IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Digital Object Identifier

Design Exploration and Code Optimizations for FPGA-Based Post-Quantum Cryptography using High-Level Synthesis

ANDREA GUERRIERI[1 2] (Senior Member, IEEE), GABRIEL DA SILVA MARQUES[1], FRANCESCO REGAZZONI[3,4], AND ANDRES UPEGUI[1]
[1]University of Applied Sciences of Western Switzerland, Geneva, 1202 Switzerland
[2]Swiss Federal Institute of Technology in Lausanne, Lausanne, 1015 Switzerland
[3]University of Amsterdam, Amsterdam, The Netherlands
[4]Università della Svizzera italiana, Lugano, Switzerland

Corresponding author: Andrea Guerrieri (e-mail: andrea.guerrieri@ieee.org).

ABSTRACT FPGAs are an extremely attractive computing platform for Post-Quantum Cryptography (PQC) due to their spatial computation capability and energy efficiency compared to traditional CPUs. The current way of designing with FPGAs is at RTL (Register-Transfer-Level) using VHDL or Verilog hardware description languages. However, creating efficient solutions is extremely time expensive, requiring months of development and verification time. With the evolution of Electronics Design Automation (EDA), technologies such as High-Level Synthesis (HLS) offer a concrete and mature solution to increase design productivity. Nevertheless, sometimes the QoR (Quality of Results) can be suboptimal due to the difficulties of HLS in handling general-purpose software code. In this paper, we explore current limitations of HLS for PQC and we propose code-level optimizations to overcome these problems, increasing QoR of generated hardware. We analyzed and improved results for the lattice-based PQC algorithm competing in the 3rd round of the NIST standardization process. We show how, starting from the original reference code submitted for the competition, original performance and resource utilization can be improved, in some cases with a speedup factor up to 200× or an area reduction of 80%.

INDEX TERMS FPGA, High-level synthesis, Post-Quantum Cryptography, Lattice-based, Saber, Crystals Kyber, Crystal Dilithium, NTRU

I. INTRODUCTION

In recent years, quantum computing gained a lot of momentum. Abiding by the rules of quantum mechanics, these computers can solve in reasonable time problems that are considered hard for classical computers. But this great advance in technology also poses a threat to classical and standard encryption algorithms. Indeed, these algorithms rely on the very same hard problems that quantum computers are now able to break. For this reason, in 2016 the American National Institute of Standard and Technology (NIST) started a contest for replacing encryption standards with post-quantum resistant cryptography, algorithms able to defy attacks from quantum computers. Almost seventy algorithms have been submitted, using six different mathematical approaches; lattice-based, multivariate, hash-based, code-based, supersingular elliptic curve isogeny cryptography, and symmetric key quantum resistance. After two rounds of hardening tests and elimination, only a tenth of the seventy algorithms is still running for the third round of standardization [1]. The result of this contest will be a new set of certified Post-quantum Cryptography (PQC) algorithms. FPGAs are an extremely attractive computing platform for PQC due to their spatial computation capability and energy efficiency compared to traditional CPUs. The current way of designing with FPGAs is at RTL (Register-Transfer-Level) using VHDL or Verilog hardware description languages. However, creating efficient solutions is extremely time expensive, requiring months of development and verification time. High-level synthesis, shortly HLS, is a mature Electronics Design Automation (EDA) solution for building hardware design very quickly.

It produces automatically HDL code for FPGAs out of C/C++, bridging the gap from algorithm to hardware design. Nowadays we can find commercial and academic tools. Both Xilinx and Intel FPGA, the two largest FPGA companies offer HLS tools targeting their FPGA families. For the NIST submission, all the algorithms have been implemented in C, a language widely supported for high-level synthesis. Furthermore, these PQC algorithms are currently objected of extensive hardening testing for the standardization process. Hence, they can be subject to revisions to improve robustness or simply for bug fixes. In this context, the adoption of EDA tools such as HLS can be extremely interesting for bridging the gap from algorithm to hardware hence reducing time-to-market. In addition, starting from the same source code, HLS is capable of producing a wide range of different microarchitectures, each one with different characteristics in terms of latency and resource utilization depending on the synthesis directives adopted.

A. WHY SYNTHESIS DIRECTIVES ARE NOT ENOUGH?

Synthesis directives allow improving performance with, in best cases, the price of increasing resources. Sometimes, improper use can increase resource usage without having any improvement on latency. Effective usage of synthesis directives requires hardware design knowledge and a clear understanding of the HLS process.

However, Figure 1 shows the main limitation of this approach; even adopting the best pragma configuration, sometimes HLS-based designs are pareto suboptimal in respect with RTL-based designs [8]. Therefore, we will show in this paper how to overcome performance bottlenecks allowing HLS to improve QoR and achieve competitive results, as shown in Figure 2.

II. CONTRIBUTION AND RELATED WORK

High-level synthesis for PQC has been used mainly to off-load CPU from heavy-computations using a hardware-software co-design approach [11]. In other cases, it has been used for doing design space exploration and comparing the characteristics of the different algorithms using synthesis directives such as loop unrolling and pipelining [7]. Our experiments differ from those related works for the following points;(1)we do not use a hardware-software co-design method, hence our module is fully synthesized in hardware for FPGA. (2) We do not limit our experiments to synthesis directives (#pragmas) that are very tool-specific, rather we understand and overcome performance limitations using code re-factorization techniques, which are instead valid for all HLS tools. General-purpose optimizations techniques for high-level synthesis have been addressed in literature as research paper [12] [21] [14] [23] [22] and reference books [13]. In this paper we leverage on these techniques, applying to complex and innovatives use case as a post-quantum cryptography algorithms. The paper is organized as follows: section III introduces the algorithms under test, section IV presents the principles of the high-level synthesis process and QoR metrics. Section V shows the model setup for hls and profiling after first synthesis. Section VI presents the optimizations adopted in this paper, how they apply, and the impact they have on each algorithm. In section VII we resume the final results and we conclude the paper with our considerations.

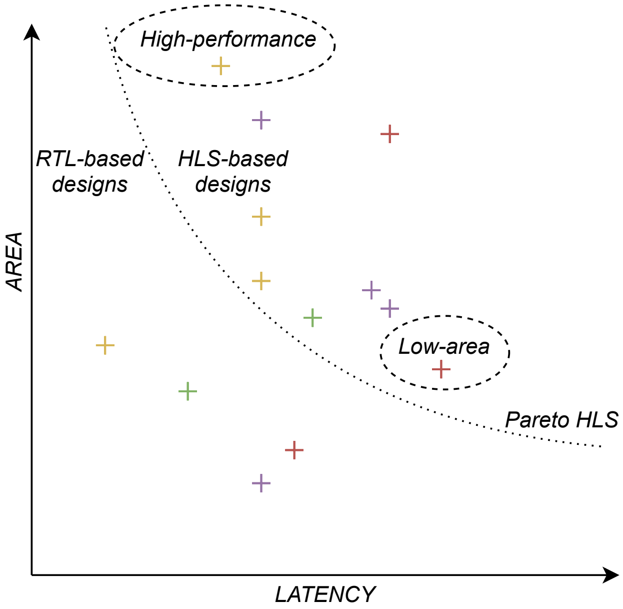

FIGURE 1: Area Latency Design Points. HLS-based design are suboptimal to RTL-based designs and bounded on the right side by the limitations imposed by the automated process [8].

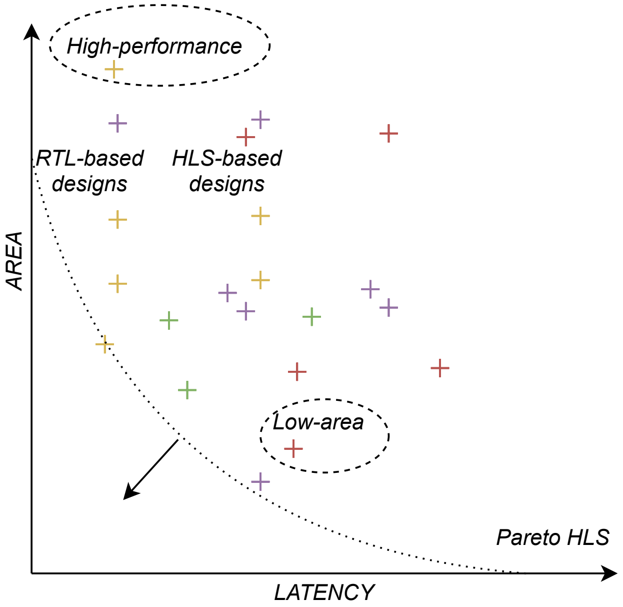

FIGURE 2: Our goal: push EDA boundaries achieving competitive results. Both RTL-based and HLS-based designs shares now the same space of solutions.

III. LATTICE-BASED POST-QUANTUM CRYPTOGRAPHY ALGORITHMS

Among the algorithms competing in the third round of the standardization process, we focus our analysis on the lattice-based. In particular, we analyzed Saber [2], Crystals Kyber [3] and Dilithium [4], NTRU [5]. In this paper we omitted Falcon, which is planned for a future work. For the limits imposed by the purpose of this manuscript we do not enter into mathematical model no into details of the software implementation of the algorithms. We limit to a brief introduction with references to the official documentation.

A. SABER

Saber relies on the hardness of the Module Learning With Rounding problem (Mod-LWR). Saber.PKE, indistinguishability under chosen-plaintext (IND-CPA) secure encryption scheme and transform it into Saber.KEM, indistinguishability under chosen-ciphertext attack (IND-CCA) secure key encapsulation mechanism, using a version of the Fujisaki-Okamoto transform [6]. Saber supports three different security levels named LightSaber, Saber, and FireSaber, respectively AES-128, AES-192, and AES-256-equivalent, which can be set using preprocessor directives $\#defines$ changing iterations and array sizes. In this work we focused on Saber, as representative model of the algorithms' family.

B. CRYSTALS KYBER

CRYSTALS Kyber is an IND-CCA2-secure key encapsulation mechanism (KEM), whose security is based on the hardness of solving the learning-with-errors (LWE) problem over module lattices. It is available in three security levels: Kyber-512, Kyber-768, and Kyber-1024 which are equivalent in terms of hardness to respectively AES-128, AES-194, and AES-256. It was developed to offer the same implementation for different levels of security, only the number of iterations changes as the number of dimensions increases by a multiple of 256. The main operations required for Kyber are variants of Keccak, additions, multiplications, Gen_matrix, and (Number Theoretic Transform) NTT. In the scope of this work, we focused on optimizing Kyber-768 which offers a good compromise between resource and latency usage and security.

C. CRYSTALS DILITHIUM

CRYSTALS Dilithium is a digital signature scheme that is strongly secure under chosen message attacks based on the hardness of lattice problems over module lattices. The security notion means that an adversary having access to a signing oracle cannot produce a signature of a message whose signature he hasn't yet seen, nor produce a different signature of a message that he already saw signed. Dilithium, for now, has the smallest public key + signature size of any lattice-based signature scheme that only uses uniform sampling. Dilithium has three versions: Dilithium2, Dilithium3, and Dilithium5 which represent 3 levels of security depending on a dimension parameter that allows using the same algorithm but with different matrix sizes and execution in more or fewer cycles. In the scope of this work, we focused on optimizing Dilithium3 which offers a good compromise between resource and latency usage and security. Dilithium and Kyber are both from the same family: CRYSTALS (Cryptographic Suite for Algebraic Lattices), and therefore they have functions in common that make it easier to adapt optimization from one algorithm to the other.

D. NTRU

NTRU is built on top of a generic transformation, coming from a correct deterministic public key encryption scheme. After the first appearance of NTRU 30-years ago, different variants have been developed. For the post-quantum standardization contest, NTRU-HRSS-KEM and NTRUEncrypt have been merged. For the third round, the submission package contains multiple versions: ntruhps2048509, ntruhps2048677, ntruhps4096821, and ntruhrss701. In this paper we focus our attention on the ntruhrss701, the version recommended by the NTRU's development team [5].

IV. BACKGROUND ON HIGH-LEVEL SYNTHESIS

In this section, we introduce the principle behind the process of high-level synthesis, with the scope to understand the optimizations steps presented in the paper. The process of high-level synthesis consists of the transformation from untimed code C/C++, into hardware description language Verilog or VHDL. The standard process of high-level synthesis is mainly composed of three phases; resource allocation, scheduling, and binding [24]. In the first phase, resource allocation, the tool identifies the number of hardware resources needed to accomplish the function. The second phase, scheduling, design control FSM (Finite-State Machine) to coordinate the operations. In the third phase, binding, the control FSM is then connected to the datapath, as well as further optimizations as resource sharing, static timing analysis, and buffer insertion to meet the design constraints. Depending on the synthesis directives HLS can generate multiple hardware designs, optimized for performance or resource utilization. To appreciate the quality of generated hardware different metrics can be used. We introduce a few here below.

1) Wall-Clock Time

Wall-clock time indicates the effective execution time needed for the execution of the task by the generated hardware. It is calculated as the product in between the minimum clock period and the latency expressed in clock cycles.

$$WallClockTime = ClockPeriod \cdot Latency \tag{1}$$

2) Latency-Area Product

Sometimes, to reduce latency a increase of parallelism is needed, which reflects into an increase in resources. Latency-area product is a meaningful parameter indicating the quality of the results achieved.

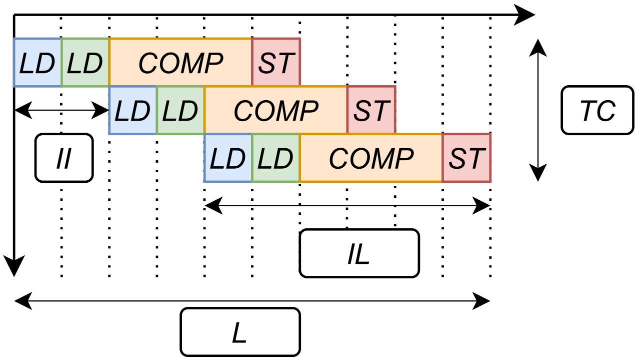

FIGURE 3: Schedule of a pipelined loop. The labels identify abstract operations present in the body loop, LD(load), COMP(compute), and ST(store)

$$LAP = WallClockTime \cdot Area \tag{2}$$

A. OTHER IMPORTANT METRICS: INITIATION INTERVAL

When pipelining a loop, the II identifies the numbers of clock cycles where the pipelines must wait before starting a new iteration of the loop. A perfect pipeline is indicated with II=1, meaning that with each clock cycle a new iteration of the loop can begin. Figure 3 shows the schedule of a pipelined loop.

$$Latency = IL + II \cdot (TC - 1) \tag{3}$$

Where

- $Latency$ = total latency of a given loop
- IL = iteration latency, the clock cycles required to execute one loop iteration
- II = initiation interval, distance between two independent iterations
- TC = trip-count, number of loop iterations

From equation 3 we observe how the latency contribution of IL is independent of the number of iteration. Conversely, the contribution of II increases proportionally with TC.

1) Factors Affecting II

The main objective of the scheduling algorithms is to achieve $II = 1$. For doing that, HLS compilers commonly leverage standard memory analysis techniques like polyhedral analysis to analyze dependencies and thus create the optimal schedule for the given pipeline. Sometimes, achieving sub-optimal results. The principal causes of $II > 1$ can be due to resource conflicts, or loop-carried dependencies. **Resource conflict** happens when a hardware resource with limited ports is accessed multiple times in the same iteration of the loop. In an example, this could be a BRAM or a specific shared custom resource. **Loop-carried dependencies** appear when the data used in the future iteration depends on a result produced in the current iteration. In this case, delaying the beginning of the next iteration by the number of clock cycles required to produce the result needed. This could be for example the memory location computed in the loop body. Sometimes loop-carried dependencies can be solved applying techniques such as tiling, transposition, or interleaving [12]. In some other cases, they cannot be solved at compile time due to runtime dependencies. To overcome these problems innovative techniques based on dynamic schedule have been introduced in HLS tools like Dynamatic [15], which produces dynamically-scheduled circuits, where the scheduling process is delegated to components [16].

V. MODELS SETUP AND SYNTHESIS WITH HLS

The submission packages contain multiple software implementations optimized for different targets platforms like CPUs with AVX (Advanced Vector Extension) or ARM Cortex-M. For our experiments, we use the reference implementation to avoid compatibility issues between the HLS compiler and target-specific instructions, or coding structures for specifically constrained computer architectures. To proceed with the synthesis of the code the identification of the top level functions is needed. For the encapsulation schemes, Saber, Kyber, and NTRU, the top level functions are as following :

- **Crypto_kem_keygen** generates the public key (*pk*) and secret key (*sk*).
- **Crypto_Kem_enc** takes as input the public key (*pk*) and generates a session key (*k*) and the ciphertext (*ck*).
- **Crypto_Kem_dec** takes in secret key (*sk*) and encapsulated key (*ck*), and outputs key (*k*).

For Dilithium, a signature scheme, these top level functions are as following:

- **Crypto_sign_keygen** generates the public key (*pk*) and secret key (*sk*).
- **Crypto_sign** takes as input the secret key (*sk*), the message (*m*), the message length (*mlen*) and generates a signed message (*sm*) and the signed message length (*smlen*).
- **Crypto_sign_open** takes as input the public key (*pk*), the signed message (*sm*) and the signed message length (*smlen*) and generates the message (*m*) and the message length (*mlen*)

HLS support for C/C++ language has historically been limited to a constrained subset. Despite the improvements in language integration, taking a software code meant to run on a classical CPU still needs an assessment. Recursive functions, unbounded loops, and third-party software libraries are not supported for hardware synthesis. Then, all random number generation functions using OpenSSL had to be exported outside the design because of their non-support by HLS tools. Thus creating a new input in top-level functions. Also, to make design validations we replaced top-level arguments pointers with defined size arrays.

A. PROFILING

The goal is the identification of the design bottleneck: the critical functions where most of the clock cycles and resources utilization are spent. Since most of the functions are common to both encapsulation/decapsulation operations, for the sake of simplicity and limits in the manuscript we report the profiling results that are valid in both cases. For doing this, we set it as an initial target frequency 100MHz, with the scope to not bias our initial results with potential timing issues or resources overhead due to the excessive usage of pipeline registers.

1) Saber

Apart from the random number generation, Saber's reference code is compatible with HLS. The latency contribution for each internal module is reported in table 1, including the resource utilization. More than 57% of the total clock cycles are spent by the matrix-vector multiplications, and almost 30% by the inner product, both executing the polynomial multiplication.

TABLE 1: Saber: Latency-Area utilization per module

Function	**Latency (cc)**	**BRAMs**	**LUTs**	**FFs**	**DSPs**
sha3_256	3506	8	18352	3529	0
sha3_512	681	8	18269	3493	0
GenMatrix	9629	10	19124	3984	0
GenSecret	5162	9	18673	3662	0
MatrixVectorMul	58696	15	13161	4459	67
InnerProd	19564	15	13092	4440	67
sha3_256	857	8	16627	3369	2

2) Crystals Kyber

Apart from the modifications presented earlier, the reference code did not require any major modification. The full results for baseline are reported in table 2. We see that blocks using the most resources are the hash, getnoise and gen_matrix functions, that is to say mostly those using Keccak. They are responsible for the use of 97 % of LUTs but in counterpart, they don't impact much latency. 85 % of latency is spent in NTT and gen_matrix.

TABLE 2: Kyber: Latency-Area utilization per module

Function	**Latency (cc)**	**BRAMs**	**LUTs**	**FFs**	**DSPs**
gen_matrix	32407	8	19568	4054	0
getnoise_eta2	1567	9	18526	3518	0
getnoise_eta2_1	1567	9	18526	3515	0
shake256	1115	8	17961	3362	0
basemul_mont	1281	1	445	456	30
basemul_mont_1	1281	1	418	452	30
basemul_mont_2	1281	1	393	446	30
pack_ciphertext	5072	0	1095	258	2
polyvec_ntt	42483	1	689	562	5
polyvec_invntt	59379	1	704	536	8
sha3_256_1	4058	8	18526	3500	0
sha3_256_2	3865	8	18439	3477	0
sha3_256	921	8	17972	3356	0
shake256_1	985	8	17960	3359	0
sha3_512	809	7	17969	3370	0

3) Crystals Dilithium

Apart from the modifications presented earlier, the reference code did not require any major modification. The full results for baseline are reported in table 3. We see that blocks using the most resources are those using Keccak as poly_challenge (15% of LUTs), poly_uniform, and poly_uniform_gamma1 (22% of LUTs each). Functions spending must of the latency are poly_challenge, poly_matrix_expand and as in Kyber, NTT and INVNTT. Dilithium is a relatively simple algorithm but with numerous loops which will increase the latency in the tens of millions.

TABLE 3: Dilithium: Latency-Area utilization per module

Function	**Latency (cc)**	**BRAMs**	**LUTs**	**FFs**	**DSPs**
poly_uniform	13115	11	57831	11857	0
p_unif_gamma1	12629	11	56931	11405	0
poly_challenge	124638	9	39284	8502	0
keccak_absorb_2	5855	2	20645	4868	0
keccak_absorb_1	6639	2	18804	3884	0
keccak_squeeze_1	118	2	17275	3198	0
keccak_squeeze_2	96	2	17258	3194	0
pvk_invntt_tomont	46195	1	886	611	6
pv_matrix_expand	393522	-	-	-	-
polyvecl_ntt	33370	-	-	-	-
polyveck_ntt	44044	-	-	-	-
polyveck_ntt	44044	-	-	-	-
while_omega	11409278	-	-	-	-
pvl_unif_gamma1	63155	-	-	-	-
polyvecl_ntt	33370	-	-	-	-
pv_matrix_montgomery	43098	-	-	-	-

p = poly, pv = polyvec, pvk = polyveck, pvl = polyveck, unif = uniform

4) NTRU

NTRU presents very simple loops and a contained resource utilization. On the other side, latency-wise is very expensive. The majority of the latency contribution is due to poly_Rq_mul function, consuming 99% of the total. This function is called once in the encapsulation and three times during the decapsulation process.

TABLE 4: NTRUHRSS701: Latency-Area utilization per module

Function	**Latency (cc)**	**BRAMs**	**LUTs**	**FFs**	**DSPs**
sha3_256	1562	8	18326	3489	0
poly_Sq_frombytes	701	0	751	94	0
poly_Sq_tobytes	2194	0	812	143	0
poly_lift	9800	1	905	249	6
poly_Rq_mul	4916113	-	-	-	-
poly_Z3_to_Zq	1400	-	-	-	-
poly_S3_tobytes	19564	-	-	-	-

VI. DESIGN AND OPTIMIZATIONS

The optimization method used in our experiments is inspired by literature [12] [21] [14] and extended for a practical complex design. It is essentially composed by the stages showed in Figure 4. First, we try to concentrate our effort to increase performance, limiting as much as possible the overhead in resources. In particular, we focus on optimize the generated pipelines and exploiting the parallelism, doing parallel computations per clock cycle. Second, we focus on resource optimization, such as sharing resources and memory. In detail, we apply:

- **Pipeline Optimizations** : to reach the maximum speed, the goal is to achieve II=1 for all the pipelined loops.
- **Exploiting Parallelism** : exploiting the spatial computing of FPGA doing parallel computation elements per clock cycle.
- **Memory optimizations** : avoiding as much as possible data movements reusing the same memory
- **Resource Sharing** : reusing computation units in different temporal moments, reducing resource utilization.

In the coming paragraph, we pass through each optimization stage explaining how this applies for each algorithm under design.

A. INCREASING PERFORMANCE

The goal of this step is to speed up the solution in generating more efficient hardware. We analyze the cause of suboptimal initiation interval and applying the optimization techniques [12] [13] we improve the pipeline to achieve $II = 1$. Furthermore, we try to exploit spatial computation feature of FPGAs, running parallel operations in the same clock cycle. The scope of this optimization step is to optimize as much as possible the available parallelism provided by the underlying computation platform. Despite the most common technique to increase parallelism in HLS consists of unrolling loops, our primary goal is to improve the execution without recurring on that hence without introducing resource overhead. Usually, state-of-the-art HLS tools automatically detect data and operation dependencies and try to exploit instruction-level parallelism (ILP). Nevertheless, the capability of this analysis is generally limited to a single module. Task-level parallelism extraction is a current limitation and should be identified and solved by the designer. How two tasks can be executed in parallel? If two tasks are both data-independent and resource-independent they can be executed in parallel without area overhead. Why synthesis directives are not enough in this case? Unfortunately, dataflow pragma can not be applied to a module containing both data-independent and data-dependent tasks, generating conservative scheduling to ensure functionality.

1) SABER

Polynomial multiplication is based on toom_cook_4way karatsuba. The toom_cook_4way is mainly composed of three parts: evaluation, multiplication, and interpolation. In the multiplication stage, seven calls to karatsuba are instantiated. The schedule diagram generated synthesizing the reference code is reported in Figure 5. The evaluation part is composed of two independent loops of $N\_SB = 64$ iterations, with $II = 2$ and $IL = 3$. The cause of suboptimal II is due in this case of memory contention due to the 4 accesses to BRAMs, with only 2 access ports. Splitting the array is possible then achieving maximum performance. Additionally, loop fusion allows reducing the total latency by merging the two loops into a single one. Internal results for karatsuba loops show $II = 64$ over a $IL = 64$, highlighting the potential difficulty in exploiting the pipeline. How to improve the pipeline in this case? The karatsuba core is mainly composed of two nested loops. While pipelining automatically all the loops, the inner loop is unrolled, although the loop-carried dependencies limit the pipeline. Using the tiling or loop interchange technique the loop-carried dependency can be solved. However, with the cost of increasing DSP resources. Avoid unnecessary loop unrolling instead, and pipelining only the inner loop an initiation interval of 7, with a no increase of DSP usage and a still a reasonable speedup. Resource contention due to the limited BRAM ports can be solved by partitioning the array. The interpolation part, instead, requires a different approach. The original schedule presents a $TC = 127$, with $IL = 11$ and $II = 7$ limited again by the BRAM limited access ports. Nevertheless, in this case, the array partition cannot be easily applied because of the array accesses pattern. In this case, we apply the loop fission or loop splitting [17] to remove the dependencies and improve overall latency. Figure 6 shows the final schedule. Applying those techniques, we achieve a total speedup of 4.85× in respect to the baseline design and 2.3×, and 15× of DSP reduction in respect to the best version using only synthesis directives. In the case of Saber, the DCFG shows the matrix generation and secret generation are both data and resource independent. In the original source code, they are called in sequence in the indcpa-kem-enc module, therefore the parallelism cannot be exploited due to the presence of other non-independent tasks. In this case, a simple re-

```
int i, j;

// EVALUATION
for (j = 0; j < N_SB; ++j) {
    //aw1-7 initializations
    //...
}
for (j = 0; j < N_SB; ++j) {
    //bw1-7 initializations
    //...
}

// MULTIPLICATION

karatsuba_simple(aw1, bw1, w1);
karatsuba_simple(aw2, bw2, w2);
karatsuba_simple(aw3, bw3, w3);
karatsuba_simple(aw4, bw4, w4);
karatsuba_simple(aw5, bw5, w5);
karatsuba_simple(aw6, bw6, w6);
karatsuba_simple(aw7, bw7, w7);

// INTERPOLATION
for (i = 0; i < N_SB_RES; ++i) {
    //...
}
```

Listing 1: Toom Cook 4 Way, original code

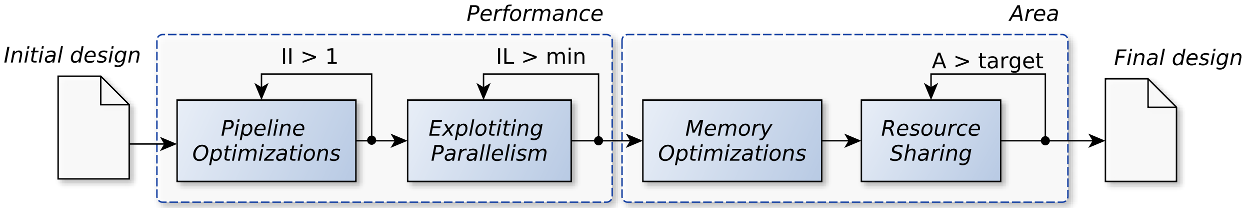

FIGURE 4: Our Proposed Optimization Process

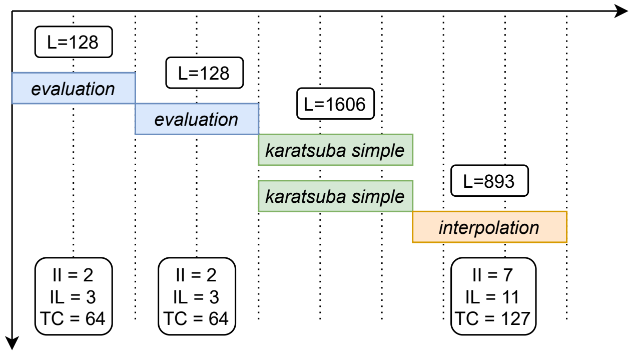

FIGURE 5: Toom Cook 4 Way - Resulting schedule

arrangement of these instructions in a sub-function allows the parallelism to be exploited. This optimization step allows a further latency reduction of 10 %, without introducing resource overhead.

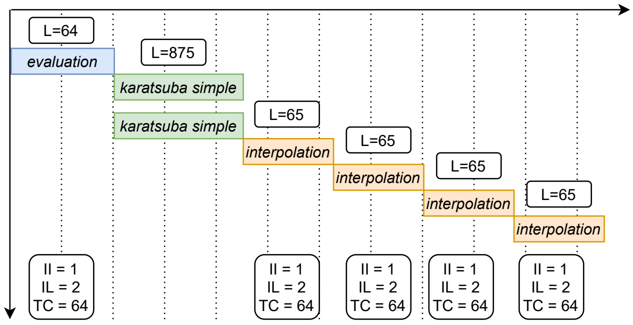

FIGURE 6: Toom Cook 4 Way - Final schedule after pipeline optimizations.

2) Crystals Kyber/Dilithium

Kyber and Dilithium share most of their functions and logic so they are both treated in this section.

NTT optimization The Number Theoretic Transform is useful to compute convolutions and is used multiple times in the algorithm. It's reference implementation is presented by listing 3. Loops of NTT have variable bounds and are too complex for HLS to understand so it fails to evaluate latency. The usage of directives allowed the tool to understand that the reduce function is executed a total amount of 896 times. Despite the global directive to the pipeline, HLS could not do so because of both loop-carry and loop-independent dependencies. Indeed, it waits for the termination of the reduce function before starting a new iteration, leading to an II=13 for the inner loop. Figure 7 represents the scheduling with data dependency. The use of directives for loop-independent dependencies and refactoring the inner loop for loop-carry dependencies allowed to pipeline the algorithm to achieve a II = 2. Furthermore, loops were modified as in listing 4 to simplify the tool analysis and make it possible to use unroll primitives to increase performance. The II = 1 can be achieved by using a supplementary BRAM or by partitioning the array. The final schedule is presented in Figure 8.

```
    int i, j;

    // EVALUATION
    for (j = 0; j < N_SB; ++j) {
        //aw1-7 initializations
        //...
        //bw1-7 initializations
        //...
    }
    // MULTIPLICATION

    karatsuba_simple(aw1, bw1, w1);
    karatsuba_simple(aw2, bw2, w2);
    karatsuba_simple(aw3, bw3, w3);
    karatsuba_simple(aw4, bw4, w4);
    karatsuba_simple(aw5, bw5, w5);
    karatsuba_simple(aw6, bw6, w6);
    karatsuba_simple(aw7, bw7, w7);

    // INTERPOLATION
  for (i = 0; i < N_SB_RES/4; i++)
  {
        //...
  }
  for (i = 0; i < N_SB_RES/4; i++)
  {
        //...
  }
  for (i = 0; i < N_SB_RES/4; i++)
  {
        //...
  }
  for (i = 0; i < N_SB_RES/4; i++)
  {
        //...
  }
}
```

Listing 2: Toom Cook 4 Way, modified code

```
k = 1;
for(len = 128; len >= 2; len >>= 1){
    for(start = 0; start < 256; start = j + len){
        zeta = zetas[k++];
        for(j = start; j < start + len; j++){
            t = fqmul(zeta, r[j + len]);
            r[j + len] = r[j] - t;
            r[j] = r[j] + t;
        }
    }
}
```

Listing 3: NTT as described in NIST submission for Kyber

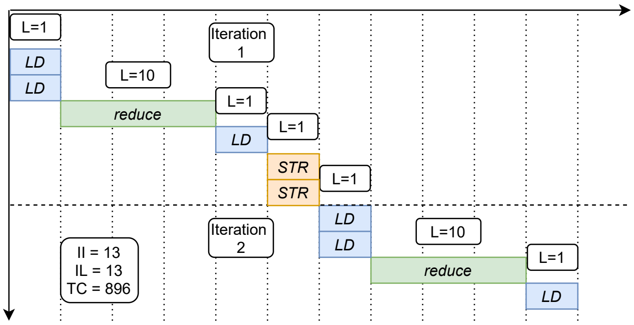

FIGURE 7: NTT initial - First two iterations of the NTT with data dependency

Loops merge. Crystals use sometimes simple functions but are repeated in loops many times for the different dimensions of matrices. Every time HLS enters and exits a loop it takes cycles and thereby many cycles are used in loop control. However, most of these loops have the same number of iterations so by using directives or by merging manually these, we can avoid wasting cycles in control.

```
k = 1;
zeta = zetas[k++];
for (len = 128; len >= 2; len >>= 1){
    limit = len;
    start = 0;
    for (int j = 0; j < 128; j++){
        uint16_t r_j_len = r[start + len];
        uint16_t r_j = r[start];
        t = fqmul(zeta, r_j_len);
        r[start + len] = r_j - t;
        r[start] = r_j + t;

        start++;
        if (start == limit)
        {
            start += len;
            limit += (len << 1);
            zeta = zetas[k++];
        }
    }
}
```

Listing 4: NTT rewritten

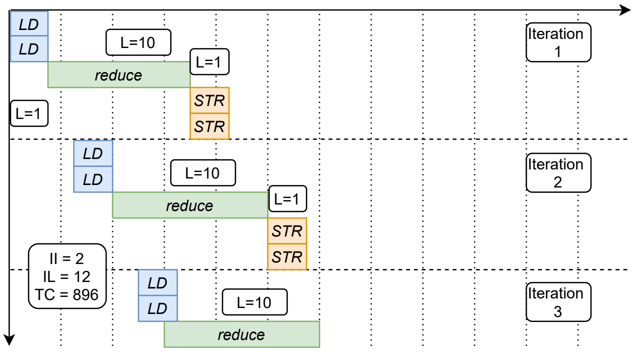

FIGURE 8: NTT pipelined - First two iterations of the NTT with loop-carry and loop-independent dependencies corrected.

3) NTRU

Pipelining NTRU did not present any significant issue with the initiation interval. The loops structures are regular and simple, perfect use-cases where high-level synthesis technology improved lot QoR over the years. Therefore, the nested loops with large iterations (701) increase the total clock cycles. To improve performance the increase of parallelism is needed. More specifically, the nested loop Poly_Rq_mul covers the main contribution of the total latency. The original code is composed of three loops: one external that iterates of a factor=701, and two internals, which presents variable a variable iteration bounds, shown in list 6. The resulting schedule is reported in Figure 9. To improve the latency of this function, applying synthesis directives such as loop unrolling is not sufficient. Few code modifications are needed for exploiting the parallelism of this loop. The first is the loop merge. Merging the two internal loops with a fixed bound reduces overall the complexity. However, even improving the bottleneck in the memory accesses between $a[]$, $b[]$, and $r[]$, the loop unroll cannot be exploited, as shown in 10. In fact, due to the structure of the loop, applying the unrolling pragma increase the resource utilization without improving performance due to the dependency to the $r[k]$. To remove the loop-carried dependency and fully exploit the hardware parallelism, the loop interchange transformation should be applied, inverting the boundary index of k and i. Using the modified code shown in Figure 6 the dependency to $r[k]$ is removed and the loop can be unrolled, as shown in Figure 11.

B. REDUCING AREA

Sharing resources is the pass allowing the area reduction, consisting in the use of the same hardware resources in different temporal moments. Efficient stack usage and preventing memory leaks are always the first concerns of a software programmer. But what happens when we use the source code with HLS? Array declarations are synthesized as BRAMs, and variables as registers. To reduce the BRAM utilization is worth analyzing the time-of-life of specific arrays, along with the potential parallel utilization, and then trying to reuse it as much as possible. Executing our analysis we managed

```
int k,i;
for(k=0; k<NTRU_N; k++)
{
 r->coeffs[k] = 0;

for(i=1; i<NTRU_N-k; i++)
{
 r->coeffs[k]+=a->coeffs[k+i]*b->coeffs[NTRU_N-i];
}

for(i=0; i<k+1; i++)
{
 r->coeffs[k] += a->coeffs[k-i] * b->coeffs[i];
}

}
```

Listing 5: Poly RQ, original code

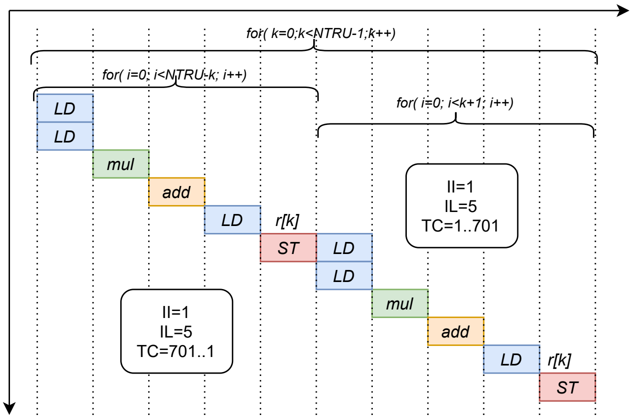

FIGURE 9: Poly RQ MUL - Resulting schedule

to reduce the BRAM utilization by almost 80% from the baseline implementation.

1) SABER

In a complex module like Saber there are multiple common functions used in the different stages. All SHA functions, for example, can be shared since they are used in different phases of the process. Furthermore, the polynomial multiplication can be shared in between the matrix-vector multiplication and the inner product, saving 67 DSPs, $\sim$15K FFs, and $\sim$7K LUTs. For Saber, after applying the area optimization steps, we obtain a total LUT reduction up to 77%.

2) Crystals Kyber/Dilithium

Kyber and Dilithium share most of their functions and logic so their resource reduction is both treated in this section. The most greedy function as defined in chapter V-A2 and V-A2 is Keccak. It is a sponge function that consists of absorption and a squeeze step. During both steps, it uses the same function KeccakF1600_StatePermute which proceeds to XOR and permute bits in multiple rounds. These operations are effectuated in 1’600 bits at each step so it takes a considerable amount of LUTs. To reduce this usage, the employment of directive allowed inline functions and made it possible for the HLS tool to share this problematic block. Furthermore, the reference implementation of Keccak was planned to effectuate two steps of permutation at once. Adapting the code to effectuate only one step reduced the usage block by 2 times. The total reduction of resources was about 4 times for Kyber and about 6 times for Keccak. The study of the memory allocation and their time of life also allowed to reuse the same arrays instead of declaring new ones and thus saving BRAMS. Finally, we detected a particular behavior that inferred a lot of unnecessary resources. Indeed, a simple instruction of copy as $z = y$; with z and y being large arrays generates an increase both in latency and resources due to registers which buffer all values of the first array and the bottleneck to read from it. Adapting the code to set values of z at the same time we set them into y to reduce the number

```
for(i=0; i<NTRU_N; i++)
{
    a0_int[i] = a1_int[i] = a->coeffs[i];
    b0_int[i] = b1_int[i] = b->coeffs[i];
}
for(i=0; i<NTRU_N; i++)
{
  for (k=0; k<NTRU_N; k++)
  {
    if ( i < NTRU_N-k )
    {
      r0_int[k] += a0_int[k+i] * b0_int[NTRU_N-i];
    }
    if ( i < k+1 )
    {
      r1_int[k] += a1_int[k-i] * b1_int[i];
    }
    r_int[k] = r0_int[k] + r1_int[k];
    }
}
for(i=0; i<NTRU_N; i++)
{
  r->coeffs[i] = r_int[1];
}
```

Listing 6: Poly RQ rewritten

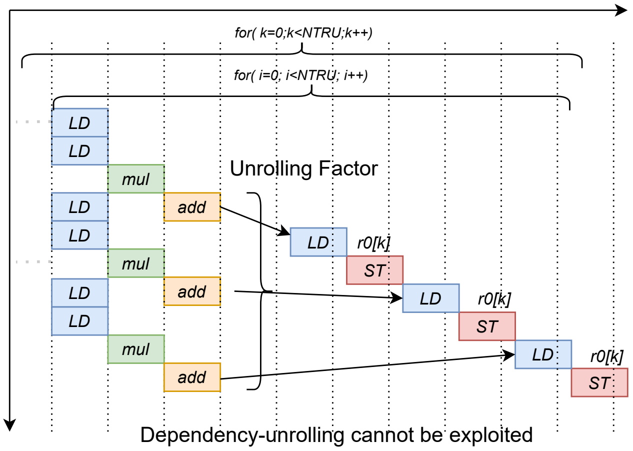

FIGURE 10: Poly RQ MUL - Schedule after pipeline optimizations.

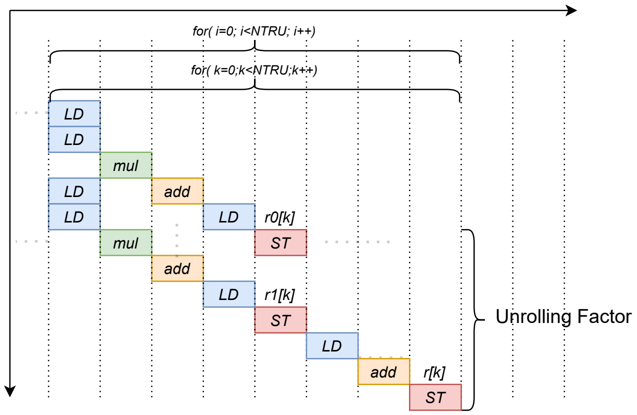

FIGURE 11: Poly RQ MUL - Final schedule after pipeline optimizations.

of Look-Up Tables (LUTs) by 20k.

3) NTRU

Among all the functions, NTRU presents very small resource usage compared to the other encryption schemes. Compared to the other lattice-based finalists, NTRU HRSS701 it is characterized by a small, regular loops with a large number of iteration. Hence, all the pipeline structures are very simple and efficient. Nevertheless, the majority of the area is concentrated in the hash functions, notably SHA-3. The optimization presented earlier for SHA-3 applies to NTRU as well.

VII. FINAL RESULTS

In this section we resume the final results after applying all the optimization steps presented earlier.

A. EXPERIMENTAL SETUP

To carry our experiments we used Vitis HLS 2020.2, targeting as FPGA a Xilinx Kintex Ultrascale+ xcku5pffvb676. The choice of the tool and target device is purely related to application needs. To obtain the minimum clock period and evaluate then the wall-clock time we rise up the frequency up to the maximum supported by the design.

B. DISCUSSING RESULTS AND BOTTLENECK

Table 5 resume the results obtained. For the limits imposed by the manuscript, we report the results of the final optimized versus the baseline. For Saber, Crystals, and Dilithium we are able to achieve a speedup of a factor between 4.8× and 7.3×, with a resource utilization reduced in some cases up to 80%, depending on the operation. For NTRU instead, the speedup is much higher, with the drawback of using more resources. However, comparing LAP we still achieve a more efficient solution. In fact, in front of 224× of speedup, we have almost 3× LUTs, 5× FFs, and 15× DSPs. Despite the different mathematical models used by the algorithms, polynomial multiplication represents the common computing-extensive module in all schemes. Hence, each of the algorithms presents different performance limitations and code modifications, as presented in section VI. Resource-usage side, the hash functions cover most of the resources, also common for all the algorithms analyzed in this paper. Figure 12 shows a wall-clock comparison between the baseline and the optimized version.

C. VERIFICATION

Although the C simulation passes all tests, there is non-proof that the RTL code will generate the same behavior as with directives we can introduce errors in the design by declaring wrong false dependencies for example. The C/RTL Co-Simulation (Cosim) works in the same way as C Simulation and uses the same files. The particularity is that it will first execute the C code on the host and produce a snapshot of input and output vectors of the top-level function. Then, it will re-execute the same test but instead of using the C code, it will simulate the RTL and compare the results with the previous one. It is important to be aware that the top level's array argument must have a defined size to the Cosim works even though that it is not necessary for synthesis and C Simulation. As a result of the Cosim, there is the success of the test and the average execution latency time which brings us a lot of information. Indeed, the synthesis tool could only give estimations but now there is real execution with the minimum, maximum, and average latency. There are some details to take into account during Cosim to have a successful test. The behavior of Vitis during this verification is to first, execute a complete C simulation and take a snapshot of inputs and outputs of the Device Under Test (DUT), then, execute again with the DUT being replaced by the RTL implementation and simulated. The RTL output is then compared to the previous snapshot and the remaining of the simulation is executed. The thing is that in this second step, inputs given to the DUT are taken from the snapshot made before and not from the actual simulation. This can lead to a failing test as, although the DUT outputs matches, the testbench fails. During the C simulation, 3 random numbers are generated among which the two first are given to the **keypair** function which produces a public and secret key. The public key and the third random number are given as input to the encapsulate function and the secret key to the decapsulate function. Both functions produce results which are then verified inside the testbench.

VIII. CONCLUSION

High-level synthesis is a powerful EDA technology for designing for FPGA in a short time. Sometimes, performance achievable is limited due to the original code. In this paper, we implement the lattice-based post-quantum cryptography algorithm with HLS, starting from the original code submitted for the standardization process. Since the limitations presented and the code optimizations are independent of the HLS tool used, the optimizations techniques can be applied to either commercial or open-source HLS tools, i.e. Intel HLS, Catapult, Synphony, Dynamatic, LegUp [20], or many more.

TABLE 5: Implementation comparison between baseline implementation and our optimized implementation for Saber, Crystals Kyber and Dilithium and NTRU

PQC	Design	Operation	Latency (cc)	Freq. Max (MHz)	WC-Time (μs)	Speedup ($\times$)	BRAMs # (+%)	DSPs # (+%)	FFs # (+%)	LUTs # (+%)
Saber	**Baseline**	Enc	169882	476	357	-	95	134	52689	157497
		Dec	205389	476	288	-	120	201	50887	137553
		Keygen	121403	476	255	-	43	67	18832	56263
	Optimized	Enc	35515	476	74	4.8	45(-52)	67(-50)	27775(-47)	35466(-77)
		Dec	37055	476	77	5.5	91(-24)	67(-66)	47757(-6)	58402(-57)
		Keygen	23008	476	48	5.2	34(-20)	67(-)	21805(15)	27252(-48)
Crystals Kyber	**Baseline**	Enc	299199	400	746	-	50	113	50183	63200
		Dec	395004	407	969	-	50	160	41167	50700
		Keygen	145225	312	464	-	29	67	17828	37242
	Optimized	Enc	48493	382	126	5.9	25(-50)	74(-34)	11184(-78)	11111(-82)
		Dec	59584	390	152	6.3	30(-40)	86(-46)	11879(-71)	11716(-76)
		Keygen	34620	413	84	5.5	21(-28)	22(-67)	8175(-54)	8635(-77)
Crystals Dilithium	**Baseline**	Sign	2136921	205	10428	-	68	88	49671	102209
		Verify	357929	205	1751	-	44	47	38120	79680
		Keygen	291940	208	1403	-	47	27	46017	97029
	Optimized	Sign	583107	410	1421	7.3	57(-16)	11(+26)	18419(-63)	15886(-84)
		Verify	141964	392	362	4.8	40(-9)	67(+42)	14980(-60)	13562(-83)
		Keygen	140765	500	281	5	33(-30)	38(+40)	11642(-75)	9761(-89)
NTRU	**Baseline**	Enc	6926955	478	14447	-	15	8	7457	23112
		Dec	20723246	476	43518	-	23	12	14114	44575
		Keygen	155845324	476	327275	-	18	7	3899	8371
	Optimized	Enc	30843	478	64	224	17(13)	134(1575)	48688(552)	90091(289)
		Dec	153642	476	322	134	35(52)	390(3150)	138998(884)	249019(458)
		Keygen	6478480	370	17492	19	19(5)	7(0)	18263(368)	38369(358)

A. BENEFITS OF USING HLS

Correctness. The synthesis process is carried by the tool; hence ensuring the functionality of the generated implementation. Nevertheless, after each of the optimization steps presented, we compiled the code with GCC and assessed functionality using the KAT (Known Answers Tests) available in the submission package.
Productivity Increased. The optimizations and code modifications shown in this paper are terribly faster compared to the hardware development in VHDL or Verilog.
Flexibility. When designing at RTL the target latency and area constraints should be usually defined at the early stages, constraining the architectural choices and reducing the flexibility of having different implementations available in a short time.
Are these the best results achievable? The final results presented in this paper represent a trade-off between performance and area. The aim of this paper is to extract the bottleneck and overcome limitations in using HLS. Nevertheless, there is still room for further performance improvements or area reduction if the final target design must be optimized for maximum performance or minimum area.

B. FINAL REMARKS

The main goal of this paper is not to overcome the performance results of already existing PQC designs. Rather, to demonstrate the opportunity of using recent design automation techniques for this new class of applications. As shown in the motivation and the review of related work, none of the techniques applied in this paper are individually original nor independently addressed in other contexts. Yet, from the best of our knowledge, this paper is the first attempt on trying to ameliorate results achievable for post-quantum cryptography algorithms. Furthermore, we believe that this paper collects and presents a concrete method that combines pragmatically and effectively the optimization techniques applied to real use cases. We plan to publish the modified optimized code for HLS on Github to make it available for the community. We believe that this is a great advantage for making hardware designers cognizant when designing lattice-based PQC for FPGAs, to profit from emerging design tools and reconfigurable platforms with negligible effort and in a short time.

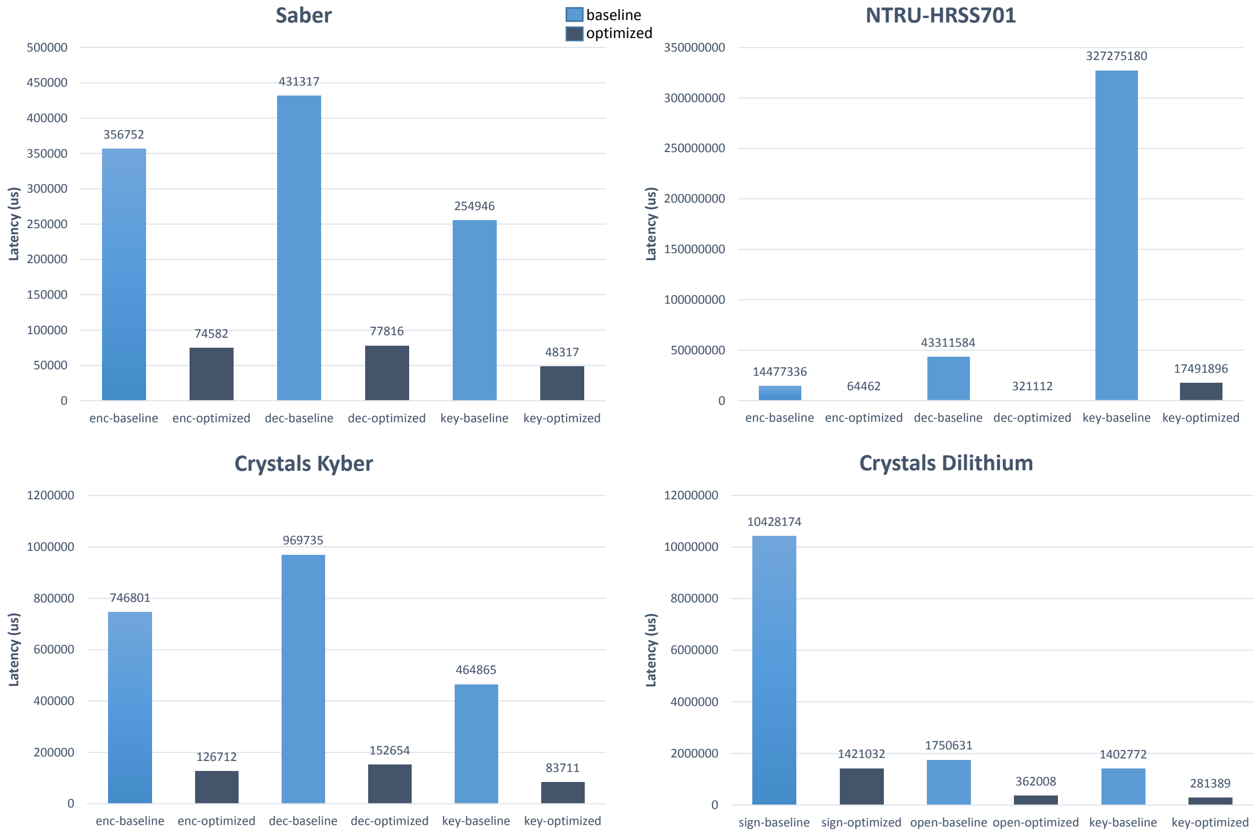

FIGURE 12: Wall-clock time comparison

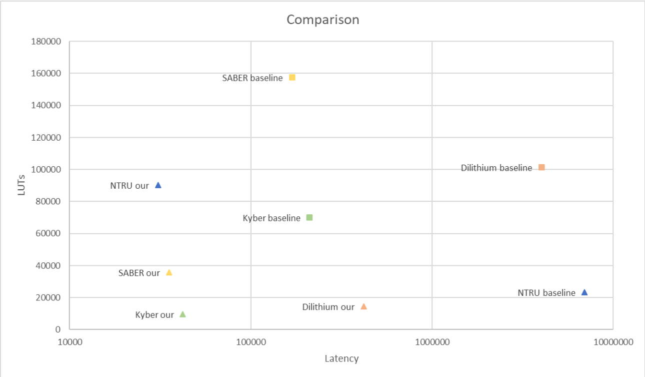

FIGURE 13: Latency vs Area Design Points

ACKNOWLEDGMENTS

This research has been carried out as part of the project "Dynamically Reconfigurable CryptoEngine" supported by Innosuisse under Grant No. 32291.1 IP-ICT, a joint collaboration with Cysec S.A. The authors thank also Alexandre Duc for his help in selecting the algorithms analyzed in this paper.

REFERENCES

[1] https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

[2] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, Frederik Vercauteren, "SABER: Mod-LWR based KEM (Round 3 Submission)"

[3] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehle, "Crystal Kyber, Algorithm Specifications And Supporting Documentation"

[4] Vadim Lyubashevsky, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehle, Shi Bai, "Crystal Dilithium, Algorithm Specifications And Supporting Documentation"

[5] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang "NTRU, Algorithm specifcations and supporting documentation".

[6] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. In Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings,volume 1666 of Lecture Notes in Computer Science, pages 537–554. Springer, 1999

[7] D. Soni, K. Basu, M. Nabeel, and R. Karri, "A Hardware Evaluation Study of NIST Post-Quantum Cryptographic Signature schemes

[8] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 5, pp. 898-911, May 2019, doi: 10.1109/TCAD.2018.2834439

[9] A. Ferozpuri, F. Farahmand, V. Dang, M. U. Sharif,J.P. Kaps, and Kris Gaj, "Hardware API for Post-Quantum Public Key Cryptosystems".

[10] V B. Dang, F. Farahmand, M. Andrzejczak, and Kris Gaj, "Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign"2019 International Conference on Field-Programmable Technology (ICFPT)

[11] F. Farahmand, D. T. Nguyen, V B. Dang, M. Andrzejczak, and Kris Gaj, "Software/Hardware Codesign of the Post Quantum Cryptography Algorithm NTRUEncrypt Using High-Level Synthesis and Register-Transfer Level Design Methodologies " 2019 29th International Conference on Field Programmable Logic and Applications (FPL)

[12] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, Torsten Hoefler, “Transformations of High-Level Synthesis Codes for High-Performance Computing“
[13] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, “Parallel Programming for FPGAs, The HLS Book“
[14] Steve Dai, Gai Liu, Ritchie Zhao, Zhiru Zhang, Enabling Adaptive Loop Pipelining in High-Level Synthesis, Asilomar 2017
[15] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Dynamatic: From C/C++ to Dynamically-Scheduled Circuits. Invited tutorial. In Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Seaside, Calif., February 2020
[16] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits. IEEE Circuits and Systems Magazine, Special Issue FPGA Evolution, Volume 21, Number 2 Second Quarter 2021
[17] Junyi Liu, John Wickerson, George A. Constantinides, “Loop Splitting for Efficient Pipelining in High-Level Synthesis“2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)
[18] Sujoy Sinha Roy and Andrea Basso, “High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware.“ In Transactions in Cryptographic Hardware and Embedded Systems 2020.
[19] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy and Ingrid Verbauwhede, “Compact domain-specific coprocessor for accelerating module lattice-based key encapsulation mechanism“ In 57th Design and Automation Conference DAC 2020.
[20] Canis Andrew, Choi Jongsok, Aldham Mark, Zhang Victor,Kammoona Ahmed, Czajkowski Tomasz, Brown Stephen, Anderson Jason, LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems, ACM Transactions on Embedded Computing Systems (TECS).
[21] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, Fabrizio Ferrandi, Inter-procedural resource sharing in High Level Synthesis through function proxies, 2015 25th International Conference on Field Programmable Logic and Applications (FPL).
[22] Lan Huang, Da-Lin Li, Kang-Ping Wang, Teng Gao1, and Adriano Tavares, A Survey on Performance Optimization of High-Level Synthesis Tools. Journal of Computer Science and Technology, May 2020.
[23] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. IEEE Trans. Comput.-Aided Design Integr. Circ. Syst. 30, 4 (2011), 473–491
[24] P. Coussy, D. D. Gajski, M. Meredith and A. Takach, "An Introduction to High-Level Synthesis," in IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8-17, July-Aug. 2009, doi: 10.1109/MDT.2009.69.

ANDREA GUERRIERI (S’15, M’17, SM’21) started working on embedded systems for industry in 2006. He graduated in Electronic Engineering from Politecnico di Torino, Italy, in 2015, becoming a Principal Engineer responsible for the development of flagship products. In 2017, he joined the Processor Architecture Laboratory at EPFL, Switzerland, where he leads and participates in research projects in collaboration with industry. Recent projects involve high-level synthesis, reconfigurable SoCs exploiting dynamic partial reconfiguration of FPGAs for future space missions, and exoplanet observation. He is also a co-developer of Dynamatic and a recipient of the Best Paper Award at FPGA’20, held in Seaside, California. He is a Senior Member of the IEEE, he published multiple articles in peer-reviewed journals and international conferences, and he is also co-author of the book Fundamentals of SoC Design in collaboration with Arm.

GABRIEL DA SILVA MARQUES started working at the University of Applied Sciences and Arts Western Switzerland of Geneva in 2018 where he studied and earned his BSc and MSc in mobile and embedded systems. His work as research assistant went through multiple and diverse projects as implementing a Human Machine Interface for an industrial controller with LabVIEW to designing PCBs for connected watches. It’s actual work includes exploring HLS solutions to implement Post Quantum Algorithms on FPGAs.

FRANCESCO REGAZZONI is assistant professor at University of Amsterdam and group leader at Università della Svizzera italiana. He received his Master of Science degree from Politecnico di Milano and his PhD degree from Università della Svizzera italiana. He held research positions at the Université Catholique de Louvain and at Technical University of Delft, and has been visiting researcher at several institutions, including NEC Labs America, Ruhr University of Bochum, and EPFL Lausanne. His research interests are in the field of secure IoT devices and embedded systems, covering in particular design automation for security, physical attacks and countermeasures, post-quantum cryptography, and efficient implementation of cryptographic primitives.

ANDRES UPEGUI is Associate Professor at the University of Applied Sciences of Western Switzerland - Geneva (HES-SO, Hepia) since 2010. He obtained a diploma on Electronic Engineering in 2000 from the UPB (Medellín, Colombia), he joined the UPB microelectronics research group from 2000 to 2001. from 2001 to 2002 he did the Graduate School on Computer Science at the EPFL, and then he joined the Logic Systems Laboratory (LSL) as PhD student. In 2006, he received the title of PhD. from the EPFL for his thesis entitled "Dynamically reconfigurable bio-inspired hardware". Afterwards he worked as senior researcher and lecturer at the REDS institute (HEIG-VD, Yverdon) and the CoRES group (Hepia, Geneva). His main research interest include self-adapting hardware, FPGA dynamic partial reconfiguration, evolutionary computation, and diverse digital hardware architectures including neural networks, optimization algorithms, evolvable systems and cryptocores.