cyclic matrix, where $N \approx 10^4$. Therefore, both the Karatsuba and Schönhage–Strassen algorithms are preferable in terms of time complexity. Between these two options, we choose the Schönhage–Strassen algorithm because of its lower time complexity, especially with sparse matrices [19], [20], [26].

IV. NTT BASED CIRCULANT MATRIX PRODUCT

The Schönhage-Strassen algorithm [25] is based on the convolution theorem. Given a vector y and a cyclic matrix $\mathbf{X}$, the cyclic matrix product $y \cdot \mathbf{X}$ is equivalent to the cyclic convolution ($conv$) between x and y, where x is the first column of $\mathbf{X}$:

$$z = conv(x, y) = y \cdot \mathbf{X} \tag{4}$$

By defining X, Y and Z as the frequency-domain representations of signals x, y and z respectively, (4) can be replaced with the element-wise multiplication:

$$Z = X \odot Y. \tag{5}$$

Thus, if one applies known algorithms to calculate the direct Fourier Tranform ($DFT(\cdot)$) and the inverse Transform ($IDFT(\cdot)$), then (4) can be rewritten as:

$$z = conv(x, y) = IDFT(DFT(x) \odot DFT(y)) \tag{6}$$

The direct Fourier Transform is conveniently replaced with the Number Theoretic Transform (NTT) [27], as it is specifically suited to transform and process integer vectors without using floating point numbers and exploiting modular arithmetic. Generally, an NTT-based multiplier involves the transform of the input values, x and y, into X and Y, their element-wise multiplication, and the inverse transform to obtain z.

The NTT for the integer vector x is defined as

$$X(k) = \sum_{i=0}^{N-1} (x(i) \cdot \alpha^{ik}) \mod P, \tag{7}$$

where N is the length of x; α and P are referred to as *radix* and *modulus*, respectively.

The values of α and P depend on N and have to be selected such that:

- The overflow condition does not occur during the computation of the convolution; thus, P must be larger than M^2N, with M being the largest integer in the input signals (it is '1' for binary vectors).
- P must be a number for which we can define a primitive root of unity r.[1]
- α is a function of r and P; if P is in the form $kN + 1$, we have $\alpha = r^k \mod P$.

Any prime value of P in the form $kN + 1$, for which a primitive root of unity can be found, is valid; then α is simply equal to $r^k \mod P$. The condition on P restricts the possible values that can be employed, but, in general, it is not mandatory for P to be prime and a low value of P has the advantage of reducing the size of the operators in the NTT.

In this work, the NTT-based multiplier is conceived for the length and type of vectors employed in LEDAcrypt and BIKE cryptosystems, which use binary vectors. In the case of the decoder, the involved operands are the ciphertext, $x = [x_0, x_1]$, and s, plus the first row of the $\mathbf{H}$ matrix, SK in the Decoder, $h = [h_0, h_1]$. The QC-MDPC codes adopted here are block codes, with n_0 block, then the variable x and $\mathbf{H}$ have a block structure.

As for the encoder, the operands are m and g_l. It is worth noting that the length of the vectors, as it is clear from Table 1, is not a power of two; instead, these lengths are prime numbers. In order to meet the NTT requirement, the vector length is extended by means of different padding approaches described in the following. Moreover, the sparsity of h is exploited by the NTT-sparse algorithm, in order to reduce the overall execution time.

The algorithm flow of the convolution-based multiplier is shown in Figure 2. Since h is sparse, it is firstly padded with a specific algorithm and then transformed by the NTT-sparse algorithm, whereas for x a trivial padding is applied and then it is transformed by the usual NTT algorithm; the two transforms are then element-wise multiplied and finally

[1] A primitive root of unity is any number such that $r^x \mod P = 1$ and $r^y \mod P \neq 1$ for any $y < x$ [27].

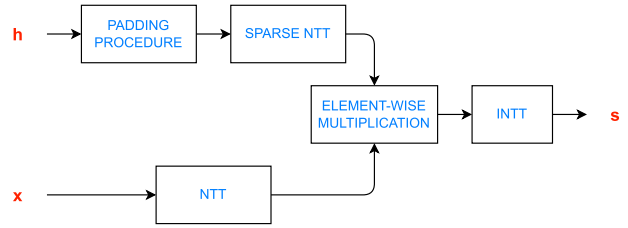


FIGURE 2. Convolution-based multiplication for Syndrome Evaluation.

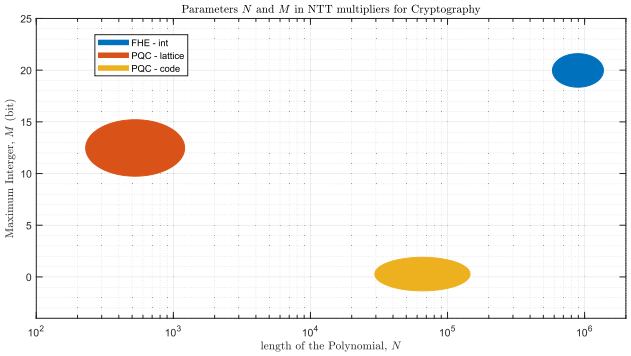


FIGURE 3. The NTT parameters compared in terms of length of the polynomial (N) and value of M.

the inverse transform (INTT) is computed to provide the syndrome s.

Code-based PQC is not the only field in which the NTT algorithm is required. It appears in a variety of applications and the parameters of the involved polynomials strongly depend on the domain. In the cryptography domain, we can distinguish between two main applications: Fully Homomorfic Encryption (FHE) and PQC. The difference in the parameters is presented in Figure 3 where, despite Lattice Based and Code-based cryptosystems are PQC primitives, their values for N, M, and P are quite different.

In the following, the NTT and NTT-sparse algorithms are described, as key elements in the implementation of the multiplication in (4), where x is the sparse vector, y the dense vector, and z the result. At the decoding side, the two algorithms are applied to the evaluation of s and upc. At the encoding side, only the NTT algorithm is required, as both vectors are dense.

A. ALGORITHMS FOR NTT COMPUTATION

The well-known radix-2, Cooley-Tuckey, decimation-in-frequency FFT structure, composed by several butterfly units (see Figure 4) is used to efficiently implement the NTT. The equation (7) can be rearranged as

$$X(k) = \sum_{j=0}^{N/2-1} x(2j)\alpha^{(2j)k} \mod P + \sum_{j=0}^{N/2-1} x(2j+1)\alpha^{(2j+1)k} \mod P \quad (8)$$

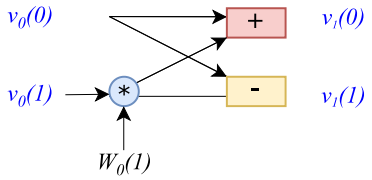


FIGURE 4. The basic butterfly unit. The input $v_0(1)$ is multiplied by the twiddle factor $W_0(1)$ and the outputs are $v_1(0) = v_0(0) + v_0(1) * W_0(1)$, and $v_1(1) = v_0(0) - v_0(1) * W_0(1)$.

Figure 5 gives the structure of the butterfly graph to compute the NTT of a vector x. The $N = 8$ example shows the sequence of multiplications and additions/subtractions computed in each stage of the NTT computation, and all of them are implemented with the basic block in Figure 4. The input vector x (listed in bit reverse order) is renamed as $v_j(i)$, where the index $j = 0, 1, 2$ is referred to the computation stage and $i = 0, \ldots, 7$.

In a straightforward approach, the whole graph is computed uniformly, with the same processing extended to all stages. However, the computation can be drastically simplified if a sparse input vector is received, where most of the elements are equal to 0. It is convenient, for sparse vectors, to process the asserted positions as separate values in the NTT-graph, by propagating them independently of the others. In the basic unit of Figure 4, no computation is needed if the inputs are 0, and reduced complexity calculations are possible when a single input is asserted: $v_1(0) = W_0(1)v_0(1)$ and $v_1(1) = -W_0(1)v_0(1)$, when $v_0(0) = 0$; or $v_1(0) = v_1(1) = v_0(0)$ when $v_0(1) = 0$ (where $W_l(j)$ indicates the twiddle factor at stage l and position j, corresponding to values $\alpha^{(2j)k}$ and $\alpha^{(2j+1)k}$ in Equation (7)).

To exploit these simplifications, in our approach, the NTT stages are split in two ranges: a sparse NTT computation is applied from the input up to a limit stage (l_{lim}); then, the asserted values are merged and the normal, dense NTT computation is applied for the remaining stages.

In the example of Figure 5, most of the inputs are 0, except $v_0(3)$ and $v_0(4)$ (asserted position $\mathbf{Pos_x} = [3, 4]$). Since stages $l = 0$ and $l = 1$ work on vectors having several elements equal to 0s, in this case, $l_{lim} = 2$, and the sparse computation is limited to stages $l = 0$ and $l = 1$. As it can be observed, the asserted element $v_0(3)$ occupies a single position at stage $l = 0$. When this input propagates to stage $l = 1$, two values need computation and four positions must be computed at stage $l = 2$.

The computation required at each propagation of an asserted value is based on the binary representation of the input positions of the asserted inputs. For example, for the asserted input $v_0(3)$, the binary representation of position 3 is [0, 1, 1]. Every bit in this representation refers to a butterfly stage, with the least significant bit corresponding to the first stage. A bit equal to 0 implies that the corresponding value only needs to be copied to the two butterfly outputs; on the

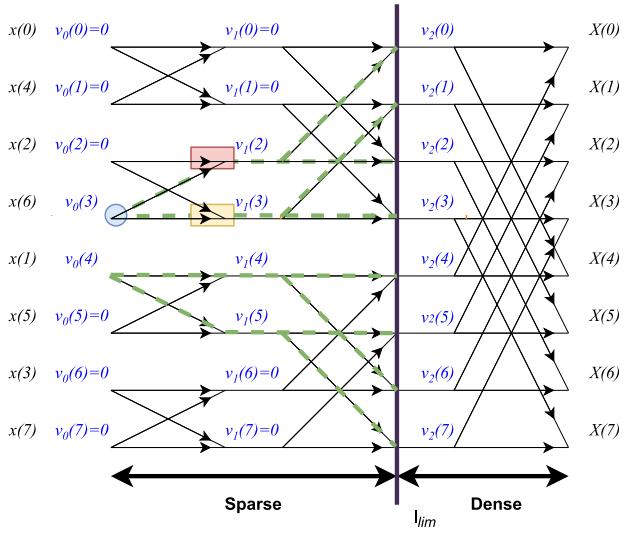


FIGURE 5. 8-points NTT with asserted bit propagation highlighted. The sparse and dense stages are divided by a bold line. The base butterfly unit is highlighted for $v_0(3)$ propagation to stage $l = 1$.

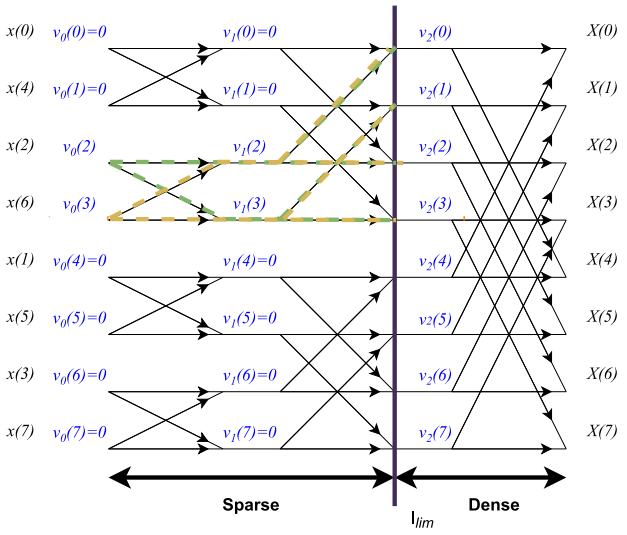


FIGURE 6. Sparse algorithm with the occurrence of two consecutive asserted bits. The following elements are computed separately and then merged at stage l_{lim}.

contrary, a bit equal to 1 implies that both outputs of the butterfly must be calculated in the simplified form. Thus, for the asserted input $v_0(3)$ of Figure 5 with binary position [0, 1, 1], we have a 1 at both stages $l = 0$ and $l = 1$. At stage $l = 0$ one multiplication and one subtraction are required to compute the two output values $v_1(2) = W_0(3)v_0(3)$ and $v_1(3) = P - W_0(3)v_0(3)$. The same kind of processing is needed at stage $l = 1$, where two multiplications are performed and four values are computed as $v_2(0)$, $v_1(0)$, $P-v_2(0)$ and $P-v_1(0)$. A different computation is done on the asserted element $v_0(4)$, having binary position [1, 0, 0]. In this case, we have bits equal to 0 for both stages $l = 0$ and $l = 1$; therefore, just replica of the asserted inputs are required at the butterfly outputs. Indeed, at stage 2, the same value, $v_2(4)$, appears in positions from 4 to 7.

Because of the linearity of the NTT, the simplified computations required for each asserted input can run independently of the others, as a separate transform contribution. When the l_{lim} stage is reached, the separate transforms are merged and the computation proceeds with the normal, dense NTT. In same cases, the merge operation is as simple as the plain concatenation of values at stage l_{lim}: this is the case for the example in Figure 5, where single values are available at the output of stage 1 in every position; therefore, they are concatenated to form the 8-input vector for the last stage. In other cases, multiple values are available for a position at the boundary between sparse and dense computation, because of the separate contributions derived with the described procedure. Under this case, the merge operation needs a sum of the multiple values at a given position, before concatenation. An example of this second case is shown in Figure 6, where the asserted inputs $v_0(2)$ and $v_0(3)$ are independently processed, leading to multiple contributions, which are added at the input of the last stage.

Algorithm 2 NTT, Sparse Computation

Input: $\mathbf{Pos}_x$, asserted position in x,
l_{lim}, the limit Stage,
Output: $\mathbf{SX}$, Sparse Transform of x;

for $i_v = 1 : nze$ **do**
  $PosBin = to_bin(\mathbf{Pos}_x(i_v))$
  for $l = 0 : l_{lim}$ **do**
    $\mathbf{SX}^{old}(i_v) = \mathbf{SX}(i_v)$
    $n_{op} = 2^l$
    for $k = 1 : n_{op}$ **do**
      if $PosBin(l) = 1$ **then**
        $\mathbf{SX}(i_v, 2k) = W_l\,\mathbf{SX}^{old}(i_v, k)$
        $\mathbf{SX}(i_v, 2k+1) = P - W_l\,\mathbf{SX}^{old}(i_v, k)$
      else
        $\mathbf{SX}(i_v, 2k) = \mathbf{SX}^{old}(i_v, k)$
        $\mathbf{SX}(i_v, 2k+1) = \mathbf{SX}^{old}(i_v, k)$
      end if
    end for
  end for
end for

In a more formal way, the proposed approach can be described with the pseudo-code in Algorithm 2, where the received inputs are: the sparse stage limit l_{lim}, the number of non-zero elements nze, and their positions in vector $\mathbf{x}$, named $\mathbf{Pos}_x$. The binary representation of these positions is reported as $PosBin$, and the single bit in the binary pattern is indicated as $PosBin(l)$ for stage l. Based on the value of $PosBin(l)$, two possible operations are performed:

1) If $PosBin(l) = 1$, a multiplication and a subtraction are executed to update the butterfly outputs;
2) If $PosBin(l) = 0$, the first butterfly input is simply copied to the two outputs.

At every stage in the range 0 to l_{lim}, the computed transform contributions are in a matrix $\mathbf{SX}(i_v, k)$, this is updated at every stage (l) and its rows, $i_v = 0 \ldots nze$ correspond to the asserted positions; the columns, k, are at most $2^{l_{lim}}$.

Later, in Section V, the multiplication by the twiddle factor (the blu circle in Figure 4) is indicated as `MPY`, while `ADD` and `SUB` correspond to the red and yellow boxes, respectively.

B. MODULAR ARITHMETIC AND NTT COMPUTATION

Modular arithmetic is fundamental in the computation of the NTT. The result of every addition, or subtraction must be in the range $[0, P-1]$, thus, if the value is outside $[0, P-1]$, supplemental computation is needed to adapt the result. Modular multiplication is similar, but it is more complex, as the result is the reminder of the division between the product and P. To have an efficient implementation, the division must be avoided, and this can be achieved by adopting the Montgomery reduction method [28], which exploits a support modulo R, to be conveniently selected.

Let us consider the product $a \cdot b = c$, where a and b, the operands, and c, the result, are in the range $[0, P-1]$. To calculate the multiplication, the operands must be converted to the Montgomery form [29]. The modulus is evaluated with respect to a number R selected such that $R > P$ and $GCD(R, P) = 1$. The conversion is performed by means of a multiplication by an integer $y_R \in [0, R-1]$, selected s.t. $P \cdot y_R = -1 \bmod R$.; thus, the equivalent of a in the Montgomery form is $a_m = (a \cdot R) \bmod P$. Similarly, $b_m = (b \cdot R) \bmod P$.

Then, the Montgomery multiplication takes place according to Algorithm 3.

Algorithm 3 Montgomery Multiplication

Input: integer factors of the multiplication, a_m, b_m; integer y_R;
Output: product in Montgomery format, c_m;

1: $x = a_m \cdot b_m$ ▷ Product
2: $x_R = x \bmod R$ ▷ Modular Reduction with R
3: $s = (x_R \cdot y_R) \bmod R$
4: $c_R = (x + s \cdot P)/R$
5: **if** $c_R < P$ **then**
6: $\quad c_m = c_R$
7: **else**
8: $\quad c_m = c_R - P$
9: **end if**

Notice that, if R is conveniently selected as a power of 2, then only multiplications by constants, bit shifting and bit masking operations are required in lines 2 to 4 of Algorithm 3. The result must be converted back from Montgomery to integer domain, which requires additional complexity. If this method is applied to compute a single operation, the input and output conversions introduce a relevant computational overhead, which limits the advantage of the overall approach. However, since the Montgomery method preserves the distributive property, it is not required to apply the conversion to each single operation; thus, only NTT inputs and INTT outputs are converted, instead.

C. PADDING

If the size of the inputs is not a power of 2, then a proper padding is required. Alternatively, in the case of an input vector length that is a prime number, the Bluestein algorithm [30] could be used. This method efficiently computes the NTT of a prime length vector with a time complexity of $N \log N$; however, it does not support sparse vectors.

Two padding methods have been selected in this work: the *PrimePadding* applied to $\mathbf{a}$ and the *ZeroPadding* applied to $\mathbf{x}$. It is convenient to apply the *PrimePadding* to the sparse vector, such that its sparsity remains unchanged, while the dense vector is padded with zeros to reduce the complexity of the computation.

1) PRIMEPADDING

The *PrimePadding* vector extension [30], is based on the idea of extending a, the first row of $\mathbf{A}$, to a' such that $\mathbf{A}'$, the new matrix, is still circulant and contains $\mathbf{A}$. The extension takes a vector of length N and provides a vector of length N', with $N' > N$ and N' a power of 2.

An example of the *PrimePadding* extension is showed in (9) for the extension from N to N', with $d_N = N' - 2N$, applied to the first column in matrix $\mathbf{A}$ given in (1).

$$\begin{bmatrix} a_0 \ \ldots \ a_N - 1 \ 0 \ \ldots \ 0 \ a_N - 1 \ \ldots \ a_1 \end{bmatrix} \tag{9}$$

The extension of the base vector consists in concatenating d_N zeros and then the mirrored $N-1$ elements of $\mathbf{a}$ on the right part of the vector.

The PrimePadding is applied to vector x, the sparse vector in Equation (4). In the complete cryptosystems, this corresponds to the first row in matrices $\mathbf{H}$ and $\mathbf{H^T}$ and to the error vectors e^{It}.

2) ZEROPADDING

The *ZeroPadding* is applied to vector x, in Equation (4). The dense vector, in the cryptosystem, are the message m and the encoding matrix $\mathbf{G}$.

The whole convolution operation requires the first N components of the output vector. The idea is shown in (10), where the multiplication is the Syndrome evaluation ($s = \mathbf{H} \cdot x$) in the Decoder,

$$s = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 & 0 & \ldots & 1 \\ 1 & 0 & 1 & \ldots & 1 & 0 & \ldots & 0 \\ 1 & 1 & 0 & \ldots & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & 0 & \ldots & 1 \\ 0 & 0 & 0 & \ldots & 1 & 1 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & 1 \\ 0 & 0 & 1 & \ldots & 0 & 1 & \ldots & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 1 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{10}$$

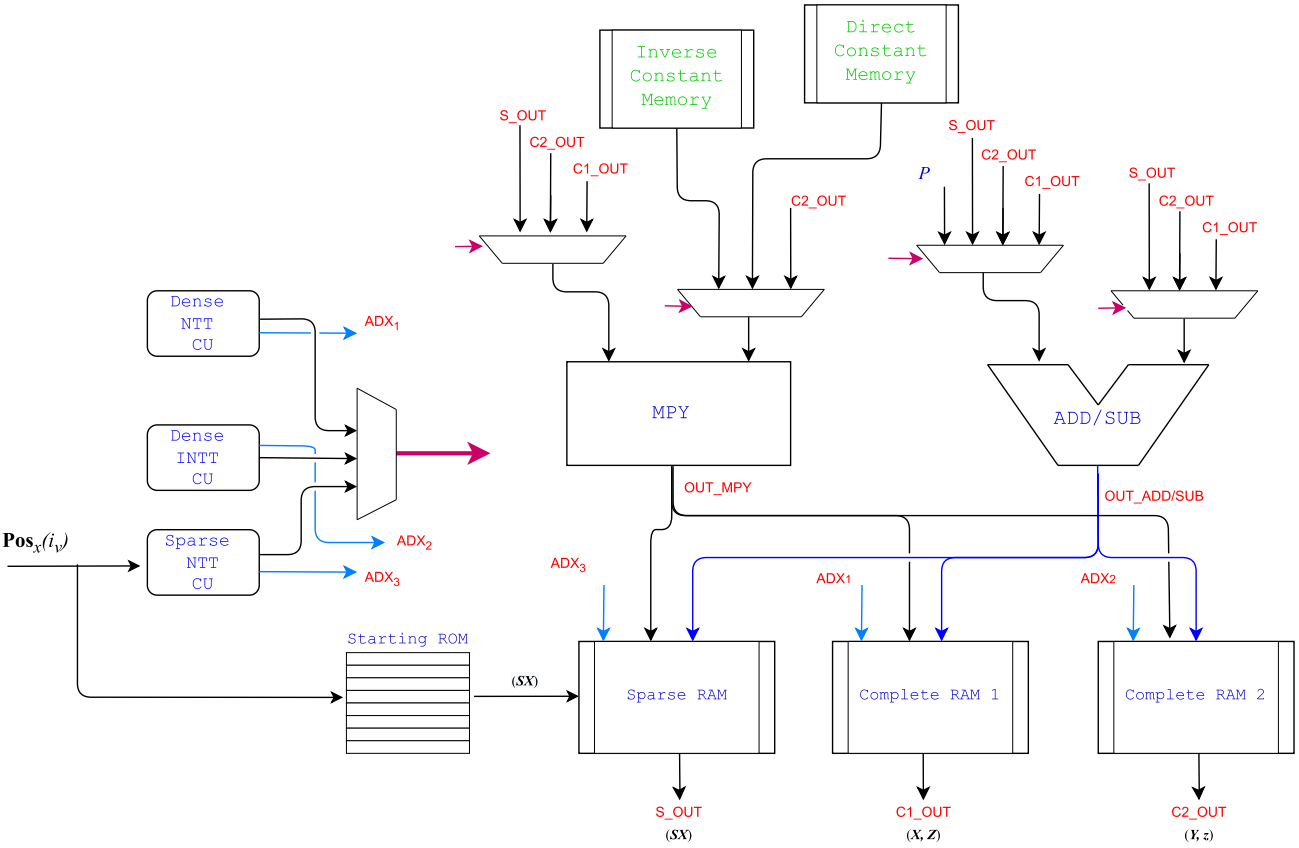


FIGURE 7. Data Path with elements for Sparse-NTT computation included. The `MPY` and `ADD/SUB` represents the basic units, in the complete architecture n_u elements has been instantiated.

In (10), the original circulant matrix is highlighted with a red square, while the dashed orange box represents the first row of the matrix, properly padded with the *PrimePadding* technique. Similarly, the input vector (blue box) has been zero-padded to N'. For the result we need to consider only the first N components,

$$\begin{bmatrix} 0 & 1 & 1 & \dots & 0 & 1 & \dots & 0 \end{bmatrix} \tag{11}$$

which are highlighted in green in (11).

The advantage of this padding technique is that the matrix is still sparse and $N' - N$ elements of the input vector are set to zero, which allows skipping part of the computation.

V. ARCHITECTURE

The NTT-based multiplier architecture handles four operations to compute the convolution: the sparse NTT computation, the dense NTT, the element wise multiplication and the INTT. These four elements of the computation involve multipliers and adders, as arithmetic units.

The whole architecture is divided in Data Path, Control Unit and Memory. To save complexity, part of the Data Path and part of the Memory are shared among the four mentioned operations.

The NTT and INTT coefficients are in the form $\alpha^{ik} \mod P$; they are pre-computed and stored in two dedicated ROMs, referred to as `Direct Constant Memory` and `Inverse Constant Memory`.

The intermediate values, involved in the computation of the product, are stored in only two dedicated RAMs, named `Complete RAM 1` and 2, that serve the two NTT, the INTT and the element wise product. The sparse NTT needs two additional memories to store the expanded positions: these are the `Starting ROM` and the `Sparse RAM`.

The core element in the Data Path is the butterfly unit of Figure 4, which includes one multiplier, one adder and a subtracter. The parallelism of the Data Path is $n_u = 8$; thus, 8 computations are handled at the same time. The basic elements of the Data Path are showed in Figure 7, and will be detailed in the following sub-sections.

The Control Unit (CU) is quite complex due to the presence of shared arithmetic units and memories. A hierarchical organization has been adopted in order to simplify the control (Figure 8).

The `Master` control unit handles three separate controllers, which manage the three main operations of the multiplier, the `Sparse NTT`, the `Dense NTT` and the `INTT` (plus the element wise product). Two additional control

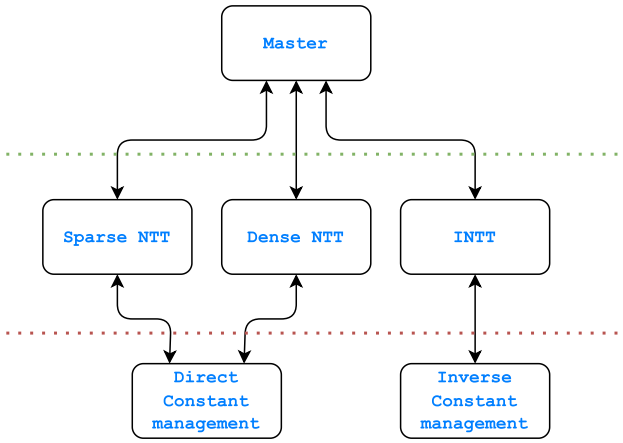


FIGURE 8. Control Unit hierarchical organization.

Starting ROM

NTT(00000001)
NTT(00000010)
NTT(00000100)
NTT(00001000)
NTT(00010000)
NTT(00100000)
NTT(01000000)
NTT(10000000)

FIGURE 9. The content of `Starting ROM`.

units regulate the access of the `MPY` and `ADD/SUB` units to the RAM memories, where the input and output values are stored and to the ROM memories containing the direct and inverse twiddle factors.

A. SPARSE-NTT

The Sparse-NTT controller has the most complex behaviour when compared to the conventional NTT and INTT. The computation is split in two phases. Initially, as described in Algorithm 2, the asserted inputs are individually processed and stored; then, after the merging, data are processed as dense vectors.

According to the architecture in Figure 7, the sparse execution works as follows: the asserted position in the sparse vector, $\mathbf{Pos}_x(i_v)$, is used to address the `Starting ROM` memory, which contains the pre-computed transform values for stage $l_p = \log_2(n_u) = 3$. Let us assume that a single input is asserted within each block: in this case, a single '1' is received, located at any position from 0 to $n_u - 1$. For each input, the resulting values up to stage l_p are stored in the memory with n_u rows, corresponding to the location of the input '1'. As an example, Figure 9 shows the content of the `Starting ROM` for the case of asserted $v_0(3)$ (Figure 5).

The selected row is stored in `Sparse RAM`, where the asserted position is expanded to stage $l = l_{lim}$. The value of $\mathbf{Pos}_x(i_v)$ is received by the `Sparse NTT` controller, which is in charge of:

- providing the correct value of the addresses to `Sparse RAM`;
- requesting the constants to `Direct/Inverse Constant Memory` through the `Direct/Inverse Constant Management`;
- executing the `MPY` then `ADD/SUB`;
- updating and expanding the content of the memory with the results of `MPY` and `ADD/SUB`.
  The `MPY` and `ADD/SUB` receive the inputs from `Sparse RAM` and at each evaluation the control unit updates and expands the content of the memory.

The `Sparse RAM` contains *nze* rows with the asserted positions expanded up to stage l_{lim}; the expansion has length $2^{l_{lim}}$. The next step merges the values from `Sparse RAM` to `Complete RAM 1`. The memory contains the transform of the sparse input vector x, from Equation (4) up to stage l_{lim} expressed as a vector. This memory content is then used by the Dense-NTT controller, which starts at stage $l_{lim} + 1$. The vector length is N': it is initially reset to 0 and then each expanded $\mathbf{Pos}_x$ is assigned to an interval of the complete N' vector; when two positions point to the same interval, they are summed together.

B. DENSE-NTT

The Dense-NTT controller evaluates the transform of the sparse vector from stage l_{lim} and the complete transform of the dense vector. As shown in Figure 7, the complete transform of the dense vector has `Complete RAM 1` as input and output of the computation. The Dense-NTT takes in input the vector of length N' and evaluates its transform at stage $l_{lim} + 1$, the vector is updated "n_u = 8 elements at time", until all N' elements of the transform are computed.

The `Dense-NTT` controller provides the addresses for memory `Complete RAM 1` that contains the inputs of the n_u `MPY`; the twiddle factors are provided by the `Direct Constant Management` given the stage and the n_u section of the complete N' that is processed. The updated values are obtained after the `ADD/SUB` is evaluated and stored in `Complete RAM 1`.

The transform of the dense vector is computed from stage $l = 0$ up to the end via the Dense-NTT, and the result is stored in `Complete RAM 2`. The processing is the same as the one described for stages $l > l_{lim}$ in sparse vector.

The transform of the sparse vector is in `Complete RAM 1` and the transform of the dense vector is in `Complete RAM 2`.

C. ELEMENT WISE MULTIPLICATION

The element wise multiplication evaluates the product between the transforms stored in `Complete RAM 1` and

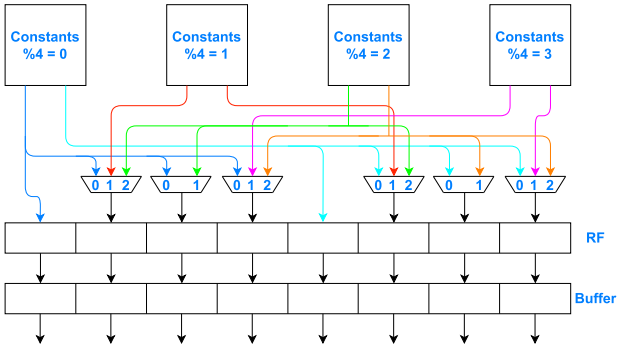


FIGURE 10. Direct/Inverse Constant Management.

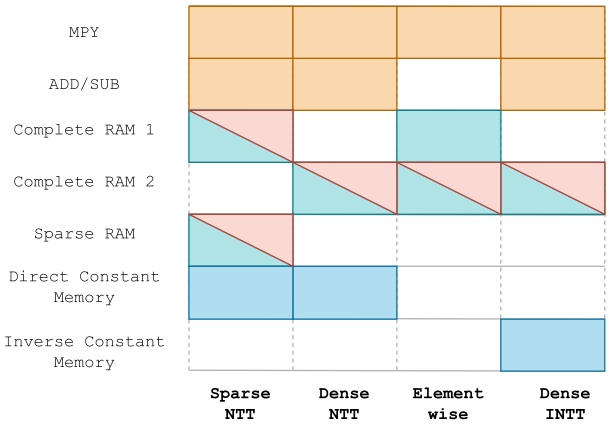


FIGURE 11. Shared resources usage during the intermediate evaluations of the NTT-based multiplier.

in `Complete RAM 2`. The computation involves the `MPY` units, with one input from `Complete RAM 1` and the second input from `Complete RAM 2` (see Figure 7). Since the data stored in `Complete RAM 2` are not needed in the following steps, they are overwritten with element wise multiplication results.

D. DENSE-INTT

This part of the system drives the inverse transform of the values stored in `Complete RAM 2`. The inverse transform works in a similar way as the Dense-NTT, under control of the `Dense INTT` controller. The inputs of the `MPY` are read from `Complete RAM 2` and the twiddle factors are provided by the `Inverse Constant Management` that is in charge of accessing `Inverse Constant Memory`.

E. TWIDDLE FACTOR MANAGEMENT

The twiddle factors are pre-computed and stored in dedicated memories. However, in the sparse processing, they are accessed according to a different procedure than in the dense case. The input twiddle factors for the `MPY` (the constants) in the Sparse and Dense NTT and INTT are provided by the the `Direct Constant management` for the NTT and the `Inverse Constant management` for the INTT, respectively. These units are connected to the memories, `Inverse Constants Memory` which store the twiddle factors.

As shown in Figure 10, the memories are organized in 4 blocks, referred to as `Constants` $\%4 = i$ (with $i = 0, 1, 2, 3$). In order to reduce the latency, n_u constants are read in parallel from the memories and forwarded to the n_u multipliers `MPY`. The `Direct Constant Management` controller is in charge of driving the multiplexers and both the `Register File (RF)` and `Buffer` units in order to effectively provide constants for both the Sparse and Dense-NTT; the execution is performed such that, while the constants loaded in `Buffer` feed the `MPY`, the `Direct Constant Management` controller loads the next set of constants in `RF`.

F. USE OF MAIN HARDWARE RESOURCES

The units in the Data Path and memory elements have been widely reused during the NTT-based multiplication between sparse and dense vectors.

The evolution of the resource use is shown in Figure 11 that shows which unit is active in each stage of the computation. The usage of the arithmetic units is highlighted in orange. As for the RAM components, the use is in light-blue when only read operations are involved (`Direct Constant Memory` and `Inverse Constant Memory`), and in light-blue/red color when the same memory is used for both read and write operations.

VI. IMPLEMENTATION RESULTS

The NTT-based multiplier has been implemented as an ASIC component, using Synopsys Design Compiler and the UMC 65 nm technology; the same architecture has also been implemented for an Artix-7 200 FPGA target, using Vivado.

A. ASIC RESULTS

The ASIC synthesis results are reported in Table 2 and include the occupied area A, when the clock period is set to $t_{ck} = 5$ ns (column 3), the shortest achievable clock period $t_{ck,min}$ before obtaining a negative slack (column 4), and the occupied area A_f when the shortest clock period is set as a constraint (column 5). The critical path, with the longest combinatorial delay, is found within the element wise multiplier and it depends on the choice of the modulus P, since P affects the size of the arithmetic units.

The Table also provides the percentage area and delay differences with respect to the synthesis case with clock period set to 5 ns: for a large N', the feasible increment of the clock frequency is higher than the corresponding area penalty.

It is worth mentioning that the area occupation is referred to a flexible kind of polynomial multiplier that performs the

TABLE 2. ASIC synthesis results for the UMC 65 nm technology. $d_v = 10$. Percentage delay and area values are given with respect to the reference synthesis with constraint $t_{cp} = 5$ ns.

N'	N	A (μm^2)	$t_{ck,min}$ (ns)	A_f (μm^2)
32,768	12,323	94, 315	3.3 (-34%)	128, 497 (+36%)
65,536	21,701 24,659	95, 543	3.5 (-30%)	113, 344 (+18%)
131,072	37,813 40,973 58,171	124, 639	3.7 (-24%)	139, 719 (+7%)

TABLE 3. FPGA resource utilization with Artix-7 200. The operating frequency is set to 100 *MHz* and $d_v = 10$.

N'	N	LUT	FF	BRAM	DSP	Latency (ms)
32,768	12,323	5396	3774	52	40	2
65,536	21,701 24,659	6259	4053	100	40	4.3
131,072	37,813 40,973 58,171	7062	4358	199	40	9.1

product of binary vectors and computes the result either in $\mathbb{GF}(2)$ or in the integer field.

B. FPGA RESULTS

The FPGA synthesis results are reported in Table 3 for several choices of the input vector length (N), while the density of the vector is set $d_v = 10$. The critical path has been set to 10 *ns* in order to enable a fair comparison against published implementations, where the same constraint was imposed. It has been verified that the effect of the density on the number of required resources is negligible. In the Table, N' is the length of the input vector extended with *PrimePadding*, as defined in Section IV-C1. Columns 2-6 provide the required number of hardware resources, namely Look-Up Tables (LUT), Flip-Flops (FF), Block RAMs (BRAM), and arithmetic processing units (DSP). It is worth noting that the number of BRAMs increases almost linearly with N', whereas the amount of allocated logic elements (LUTs and FFs) is sub-linear with N', leading to hardware utilization improvement.

C. LATENCY

The multiplier latency is reported in Table 4 in terms of number of clock cycles, for a few choices of vector length and sparsity. The last two columns also show the latency value in ms, assuming for each case the corresponding highest clock frequency in the ASIC and FPGA synthesis. From Table 4, it can be noticed that the latency scales linearly with N', while it scales logarithmically with the matrix density.

VII. COMPARISON

The latency values provided in Table 4 include three components deriving from the three fundamental processing

TABLE 4. The execution time with different length N' and density (d_v). The results are provided in terms of both number of clock cycles and *ms*, assuming the largest achievable clock frequency.

N'	d_v	Clock cycles	Time (ms)	
			ASIC	FPGA
32, 768	10	207k	0.83	2
	100	226k	0.90	2.2
	1000	244k	0.97	2.4
	$\approx N'$	261k	1	2.6
65, 536	10	437k	1.7	4.4
	100	474k	1.9	4.7
	1000	512k	2	5.1
	$\approx N'$	565k	5.6	
131, 072	10	919k	3.4	9.2
	100	996k	3.7	1
	1000	1M	3.7	1
	$\approx N'$	1.2M	4.4	1.2

TABLE 5. QC-MDPC encoding time in terms of number of clock cycles and corresponding amount of allocated LUTs.

N	Total number of clock cycles / allocated LUTs			
	Proposed	[19]	[17]	[18]
12, 323	261k / 5396	1k / 65k	3.5M / 1327	1.1M / 4700
24, 659	565k / 6259	3k / 65k	7M / 1327	9.5M / 4400
37, 813	1.2M / 7062	5.6k / 65k	11M / 1327	44M / 2800

parts: i) the transform of the two input operands to the frequency domain, ii) the element-wise multiplication, and iii) the final inverse-transform. However, in LEDAcrypt and BIKE decoders for PQC, the result of a cyclic product frequently becomes the input of another cyclic product. In such cases, it is convenient to keep the results in the frequency domain and avoid unnecessary transforms and inverse-transforms. Therefore, a comparison among complete systems that incorporate the polynomial product as one of the key operations is more meaningful than the comparison between standalone multiplier components.

In this paper, we consider the use of the NTT-based multiplier in the context of a PQC application, encompassing both encryption and decryption. We assume the adoption of the QC-MDPC McEliece crytopsystem with the parameters given in Table 1.

The encoding part requires one cyclic multiplication with two dense vectors, as in (2), and the sum with the error vector. For these operations, Table 5 reports both the latency (expressed as the total number of required clock cycles) and the complexity (limited for simplicity to the number of allocated LUTs) when using the proposed NTT-based architecture and three alternative multipliers from the recent literature.

For the implementation in [17], the reported numbers of cycles are derived from the available formulas. The number of LUTs in [18] is referred to the whole system for the parallelism set to $n_u = 8$. In [19], the area and execution time are related to a multiplier suitable for binary polynomial multiplication, as necessary in the LEDAcrypt and BIKE encoders. An additional implementation proposed in [16] has

TABLE 6. The clock cycles required in each iteration of the decoder.

It	Step in Alg1	Total Clock Cycles						
		NTT-PQC	[17]	[18]		[16]		
		$n_u = 8$	$n_u = 8$	$n_u = 32$	$n_u = 128$	$n_u = 32$	$n_u = 64$	$n_u = 128$
1^{st} It	$\mathbf{s}^0$ $\mathbf{upc}^0$ $\mathbf{e}^0$	524k	346k	32k	2k	325k	100k	38k
2^{nd} It and following ones	$\mathbf{s}^1$ $\mathbf{upc}^1$ $\mathbf{e}^1$	236k	346k	32k	2k	325k	100k	38k
LUT		5396	1327	4480	57878	9380	16140	30430

not been included in Table 5 as it is referred to an encoder with a sparse multiplication.

The NTT-based products in the decoder can be arranged by exploiting the properties of the convolution, with the purpose of reducing the overall complexity. The decoding Algorithm 1 includes two multiplications in sequence, at steps 1 (Syndrome Evaluation) and 3 (UPC Evaluation). Therefore, in the first iteration, one transform can be saved by computing the syndrome transform S^{It} as:

$$S^{It} = NTT(x^{It}) \odot NTT(h),$$

where h is the first row of matrix $\mathbf{H}$. The evaluation of upc^{It} is then

$$upc^{It} = INTT(S^{It} \odot NTT(h^T)).$$

where h^T is the first row of matrix $\mathbf{H}^T$. Thus, one inverse transform can be saved in the first iteration.

The upc^{It} vector is processed in the time domain, since the error position over the threshold has to be derived. The threshold is computed from the Syndrome sum, in the frequency domain, which corresponds to the value of $S^{It}(0)$, associated to $\alpha^{i \cdot 0}$ from Equation (7).

The error vector, e^{It}, in the time domain is transformed in the frequency domain to E^{It} with the Sparse-NTT Algoritm.

The syndrome vector is then updated by exploiting the linearity of the transform, the updated syndrome is:

$$S^{It+1} = S^{It} + E^{It} \bmod P \tag{12}$$

Then, as in the decoder Algorithm 1, after the check on the syndrome weight a new iteration of the procedure starts.

The execution time of the first iteration is strongly reduced if some NTT/INTT unnecessary transforms are skipped. Moreover, the execution time for the remaining iterations is further reduced because the transforms of h^T and h, have been already computed during the first iteration and one Sparse-NTT transform is enough.

The resulting decoding execution time, with mixed frequency and time domain computation, is reported in the third column of Table 6, in terms of number of cycles. The remaining columns give the same information for three previous implementations, while the last row shows the required number of LUTs.

TABLE 7. LC figure of merit for combined encoder and decoder: comparison between the proposed solution and state-of-the-art solutions.

N	LC			
	NTT-PQC	[17]†	[18]†	[16]*
12,323	112	786	324	92
24,659	282	1044	1605	N/A
37,813	677	7352	2984	N/A

*Sparse-dense multiplication in the encoder.
† The execution times of the encoder and decoder have been scaled to the nearest N, while the number of LUTs is kept constant for similar values of N.

In Table 6, the total numbers of clock cycles reported in [17] and [18] have been scaled to have $N = 12,323$ and $d_v = 100$, using the formulas provided in the two works. When comparing the different decoder implementations, one can see that the proposed solution achieves a similar latency as [16] and [17] ($n_u = 32$), while it is much slower than [18]. In terms of LUTs, the proposed decoder is better than [16], very similar to [18] and more expensive than [17].

We now consider the implementation of the cyclic multiplier in the context of a complete system, able to perform both encoding and decoding, and supporting both binary and integer formats. To compare the alternative implementations, we introduce the *LC* figure of merit, defined as a latency-complexity product, where the complexity C is given by the global number of LUTs allocated in the synthesis of both encoder and decoder, while the latency L is the execution time (in s) evaluated for encoder and decoder, assuming for the case $N = 12,323$, $d_v = 100$, a clock frequency of 100 MHz, and $It_{max} = 6$ (when possible the numbers are scaled to match these requirements).

Although the proposed NTT-based multiplier is not the best option when comparing standalone arithmetic units, the results in Table 7 show that it is more efficient than the other implementations proposed in the literature, when the unit is used in a complete application, where the same component must be exploited for multiple products in the encoding and decoding stages, which are characterized by different data formats and computational structures. As seen from Table 7, the advantage in terms of *LC* product over the alternative approaches grows quickly with the size of the problem: when

N moves from 12,323 to 37,813, LC is 5 times the initial one. Considering the same N our proposal is almost 7 times more efficient when compared to [17] and 3 times more efficient than [18]. Moreover, the increase of N takes advantage of the Schonhage Stressen algorithm and makes out proposal even more efficient. The reported LC figure for [16] is better than all the other ones in Table 7; however, this comparison is not fair, because the encoding in [16] involves a product between a sparse vector and a dense vector, which drastically simplifies the computation with respect to the considered applications.

VIII. CONCLUSION

The NTT-based multiplier offers a series of advantages when applied to code-based cryptosystems. One relevant advantage is the logarithmic increase of the latency with the increase of the density of the variables. This property is important for the implementation of a McEliece Cryptosystem, where both sparse and dense cyclic products are required. This advantage is more relevant if the complete execution time is considered for both encoder and decoder, which involve operands with different density values.

Moreover, in the decoding procedure, the computation can be arranged to skip or reduce several operations, leading to a shorted execution time, in the computation of the multiplication.

A second advantage is in its adaptability of the NTT-based approach to different data types. This is particularly important for the implementation of code-based cryptosystems, where variables with different densities and coefficients are used. The flexibility of the proposed solution is not limited to LEDAcrypt and BIKE cryptosystems, but it can be extended to other primitives in the PQC domain, where the polynomial product plays a key role is several primitives.

The present work proves that a NTT-based multiplier, with proper design choices, is efficient for code-based Post-Quantum Cryptography, making the proposed unit a valid accelerator for different applications.

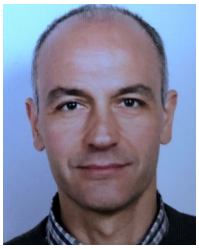
REFERENCES

[1] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, Sep. 2017, doi: 10.1038/nature23461.
[2] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.
[3] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory Comput. (STOC)*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 212–219, doi: 10.1145/237814.237866.
[4] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: Quantum resource estimates," 2015, *arXiv:1512.04965*.
[5] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *Deep Space Netw. Prog. Rep.*, vol. 44, pp. 114–116, Jan. 1978.
[6] M. Baldi, F. Chiaraluce, and R. Garello, "On the usage of quasi-cyclic low-density parity-check codes in the McEliece cryptosystem," in *Proc. 1st Int. Conf. Commun. Electron.*, Oct. 2006, pp. 305–310, doi: 10.1109/CCE.2006.350824.
[7] R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. L. M. Barreto, "MDPC-McEliece: New McEliece variants from moderate density parity-check codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2013, pp. 2069–2073, doi: 10.1109/ISIT.2013.6620590.
[8] National Institute of Standards and Technology. *Post-Quantum Cryptography, Round 3 Submissions*. Accessed: Nov. 1, 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions
[9] *Classic McEliece*. Accessed: Nov. 1, 2022. [Online]. Available: https://classic.mceliece.org/nist.html
[10] *BIKE Suite*. Accessed: Nov. 1, 2022. [Online]. Available: https://bikesuite.org/
[11] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J. Bos, J. Deneuville, A. Dion, P. Gaborit, J. Lacan, E. Persichetti, J. Robert, P. Véron, and G. Zemor. (2020). *Hamming Quasi-Cyclic (HQC)—Third Round Version—Updated Version 10/01/2020*. [Online]. Available: http://pqc-hqc.org/doc/hqc-specification%5C_2020-10-01.pdf
[12] Y. Xing and S. Li, "A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 2, pp. 328–356, Feb. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8797, doi: 10.46586/tches.v2021.i2.328-356.
[13] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. *Crystals-Kyber*. Accessed: Nov. 1, 2022. [Online]. Available: https://pq-crystals.org/kyber/resources.shtml
[14] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.
[15] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computer*, vol. 7, no. 3, pp. 281–292, Sep. 1971, doi: 10.1007/BF02242355.
[16] J. Richter-Brockmann, J. Mono, and T. Güneysu, "Folding BIKE: Scalable hardware implementation for reconfigurable devices," *IEEE Trans. Comput.*, vol. 71, no. 5, pp. 1204–1215, May 2022, doi: 10.1109/TC.2021.3078294.
[17] K. Koleci, P. Santini, M. Baldi, F. Chiaraluce, M. Martina, and G. Masera, "Efficient hardware implementation of the LEDAcrypt decoder," *IEEE Access*, vol. 9, pp. 66223–66240, 2021, doi: 10.1109/ACCESS.2021.3076245.
[18] D. Zoni, A. Galimberti, and W. Fornaciari, "Efficient and scalable FPGA-oriented design of QC-LDPC bit-flipping decoders for post-quantum cryptography," *IEEE Access*, vol. 8, pp. 163419–163433, 2020, doi: 10.1109/ACCESS.2020.3020262.
[19] D. Zoni, A. Galimberti, and W. Fornaciari, "Flexible and scalable FPGA-oriented design of multipliers for large binary polynomials," *IEEE Access*, vol. 8, pp. 75809–75821, 2020.
[20] K. Millar, M. Lukowiak, and S. Radziszowski, "Design of a flexible Schönhage-strassen FFT polynomial multiplier with high-level synthesis to accelerate HE in the cloud," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2019, pp. 1–5.
[21] E. Berlekamp, R. J. McEliece, and H. C. A. Van Tilborg, "On the inherent intractability of certain coding problems (corresp.)," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 3, pp. 384–386, May 1978, doi: 10.1109/TIT.1978.1055873.
[22] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.
[23] 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*, T. Lange and R. Steinwandt, Eds. Cham, Switzerland: Springer, 2018, pp. 3–24.
[24] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini. *Ledacrypt Home*. Accessed: Nov. 1, 2022. [Online]. Available: https://www.ledacrypt.org/
[25] V. Strassen and A. Schönhage, "Multiplication of many-digital numbers by automatic computers," *Computing*, vol. 7, nos. 3–4, pp. 281–292, 1971.
[26] X. Feng, S. Li, and S. Xu, "RLWE-oriented high-speed polynomial multiplier utilizing multi-lane Stockham NTT algorithm," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 3, pp. 556–559, Mar. 2020.
[27] T. Nagell, *Introduction to Number Theory*. Providence, RI, USA: American Mathematical Society, 2001.
[28] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York, NY, USA: Oxford Univ. Press, 1999.
[29] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[30] L. Bluestein, "A linear filtering approach to the computation of discrete Fourier transform," *IEEE Trans. Audio Electroacoust.*, vol. AU-18, no. 4, pp. 451–455, Dec. 1970, doi: 10.1109/TAU.1970.1162132.

KRISTJANE KOLECI (Graduate Student Member, IEEE) received the M.S. degree in electronic engineering from the Politecnico di Torino, Italy, in 2019, where she is currently pursuing the Ph.D. degree with the VLSI-Laboratory Group. Her research interests include the design hardware architectures of algorithms for post-quantum cryptography and error correcting codes. She has been the Vice-Chair of the IEEE Women in Engineering Student Branch Affinity Group, Politecnico di Torino.

PAOLO MAZZETTI received the M.S. degree in electronic engineering from the Politecnico di Torino, Italy, in 2021. He has worked during his thesis project on the design of a vector by circulant matrix product, as part of a code-based post-quantum cryptography algorithm.

MAURIZIO MARTINA (Senior Member, IEEE) received the M.S. and Ph.D. degrees in electronic engineering from the Politecnico di Torino, Italy, in 2000 and 2004, respectively. He is currently a Full Professor with the VLSI-Laboratory Group, Politecnico di Torino. His research interests include VLSI design and implementation of architectures for digital signal processing, video coding, communications, artificial intelligence, machine learning, and event-based processing. He has more than 100 scientific publications. He is also an Associate Editor of IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS. He is the Counselor of the IEEE Student Branch at the Politecnico di Torino and a Professional Member of IEEE HKN.



GUIDO MASERA (Senior Member, IEEE) received the Dr.-Ing. (summa cum laude) and Ph.D. degrees in electronic engineering from the Politecnico di Torino, Italy. He has been a Professor with the Electronic Department, Politecnico di Torino, since 1992. His research interests include several aspects in the design of digital integrated circuits and systems, with a special emphasis on high-performance architectures for communications, forward error correction, image and video coding, cryptography, and hardware accelerators for machine learning. He has more than 200 publications and two patents, and was a designer of several ASIC components. He is an Associate Editor of *Electronics* (MDPI) and a Former Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS, and the *IET Circuits, Devices & Systems*.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement