HLS-Based HW/SW Co-Design of the Post-Quantum Classic McEliece Cryptosystem

Vatistas Kostalabros
Barcelona Supercomputing Center
Bacelona, Spain
vaistas.kostalabros@bsc.es

Jordi Ribes-González
Universitat Rovira i Virgili
Tarragona, Spain
jordi.ribes@urv.cat

Oriol Farràs
Universitat Rovira i Virgili
Tarragona, Spain
oriol.farras@urv.cat

Miquel Moretó
Barcelona Supercomputing Center
Barcelona, Spain
miquel.moreto@bsc.es

Carles Hernandez
Universitat Politècnica de València
Valencia, Spain
carherlu@upv.es

***Abstract*—While quantum computers are rapidly becoming more powerful, the current cryptographic infrastructure is imminently threatened. In a preventive manner, the U.S. National Institute of Standards and Technology (NIST) has initiated a process to evaluate quantum-resistant cryptosystems, to form the first post-quantum (PQ) cryptographic standard. Classic McEliece (CM) is one of the most prominent cryptosystems considered for standardization in NIST's PQ cryptography contest. However, its computational cost poses notable challenges to a big fraction of existing computing devices. This work presents an HLS-based, HW/SW co-design acceleration of the CM Key Encapsulation Mechanism (CM KEM). We demonstrate significant maximum speedups of up to 55.2×, 3.3×, and 8.7× in the CM KEM algorithms of key generation, encapsulation, and decapsulation respectively, comparing to a SW-only scalar implementation.**

I. Introduction

The advent of large-scale quantum computers may have a positive impact on many computational disciplines. However, the most popular public-key algorithms that we use today could be efficiently broken by a sufficiently strong quantum computer. This could have a disruptive effect on our society. While ongoing advances in quantum computing bring large-scale quantum computers closer to reality, the need to come up with quantum-resistant replacements for traditional cryptography is imperative.

Post-Quantum Cryptography (PQC) is a branch of cryptography that aims to develop PQ cryptographic solutions that will be able to execute on today's non-quantum computers. These PQ cryptosystems should be resistant to both conventional and quantum attacks. In this direction, public institutions are making a coordinated effort to standardize the use of PQ cryptosystems.

Intending to define the first PQC standard, NIST launched in 2017 an open contest to evaluate PQ cryptosystems [1]. Its 3rd evaluation round currently features 4 finalist PQ candidates that advanced from the 69 initial submissions [2]. Our work focuses on Classic McEliece [3], one of the most prominent finalists to form the core of the first PQC standard.

CM is a code-based cryptosystem. Its security is based on the hardness of decoding a hidden error-correcting code. While its large public key (PK) matrix complicates its hardware implementation, the fact that since its discovery at 1978 it remains unbroken, renders its security properties particularly appealing [3]. Nevertheless, it lacks performance evaluation studies, especially on heterogeneous platforms.

Our work bridges this gap by proposing an acceleration of CM on heterogeneous platforms containing CPUs and FPGAs. To exploit the advantages of the heterogeneity of the target architecture, we utilize a HW/SW co-design methodology. To gain the speed efficiency of hardware designs with reduced design effort, we use HLS-based techniques on the FPGA. Finally, HLS-based HW/SW co-design acceleration is the ideal methodology in terms of flexibility and performance gain to address any possible changes of CM prior to its standardization.

This paper makes the following contributions:

- The first, to the best of our knowledge, HLS-based HW/SW co-design acceleration of CM on CPU+FPGA heterogeneous platforms.
- An effective HW/SW co-design approach that addresses the large PK size of CM, providing speedups of up to 55.2× to the most time-consuming part of CM.
- A design space exploration that delivers a balanced solution to the performance-security tradeoff of CM implementation.
- An analysis of the CM acceleration potential on heterogeneous platforms including scalar or vector-processing cores.
- A highly-portable and OpenCL-based HW/SW co-design, across embedded and data-centric heterogeneous acceleration platforms containing FPGA devices.
- Open-source access to the developed source code at https://github.com/beatsnbytes/classic_mceliece.

This paper has the following structure: Section II introduces the CM cryptosystem and presents the motivation for this work. Section III analyzes our HW/SW co-design proposal. Next, Section IV explains our experimental methodology.

Section V demonstrates the performance evaluation of our proposal. Finally, Section VI presents the related work and Section VII concludes the current study.

II. Background and Motivation

Key Encapsulation Mechanisms. Candidates of the NIST PQC contest span two categories; KEMs and Digital Signature algorithms. In this work, we focus on the CM KEM. KEMs enable a server and a client to privately establish a common session key. They consist of three algorithms: Key Generation (KeyGen), Encapsulation, and Decapsulation. At first, a server executes KeyGen to generate a public key (PK) and secret key (SK) pair, and distributes the PK safely to the client. The client then uses this PK on Encapsulation to generate a session key in plaintext and ciphertext (CT) forms. The client then sends this CT to the server. Finally, the server uses Decapsulation with its own SK to decrypt the CT and obtain the session key.

Classic McEliece. CM is among the four NIST contest finalist KEMs. It is based on the public-key encryption schemes of McEliece and Niederreiter, with some custom algorithmic optimizations [4], [5]. The main advantages of CM are: i) its long-standing security, having resisted classical and quantum attacks without significant modifications for over 40 years [3], ii) its encryption/decryption speed, and iii) its unusually small CTs, which are useful in some applications [6]. A well-known drawback is its large PK size, which poses significant challenges to its hardware implementation in a big fraction of existing computing devices and may hinder its suitability for some network protocols [7], [8]. The parameters of CM are:

- Three positive integers m, n, t with $n \leq 2^m$ and $m \cdot t < n$. These also define two integers $k = n - m \cdot t$ and $q = 2^m$.
- Two monic irreducible polynomials f, F that define the finite fields $\mathbb{F}_q = \mathbb{F}_2[z]/(f(z))$ and $\mathbb{F}_{q^t} = \mathbb{F}_q[y]/(F(y))$.

The CM cryptosystem provides five parameter sets, belonging to security levels 1, 3 and 5. Parameter sets alter the inner configuration of the cryptosystem according to the desired security strength the user wants to achieve. They ultimately dictate the PK, SK, and CT sizes, thus increasing/decreasing the provided security. We reproduce the most significant parameters, as well as their respective values, of CM in Table I.

CM uses the hash algorithm H = SHAKE256. In the next section we will describe the computational steps of the three CM KEM algorithms.

TABLE I
CM parameter sets and sizes of Public Key (PK), Secret Key (SK), and Ciphertext (CT) [8].

Security level	Parameter set	m	n	t	PK size [KB]	SK size [KB]	CT size [Bytes]
L1	`mceliece348864`	12	3488	64	255	6.3	128
L3	`mceliece460896`	13	4608	96	511.8	13.2	188
$L5_1$	`mceliece6688128`	13	6688	128	1020.5	13.56	240
$L5_2$	`mceliece6960119`	13	6960	119	1022.2	13.58	226
$L5_3$	`mceliece8192128`	13	8192	128	1326	13.75	240

KeyGen:

1) Generate a random monic irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t.
2) Choose random distinct $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$.
3) Compute the $t \times n$ matrix $\left(\alpha_j^{i-1}/g(\alpha_j)\right)_{i,j}$.
4) Extend this matrix to an $mt \times n$ matrix by writing elements as column m-bit vectors.
5) If possible, systematize this matrix, obtaining (I_{n-k}, T). If this fails, go to step 1.
6) Randomly generate $s \in \{0,1\}^n$.
7) Output $(s, g, \alpha_1, \ldots, \alpha_n)$ as the secret key, and T as the public key.

Encapsulation(T):

1) Let $e \in \mathbb{F}_2^n$ a random vector with exactly t ones.
2) Compute $C_0 = (I_{n-k}, T) \cdot e \in \mathbb{F}_2^{n-k}$.
3) Compute $C_1 = \mathsf{H}(2, e)$.
4) Compute $K = \mathsf{H}(1, e, (C_0, C_1))$.
5) Output the session key K and its ciphertext (C_0, C_1).

Decapsulation($s, g, \alpha_1, \ldots, \alpha_n, (C_0, C_1)$):

1) Let $v \in \mathbb{F}_2^n$ be C_0 padded with k zeros.
2) Compute the $2t \times n$ matrix $\left(\alpha_j^i/g^2(\alpha_j)\right)_{i,j}$.
3) Extend this matrix to a $2mt \times n$ matrix $H^{(2)}$ by writing elements as column m-bit vectors.
4) Compute the syndrome $H^{(2)} \cdot v$.
5) Find the error locator polynomial $\sigma(x)$ of the syndrome, using Berlekamp-Massey decoding.
6) Let $c = (c_i)_i \in \mathbb{F}_2^n$ with $c_i = 1$ if and only if $\sigma(\alpha_i) = 0$, and $c_i = 0$ otherwise.
7) Let $e' = v + c$. If $C_0 = (I_{n-k}, T) \cdot e'$ and $C_1 = \mathsf{H}(2, e')$ and e' has exactly t ones, then output $K = \mathsf{H}(1, e', (C_0, C_1))$. Otherwise, output $K = \mathsf{H}(0, s, (C_0, C_1))$.

Our HW/SW acceleration focuses on the most time-consuming parts of the three KEM algorithms: step 5 of KeyGen, step 2 of Encapsulation, and steps 2, 3, 4 of Decapsulation.

Motivation. CM KEM is one of the most prominent cryptosystems to form the core of the first PQC standard. However, as the standardization process advances, CM lacks an extensive performance evaluation on heterogeneous platforms. Therefore, we assist the hardware benchmarking process of CM by providing an efficient HW/SW co-design acceleration, based on heterogeneous CPU+FPGA platforms. The finalist KEM's may undergo significant changes to conform to stricter security, and to correct any possible flaws prior to their standardization process. We effectively address this need by providing a flexible, reduced design-effort HLS acceleration that is also portable across CPU+FPGA heterogeneous platforms. Finally, broadening the scope of our work, we evaluate the performance of our proposal in heterogeneous platforms containing different classes of scalar or vector processors.

III. Hardware/Software Co-Design

Our HW/SW co-design acceleration of CM focuses on heterogeneous systems-on-chip (SoCs) including hard processors

```
1 for (i = 0; i < (PK_NROWS + 7) / 8; i++){
2     for (j = 0; j < 8; j++){
3         row = i*8 + j;
4         if (row >= PK_NROWS) break;
5         // forward elimination
6         for (k = row + 1; k < PK_NROWS; k++){
7             mask = mat[row][i] ^ mat[k][i];
8             mask >>= j;
9             mask &= 1;
10            mask = -mask;
11            for (c = 0; c < SYS_N/8; c++)
12                mat[row][c] ^= mat[k][c] & mask;
13        }
14        if ( ((mat[ row ][ i ] >> j) & 1) == 0 ){
15            // return if not systematic
16            return -1;
17        }
18        // backwards substitution
19        for (k = 0; k < PK_NROWS; k++){
20            if (k != row){
21                mask = mat[ k ][ i ] >> j;
22                mask &= 1;
23                mask = -mask;
24                for (c = 0; c < SYS_N/8; c++)
25                    mat[k][c]^= mat[row][c]&mask;
26            }
27        }
28    }
29 }
```

Listing 1. Gaussian systemizer's main computational loop source code.

and embedded FPGAs. Examples of these type of devices are the Xilinx Zynq multi-processor SoC family and some more recent SoCs like the RISC-V based Arnold chip and the DHALIA radiation hardened SoC [9]–[11]. This section explains the reasons behind our choices on the acceleration techniques, both in the CPU and FPGA part of the target acceleration platform. Performance and resource consumption results are presented in Section V.

Acceleration Methodology Rationale. We accelerate CM by designing three accelerators decoupled from the CPU's pipeline. Each accelerator implements the most computational intensive parts of the KEM algorithms. These parts were identified by a software profiling process whose findings are presented in Section IV. The designed accelerators are: i) a Gaussian systemizer for the acceleration of KeyGen, ii) a syndrome encoder for the acceleration of Encapsulation, and iii) a syndrome decoder module for the acceleration of Decapsulation.

Some works consider unnecessary the acceleration of KeyGen [12]. Their approach is based on the strong assumption that KeyGen is not called frequently in the cryptosystem execution and therefore is implemented on a separate device than the rest of the KEM, or even executed entirely in software. However, when the application scenario mandates the use of frequent PK-SK pair generation [13], the acceleration of KeyGen is performance-critical, since it accounts for 95% of the total execution time consumed by the CM KEM algorithms.

HLS Primer. Here we explain the specific HLS techniques we use to design the accelerators and elaborate on the way they impact their performance. HLS directives impact the hardware generation in two ways: i) they increase performance, and ii) they control resource usage. To increase the FPGA to DRAM communication performance, we wrap every accelerator in an AXI interface. Arrays placed at the FPGA DRAM

```
1 for (i = 0; i < PK_NROWS; i++)
2 {   // row initialization loop
3     for (j = 0; j < SYS_N/8; j++)
4         row[j] = 0;
5     for (j = 0; j < PK_ROW_BYTES; j++)
6         row[SYS_N/8-PK_ROW_BYTES+j] = pk_ptr[j];
7     row[i/8] |= 1 << (i%8);
8     b = 0;
9     // perform the multiplication
10    for (j = 0; j < SYS_N/8; j++)
11        b ^= row[j] & e[j];
12    b ^= b >> 4;
13    b ^= b >> 2;
14    b ^= b >> 1;
15    b &= 1;
16    s[ i/8 ] |= (b << (i%8));
17    pk_ptr += PK_ROW_BYTES;
18 }
```

Listing 2. Syndrome encoder's main computational loop source code.

by the CPU direct memory access (DMA) are loaded/stored via AXI-bursts from/to local BRAMs of the accelerator. AXI-bursts are implicitly inferred by defying HLS primitives at the accelerator load/store loops. Array elements can be partitioned to multiple BRAMs and accessed through separate BRAM read/write ports using specific HLS directives. This way we mitigate memory bottlenecks and achieve parallelism by accessing multiple elements of the same array at the same clock cycle at the expense of increased BRAM usage. Data parallelism can be exploited by unrolling and pipelining computational loops to increase the design's throughput. Finally, to avoid using up specific FPGA resources, we leverage HLS directives to map operations to specific FPGA computational cores (e.g DSP, LUT).

Gaussian Systemizer. The source code of the Gaussian systemizer comprises two main computational loops: forward elimination and backwards substitution (Listing 1). It is executed in step 5 of KeyGen (see Section II). In our accelerator, every new iteration of the outer loop (line 2), caches the matrix row `mat[row]` in separate BRAMs than the rest of the matrix `mat`. This decoupling enables the concurrent and quick access to all matrix elements involved in computations at lines 12, 25. This way we pipeline and partially unroll the forward elimination inner loop (line 11), increasing its throughput at one row element per clock cycle. To take further advantage of the internal parallelization potential of the Gaussian systemizer, we completely unroll and pipeline the backwards substitution loop (lines 19-27). Its two computational loops (lines 19, 24) are merged, thus decreasing the latency. To increase the throughput, we use array partitioning for the `mat` matrix. The extent to which we partition the `mat` array defines the throughput of the backward substitution loop.

Syndrome Encoder. Syndrome encoding is a part of the Encapsulation of CM (step 2 at Section II). We reproduce its source code in Listing 2. Pipelining and unrolling the outermost loop (line 1) seems to be the best solution from a performance perspective, but this is not the case for syndrome encoder for two basic reasons. First, this accelerator performs computations based on the PK of CM. Due to its large size, moving the PK from the FPGA DRAM to the local BRAMs

```
1     c = (r[i/8] >> (i%8)) & 1;
2     e = eval(f, L[i]);
3     e_inv = gf_inv(gf_mul(e,e));
4     for (j = 0; j < 2*SYS_T; j++){
5         out[j] = gf_add(out[j], gf_mul(e_inv, c));
6         e_inv = gf_mul(e_inv, L[i]);
7     }
8 }
```

Listing 3. Syndrome decoder's main computational loop source code.

of the accelerator takes up >80% of its total execution time. Second, completely pipelining and unrolling the loop at line 1 causes a considerable resource consumption increase in the accelerator (≈50% of total BRAMs), with a negligible performance increase (< 1% reduction in execution time), as the total execution time is governed by the PK loading from the DRAM. We manage to provide a balanced solution, regarding performance and resource consumption, by adhering to the following approach. Initially, we focus our attention on the most computationally intensive loops (lines 5, 10), and we increase their performance by unrolling and pipelining them according to the array partition we perform for the `pk_ptr`, `row` and `e` matrices. Next, we split the computational task into multiple independent accelerators of equal size. By executing these accelerators in parallel, we effectively hide the PK transfer time by a factor proportional to the accelerator multiplicity. Each of these smaller accelerators, by not being fully HLS-optimized (internal parallelization, pipelining), consumes less FPGA resources than a fully-optimized monolithic accelerator.

Syndrome Decoder. The syndrome decoder belongs to the Decapsulation of the CM KEM (steps 2, 3, and 4 at Section II). Its main computational loop calls four different Galois Field (GF) arithmetic functions: `gf_mul`, `gf_add`, `gf_inv`, and `eval` (Listing 3). They perform GF multiplication, addition, inversion, and evaluation of a polynomial at a field element, respectively. The addition is implemented as bitwise XOR, while the rest of the functions iterate over the execution of `gf_mul`. Therefore, we focus on the performance optimization of `gf_mul`. In the acceleration of syndrome decoder we balance the tradeoff between hardware overhead and speedup. Specifically, we pipeline only the `gf_mul` execution and the computational intensive loop at line 4. We inline the rest of the GF arithmetic functions, replicating their hardware implementation each time they are called. Finally, we split the main computational loop (line 4) into smaller and equally-sized parts, forming multiple accelerators that execute concurrently on the FPGA. This way, we achieve significant speedup increase and smaller resource consumption in comparison to that of a single, fully HLS-optimized accelerator instance.

Task-Level Parallelism. To further increase the throughput and decrease the latency of our design we analyzed the behavior of the three accelerators under the dataflow HLS optimization. The dataflow optimization converts a series of sequential software tasks in concurrent hardware processes, taking advantage of the task level parallelism. Pipelining at the task level allows functions and loops to execute concurrently. However, we observed that the required modifications to increase the performance of the accelerators (i.e increased amount of data streamed between CPU and accelerators, modification of the accelerators' internal architecture) severely limit the acceleration gains. We conclude that this type of optimization better serves a fully detached accelerator design approach.

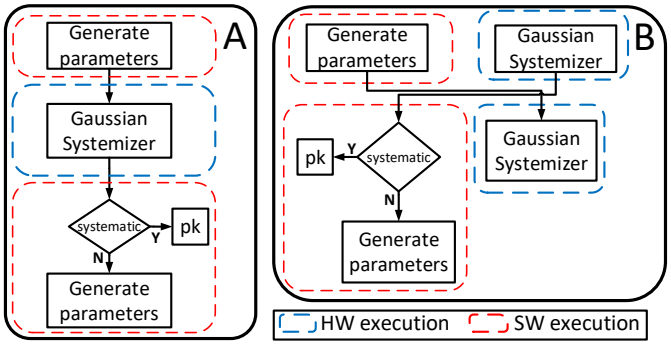


Fig. 1. (A) Sequential and (B) parallel execution of Gaussian systemizer and parameter regeneration in case of systemization failure.

TABLE II
SW profiling results of the CM KEM. **Bold**=functions chosen for acceleration, B=Baseline, V=auto-vectorized, Hash=SHAKE256.

KeyGen			Encapsulate			Decapsulate	
Function	Exec. time [%]		Function	Exec. time [%]		Function	Exec. time [%]
	B	V		B	V		B/V
`pk_gen` **Gaussian systemizer**	**81.8**	**97.6**	**`syndrome`**	**66.1**	**85.4**	**`gf_mul`**	**60.5**
						`synd`	**8.1**
`int32_sort`	9.8	0.4	`randombytes`	21	9.2	**`gf_add`**	**1.9**
						`gf_inv`	**1**
`gf_mul`	3.5	0.1	Hash	6.1	3.2	**`eval`**	**0.9**
Other	4.9	1.9	Other	6.8	2.2	Other	27.6

CPU-Side Optimizations. Besides the performance gain that the FPGA accelerators provide, we get a considerable speedup by employing HW/SW co-design techniques on the CPU of the acceleration platform. Specifically, we focus on the optimization of the software part of the PK generation. Fig. 1 depicts the approach we follow. While systemization is taking place in the Gauss systemizer accelerator, we harness the idle time of the CPU to compute a second set of input parameters to the Gaussian systemizer and load them to the FPGA DRAM. In case the matrix systemization fails, this allows us to swiftly perform a subsequent Gaussian systemization, without waiting to sequentially recompute the input parameters of the accelerator between accelerator calls. This technique provides significant speedup increase to KeyGen.

IV. Experimental Methodology

HW/SW Partitioning. Using the Valgrind profiling tool [14], we profile the execution of the CM software implementation on all security levels, both with and without the use of auto-vectorized code. To include auto-vectorized code in software,

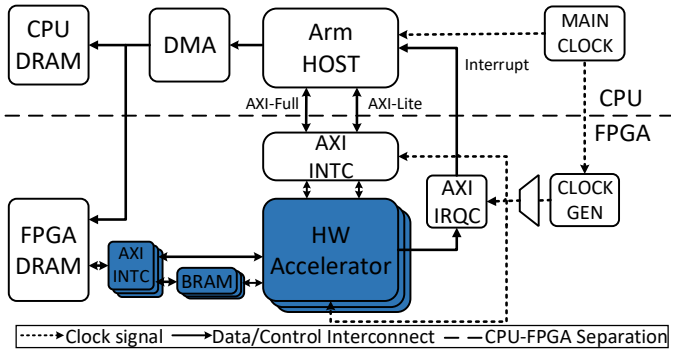


Fig. 2. Base accelerator platform with HW components and their basic interconnection signals. Blue color denotes the reconfigurable, dynamic region of the FPGA device that gets modified with every new accelerator design.

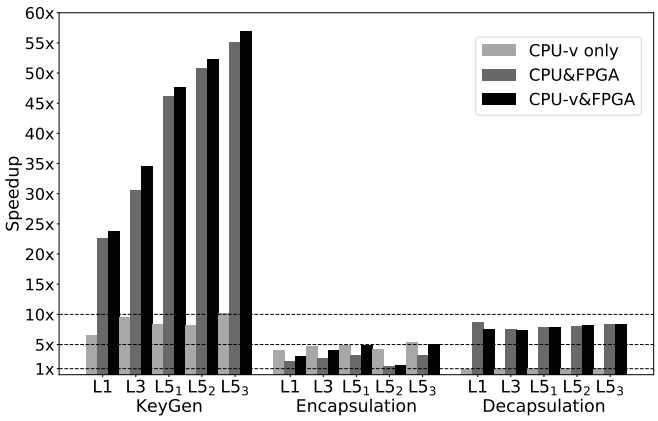


Fig. 3. CM KEM speedup for all security levels. Baseline=CM SW implementation without auto-vectorization (CPU). (CPU-v=SW auto-vectorization, FPGA=HW accelerators)

we use the code auto-vectorization feature of the gcc compiler. Software auto-vectorization optimization is included by default in the `-O3` optimization level of gcc, and it can be turned off with the respective compiler flag `-fno-tree-vectorize`.

In Table II, we report the execution time percentage of the most time-consuming functions of each CM KEM algorithm. The values are averaged over all security levels and 100 executions. We consider as ideal acceleration candidates, the most time consuming functions of each KEM algorithm. Regarding the total execution time of all three KEM algorithms, we report that KeyGen takes 95% of it, while Decapsulation and Encapsulation take 4% and $< 1\%$, respectively.

Experimental Platform. We implement our proposal on the Xilinx zcu102 heterogeneous CPU+FPGA platform. The platform comprises a quad-core Arm Cortex-A53, running at 1.1 GHz, and a Zynq-UltraScale+ FPGA-device [9]. 4GB and 512MB DRAMs are attached to the CPU and FPGA devices, respectively. The software application uses one of the four available cores. Fig. 2 presents a high-level overview of the acceleration platform that supports our HW/SW co-design acceleration approach. The accelerators communicate via AXI-Full/Lite interface with the rest of the FPGA and CPU. DMA between the CPU and the DRAM FPGA is supported. Petalinux, a Xilinx's Linux operating system (OS) distribution, boots on top of the platform and takes care of OS-related tasks such as address translation, DMA communication, FPGA interrupt servicing, etc.

Development Tools and Functional Verification. The efficient implementation of both the software part of CM on the CPU and of the hardware accelerators on the FPGA device is of paramount importance to achieve high performance. Moreover, the interaction between hardware accelerators and CPU is also a crucial factor for performance. To meet these needs, we use the OpenCL programming framework. The native support of OpenCL from the Xilinx runtime lets the hardware designer focus on the efficient design of the hardware accelerators by abstracting away complex tasks such as the data movement through the AXI interface and the DMA data transfer from the CPU to the FPGA DRAM. It also offers high-portability between different heterogeneous platforms, with minimal design modifications.

We test the functional correctness of our design, by using real FPGA runs and the Known Answer Test (KAT) utility provided by the NIST contest.

Constant-Time Evaluation. The software implementation of CM KEM is verified to be constant-time [3], and our software changes do not introduce timing leaks. We leave the constant-time analysis of our HLS implementation for future work.

V. Evaluation

KEM Speedup. We evaluate the impact of our proposal on the execution time of the CM KEM algorithms for all security levels. Fig. 3 presents the maximum achieved speedup for each CM KEM algorithm across the 5 security levels. The baseline is the software implementation of CM on a single Arm core, without the use of auto-vectorized instructions.

Our proposal outperforms the baseline for all KEM algorithms and security levels. KeyGen achieves a significant speedup ranging from 22.6× to 55.2×, taking advantage of the internal parallelization potential of the Gaussian systemizer. Decapsulation executes between 7.5× and 8.7× faster than the baseline, by splitting the syndrome decoding computation between multiple concurrent accelerators. Finally, overcoming the big timing overhead that PK loading puts on the syndrome encoder, we provide Encapsulation with speedups ranging from 1.4× up to 3.3×.

Security-Performance Tradeoff. For high CM security levels, the sizes of PK, SK, and CT increase (see Table I). Table III shows that our proposal does not have to sacrifice performance to provide increased security. Contrarily, we efficiently provide bigger speedup as the security levels increase.

KeyGen increases its speedup from 22.6× to 55.2× (144%) from L1 to L5$_3$. This happens because the Gaussian systemizer exploits the bigger internal parallelization that the PK matrix offers. Encapsulation raises its speedup from 2.2× to 3.3× (50%) from L1 to L5$_3$. The 1.4× speedup value of L5$_2$ is an outlier in terms of performance. The size of the accelerator matrices at L5$_2$ does not allow a balanced split of the syndrome

TABLE III
Maximum achieved speedup & FPGA resource consumption

Accelerator	Security level	Resource Consumption [%]				Speedup [x]
		LUT	BRAM	DSP	FF	
Gaussian systemizer	L1	6.5	20.4	0	3.7	22.6
	L3	8.3	41.4	0	5.4	30.6
	$L5_1$	14.4	63.3	0	8.2	46.1
	$L5_2$	8.8	55.8	0	5.5	50.8
	$L5_3$	9.8	64.8	0	7.4	55.2
Syndrome encoder	L1	15.6	24	0.4	6.2	2.2
	L3	18.2	52.4	1	6.4	2.8
	$L5_1$	14.1	54.4	1	5	3.2
	$L5_2$	6	44	1.2	1.9	1.4
	$L5_3$	15.3	76	1.2	7	3.3
Syndrome decoder	L1	18.6	2.7	67.8	7.2	8.7
	L3	40.6	43.4	28.1	33.6	7.5
	$L5_1$	49.6	68.4	25.8	36.5	7.8
	$L5_2$	53.2	41.2	35.1	33	8
	$L5_3$	55	64.4	37.5	47.3	8.3

encoder to more than one computational unit, thus providing a smaller speedup. Finally, the Decapsulation algorithm shows a speedup from 7.5× to 8.3× (16%) from L2 to $L5_3$. Interestingly, L1 achieves an 8.7× speedup, the highest speedup out of all security levels. This happens due to the small matrix sizes involved in L1 that enable a single accelerator configuration to be fully HLS-optimized and provide high speedup while fitting in the FPGA device area.

Resource Consumption. Table III shows the resource consumption and the respective speedup for each accelerator and security level of the CM KEM.

The resource consumption of the Gaussian systemizer is mainly dominated by BRAMs. BRAM usage is directly proportionate to the amount of dual-port BRAMs we utilize in the accelerator, to provide concurrent accesses to different elements of the same array in the same clock cycle. The Gaussian systemizer uses no DSP resources and the amount of LUTs and FFs shows a small increase across the 5 security levels. In a similar manner, the syndrome encoder utilizes BRAMs to perform concurrent accesses to the PK matrix, while the rest of the FPGA resources show a small increase. Finally, the syndrome decoder demonstrates a more balanced profile in its resource consumption. Its LUT and DSP usage is attributed to the bitwise boolean operations and GF multiplications. Moreover, its BRAM consumption is associated with array partitioning. The L1 implementation of syndrome decoder uses a single accelerator configuration. To provide speedup it fully pipelines and unrolls the `gf_mul` computation, thus resulting in high DSP usage. The rest of the FPGA resources exhibit a low usage, as their increased consumption is observed at the creation of multiple concurrent accelerators. Table III reveals that $L5_2$ provides the lowest resource consumption among all L5 implementations. This is so because $L5_2$ has the smallest t value among all L5 KEM parameters, resulting in smaller matrices and decreased FPGA resource usage.

TABLE IV
Duration in 10^3 clock cycles [$k$cc] and overhead over total execution time for the computation and data movement part of the hardware accelerators.

Accelerator	Computation		Data movement		Total
	[$k$cc]	Overhead [%]	[$k$cc]	Overhead [%]	[$k$cc]
Gaussian S.	72913.2	98.6	1043.4	1.4	73956.5
Synd. Enc.	13.8	17.2	65.8	82.8	79.5
Synd. Dec.	2444.4	99.97	0.9	0.03	2445.3

Impact of Vector Instructions. This section quantifies the impact of vector instructions on the performance of our proposal. Fig. 3 shows a 5× and 8.18× average speedup increase over the speedup of auto-vectorized code for KeyGen and Decapsulation, respectively. For the case of Decapsulation we observe that the software auto-vectorization does not provide significant performance increase compared to the non-vectorized code. That is, due to the fact that the Decapsulation software part that can benefit from auto-vectorization is already accelerated on the FPGA accelerators. Table IV shows the runtime percentage of each kernel spent between data movement and data computation. Unlike Gaussian systemizer and Syndrome decoder, the Syndrome encoder accelerator has a high data movement overhead of 82%. The large amount of time spent in data communication, due to the PK data movement, hinders the acceleration gains for the Encapsulation KEM part. For this reason hardware-acceleration of Encapsulation cannot outperform the speedup of software auto-vectorization. Therefore, when executed on a vector processor, software auto-vectorization could be the acceleration method of choice for the Encapsulation part, thus avoiding the accelerator's hardware cost. However, Encapsulation takes $< 1\%$ of the total CM execution time, so we still provide significant speedup over the whole CM KEM application.

We also compare our proposal with the *vec* CM KEM implementation. *vec* is a hand-coded, vectorized across 64-bits software implementation, part of the CM KEM NIST submission. Note that manually vectorized implementations provide performance gains only when they execute on vector processors. However, reconfigurable SoCs do not usually include cores with vector units [10], [11]. Furthermore, unlike HLS-based designs, they require a lot more effort to be developed. Finally, the software part of our proposal is not manually-vectorized, giving us an a priori handicap in this comparison. Nevertheless, in the case of KeyGen, we observe a substantial performance increase of 2.3× in speedup. With KeyGen being the most time-consuming part of the whole KEM, taking up 95% of its total execution time (see Section IV), this comparison demonstrates the solid performance gain we provide to the whole CM KEM. Encapsulation and Decapsulation show speedups of 0.5× and 0.1×, respectively. However, since Encapsulation and Decapsulation have a combined KEM execution time of 5%, we still outperform *vec* for the whole CM KEM execution.

TABLE V
Speedup & resource consumption for different accelerator configurations. For Synd Enc, Synd Dec. #=accelerator's multiplicity, h=High resource consumption config. For Gaussian S. #=(internal parallelization-parallel/sequential parameter regeneration, see Fig 1).

Accelerator	#	Freq.	Resource Consumption [%]				Speedup [x]	
		[MHz]	LUT	BRAM	DSP	FF	over baseline	# gain[%]
Gaussian systemizer	44-s	200	8	41.1	0	5.8	25.7	0
	44-p						31.2	21.4
	128-s	170	14.5	63.2	0	8.2	35	0
	128-p						46.1	31.7
Syndrome encoder	1h	250	6.8	46	3.9	4.5	1.4	-44
	1	333	3.1	37.8	0.3	1	2.5	0
	2	333	6	43.3	0.5	1.7	3	20
	4	333	12.1	54.4	1	3.7	3.2	28
	8	333	25.2	77.2	2	7.5	3.1	24
Syndrome decoder	1h	333	18.9	3.5	71.9	8.5	3.9	18.1
	1	333	2.9	2.6	2.3	2.4	3.3	0
	2	333	6.6	5	4.7	4.9	4.8	45
	4	333	12.5	10.3	9.4	9.7	6.2	87
	8	333	25.7	26.3	18.7	21.3	7.1	115
	16	333	53.5	64.3	37.4	46	7.5	127
	18	333	61.8	76.6	42.1	53.3	7.5	127

Design Space Exploration. The HLS techniques we used in the accelerators design provide our proposal with a balanced solution to the performance-area tradeoff. Table V presents the speedup that different accelerator configurations provide.

Gaussian systemizer takes advantage of its internal parallelization potential to provide acceleration. We point out that the CPU-side optimization of the parallel parameter regeneration (see Section III) provide a performance increase of up to 31.7%. In the case of syndrome encoder and syndrome decoder, we observe that the speedup from executing multiple concurrent accelerators is superior to that of a single fully HLS-optimized accelerator. However the performance gain of KEM plateaus after a certain accelerator multiplicity. For syndrome encoder and syndrome decoder, this saturation is evident for 8 and 16 parallel accelerators, respectively. As security levels and resource consumption of accelerators rise, this number decreases. For all accelerators, along with the increased performance gain, we observe a respective increase in their overall resource consumption. This behavior defines the Pareto optimal solution for the performance-area tradeoff.

VI. Related Work

Several works have studied the implementation of NIST contest candidates and their acceleration in hardware [15]–[24]. Farahmand et al. [25] and Nguyen et al. [26], [27] present a HW/SW co-design methodology in the context of lattice-based PQ KEMs. Their work validates our choice for adopting an HLS-based design, since it provides a good tradeoff between hardware overhead and speedup. The original McEliece cryptosystem has been the subject of hardware implementation studies [12], [28]–[33]. The CM KEM is based on the Niederreiter cryptosystem, which is implemented in an FPGA device by Heyse et al. [34]. Wang et al. [35], [36] also provide a fully-RTL (Verilog) FPGA implementation of the Niederreiter cryptosystem and use the same core mathematical operations as CM, thus making their implementation applicable to the CM KEM. However, their solution does not provide an end-to-end CM KEM implementation, and thus, it cannot be compared with our proposal. In any case, a fully-hardware implementation of CM can naturally surpass the performance of a HW/SW acceleration approach. However, our proposal relies on a reconfigurable SoC and can therefore be adapted to potential CM KEM changes prior to its standardization. Interestingly, our HLS-based accelerators of the CM are able to achieve higher frequencies than the ones reported by Wang et al. [35], [36]. Basu et al. [37] present a purely-hardware (HLS) design of the Encapsulation and Decapsulation algorithms in an FPGA device. Contrarily, our proposal follows the more flexible approach of HW/SW co-design. Additionally, KeyGen which is accelerated by our proposal and not addressed by Basu et al. [37], is the most time consuming part of CM KEM (95% of the total execution time).

VII. Conclusions

To the best of our knowledge, this paper presents the first HW/SW co-design acceleration of the CM KEM based on HLS coding. Our proposal demonstrates significant speedups for the most time consuming parts of the CM KEM. The design of our proposal is highly-portable and can be implemented in current heterogeneous CPU+FPGA platforms. Finally, the source code of this study is available, under an open-source license, at https://github.com/beatsnbytes/classic_mceliece.

VIII. Acknowledgements

We are profoundly grateful to all our anonymous reviewers for their constructive feedback throughout the paper drafting.

This research was supported by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible, under the DRAC project [001-P-001723]. It was also supported by the Spanish goverment (grant RTI2018-095094-B-C21 "CONSENT"), by the Spanish Ministry of Science and Innovation (contracts PID2019-107255GB-C21, PID2019-107255GB-C21) and by the Catalan Government (contracts 2017-SGR-1414, 2017-SGR-705). This work has also received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No. 871467.

V. Kostalabros has been partially supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia under "Ajuts per a la contractació de personal investigador novell" fellowship No. 2019FI B01274. M. Moreto was also partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under "Ramón y Cajal" fellowship No. RYC-2016-21104.

REFERENCES

[1] "NIST announce the release of draft nistir 8105, report on post-quantum cryptography for public comment," 2016. [Online]. Available: https://csrc.nist.gov/News/2016/NIST-Announce-the-Release-of-DRAFT-NISTIR-8105

[2] "PQC standardization process: Third round candidate announcement," 2020. [Online]. Available: https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement

[3] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier *et al.*, "Classic mceliece: conservative code-based cryptography," *NIST submissions*, 2017.

[4] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[5] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Prob. Contr. Inform. Theory*, vol. 15, no. 2, pp. 157–166, 1986.

[6] A. Hülsing, K.-C. Ning, P. Schwabe, F. Weber, and P. R. Zimmermann, "Post-quantum wireguard," Cryptology ePrint Archive, Report 2020/379, 2020, https://eprint.iacr.org/2020/379.

[7] E. Crockett, C. Paquin, and D. Stebila, "Prototyping post-quantum and hybrid key exchange and authentication in tls and ssh," Cryptology ePrint Archive, Report 2019/858, 2019, https://eprint.iacr.org/2019/858.

[8] M. R. Albrecht, D. J. Bernstein, T. Chou, C. Cid, J. Gilcher, T. Lange, V. Maram, I. von Maurich, R. Misoczki, R. Niederhangen, K. G. Paterson, E. Perischetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, C. J. Tjhai, M. Tomlinson, and W. Wang, "Classic McEliece: conservative code-based cryptography," 2020. [Online]. Available: https://classic.mceliece.org/nist/mceliece-20201010.pdf

[9] Xilinx Inc., "Ultrascale architecture and product data sheet: Overview," 2020. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[10] P. D. Schiavone, D. Rossi, A. D. Mauro, F. Gurkaynak, T. Saxe, M. Wang, K. C. Yap, and L. Benini, "Arnold: an eFPGA-augmented RISC-V SoC for flexible and low-power IoT end-nodes," 2020, arXiv:2006.14256.

[11] J. Poupat, T. Helfers, P. Basset, A. G. Llovera, M. Mattavelli, C. Papadas, and O. Lepape, "DAHLIA, very high performance microprocessor for space applications," dahlia-h2020.eu, Tech. Rep., 2019.

[12] S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede, "A speed area optimized embedded co-processor for mceliece cryptosystem," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2012, pp. 102–108.

[13] P. Schwabe, D. Stebila, and T. Wiggers, "Post-quantum TLS without handshake signatures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, p. 1461–1480.

[14] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[15] J. W. Bos, S. Friedberger, M. Martinoli, E. Oswald, and M. Stam, "Fly, you fool! faster Frodo for the ARM Cortex-M4," Cryptology ePrint Archive, Report 2018/1116, 2018, https://eprint.iacr.org/2018/1116.

[16] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard lattice-based key encapsulation on embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 3, pp. 372–393, Aug. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7279

[17] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner, "Efficient FPGA implementations of LowMC and Picnic," in *Topics in Cryptology – CT-RSA 2020 - The Cryptographers Track at the RSA Conference 2020, Proceedings*, S. Jarecki, Ed., vol. Lecture Notes in Computer Science. Springer, 2 2020, pp. 417–441.

[18] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking nist pqc on arm cortex-m4," Cryptology ePrint Archive, Report 2019/844, 2019, https://eprint.iacr.org/2019/844.

[19] B. Koziel, R. Azarderakhsh, and M. Kermani, "A high-performance and scalable hardware architecture for isogeny-based cryptography," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1594–1609, nov 2018.

[20] P.-C. Kuo, W.-D. Li, Y.-W. Chen, Y.-C. Hsu, B.-Y. Peng, C.-M. Cheng, and B.-Y. Yang, "High performance Post-Quantum key exchange on FPGAs," Cryptology ePrint Archive, Report 2017/690, 2017, https://eprint.iacr.org/2017/690.

[21] T. Oder and T. Güneysu, "Implementing the newhope-simple key exchange on low-cost fpgas," in *Progress in Cryptology - LATINCRYPT 2017 - 5th International Conference on Cryptology and Information Security in Latin America, Havana, Cuba, September 20-22, 2017, Revised Selected Papers*, T. Lange and O. Dunkelman, Eds., vol. 11368. Springer, 2017, pp. 128–142.

[22] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Selected Areas in Cryptography – SAC 2013*, T. Lange, K. Lauter, and P. Lisoněk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 68–85.

[23] J. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 11, pp. 1305–1319, 2005.

[24] S. M. Qasim, S. A. Abbasi, and B. Almashary, "A proposed fpga-based parallel architecture for matrix multiplication," in *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, 2008, pp. 1763–1766.

[25] F. Farahmand, D. T. Nguyen, V. B. Dang, A. Ferozpuri, and K. Gaj, "Software/hardware codesign of the post quantum cryptography algorithm NTRUEncrypt using high-level synthesis and register-transfer level design methodologies," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 225–231.

[26] D. T. Nguyen, V. B. Dang, and K. Gaj, "High-level synthesis in implementing and benchmarking number theoretic transform in lattice-based post-quantum cryptography using software/hardware codesign," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2020, pp. 247–257.

[27] ——, "A high-level synthesis approach to the software/hardware codesign of NTT-based post-quantum cryptography algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 371–374.

[28] A. Shoufan, T. Wink, G. Molter, S. Huss, and F. Strentzke, "A novel processor architecture for mceliece cryptosystem and fpga platforms," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2009, pp. 98–105.

[29] T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar, "Microeliece: Mceliece for embedded devices," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2009, pp. 49–64.

[30] S. Ghosh and I. Verbauwhede, "BLAKE-512-based 128-bit CCA2 secure timing attack resistant mceliece cryptoprocessor," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1124–1133, 2012.

[31] S. Ghosh, "On the implementation of mceliece with CCA2 indeterminacy by sha-3," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2804–2807.

[32] P. M. C. Massolino, P. S. Barreto, and W. V. Ruggiero, "Optimized and scalable co-processor for mceliece with binary goppa codes," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, pp. 1–32, 2015.

[33] M. López-García and E. Cantó-Navarro, "Hardware-software implementation of a mceliece cryptosystem for post-quantum cryptography," in *Future of Information and Communication Conference*. Springer, 2020, pp. 814–825.

[34] S. Heyse and T. Güneysu, "Towards one cycle per bit asymmetric encryption: Code-based cryptography on reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 340–355.

[35] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based key generator for the Niederreiter cryptosystem using binary goppa codes," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 253–274.

[36] ——, "FPGA-based Niederreiter cryptosystem using binary goppa codes," in *International Conference on Post-Quantum Cryptography*. Springer, 2018, pp. 77–98.

[37] K. Basu, D. Soni, M. Nabeel, and R. Karri, "Nist post-quantum cryptography-a hardware evaluation study." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 47, 2019.