Towards Automating Cryptographic Hardware Implementations: a Case Study of HQC

Carlos Aguilar-Melchor, Jean-Christophe Deneuville, Arnaud Dion, James Howe, Romain Malmain, Vincent Migliore, Mamuri Nawan, Kashif Nawaz

Towards Automating Cryptographic Hardware Implementations: a Case Study of HQC

Carlos Aguilar-Melchor[1], Jean-Christophe Deneuville[2], Arnaud Dion[3], James Howe[1], Romain Malmain[4], Vincent Migliore[5], Mamuri Nawan[6], and Kashif Nawaz[6]

[1] SandboxAQ, Palo Alto, USA. {carlos.aguilar,james.howe}@sandboxaq.com
[2] ENAC, University of Toulouse, France, jean-christophe.deneuville@enac.fr
[3] ISAE-SupAero, University of Toulouse, France, arnaud.dion@isae-supaero.fr
[4] EURECOM, France, romain.malmain@eurecom.fr
[5] INSA / LAAS-CNRS, University of Toulouse, France, vincent.migliore@laas.fr
[6] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE, mamuri@tii.ae, kashif.nawaz@tii.ae

Abstract. While hardware implementations allow the production of highly efficient and performance oriented designs, exploiting features such as parallelization, their longer time to code and implement often bottlenecks rapid prototyping. On the other hand, high-level synthesis (HLS) tools allow for faster experimentation of software code to a hardware platform while demonstrating a reasonable extrapolation of the expected hardware behavior. In this work, we attempt to show a rapid, fast prototyping of the well known HQC algorithm, using HLS, and show how with a modification of certain parameters, varying degrees of comparable results can be obtained. These results, in turn, could be used as a guide for HDL-RTL developers to enhance their designs and better prototyping time in the future. Additionally, we also demonstrate that it is possible to benefit from HQC's versatility; by achieving a low hardware footprint whilst also maintaining good performances, even on low-cost FPGA devices, which we demonstrate on the well known Artix-7 xc7a100t-ftg256-1.

1 Introduction

Quantum-resistant cryptography, more colloquially known as Post-Quantum Cryptography (PQC), has emerged as one of the leading research fields in the broader scope of theoretical and applied cryptography. This research field has appeared due to the likely realization of quantum computers in the next few decades, which threaten the current public-key cryptography standards ubiquitously used today. Indeed, with a large fault-tolerant quantum computer, quantum algorithms are able to trivially solve discrete logarithms and factor very large numbers, which has been the cornerstone of our public-key cryptography standards for the last few decades. In 2016, the National Institute of Standards and Technology (NIST) [15] initiated an open call for post-quantum cryptographic algorithms for public evaluation. This standardization process started

with having 69 accepted submissions for either a key encapsulation mechanism (KEM) or a digital signature scheme (DSS) to in 2022 where we received the first PQC standards; one KEM selection, CRYSTALS-Kyber, and three DSS selections, CRYSTALS-Dilithium, SPHINCS$^+$, and Falcon [4]. Additionally, four KEM candidates were promoted to a fourth round for further analysis, with the potential of them being standardized in the future. Three of these KEMs are based on code-based cryptography, being seen as a good alternative to CRYSTALS-Kyber, a lattice-based cryptography scheme, and which would add diversity to NIST's PQC suite of standards. These three candidates are BIKE, HQC and Classic McEliece. NIST have stated that at the end of the fourth round they intend to standardize at most one of the two former candidates, and also that they are in no rush to standardize the latter [4].

This standardization process partially motivates the purpose of this research. Since the beginning, NIST have stated their desire for hardware designs of these PQC candidates, and have in the past used these results to compare similar proposals in their decision process. We add to this line of research by proposing hardware designs for the code-based KEM HQC, specifically for HQC-128. We utilize tools and techniques for the rapid prototyping of schemes in hardware, specifically high-level synthesis (HLS), which has proven in the past to significantly increase design time for hardware engineers by converting especially designed software code to hardware languages such as VHDL. This strategy was recently shown successful by Guerrieri, Da Silva, Regazzoni, and Upegui for PQC candidates based on lattice-based cryptography [11].

1.1 Design Artifacts

The source code of the HLS designs are available for download at under an open source license at https://pqc-hqc.org/implementation.html.

1.2 Outline of the Paper

The remainder of the paper is structured as follows. Section 2 gives some mathematical preliminaries and a background on the HQC algorithm. Section 3 details the HLS synthesis design and implementation. Section 4 presents our results and compares this HQC design to other, existing designs, both in hardware and software. Section 5 concludes the paper and provides future research directions.

2 Preliminaries and Background

Hamming Quasi-Cyclic (HQC) is a public-key encryption scheme that relies on the hardness of —as its name suggests— decoding random quasi-cyclic codes in Hamming metric. The construction itself shares similarities with Alekhnovich's cryptosystem [5], which uses random linear codes, making it inefficient in practice. HQC was originally proposed in 2016 by Aguilar *et al.* using BCH codes tensored with a repetition code [2, 3]. Aragon *et al.* later proposed an improved

version using Reed-Muller concatenated with Reed-Solomon codes [6] named HQC-RMRS. The version that is currently considered in the NIST standardization process is an IND-CCA2 KEM variant (see [1, Section 2.3.2]) of HQC-RMRS, obtained by applying the Fujisaki- Okamoto ($\mathrm{FO}^{\perp}$) transform to the IND-CPA public-key encryption scheme [12].

This section describes the notations used throughout this paper, and recalls the description of the HQC encryption scheme. For conciseness, we refer the reader:

- to [13] for additional details on Reed-Muller and Reed-Solomon codes;
- to [10] for an introduction to code-based cryptography;
- to [3, 6] for full details about HQC (original and RMRS versions), including the security proof and decryption failure analysis; and
- to [12] for full details about the PKE-KEM conversion.

2.1 Notations

Throughout this document, $\mathbb{Z}$ denotes the ring of integers and $\mathbb{F}_2$ the binary field. Additionally, we denote by $\omega(\cdot)$ the Hamming weight of a vector *i.e.* the number of non-zero coordinates, and by $\mathcal{S}_w^n(\mathbb{F}_2)$ the set of words in $\mathbb{F}_2^n$ of weight w. Formally:

$$\mathcal{S}_w^n(\mathbb{F}_2) = \{\boldsymbol{v} \in \mathbb{F}_2^n, \text{ such that } \omega(\boldsymbol{v}) = w\}.$$

$\mathcal{V}$ denotes the vector space $\mathbb{F}_2^n$ of dimension n over $\mathbb{F}_2$ for some positive $n \in \mathbb{Z}$. Elements of $\mathcal{V}$ can be interchangeably considered as row vectors or polynomials in $\mathcal{R} = \mathbb{F}_2[X]/(X^n-1)$. Vectors/Polynomials (resp. matrices) will be represented by lower-case (resp. upper-case) bold letters. For a vector $\boldsymbol{v}$, v_k denotes its k-th coordinate. For the sake of conciseness, we will say that a prime integer n is primitive if 2 is a primitive n-th root of unity, equivalently if the polynomial $(X^n-1)/(X-1)$ is irreducible in $\mathbb{F}_2[X]$.

For $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{V}$, we define their product similarly as in $\mathcal{R}$, *i.e.* $\boldsymbol{uv} = \boldsymbol{w} \in \mathcal{V}$ with

$$w_k = \sum_{i+j \equiv k \mod n} u_i v_j, \text{ for } k \in \{0, 1, \ldots, n-1\}. \tag{1}$$

HQC takes great advantage of matrices with a cyclic structure. Following [3], $\mathbf{rot}(\boldsymbol{v})$ for $\boldsymbol{v} \in \mathcal{V}$ denotes the circulant matrix whose i-th column is the vector corresponding to $\boldsymbol{v}X^i$. This is captured by the following definition.

Definition 1 (Circulant Matrix). *Let* $\boldsymbol{v} = (v_0, \ldots, v_{n-1}) \in \mathbb{F}_2^n$. *The* circulant matrix *induced by* $\boldsymbol{v}$ *is defined and denoted as follows:*

$$\mathbf{rot}(\boldsymbol{v}) = \begin{pmatrix} v_0 & v_{n-1} & \ldots & v_1 \\ v_1 & v_0 & \ldots & v_2 \\ \vdots & \vdots & \ddots & \vdots \\ v_{n-1} & v_{n-2} & \ldots & v_0 \end{pmatrix} \in \mathbb{F}_2^{n \times n} \tag{2}$$

As a consequence, it is easy to see that the product of any two elements $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{R}$ can be expressed as a usual vector-matrix (or matrix-vector) product using the $\mathbf{rot}(\cdot)$ operator as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u} \times \mathbf{rot}(\boldsymbol{v})^\top = \left(\mathbf{rot}(\boldsymbol{u}) \times \boldsymbol{v}^\top\right)^\top = \boldsymbol{v} \times \mathbf{rot}(\boldsymbol{u})^\top = \boldsymbol{v} \cdot \boldsymbol{u}. \tag{3}$$

Finally, the HQC version considered for standardisation in the NIST PQC process has been modified to use Reed-Muller codes concatenated with Reed-Solomon codes, between the 2nd and 3rd rounds.

Definition 2. [Concatenated codes [1, Section 2.5.1]]
A concatenated code consists of an external code $[n_e, k_e, d_e]$ over $\mathbb{F}_q$ and an internal code $[n_i, k_i, d_i]$ over $\mathbb{F}_2$, with $q = 2^{k_i}$. We use a bijection between elements of $\mathbb{F}_q$ and the words of the internal code, this way we obtain a transformation:

$$\mathbb{F}_q^{n_e} \to \mathbb{F}_2^N$$

where $N = n_e n_i$. The external code is thus transformed into a binary code of parameters $[N = n_e n_i, K = k_e k_i, D \geqslant d_e d_i]$.

2.2 Background on HQC

We now recall the HQC scheme in Figure 1. In [3], the code $\mathcal{C}$ used for decoding is a tensor product of BCH and repetition codes. But since this code is public, its structure has no incidence on security, and one can choose any code family, influencing only the DFR and the parameter sizes.

- $\mathsf{Setup}(1^\lambda)$: generates and outputs the global parameters $\mathsf{param} = (n, k, \delta, w, w_{\boldsymbol{r}}, w_{\boldsymbol{e}})$.
- $\mathsf{KeyGen}(\mathsf{param})$: samples $\boldsymbol{h} \xleftarrow{\$} \mathcal{R}$, the generator matrix $\boldsymbol{G} \in \mathbb{F}_2^{k \times n}$ of the public code $\mathcal{C}$, $\mathsf{sk} = (\boldsymbol{x}, \boldsymbol{y}) \xleftarrow{\$} \mathcal{R}^2$ such that $\omega(\boldsymbol{x}) = \omega(\boldsymbol{y}) = w$, sets $\mathsf{pk} = (\boldsymbol{h}, \boldsymbol{s} = \boldsymbol{x} + \boldsymbol{h}\boldsymbol{y})$, and returns $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Encrypt}(\mathsf{pk}, \mathbf{m})$: generates $\boldsymbol{e} \xleftarrow{\$} \mathcal{R}$, $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2) \xleftarrow{\$} \mathcal{R}^2$ such that $\omega(\boldsymbol{e}) = w_{\boldsymbol{e}}$ and $\omega(\boldsymbol{r}_1) = \omega(\boldsymbol{r}_2) = w_{\boldsymbol{r}}$, sets $\boldsymbol{u} = \boldsymbol{r}_1 + \boldsymbol{h} \cdot \boldsymbol{r}_2$ and $\boldsymbol{v} = \boldsymbol{m}\boldsymbol{G} + \boldsymbol{s}\boldsymbol{r}_2 + \boldsymbol{e}$, returns $\boldsymbol{c} = (\boldsymbol{u}, \boldsymbol{v})$.
- $\mathsf{Decrypt}(\mathsf{sk}, \boldsymbol{c})$: returns $\mathcal{C}.\mathsf{Decode}(\boldsymbol{v} - \boldsymbol{u}\boldsymbol{y})$.

Fig. 1: Description of HQC.

Based on this observation, Aragon *et al.* suggest to use Reed-Muller concatenated with Reed-Solomon codes to reduce the size of the resulting parameters, yielding HQC-RMRS.

For the external code, HQC-RMRS uses a Reed-Solomon code of dimension 32 over $\mathbb{F}_{256}$ and for the internal code, a Reed-Muller code $[128, 8, 64]$ duplicated

3 or 5 times (*i.e.* duplicating each bit to obtain codes of parameters $[384, 8, 192]$ and $[640, 8, 320]$).

For decoding, a maximum likelihood decoding algorithm is first performed onto the internal code, yielding a (noisy) vector in $\mathbb{F}_q^{n_e}$, that is hence decoded using an algebraic decoder for the external Reed-Solomon code.

All technical details regarding the encoding and decoding of Reed-Muller and (shortened) Reed-Solomon are provided in the NIST submission package of HQC [1, Sections 2.5.2 to 2.5.7].

3 HLS Design Implementation of the HQC

In this section, we detail the High-level synthesis (HLS) and the Software (SW) implementations which we optimize. We describe these synthesis and optimizations for HQC with parameters which target NIST Level 1 security, that is 128 bits of security, i.e., HQC-128.

3.1 HLS implementation: Basics

Traditional RTL development, using a HDL-based language, most commonly Verilog or VHDL dates back a few decades providing a robust and concrete methodology in which almost all (if not all) digital designs are conceived, written (in code) and implemented. Clearly, these languages have been resilient and have resisted much change (compared to their more dynamic counterparts in the corresponding software world) and are the *de-facto* jargon of all digital designers and engineers alike. The final implementation of code developed using these languages finds implementations in devices like FPGAs and ASICs, for prototyping or production. The design cycle from concept to the final bit-stream (in case of the FPGA) or the GDSII (in case of ASIC) often involves considerable time and design effort[1] and hence, correspondingly a higher time-to-market. Additionally, if there are changes required to be made on the design, post-routing and post-implementation, it involves considerable debugging in case of FPGAs, to understand the nuances of the proprietary synthesizer engines, whereas in the case of ASICs, if these are detected post-fabrication, in involves a complete reversal to the RTL design phase.

Although digital designers have ways to circumvent and prevent such catastrophic failures, software engineers, or hardware-software co-designers/system architects, cannot simply afford the time to port their software code to a RTL design based flow and debug all the way back. High-level synthesis offers a cheap, quick and versatile design flow methodology wherein the software designer can predict what their code would perform like in a hardware setting, what will be the resources and performance numbers, acknowledging nevertheless, that handcrafted RTL could outperform, but at the cost of increased time and effort. We now briefly review some terminologies specific to the HLS design methodology.

[1] we note here that compared to FPGAs, ASICs have a much higher and longer turnaround time.

1. Initiation-Interval, II, is defined as the minimum number of clock cycles between any two successive operations. In the case of HLS, the initiation interval is defined w.r.t the loop iterations, i.e., the number of clock cycles between any given two loop iterations. In an ideal pipelined based flow, the expected value of the II=1 [11].
2. Loop iteration, n, is defined as the number of counts an operation is repeated; in a design implementing pipelines, the loop iteration is simply number of times a pipeline is full when performing operations.
3. Iteration latency, t_{il}, the number of (minimum) clock cycles required to complete one loop iteration (i.e, for $n = 1$)
4. Latency-Area product (LAP), the classical LAP metric, in the context of HLS methodology [11], can be defined as

$$LAP = (t_{ii} \times (n-1) + t_{il}) \times Area \tag{4}$$

where $(t_{ii} \times (n-1) + t_{il})$ represents the total latency and Area, the number of Slices (or LUTs).

3.2 Methodology and Implementation

The NIST submission package for the HQC, available from the HQC designers' web-page[2], contains the reference, optimized and the hardware implementations available for download. The reference implementation is the NIST KEM submitted version which contains the source C-files from the authors' of the algorithm. The README file details the conversion requirements as per NIST FAQ #13 and describes the different variants of the submission, which we omit here for brevity. Additionally, the submission provides the reference implementations for all the 3 proposed versions of the HQC-algorithm, namely, hqc-128, hqc-192, and hqc-256. During the build, the corresponding binaries are generated in the build folder.

The optimized implementation consists of the AVX2 implementation and is constant time (as per the latest submission specification) and avoids any secret-key dependent memory access [1].

The hardware implementation consists of a HLS-compatible C implementation (although the authors' explicitly specify the C++ extension) which can be compiled standalone as a C code (with the same functionality as the golden reference implementation) or translated into a VHDL implementation using the HLS design flow methodology.

Additionally, we provide an area-friendly `compact` and a performance-oriented `perf` version to allow for trade-offs between an area-footprint and throughput.

In addition to the above, the vanilla (or the pure) version also adds an optimized version, wherein we manually refactor some of the HLS-synthesized VHDL-generated code to remove possible duplications of the modules (as in the case of keccak which we explore in the next section).

[2] See https://pqc-hqc.org/implementation.html.

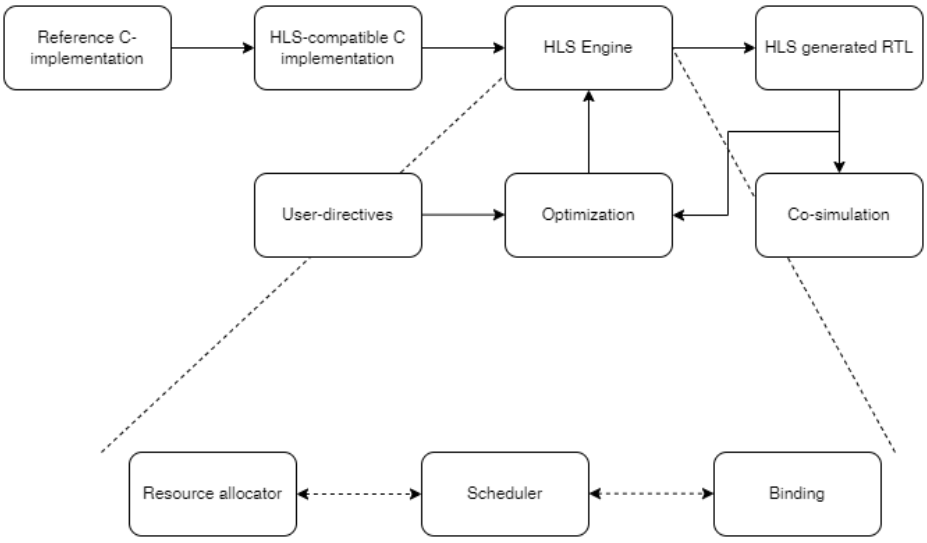

Fig. 2: A highly simplified overview of the HLS flow. Adapted from [17]

Design flow methodology The methodology consists of converting a design, written in a high-level programming language such as C/C++, to a hardware description language (HDL), such as Verilog or VHDL. HLS consists of 3 main steps: Resource allocation (or sharing), Scheduling (using a scheduler) and binding [8]. As the name implies, Resource allocator allocates or allows sharing of multiple resources (such as functional blocks like Block-RAMs, DSP units, LUTs, Registers etc.) between different code blocks. Scheduler is responsible for the actual implementation of the target operation, corresponding to the operation defined in the C-code. For instance, a multiplier operation between, any two variables, x and y, and the corresponding product, z would entail, retrieval of x and y, from their corresponding stored locations (which could a ROM, RAM, or simply registers holding on to those values), determining their bit widths, looking for optimizations (if could be applied such as using a DSP unit), computing the product z, and finally writing back to the specified target location (in RAM) or holding onto the value for subsequent computations. For all of these enumerated operations, the scheduler determines (or rather estimates) the number of clock cycles required and schedules the operation either in a single clock cycle or over a span of multiple clock cycles. This then allows the scheduler to determine if parallelism can be exploited to reduce the number of clock cycles, by also computing the number of available resources at hand and checking for any instructions that could (potentially) have some data-dependencies [8]. Binding allows for the variable to be linked to a functional unit (storage or otherwise) to allow for better optimization.

HLS implementation of HQC The HQC algorithm submission package consists of 3 main algorithms, as defined in the preceding section, namely the keypair generation, the encapsulation function and the decapsulation functions. These can be categorized according to the top level functions, first introduced in [17], namely, the `crypto_kem_keygen`, `crypto_kem_enc` and `crypto_kem_dec` functions. Each of these functions could then be further divided into smaller functions based on their respective modules; for instance, `crypto_kem_enc` would consist of the `crypto_kem_pke_ind_cpa`. As mentioned above, the IND-CPA scheme is transformed into a secure CCA-2 KEM, thanks to the FO transform.

1. `crypto_kem_keygen` outputs the public key, pk and the secret key sk
2. `crypto_kem_enc` outputs the shared key K and the ciphertext ck taking the public key pk as input
3. `crypto_kem_dec` outputs the key K, taking in the secret key sk and the ciphertext ck as inputs

While HLS allows for rapid prototyping and outputs an HDL netlist, it must be highlighted that the tool in itself is restricted in terms of the functionalities it can implement; for instance, converting recrusive functions, unspecified length of execution loops are currently not supported by the HLS synthesis engines, integration of open source libraries, which software implementations rely upon, for instance, during random number generation, present a scope for further optimizations and enhancing the synthesis capabilities of such tools [11].

4 Results and Comparisons

In this section, we present and describe our synthesis, implementation and simulation results for the High-level synthesis of the HQC algorithm. Although the results described here are for HQC-128, we hypothesize similar trends for the other variants, i.e., HQC-192 and HQC-256 and leave these implementations for future works.

4.1 Target settings

The HLS has been synthesized and implemented for a target frequency of $\geq$ 125 MHz (i.e., a clock period of 8ns). In principle, achieving higher target clock frequencies (especially for performance oriented applications) is desirable, nevertheless, our goal is to demonstrate the versatility of HLS in general and a first-pass at the design, so as to benchmark the overall performance/area of a design over a broad range of applications quickly, rather than elaborate into timing closure and optimizations issues (which require finer tuning of the directives settings and the code itself). The target FPGA for these settings is the Xilinx artix7 xc7a100t-ftg256-1.

4.2 Synthesis Results

In this section, we present our HLS results for the given target FPGA for each module (i.e., individual functions). This allows for a granular understanding of how the HLS engine is able to resource allocate and share similar blocks across different operations. Note, for brevity, we present only the values for the optimized perf version of our implementation.

1) `crypto_kem_keygen`: Table 1 presents the synthesis and implementation results for the modules (functions) comprising the keypair generation module. For the sake of brevity, we report the minimum values of the latency (both in terms of the number of clock cycles required and the absolute time) to demonstrate the competitiveness of the HLS based designs. Additionally, we provide the post-implementation results to highlight the optimizations the tool (in this case Vivado, and not the Vitis or HLS tool) is able to implement the HLS synthesized netlist. We also compare our design with the state-of-art for the available modules, for instance, the polynomial multiplier and adder module from [9] is compared with our vector multiplier and adder module. Although handcrafted RTL is superior in terms of the number of clock cycles (required for the computation), nevertheless it provides a close enough approximation for a software designer to optimise the design. We also note that the RTL design (from [9]) uses four BRAM modules compared to zero from the HLS design.

Table 1: Post-synthesis latency and area results for the individual `keygen` functions.

Module (function)	Latency		BRAM	DSP	FF	LUT
	Clocks	Time				
seedexpander_init	51	0.337 µs	0	0	3 384	9 552
seedexpander_mult_ty	3048	20.117 µs	0	0	3 723	9 240
shake_prng	4	26.40 ns	0	0	3 380	9 055
shake_prng_init	70	0.462 µs	1	0	3 572	10 311
vect_mul_add	21418	0.141 ms	0	0	3 701	6 175
poly_mult & add module[3] [9]	18976	83 µs	4	-	906	2137
vect_set_random_fixe	2573	16.982 µs	0	0	4 354	9 942
vect_set_random_fixe_1	2478	16.335 µs	0	0	72	225
fixed weight generator module [9]	3649	16.39 µs	2	0	124	316

2) `crypto_kem_enc`: Table 2 presents the synthesis and implementation results for the modules (functions) comprising the encapsulation module. Again, for the sake of brevity, we report the minimum values of the latency (both in terms of the number of clock cycles required and the absolute time) to demonstrate the competitiveness of the HLS based designs like above. Additionally, we omit the modules which are shared across the different functions for instance, the `shake_prng_init`, the `vect_set_random_fixe` functions described above.

Table 2: Post-synthesis latency and area results for the individual `enc` functions.

Module (function)	Latency		BRAM	DSP	FF	LUT
	Clocks	Time				
reed_solomon_encode	803	5.3 µs	0	0	423	1 523
reed_muller_encode	6 441	42.511 µs	0	0	180	496
vect_add	2 211	14.593 µs	0	0	54	112
shake256_512_ds	238	1.57 µs	0	0	3 721	10 343
shake256 [7]	270	1.80 µs	0	0	270	2017
hqc_ciphertext_to_st	4 489	29.627 µs	0	0	177	483
hqc_public_key_from_s	559	3.689 µs	0	0	118	164

From Table 2, specifically for the SHAKE256 module, HLS outperforms the handcrafted RTL design in terms of latency. Although, the reader may be tempted to point to the larger usage area using HLS, which indeed can be further optimized, the design from [7] uses a parallel slice based design, and the total area is computed using the LUTs as both logic and memory, which amortizes the total overall area cost.

3) `crypto_kem_dec`: Table 3 presents the synthesis and implementation results for the modules (functions) comprising the encapsulation module. Again, for the sake of brevity, we report the minimum values of the latency (both in terms of the number of clock cycles required and the absolute time) to demonstrate the competitiveness of the HLS based designs like above. Additionally, we omit the modules which are shared across the different functions for instance, the `shake_prng_init`, the `vect_set_random_fixe` functions described above.

Table 3: Post-synthesis latency and area results for the individual `dec` functions.

Module (function)	Latency		BRAM	DSP	FF	LUT
	Clocks	Time				
reed_solomon_decode	12 774	0.11 ms	3	0	1 910	7 894
reed_muller_decode	55 845	0.482 ms	0	0	659	2260
vect_compare_64	18	0.115 µs	0	0	13	100
vect_compare	2 210	19.070 µs	0	0	19	143
hqc_ciphertext_from_s	4 465	38.528 µs	0	0	152	413
hqc_public_key_from_s	559	4.824 µs	0	0	118	164

4.3 Modular comparisons among different versions

To the best of our knowledge, this is the first work which targets a high-level synthesis hardware implementation of the HQC algorithm, although a very recent work which targets a complete handcrafted Hardware implementation of HQC is available from [9].

In this subsection, we provide a detailed breakdown of the hardware components utilized for all the 3 functions, for the sake of completeness of our results, in Table 4.

Table 4: Implementation comparisons of HQC-128 across the different implementation variants with state-of-the-art.

Target	**Alg.**	**Design**		**Freq (MHz)**	**Slices**	**LUT**	**FF**	**BRAM**	**Latency**	
									Clocks	ms
HLS (this work)	keygen	pure	Perf	153	8 359	24 746	21 746	7	40 427	0.27
			Comp	132	2 470	7 907	9 544	7	626 589	5.01
		optimized	Perf	150	3 921	11 484	8 798	6	40 427	0.27
			Comp	130	1 541	4 676	9 544	6	626 589	5.01
	encaps	pure	Perf	148	9 955	29 496	26 333	11	89 131	0.59
			Comp	131	3 075	9 544	9 544	11	1 482 332	11.85
		optimized	Perf	152	5 575	16 487	13 390	10	89 110	0.59
			Comp	129	2 122	9 544	9 544		1 482 332	11.85
	decaps	pure	Perf	150	8 434	24 898	21 680	18	193 004	1.27
			Comp	129	3 168	9 544	9 544	21	2 152 313	17.21
		optimized	Perf	152	6 223	18 739	15 243	18	193 082	1.27
			Comp	130	2 678	9 544	9 544	21	2 152 313	17.21
RTL [9]	keygen	single clock	-	164	-	2 350	1 106	9.5	23 480	0.14
		dual clock	-	242	-	3 094	879	14.5	27 013	0.12
	encaps	single clock	-	164	-	2 725	2 060	15.5	52 757	0.32
		dual clock	-	218	-	2 609	2 070	15.5	45 739	0.30
	decaps	single clock	-	164	-	8 426	6 642	36	78 233	0.48
		dual clock	-	204	-	8 434	6 652	36	71 199	0.43

Figure 3 provides a comparison of the area (measured in the number of LUTs) between the different variants the HLS HQC-128 design offers. Clearly,

- The area utilization of the perf version is $\times 2$ - $\times 3$ the comp version, which is expected. This is further elaborated in the difference (both in terms of absolute latency (measured in ms and the number of clock cycles) and the frequency of operation, as detailed in Table 4.
- The optimized version clearly outperforms the pure or the baseline version, for both the perf and comp variants, across all the functional modules, i.e., keygen, encapsulation and decapsulation.
- Interestingly, the gain (in terms of area, higher is better) is better for the optimized version across the perf and comp variants compared to the baseline implementation. This demonstrates that irrespective of the architecture deployed, (i.e., round based or performance oriented), HLS is able to optimize the overall design in a better fashion.

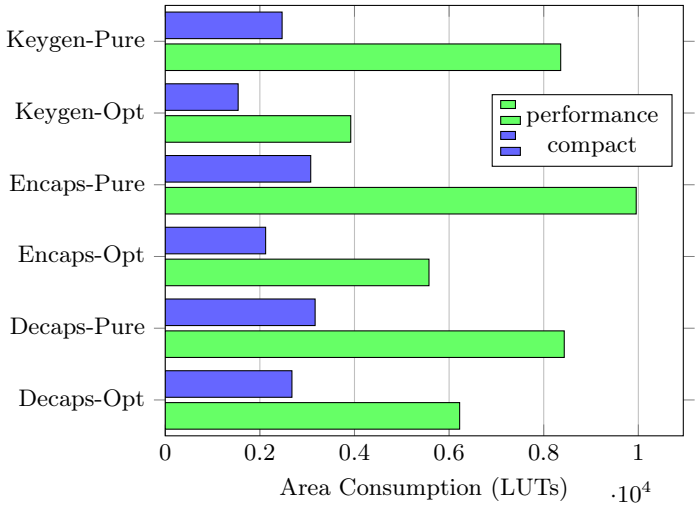

Fig. 3: Area (measured in LUTs) optimizations between perf and compact for the pure (baseline) and optimized versions.

4.4 Comparisons with Software Implementations

In Table 5 we provide results for our proposed HLS designs of HQC-128 compared to those in software, taken from SUPERCOP [18]. For variety, the table shows results from a variety of different CPU targets, and thus provide an overall indicator of how performant the proposed implementations are in comparison. Our performance enhanced design makes significant savings in clock cycles compared to high-end CPUs, with savings between 4-7x on average. Our compact enhanced design has at least a 2x saving in clock cycles compared to the low-end CPU results. Overall we see that HLS designs are a viable option for implementing HQC, for both high-end and low-end devices and applications.

Table 5: Benchmarking results of HQC-128, comparing those in software, taken from SUPERCOP-20220506 (using 50% median values), to our results for HLS.

Platform	**Clock Cycles**		
	Keygen	**Encaps**	**Decaps**
Intel Xeon Skylake (2015)	202 120	351 273	645 728
AMD Ryzen 7 (2017)	307 486	661 913	1 259 627
ARM Cortex A53 (2018)	1 509 404	3 029 021	5 179 020
Artix 7 FPGA (perf)	40 427	89 110	193 082
Artix 7 FPGA (comp)	626 589	1 482 332	2 152 313

4.5 Comparison with state-of-art Hardware Implementations

We now compare our HLS HQC implementations with the available state-of-the-art handcrafted RTL implementations in Table 6 targeting code-based post-quantum implementations, to allow for a fairer and sensible comparison. Clearly, we note

1. HLS-HQC-128 version outperforms all of the handcrafted RTL implementations, most notably in terms of the area optimizations (i.e., LUTs). Notably, HLS has a lower BRAM footprint compared to most handcrafted RTL implementations.
2. In terms of the frequency of operation, HLS (both comp and perf) variants offer a comparable frequency of operation. We note that this is limited to the loop unrolling, which is typically preferred in HLS design, and can impact frequency. Additionally, given the overall optimizations the tool performs, trying to limit the overall area, could also result in degradation of the frequency.
3. The overall latency (in terms of the absolute value in ms, for the perf version is comparable to the handcrafted RTL [9] (we do not claim that HLS outperforms them) but rather point out that such values, if obtained very earlier in the design cycle, allow for a better optimization of the existing code. A similar argument can be made for the clock cycles required for each operation. Nevertheless, we clearly see our HLS based design outperforms the existing BIKE (level 1) and classic McEliece and the SIKE RTL-based implementations.

Table 6: FPGA design comparisons of post-quantum code-based KEMs at NIST L1 security level across implementation variants lightweight (LW) and high-speed (HS), for our works this corresponds to comp and perf, respectively. For HQC-128-RTL we provide single- (SC) and dual-clock (DC).

PQC Scheme	**Imp.**	**LUT**	**FF**	**BRAM**	**DSP**	**Freq (MHz)**	**Latency (cc/10^6, ms)**					
							keygen		**encaps**		**decaps**	
HLS-HQC-128 (this work)	LW	8 876	6 405	28	0	132	0.62	5.01	1.48	11.85	2.15	17.21
	HS	20 169	16 374	25	0	148	0.04	0.27	0.09	0.59	0.19	1.27
Classic McEliece [7]	LW	23 890	45 658	139	5	112	8.88	79.20	0.13	1.10	0.17	1.50
	HS	40 018	61 881	178	4	113	0.97	8.60	0.03	0.30	0.10	0.90
BIKE-L1 [16]	LW	12 868	5 354	17	7	121	2.67	21.90	0.20	1.20	1.62	13.30
	HS	52 967	7 035	49	13	96	0.26	2.60	0.01	0.10	0.19	1.90
HQC-128-RTL [9]	SC	16 320	10 044	61	0	164	0.02	0.14	0.05	0.32	0.08	0.48
	DC	16 956	9 837	66	0	204	0.03	0.12	0.06	0.30	0.08	0.43
SIKE [14]	LW	11 943	7 202	21	57	145	-	25.60	-	27.20	-	15.10
	HS	22 673	11 661	37	162	109	-	15.30	-	16.30	-	9.10

5 Conclusions

Given the interesting results HLS has generated, the importance of HLS in rapid prototyping and in HW-SW co-designs cannot be understated, although the authors note that HLS cannot be a (complete) alternative to RTL developed using handwritten code. Rather, we emphasize that for certain designs where a quick understanding of the bottleneck parts of a larger algorithm need to be identified and quickly reworked upon, then HLS is the perfect candidate for such. We would also like to extend our work to the hqc-192 and hqc-256 versions, in addition to HLS implementations of other code-based cryptographic schemes as open future works. Additionally, this paves the way for further design automation in hardware based designs and allows for designing better and efficient implementations with the minimal effort and time [11].

References

[1] Aguilar Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Bos, J., Deneuville, J.C., Dion, A., Gaborit, P., Lacan, J., Persichetti, E., Robert, J.M., Véron, P., Zémor, G.: HQC (October 2020, updated 06/06/2021) NIST Round 3 submission for Post-Quantum Cryptography. *3*, *4*, *5*, *6*

[2] Aguilar Melchor, C., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. CoRR **abs/1612.05572** (2016) *2*

[3] Aguilar Melchor, C., Blazy, O., Deneuville, J., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. IEEE Trans. Inform. Theory **64**(5) (2018) 3927–3943 *2*, *3*, *4*

[4] Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: NIST IR 8413: Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. (2022) *2*

[5] Alekhnovich, M.: More on average case vs approximation complexity. In: 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings., IEEE (2003) 298–307 *2*

[6] Aragon, N., Gaborit, P., Zémor, G.: Hqc-rmrs, an instantiation of the hqc encryption framework with a more efficient auxiliary error-correcting code (2020) *3*

[7] Chen, P., Chou, T., Deshpande, S., Lahr, N., Niederhagen, R., Szefer, J., Wang, W.: Complete and improved FPGA implementation of classic mceliece. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(3) (2022) 71–113 *10*, *13*

[8] Coussy, P., Gajski, D.D., Meredith, M., Takach, A.: An introduction to high-level synthesis. IEEE Design & Test of Computers **26**(4) (2009) 8–17 *7*

[9] Deshpande, S., Nawan, M., Nawaz, K., Szefer, J., Xu, C.: Towards a fast and efficient hardware implementation of hqc. Cryptology ePrint Archive, Paper 2022/1183 (2022) https://eprint.iacr.org/2022/1183. *9*, *10*, *11*, *13*

[10] Gaborit, P., Deneuville, J.C.: Code-based cryptography. In: Concise Encyclopedia of Coding Theory. Chapman and Hall/CRC (2021) 799–822 *3*
[11] Guerrieri, A., Marques, G.D.S., Regazzoni, F., Upegui, A.: Design Exploration and Code Optimizations for FPGA-Based Post-Quantum Cryptography using High-Level Synthesis. (3 2022) *2*, *6*, *8*, *14*
[12] Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Theory of Cryptography Conference, Springer (2017) 341–371 *3*
[13] Huffman, W.C., Kim, J.L., Solé, P.: Basics of coding theory. In: Concise Encyclopedia of Coding Theory. Chapman and Hall/CRC (2021) 3–44 *3*
[14] Massolino, P.M.C., Longa, P., Renes, J., Batina, L.: A compact and scalable hardware/software co-design of SIKE. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2) (2020) 245–271 *13*
[15] NIST: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2016) https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. *1*
[16] Richter-Brockmann, J., Mono, J., Güneysu, T.: Folding BIKE: scalable hardware implementation for reconfigurable devices. IEEE Trans. Computers **71**(5) (2022) 1204–1215 *13*
[17] Soni, D., Basu, K., Nabeel, M., Karri, R.: A hardware evaluation study of nist post-quantum cryptographic signature schemes. In: Second PQC Standardization Conference, NIST (2019) *7*, *8*
[18] SUPERCOP: System for unified performance evaluation related to cryptographic operations and primitives. https://bench.cr.yp.to/supercop.html *12*