TYRCA: A RISC-V Tightly-coupled accelerator for Code-based Cryptography

Alessandra Dolmeta*†, Stefano Di Matteo†‡, Emanuele Valea‡, Mikael Carmona†, Antoine Loiseau†, Maurizio Martina*, Guido Masera*
*Department of Electronics and Telecommunications, Politecnico di Torino, Torino, Italy
†Univ. Grenoble Alpes, CEA, Leti, F-38000 Grenoble, France
‡Univ. Grenoble Alpes, CEA, List, F-38000 Grenoble, France

***Abstract*—Post-quantum cryptography (PQC) has garnered significant attention across various communities, particularly with the National Institute of Standards and Technology (NIST) advancing to the fourth round of PQC standardization. One of the leading candidates is Hamming Quasi-Cyclic (HQC), which received a significant update on February 23, 2024. This update, which introduces a classical dense-dense multiplication approach, has no known dedicated hardware implementations yet. The innovative Core-V eXtension InterFace (CV-X-IF) is a communication interface for RISC-V processors that significantly facilitates the integration of new instructions to the Instruction Set Architecture (ISA), through tightly connected accelerators. In this paper, we present a TightlY-coupled accelerator for RISC-V for Code-based cryptograAphy (TYRCA), proposing the first fully tightly-coupled hardware implementation of the HQC-PQC algorithm, leveraging the CV-X-IF. The proposed architecture is implemented on the Xilinx Kintex-7 FPGA. Experimental results demonstrate that TYRCA reduces the execution time by 94% to 96% for HQC-128, HQC-192, and HQC-256, showcasing its potential for efficient HQC code-based cryptography.**

***Index Terms*—RISC-V, CV-X-IF, HQC, FPGA, PQC.**

I. Introduction

Quantum computers, poised to revolutionize fields like medicine, materials science, and artificial intelligence, also pose significant threats to current communication security. Cryptosystems such as RSA, Diffie-Hellman, and Elliptic Curve Cryptography (ECC) are vulnerable to quantum attacks that can solve discrete logarithms and factorize large numbers [1]. To counter this threat, Post-Quantum Cryptography (PQC) aims to develop cryptosystems that are resilient against quantum attacks. Since 2016, the National Institute of Standards and Technology (NIST) has led a standardization process to identify and evaluate PQC algorithms, now in its fourth round, covering several mathematical families. Recently, NIST selected the first standards, mostly based on the lattice-based cryptographic family [2]. NIST's selection process continued with the evaluation of additional cryptographic schemes, ultimately choosing alternatives based on diverse mathematical families to ensure resilience against potential future attacks on lattice-based cryptography. At the moment, the favorite alternative is code-based cryptography, with crypto-schemes like Classic McEliece, BIKE, and HQC, the latter being the focus of this paper. HQC employs structured codes, relying on the difficulty of decoding random quasi-cyclic codes in the Hamming metric. HQC's implementation efficiency lags behind traditional cryptosystems like RSA and ECC, and its hardware implementations have been minimally explored. Existing HQC implementations are either fully hardware-based [3]–[5], or utilize loosely-coupled accelerators for specific parts of the algorithm [6], [7]. Fully hardware-based implementations offer the best performance but they induce a consistent implementation footprint and limited flexibility. Loosely-coupled accelerators offer more versatility since the same accelerator can be shared by several applications. In general, hardware accelerators represent a significant cost for System-on-Chip (SoC) designers who need to face integration efforts and write dedicated software drivers.

An alternative approach is the tightly coupled acceleration. It relies on identifying simple operations, that are recurrent in the algorithm, and that can be encapsulated inside a custom instruction that will be added to the Instruction Set Architecture (ISA) of the target processor. The implication is that the hardware accelerator is integrated inside the pipeline of the Central Processing Unit (CPU) and the use of the accelerator is completely transparent to the application developer. Tightly coupled acceleration is becoming extremely widespread in microprocessor design, thanks also to the rise of the RISC-V standard and its possibility of proposing custom ISAs in the open-source domain. However, from an implementation point of view, integrating tightly coupled accelerators is more challenging than their loosely coupled counterparts. Extending the ISA of a processor usually involves modifying the decode unit of the CPU and the compilation toolchain.

In this paper, we overcome these limitations by using a novel approach for tightly coupled acceleration that is being standardized in the RISC-V community: the Core-V eXtension InterFace (CV-X-IF) [8] [1]. This interface offers a seamless way to support tightly-coupled accelerators by enhancing the CPU with custom or standardized instructions, without modifying the decode unit. This method maintains the advantages of tightly coupled accelerators while significantly reducing the

This work received funding from the France 2030 program, managed by the French National Research Agency under grant agreement No. ANR-22-PETQ-0008 PQ-TLS. This work was funded by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

[1] https://github.com/openhwgroup/core-v-xif/tree/main

978-3-9826741-0-0/DATE25/© 2025 EDAA

integration complexity. Our work aims to explore the feasibility of designing and implementing the first tightly coupled hardware acceleration for HQC. The HQC software implementation considered in this work is taken from the official website, and updated to the last version (round 4 - 02/23/2024) [9].
The novelties proposed in this paper are the following:

- We show how the CV-X-IF interface facilitates smooth integration with a RISC-V CPU core for implementing cryptographic tasks.
- We provide TYRCA, the first CV-X-IF-based tightly coupled implementation of round-4 HQC. The proposed architecture has been deployed on FPGA to show area and performance results.

The paper is organized as follows: section II provides background on the working principles of HQC, and describes existing implementations. The CV-X-IF interface is described in section III, while the TYRCA architecture is presented in section IV. Results are shown in section V. Section VI concludes the paper.

II. Background

This section provides some context on HQC cryptography. First, in subsection II-A, we briefly overview the HQC algorithm and discuss its main bottlenecks. Then, in subsection II-B we present the state of the art on existing HQC hardware implementations.

A. HQC

HQC [9] is a Key Encapsulation Mechanism (KEM) that relies on the hardness of the syndrome decoding problem in linear quasi-cyclic codes. The encryption process involves encoding the message with a publicly known error-correcting code and adding an error that exceeds the code's correction capacity. This error is derived from the public key, and only the private key can effectively reduce it during decryption, allowing the recovery of the original message. The key pair is generated from a random quasi-cyclic code, concatenating Reed-Muller (RM) and Reed-Solomon (RS) codes, and a syndrome. The HQC KEM is composed of three functions: Key Generation (KeyGen), Encapsulation (Encaps), and Decapsulation (Decaps). HQC is available in three different security levels, namely HQC-128, HQC-192, and HQC-256, each with a different parameter set. The higher the level of security, the larger the size of vectors used and, therefore, the complexity of the algorithm.

Bottlenecks. We conducted a profiling of HQC-128 on a 32-bit SoC based on the CV32E40PX core (more details in subsection IV-A). The considered software is the latest version of HQC compiled with optimization level 2 (`-O2`).
From Table I, it is clear that the execution time of the HQC algorithm is significantly influenced by the polynomial multiplication (over 95% of the total execution time). Keccak [10], though minimally contributing, is consistently used in the algorithm. The RS-RM codes, particularly encoding and decoding, present notable bottlenecks during encapsulation and decapsulation, contributing up to 5% of the total execution time.

TABLE I: Kcycles of the main sub-functions of HQC-128 (round 4 - 2024)

Function	KeyGen	Encaps	Decaps
Total	**66,026.999**	**133,331.422**	**208,550.842**
Arithmetic in $\mathcal{R}$	64,603.234	129,206.468	193,809.702
	[97.84%]	[96.9%]	[92.93%]
· vect_mul	64,599.335	129,198.670	193,798.005
· vect_add	3.899	7.798	11.697
Keccak	27.876	83.270	81.886
	[0.042%]	[0.062%]	[0.039%]
RS-RM Code	-	995.100	10.565.171
	-	[0.746%]	[5.067%]
· encode	-	995.100	995.100
· decode	-	-	9,570.071

B. State-of-the-Art HQC Hardware Implementations

HQC has recently garnered significant research interest in efficient hardware implementations. Researchers are focusing on either full hardware solutions, accelerating the entire algorithm, or using specialized loosely-coupled accelerators, enhancing its most computationally intensive functions. For example, [7] improves HQC implementations by addressing polynomial multiplication and decryption inefficiencies. [6] presents HQC's first HW/SW co-design for IoT applications, with optimized software and hardware balancing for competitive energy consumption and performance. [3] optimizes HQC KEM for FPGAs with high performance and minimal area overhead. Authors in [4] provide a high-level synthesis design updated to third-round specifications, guiding RTL developers with parameter tweaks. Recently, [5] introduced a constant-time hardware design with an optimized encoder and decoder, efficient polynomial arithmetic, and novel modulo operations.

TABLE II: Implementation results of existing works of HQC

Ref. Code	Method		FPGA resources [LUT/FF/BRAM]	Latency [cc]	Latency [μs]
[3]		K	2,369/901/10	15,759	88
round 4	Fully-HW	E	4,145/2,128/13	33,438	186
2023		D	8,984/6,596/20	48,212	251
[4]		K	8,324/5,273/6	40,427	270
round 3	Fully-HW	E	13,179/9,851/10	89,110	586
2021		D	15,304/11,704/18	193,082	1270
[5]		K	6,827/5,444/20	6,203	30
round 4	Fully-HW	E	7,064/6,378/20	13,314	79
2023		D	13,327/9,488/20.5	19,880	119
[6]		K		56,000	56
round 3	Loosely	E	8,000/2,400/3	131,000	131
2021		D		557,000	557

Compatibility. Previous works focused on implementing hardware solutions or accelerators for older HQC versions, such as round 3 (2021) and round 4 (2023). These versions are characterized by the usage of a *dense-sparse* implementation of the polynomial multiplication, which opens to more optimization opportunities than the latest implementation, based on a *dense-dense* approach for polynomial multiplication. Due to this difference in the implementation of the polynomial multiplication, direct comparison across these implementations is impractical. Despite being more secure by avoiding potential

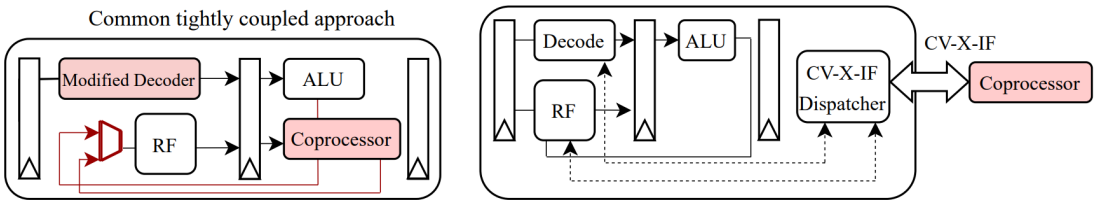

Fig. 1: Common tightly-coupled approach vs CV-X-IF

information leakage [9], using classical dense-dense multiplication is significantly more complex and slower than sparse-dense multiplication. This is because dense-dense multiplication processes all terms, whereas sparse-dense multiplication can skip many zero terms, thereby reducing the number of operations and execution time. Nevertheless, existing FPGA implementations are detailed and compared in Table II, offering insights into performance metrics and resource utilization for KeyGen (K), Encaps (E), and Decaps (D) operations. Our proposal pioneers tightly-coupled acceleration using the latest version of HQC, from round 4 (2024) [9], that employs dense-dense polynomial multiplication.

III. CV-X-IF

Adding a custom instruction to a RISC-V processor typically involves several key steps: defining the instruction format and opcode, designing the coprocessor, and updating some hardware components of the CPU such as the decoder, datapath, and control signals. Additionally, software modifications are required, including updating the assembler, compiler (to support intrinsic functions), simulators, debugging tools, and software libraries and drivers. The Core-V eXtension interface (CV-X-IF) is a RISC-V extension interface designed to simplify the integration of custom coprocessors and instruction set extensions into existing RISC-V processors without altering the CPU or the toolchain. This interface includes dedicated channels for offloading instructions to accelerators, regardless of their type, and for writing back the resulting data using organized and standardized signals. The CV-X-IF was first introduced as a feature of the CV32E40X core [11], a standard-compliant 32-bit RISC-V processor, supporting the base RISC-V integer instruction set (RV32I, RV32E) along with many other standard extensions. Recently, the CV-X-IF was successfully integrated into the CV32E40P core, resulting in a variant labeled CV32E40PX [2].

Fig.1 shows the key differences between the traditional approach and the CV-X-IF approach. In a standard RISC-V core, adding a custom instruction requires modifications to the decode and execution stages to recognize and handle new custom opcodes, ensuring correct operand routing and the integration of the Coprocessor with the existing datapath. In contrast, with the CV-X-IF, the RISC-V core is equipped with a dispatcher that sends instructions to the interface. This setup does not modify the CPU but simply redirects all necessary signals to the CV-X Interface.

[2]https://github.com/esl-epfl/cv32e40px

Hardware accelerators can be connected to the interface through a dedicated controller. In our work, we implemented the CV-X-IF controller, which works in parallel with the core's decoding unit. This controller is responsible for decoding incoming instructions, directing them to the appropriate accelerator, tracking the identification of the offloaded instruction, and managing the destination register address for the result. Specifically, as shown in Fig.3, the CV-X-IF controller examines the instruction opcode extracted from the issue request signal (*issue req*), which is coming directly from the decoding stage of the core. If there is a match with one of the custom instruction opcodes, it activates a valid signal of the response package (*issue resp*), indicating the accelerator's readiness to receive data. Upon validation, the accelerator samples its inputs from the core's register file, ensuring synchronous and efficient data transfer. Subsequently, the co-processor activates the appropriate block from the range of integrated accelerators. After executing the instruction, if the core is prepared to receive the result and the result is valid (*result ready* and *result valid*), it is written back into the register file, at the destination address previously saved. This is done through the result interface (*result*). Notably, instructions offloaded by the RISC-V core during its decode stage are speculative, necessitating thorough verification before the accelerator can write back results through the commit and result interfaces.

IV. TYRCA

In this section, we describe the proposed architecture. In subsection IV-A we provide an overview of the complete SoC. Details on TYRCA and the methodology for the choice of the accelerators are presented in subsection IV-B. The integration method is explained in subsection IV-C, showing the simplicity of this new approach, which allows the addition of custom instructions without modifying the CPU's decode unit, maintaining efficiency and reducing integration complexity.

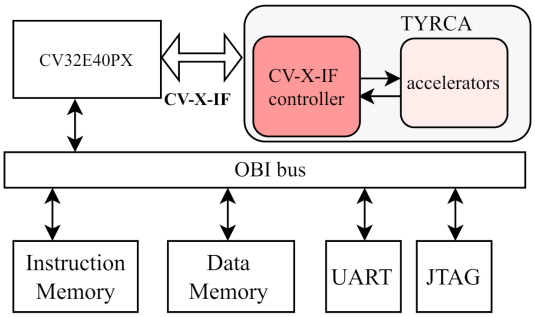

Fig. 2: SoC architecture

A. Processing System Architecture

The new CV-X-IF provides a smooth method for bolstering the CPU by integrating custom instructions, all without altering the CPU's decoding unit. This interface ensures low latency with tightly integrated read and write access to the CPU register file. New extensions can be implemented as separate units external to the CPU and then connected to the CPU through the interface. All newly developed features are accessed using unused opcodes within the CPU.

Our methodology requires a processing system that allows instruction set extensions with the CV-X-IF. Therefore, we chose an adaptive platform with a CV32E40PX core, allowing us to leverage the CV-X-IF for our accelerators. Figure 2 shows our SoC's final architecture.

Apart from TYRCA and the RISC-V core, the main elements of the SoC are instruction and data memories. The JTAG module provides access to the RISC-V core's memories and the register file. The UART module handles serial communication, enabling data exchange between the system and external devices. Additionally, an OBI bus [12], chosen for being an open-source RISC-V standard, facilitates communication between these components.

B. TYRCA Architecture

TYRCA architecture is shown in Fig.3. It is mainly composed of two modules: the CV-X-IF controller and various accelerators.

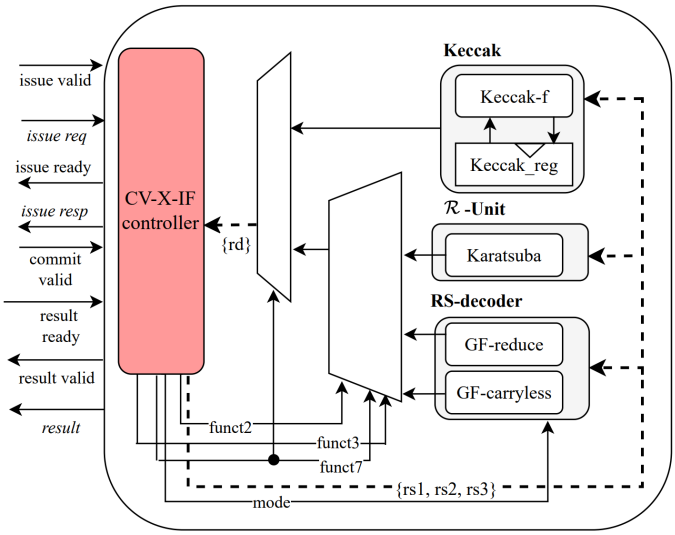

Fig. 3: TYRCA architecture

Following an extensive examination of the HQC algorithm, considering the results of Table I, three main accelerators are implemented: $\mathcal{R}$-Unit, RS-decoder, and Keccak (as shown in Fig.3). Each of these accelerators is customized to execute one or more specific instructions. The CV-X-IF interface, as mentioned in subsection IV-B, is connected directly to the register file of the CPU. Therefore, functions with input and output sizes of 32 bits or less are prime candidates for acceleration.

$\mathcal{R}$-Unit The polynomial multiplication accelerator stands out as the most significant among those implemented. It contributes the most to the performance enhancements achieved, primarily because polynomial multiplication, which is handled by this accelerator, accounts for the largest portion of the total execution time (as reported in Table I). The Karatsuba algorithm required some software adjustments to work correctly with 32-bit inputs and it needed an additional custom instruction for complete result acquisition. The original implementation uses a bitwise Karatsuba multiplication algorithm for 64-bit integers, and it operates on 64-bit integers using bitwise operations and precomputed tables for efficient multiplication. Our version operates on 32-bit integers, breaking each 64-bit integer into two 32-bit chunks (A_0, A_1, B_0 and B_1). Then, it performs conventional multiplication and properly combines the intermediate results. The basic working principle is shown in Fig.4.

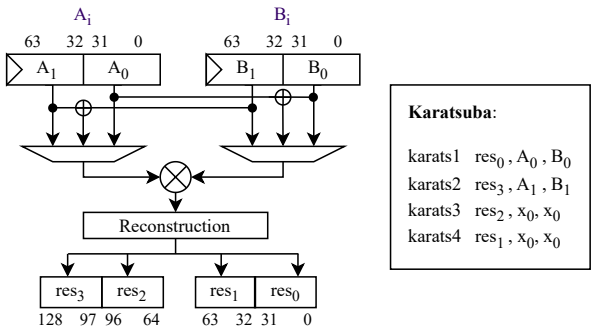

Fig. 4: Proposed Karatsuba architecture

Indeed, since the multiplication involves two 64-bit integers, the result is on 128 bits, whereas our processing system operates on 32-bit units. However, this issue is easily resolved. During the first iteration (`karats1` - first custom instruction of Fig.4), the 32 least significant bits (LSBs) are computed and they can be saved directly into the final result. Similarly, the 32 most significant bits (MSBs) are processed and saved during the second iteration (`karats2` - second custom instruction). The remaining 64 bits are handled in the third iteration (`karats3` - third custom instruction), allowing us to store them 32 bits at a time using an additional store instruction (`karats4` - fourth custom instruction). From an implementation perspective, the accelerator comprises a unique carry-less multiplier along with the necessary logic to manage inputs, manage outputs (Reconstruction module), store intermediate results, and perform XOR operations.

RS-decoder. The Reed-Solomon decode function involves several steps. While we will not delve into the detailed description of each step, it is important to note that we have accelerated certain atomic functions within these steps to enhance performance.

- **GF-reduce.** Galois-Field-reduce function reduces a polynomial x modulo a given primitive polynomial. The straightforward implementation iteratively shifts and XORs parts of x to eliminate higher-degree terms. Three

customized instructions are introduced, to perform part of the processing in hardware. In this way, we can avoid useless intermediate results, by performing multiple operations in a single clock cycle. One of these new instructions is the `trailing zero` function, which counts the number of trailing zero bits in a 16-bit integer using loop and bitwise operations. It iterates through each bit, adding to the count if the bit is zero and updating a mask to stop counting once a non-zero bit is encountered. It is used multiple times inside the polynomial reduction function. Thanks to the ISA extension, a single custom instruction is sufficient, being both input and output on 16 bits.

- **GF-carryless.** This function performs carry-less multiplications of two 8-bit polynomials storing the result in a 16-bit polynomial. This multiplication is similar to binary polynomial multiplication but without carrying over when bits exceed 1. The original function uses precomputed values and bitwise operations to achieve this efficiently. The new function is optimized using four custom instructions to perform carry-less multiplication directly on TYRCA, which enhances efficiency and speed. Leveraging specific instructions for this task eliminates the need for loops and manual bitwise operations present in the original software. The function computes intermediate results and updates the lower and higher parts of the result more efficiently.

Keccak. The Keccak algorithm presents a different scenario. Operating on 1600-bit data, this accelerator requires numerous load and store instructions to fill its internal register. However, the overhead from the load/store process with the CV-X-IF is less than the overhead that would result from attaching the Keccak accelerator as a loosely coupled accelerator to the bus. Since the area requirements for both approaches are approximately the same, we have shown that the CV-X-IF has significant potential for multi-cycle operations as well. The Keccak accelerator features a 1600-bit state and performs 24 rounds of permutation. The Keccak algorithm processes data through a series of compression rounds, each consisting of five distinct steps: θ, ρ, π, χ, and ι [10]. The main challenge with this accelerator lies in the size of the state, which exceeds the capacity of the CPU's register file. To address this, Keccak is the only TYRCA accelerator equipped with a dedicated register. This 1600-bit register allows the entire permutation state to be stored, processed, and returned to the core efficiently. Three custom instructions have been introduced to facilitate this: a specialized `load` operation to upload the state matrix into the Keccak register 64 bits at a time; a `start` instruction to initiate the 24-round process; and a tailored `store` operation to save the result back to the core's register file 32 bits at a time.

TABLE III: Performance improvement of TYRCA operations

Function	SW [cc]	TYRCA [cc]	Calls	Speed-up (%)
Karatsuba	6,072	52	59,142	99.14
GF-carryless	300	56	4,998	81.33
GF-reduce	1,660	145	5,589	91.27
Keccak-*f*	26,825	2,538	143	90.54

Table III compares the different functions described in subsection IV-B, comparing the original clock cycles to the one obtained with TYRCA. The *Calls* column indicates how many times each function is invoked in HQC-128.

TABLE IV: TYRCA custom instructions

Type	Instruction
R-type	".insn r 0x3b, 0xα, 0xβ, rd, rs1, rs2"
R4-type	".insn r 0x4b, 0xα, 0xγ, rd, rs1, rs2, rs3"

C. Integration

Inline assembly in C is employed to leverage the specialized capabilities of TYRCA efficiently. The inline assembly allows assembly language instructions to be embedded directly within C code. In our implementation, custom instructions are defined using the `.insn` directive within the inline assembly block. This directive facilitates the creation of new instructions that interact with the tightly-coupled accelerators. By specifying the opcode, function codes, and operands, these instructions execute specialized operations directly on TYRCA. According to the RISC-V base opcode map [13], there is a limited set of available opcodes. For introducing new R- and R4-type instructions, we have chosen two free opcodes and used `funct3` (α), `funct7` (β) and `funct2` (γ) fields to differentiate between various functions, as these fields allow the encoding of specific parameters required for the distinct operations (see Table IV).

R-type instructions exploit two source registers, while R4 instructions exploit three source registers. The use of inline assembly ensures that input and output operands are correctly mapped to the C variables, allowing seamless integration with the rest of the program.

V. Results

TYRCA is implemented at the RTL using SystemVerilog and deployed on the Xilinx Kintex-7 FPGA, specifically on the Digilent Genesys 2 board. Synthesis and Place&Route are performed using Xilinx Vivado. When synthesized standalone, TYRCA achieves a maximum operating frequency of 150 MHz on the same FPGA.

A. Performance and Code Size

Table V reports the execution time of the original HQC implementation [9] compared to the optimized TYRCA implementation. The code was compiled with optimization level 2 (`-O2`) and ran on FPGA. The results in square brackets represent the gain in percentage with respect to the software implementation.

TABLE V: Execution time (in kcycles) and performance gain (round 4 - 2024)

Version	Level	KeyGen	Encaps	Decaps
SW [9]	HQC-128	66,026.999	133,331.422	208,550.842
	HQC-192	195,307.650	392,977.384	600,490.993
	HQC-256	357,245.410	719,137.771	1,106,009.231
TYRCA	HQC-128	3,108.737 [**-95.29%**]	6,302.661 [**-95.27%**]	11,095.034 [**-94.68%**]
	HQC-192	10,847.998 [**-95.22%**]	22,578.534 [**-95.27%**]	34,895.339 [**-94.97%**]
	HQC-256	20,153.688 [**-95.06%**]	41,469.541 [**-95.13%**]	64,999.457 [**-94.82%**]

The results in Table V illustrate a substantial decrease in clock cycles with the optimized TYRCA implementation compared to the original SW implementation. Across all security levels—HQC-128, HQC-192, and HQC-256—the TYRCA

implementation achieves approximately a 95% reduction in clock cycles for KeyGen, Encaps, and Decaps. This consistent enhancement across various operations and security levels highlights the efficacy of the optimization techniques integrated inside TYRCA. In TYRCA, compared to the original HQC implementation, the instruction memory usage is reduced by 4916 B for HQC-128, 4716 B for HQC-192, and 4852 B for HQC-256. The observed reduction in code size is due to the replacement of entire code sections with single instructions invoking TYRCA operations.

B. Area Overehead

Table VI presents the resource utilization on the Kintex-7 FPGA. Among all the accelerators, the Keccak Unit presents the largest area, which was expected due to the additional 1600-bit register required to store its internal state. TYRCA occupies less than 30% of the total SoC area. The $\mathcal{R}$-Unit, while occupying less than 10% of the total area, is the one that provides the highest performance gain, as shown in Table III.

TABLE VI: Resource utilization (Kintex-7 FPGA)

	LUT	**Registers**	**BRAM**
SoC top-level	33,701	21,806	148
◇ TYRCA Wrapper	8,894	3,710	0
• Accelerators	8,819	3,547	0
○ Keccak	5,418	1,628	0
○ $\mathcal{R}$-Unit	905	255	0
○ RS-decoder	222	0	0
• CV-X-IF Controller	75	163	0

C. Comparison with related works

Table VII presents a comparative analysis of resource utilization and speed-up percentages between our approach and [6], which is the only work that proposes a loosely-coupled approach, presenting different accelerators for selected HQC sub-functions. To enable comprehensive FPGA design comparisons tailored for specific targets, we utilize the *equivalent slice* (eSlice) metric. This metric integrates the contribution of lookup tables (LUTs), flip-flops (FFs), and block memories (BRAMs). Unlike exact slice counts, eSlice provides a standardized benchmark across diverse FPGA platforms. In the context of Xilinx 7-Series FPGAs, a slice comprises four 6-input LUTs and eight FFs. Each BRAM36 can store up to 32 Kib of data, contributing to the eSlice count based on its storage capacity converted into LUT equivalents. The formulation for eSlices is expressed as $\text{eSlices} = \max\left\{\frac{\text{LUT}}{4} + \text{BRAM} \times 128, \frac{\text{FF}}{8}\right\}$. This approach aligns with the methodology outlined in [5].

TABLE VII: Resources, speed-up and efficiency comparison

Ref.	**LUT/FF/ BRAM**	**eSlice**	**Speed-up [%]** K/E/D	**Speed-up/ eSlices**
[6] round 3	7920/2370/3	2364	99/99.05/97.2	41.8/41.9/41.1
OURS round 4	8894/3710/0	2224	95.29/95.27/94.6	42.8/42.8/42.5
OURS w/o Keccak round 4	3464/3168/0	866	94.1/93.89/93.63	108.6/108.4/ 108.1

It is important to note that TYRCA is implementing the latest version of HQC. Thus, direct latency and clock cycle comparisons are not entirely equitable. To facilitate a fairer comparison, we introduce speed-up/eSlices, the ratio of speed-up percentages achieved by the two implementations, normalized by eSlices. This normalization method helps mitigate disparities arising from different HQC versions, providing a relative measure independent of specific implementation details. The higher the speed-up/eSlices ratio, the better it is. As demonstrated in Table VII, our results are better than those of [6], especially in the case in which Keccak is not accelerated. The three values reported in the speed-up and speed-up/eSlices columns refer to KeyGen, Encaps, and Decaps functions. TYRCA performances and resource occupation are also reported without considering the Keccak accelerator, this being the accelerator that occupies the largest area, but not the one that is providing the highest performance gain. In this case, the normalized speed-up is more than doubled.

VI. Conclusions

HQC is an emerging algorithm in the context of PQC, that is lacking of efficient hardware implementations. This is even more true since the release of the latest version which offers fewer opportunities for optimization. We proposed TYRCA, a tightly-coupled acceleration solution for RISC-V architectures based on the CV-X-IF, that offers a seamless integration for extending RISC-V ISA. Experimental results confirm TYRCA's effectiveness, achieving an execution time reduction of 94% to 96% across HQC-128, HQC-192, and HQC-256. This underscores TYRCA's potential in robust post-quantum cryptography, establishing a new standard in HQC hardware implementations.

References

[1] D. Ott, C. Peikert, and et al., "Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility," *CoRR*, vol. abs/1909.07353, 2019.

[2] G. Alagic, D. Apon, and et Al., "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process," tech. rep., National Institute of Standards and Technology (NIST), July 2022. The original report has been updated by an errata update, NIST IR 8413-upd1. Planning Note (09/29/2022).

[3] S. Deshpande, C. Xu, M. Nawan, K. Nawaz, and J. Szefer, "Fast and Efficient Hardware Implementation of HQC." Cryptology ePrint Archive, Paper 2022/1183, 2022. https://eprint.iacr.org/2022/1183.

[4] C. Aguilar-Melchor, J.-C. Deneuville, A. Dion, J. Howe, R. Malmain, V. Migliore, M. Nawan, and K. Nawaz, "Towards Automating Cryptographic Hardware Implementations: A Case Study of HQC," in *Code-Based Cryptography Workshop*, pp. 62–76, Springer, 2022.

[5] F. Antognazza, A. Barenghi, G. Pelosi, and R. Susella, "A High Efficiency Hardware Design for the Post-Quantum KEM HQC," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 431–441, 2024.

[6] M. Schoffel, J. Feldmann, and N. Wehn, "Code-based Cryptography in IoT: A HW/SW Co-Design of HQC," in *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, pp. 1–7, 2022.

[7] C. Li, S. Song, J. Tian, Z. Wang, and C. K. Koc, "An Efficient Hardware Design for Fast Implementation of HQC," in *2023 IEEE 36th International System-on-Chip Conference (SOCC)*, pp. 1–6, 2023.

[8] "CV-X-IF eXtension Interface." https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/x_ext.html. Accessed: April 3, 2024.

[9] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zemor, and I. Bourges, "Hamming Quasi-Cyclic (HQC) Fourth Round Version." Online, 2024. Updated version 23/02/2024.

[10] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and V. K. Ronny, "FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," 2015.

[11] O. Group, *CV32E40X User Manual*, 2024. Accessed: 2024-06-11.

[12] OpenHW Group, “Open Bus Interface Protocol.” https://github.com/openhwgroup/obi, Accessed 2024-04-07.
[13] “RISC-V Instruction Set Manual Volume I: User-Level ISA, Version 2.2.” https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. Accessed: March 15, 2024.