Invited Paper: Dilithium Hardware-Accelerated Application using OpenCL-based High-Level Synthesis

Islam Alexander El-Kady
University of Patras and
Industrial Systems Institute/R.C. ATHENA
Patras, Greece

Apostolos P. Fournaris
Industrial Systems Institute/R.C. ATHENA
Patras, Greece

Vassilis Paliouras
University of Patras
Patras, Greece

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323770

Abstract—**Post-quantum cryptography (PQC) has been gaining attention in the last few years due to the evolution of quantum computers and the need to replace traditional, quantum-attack-insecure cryptography schemes with quantum-attack-resistant schemes. Lattice-based cryptography (LBC) constitutes a highly promising post-quantum solution (Quantum Resistant), but implementations in software or hardware are challenging due to the use of operations based on large-size polynomials. LBC selected schemes for standardization by the National Institute of Standards and Technology (NIST), rely on matrix-to-matrix multiplications of high-order polynomials, having performance bottlenecks that are solved using the Number-Theoretic Transform (NTT). One of the NIST-selected schemes for digital signature (DS) is the CRYSTALS-Dilithium scheme, which uses $N = 256$ degee polynomials. In this paper a Hardware/Software (HW/SW) co-design solution is proposed for all security levels of Dilithium, utilizing the OpenCL framework and Vitis High-Level Synthesis (HLS) tool. In our work, the HW/SW co-design OpenCL mechanisms are analyzed extensively and communication overheads between the hardware kernel and an ARM processor are identified, while appropriate techniques are proposed in order to bypass the I/O time-bottleneck on a real-world application. The proposed implementation runs on the ARM Processing System (PS) of a Xilinx Multi-Processor System on Chip (MPSoC) system, utilizing the MPSoC FPGA Programmable Logic (PL) in order to accelerate the calculations relative to the heavy matrix-multiplication operation. Finally, the proposed HW/SW codesigned solution is realized as a real-world Linux-based Dilithium DS executable and manages to achieve realistic performance gain, in terms of time execution, versus a CPU-only execution ranging from 2–23% (depending on the utilized CPU Clock Frequency).**

Index Terms—**Post-quantum cryptography, High-Level Synthesis tool, Number-Theoretic Transform, Hardware-Accelerated Application, OpenCL framework**

I. Introduction

Classical public-key cryptography algorithms rely on mathematical problems that are believed to be hard-to-solve using classical computers; however, some of these problems can be solved efficiently using quantum computers. Cryptographic schemes such as RSA, Diffie-Hellman and elliptic curve cryptography (ECC), will become vulnerable to attacks once large-scale quantum computers become a reality [1]. Post-quantum cryptography offers a solution to this problem by providing cryptography algorithms that are designed to be secure against attacks from both classical and quantum computers. These algorithms are typically based on mathematical problems that are believed to be hard even for quantum computers to solve, such as the learning with errors (LWE) problem or the hash-based Merkle tree signature scheme [2].

After careful consideration during the third round of the NIST PQC Standardization Process, NIST has announced four finalist algorithms suitable for standardization. NIST will recommend two primary algorithms to be implemented for most use cases, which are CRYSTALS-KYBER [3] for key-establishment and CRYSTALS-Dilithium [4] for digital sign. In addition, the DS schemes FALCON [5] and SPHINCS [6] will also be standardized. Three out of those four algorithms rely on Lattice-Based Cryptography (LBC), which seems to dominate the PQC research domain due to its cryptoanalysis complexity against quantum-computer attacks. However, the CRYSTALS PQC cryptosuite as well as, partially, FALCON rely on polynomial rings and use matrix-to-matrix and matrix-to-vector polynomial operations that, when the degree of polynomials is high (eg. 256 for Dilithium, 512 or 1024 for FALCON), impose a substantial overhead on the performance of those PQC schemes.

Early works like [7] and [8] tackle the above performance problem by focusing on designing small, sequential FPGA-based NTT multipliers using combination of registers, RAMs (or BRAMs in FPGAs) and ROMs to minimize the overall resource cost and achieve a multiplication speedup compared to software implementations. In other early NTT multiplier approaches, instead of precomputing and storing in ROM the NTT/INTT twiddle factors, additional run-time multiplications have been proposed to compute the required twiddle factor each time [9]. As expected, the utilized resources (in memory usage and FPGA slices number) of on-the-fly computations are potentially smaller than precomputation approaches, but at an extra time delay cost due to the new runtime operations. As new works gradually emerge, significant focus for NTT multipliers has been placed on optimizing the NTT and INTT operation itself during multiplication [10] [11]. This has even been extended to the use of high-radix (e.g., Radix-4) NTT

[12] [13] to reduce the complexity of NTT. In various studies, High Level Synthesis (HLS) tools have been employed to reduce the design time for NTT multipliers and enhance design flexibility. These tools allow for the extraction of a hardware design from properly formulated software code. Such an approach is followed by Millar [14], where HLS directives such as PIPELINE, dependence, LOOP_FLATTEN, UNROLL and ARRAY_PARTITION have been used for NTT/INTT or by Kawamura *et al.* [15], where loop flattening, trip count reduction and loop expansion assisted by HLS directives, are proposed to optimize the synthesized code.

Eventually, however, NTT multipliers must be considered under a given PQC application, i.e., a full PQC scheme. While there exist several purely hardware PQC Key Encapsulation Mechanism (KEM) or DS designs derived using HLS tools [16] or traditional hardware design approaches [17] [18], such solutions lack the flexibility that a software implementation can offer. A compromise on flexibility versus speed can be found by hardware-software co-design solutions that fit ideally to latest System-on-Chip (SoC) FPGA approaches. Typical hardware/software co-design approaches identify the bottleneck operations on a given software design and transform such operations into hardware accelerators that communicate through the processor's bus to the executed software. In recent years, PQC hardware / software solutions tend to dominate the relevant research literature [19] [20]. While such works provide competitive results in terms of overall performance (in execution time) with relatively constrained resources (in chip covered area, number of Look-Up Tables (LUTs) in FPGAs, Flip-Flops, etc.) they do not take into account the aspects of a Hardware/Software codesign that have an impact and can reduce the performance, such as the data communication cost between hardware and software, or the fact that a PQC scheme in a realistic scenario is executed in parallel with other applications, for example, in an embedded Linux OS.

In this paper, an HLS-based Dilithium NTT polynomial matrix multiplier is designed, and a full hardware/software FPGA-based System-on-Chip (SoC) co-design for a Dilithium DS implementation is proposed under a real-world scenario that takes into account all communication overheads between the FPGA Programmable-Logic (PL) and CPU Processing System (PS) as well as any overhead introduced by the SoC operating system. Specifically, an analysis is conducted to determine the performance overhead in terms of execution time and resources that is caused by the hardware/software intercommunication in Dilithium DS applications (key-generation, verify, sign), where the Dilithium NTT multiplication is performed by a faster than software hardware accelerator that, when operating as stand-alone IP Core, achieves an execution time gain of 34% to 46% (depending on the Dilithium security mode) compared to software NTT multiplication. However, the above analysis shows that this gain is significantly reduced when all performance costs are considered, especially when compared with a high-frequency CPU. By proposing an adaptation of the OpenCL framework used, for the software to hardware interfacing as well as the proper use of AXI bus interface (using DMA and BRAMs) we manage to reduce the intercommunication performance overhead and achieve execution time gains compared to pure software solutions. This speed performance gain of the entire hardware/software Dilithium application can reach 10–20% in the highest security mode of Dilithium, depending on the CPU frequency.

The remainder of the paper is organized as follows. In Section II we provide the necessary background to our research work. In Section III the proposed hardware kernel of the NTT multiplier is presented and analyzed. In Section IV the full Dilithium DS hardware/software co-design scheme is presented, and in Section V the results of the proposed work are described. Finally, Section VI concludes the paper.

II. CRYSTALS-Dilithium Signature Scheme

The CRYSTALS Dilithium scheme is a post-quantum secure DS algorithm based on the hardness of the LWE problem, relying on LBC. The security level of the Dilithium signature scheme is determined by the number of mathematical operations required to generate a signature. The scheme offers three different security levels, as dictated by the NIST requirements, known as Dilithium 2, 3, and 5. Each level provides increasing levels of security, with Dilithium 5 providing the highest level of security.

A. Basic operations

Note that each element of the multiplied vectors in Dilithium DS is a polynomial on the polynomial ring $\mathbb{R}_q = \mathbb{Z}_q[X]/(X^N+1)$, where $q = 2^{23} - 2^{13} + 1 = 8380417$ (23-bit integer values) and $N = 256$. The DS scheme consists of three basic operations [4], specifically

1) Key Generation: The first step in the Dilithium signature scheme is to generate a public-private key pair. This is done by sampling two secret key vectors and applying a vector multiplication procedure with a sampled matrix.
2) Signature Generation: To sign a message, the signer uses their secret key to generate a signature that is unique to that message. This process can be executed multiple times on a loop until a series of security conditions are met. In this process, the main operations are hashing and vector multiplications.
3) Signature Verification: To verify a signature, the verifier uses the signer's public key and the message to check that the signature is valid. This involves a series of vector multiplications and hashing functions in order to verify the challenged message.

Profiling the DS scheme on security level V, which is the most demanding security level in terms of operations, we can see the most time-consuming parts of the algorithm in Fig. 1. Column "Others" refers to components such as Sampler, Expansion, etc. Despite Keccak hashing function (as part of SHAKE256 used in Dilithium) being the most executed part of the algorithm with 32%, the execution time of the NTT matrix-multiplication, which is roughly the sum of INTT, NTT, Pointwise-Multiplication (PWM) and PW-Addition (PWA) times, is approximately 54%, a percentage

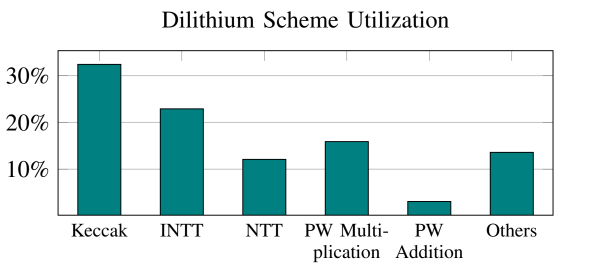

Fig. 1. Profiling of Dilithium security-level 5 obtained using Vitis-profiler

that is much larger than the Keccak utilization. Therefore our effort is focused on the optimization of this part of the algorithm as it takes place on all operations of the DS scheme, Key-Generation, Sign and Verify.

B. NTT preliminaries

NTT (Number Theoretic Transform) multiplication algorithm is a fast multiplication technique for efficient polynomial and integer multiplication. It is closely related to the Fast Fourier Transform (FFT) algorithm, which is used to efficiently compute the Discrete Fourier Transform (DFT).

The NTT multiplication algorithm is based on the principle of representing a polynomial as a sequence of integers and performing a Point-Wise Multiplication (PWM) in the NTT domain instead of the classic multiplication, thus reducing the complexity to $O(n\log(n))$ instead of the traditional polynomial multiplication algorithm that has a complexity of $O(n^2)$. Dilithium's software implementation takes advantage of this property.

Let f be a polynomial of degree n, where $f = \sum_{i=0}^{n-1} f_i X^i$ and $f_i \in \mathbb{Z}_q$, q is a prime number and ω is the primitive n-th root of unity or otherwise called twiddle factor, which must satisfy the conditions $\omega^n \equiv 1 (\text{mod } q)$ and $\omega^i \neq 1 \pmod q$, $\forall i < n$, where $q \equiv 1 \pmod n$. Under these conditions, the forward NTT is then defined as

$$\hat{f}_i = \text{NTT}(f) = \sum_{j=0}^{n-1} f_j \omega_n^{ij} \bmod q. \tag{1}$$

Respectively, the INTT operation is defined as:

$$f_i = \text{INTT}(\hat{f}) = n^{-1} \sum_{j=0}^{n-1} f_j \omega_n^{-ij} \bmod q. \tag{2}$$

C. Montgomery Reduction

When multiplication using NTT is performed, the Montgomery reduction can be used to reduce the cost of the modular reductions needed during the computation. The basic idea is to first transform the operands into the Montgomery form, perform the NTT multiplication, and then apply the Montgomery reduction to obtain the final result. To transform an integer a into the Montgomery form with respect to a modulus m, we compute $a' = (a \cdot R) \bmod m$, where R is a constant chosen such that $R \bmod m = -1$. The value R is typically chosen as a power of 2, which makes the multiplication by R more efficient.

To apply Montgomery reduction to the product of two integers a and b in Montgomery form, we compute $c = (a \cdot b \cdot R^{-1}) \bmod m$, where R^{-1} is the inverse of R modulo m. The value c is also in Montgomery form, so to obtain the final result, we need to convert it back to the standard representation by computing $c' = (c \cdot R^{-1}) \bmod m$.

III. Hardware Kernel

In the context of FPGA-based system development, a *kernel* is a term used to describe a hardware unit that can be loaded onto an FPGA to perform a specific task. In our case, a kernel is designed, related to the NTT matrix-multiplication calculations with parameters based on Dilithium's DS implementation. Subsequently, this kernel is used to provide a hardware-accelerated function to the software, and, combined, they create the final DS application.

A. Dilithium Matrix-Multiplier

In the Dilithium software application, the code section relative to the NTT matrix-multiplication to be accelerated contains four function calls, which can be seen in Algorithm 1. The code includes a) L executions of NTT, b) a matrix-pointwise multiplication between a $K \times L \times N$ matrix and NTT output vector $L \times N$ (including Montgomery reductions to avoid multiplication overflows), c) a Montgomery reduction of the accumulated multiplication result, and d) K executions of INTT (including Montgomery reduction on the output vector). These lines of code are executed in all three basic DS operations, therefore, we create a kernel that receives as input an L-poly-vector, a K-matrix of L-poly-vectors (already in the NTT domain) and outputs their product on a K-poly-vector, where a poly-vector is defined as a vector of N coefficients (representing a polynomial in $[R]_q$).

Algorithm 1 Dilithium matrix-multiplication code

```
polyvecl_ntt(&$z$);
polyvec_matrix_pointwise_montgomery(&$w1$, $mat$, &$z$);
polyveck_reduce(&$w1$);
polyveck_invntt_tomont(&$w1$);
```

The matrix-multiplication code is rearranged in two sections for the HLS compiler, the NTT section (function polyvecl_ntt of Algorithm 1) and the PWM_INTT section, which includes all other computations (the remaining three functions of Algorithm 1). This type of hierarchy helps the HLS compiler to minimize the memory needed by INTT and Montgomery reduction. On our top-hierarchy function, the dataflow Pragma is applied, which enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput rate of the design. The modified algorithm is presented as Algorithm 2.

B. NTT

Initially, the NTT operation is applied L times on the input L-poly-vector, for each poly-vector. NTT and INTT take as

TABLE I
HLS PRAGMAS FOR DILITHIUM MULTIPLICATION

Variables	HLS Pragma
-	dataflow
in, in'	array_partition factor $= L$
mat, out	array_partition factor $= K$
in, out	interface m_axi bundle = gmem0
mat	interface m_axi bundle = gmem1

Algorithm 2 Dilithium multiplication HLS-function

```
    Input $L \times N$ polynomial $in(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
    Input $K \times L \times N$ polynomial $mat(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
    Output $K \times N$ polynomial $out(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
    #pragma HLS dataflow
    NTT:
    for ($i = 0; i < L; i++$) do
        $in'_i \leftarrow$ NTT($in_i$)
 5: end for
    PWM_INTT:
    for ($i = 0; i < K; i++$) do
        $c \leftarrow Pointwise\_acc(mat_i, in')$
        $out_i \leftarrow$ INTT($c$)
10: end for
    return $out(x)$
```

input an array of N coefficients (a poly-vector) and output an array of N coefficients on the opposite domain (NTT and regular arithmetic domain, respectively). The particular modules have been implemented for a Radix-2 architecture and have been analyzed and optimized for Dilithium DS parameters [11] [21], using the Vitis-HLS tool. To pipeline the design and allow for full exploitation of the HLS compiler optimizations, we introduce BRAMs on variables between the intercommunication sections of NTT, PWM_INTT and the I/O, as can be seen in Table I. Therefore, the variables in and in' are on BRAMs split in chunks, equal to their polyvector dimension L. The final architecture of the hardware kernel is shown in Fig. 2.

C. Point-wise Multiplication, Addition and INTT

The matrix pointwise multiplication can be grouped in K loops of multiplication, accumulation, Montgomery reduction (Pointwise_acc function), and INTT, as shown in lines 7–10 of Algorithm 2. Each call of Pointwise_acc function includes a pointwise multiplication of L-poly-vectors, Montgomery reduction, and accumulation of the L results on a temporary array of length N, lines 2–6 of Algorithm 3. This minimizes the memory needed, as the inputs are already on BRAMs and allows the design to be pipelined and unrolled to exploit

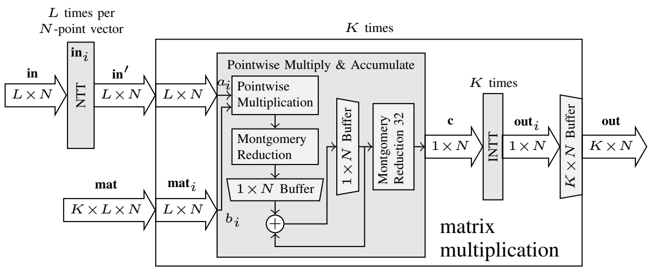

Fig. 2. NTT matrix-multiplication architecture

Algorithm 3 Pointwise_acc HLS-function

```
    Input $L \times N$ polynomial $a(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
    Input $L \times N$ polynomial $b(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
    Output $1 \times N$ polynomial $c(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$
    $acc_i \leftarrow 0,\ i = 1, \dots, N$
    for ($i = 0; i < L; i++$) do
        for ($j = 0; j < N; j++$) do
            $tmp_j \leftarrow a_{i,j} \cdot b_{i,j}$ mod $q$
 5:         $acc_j \leftarrow acc_j + tmp_j$
        end for
    end for
    32-Montgomery reduce:
    for ($j = 0; j < N; j++$) do
10:     $c_j \leftarrow acc_j$ mod $q$
    end for
    return $c(x)$
```

trade-offs between resources and execution time. The result of Pointwise_acc is the input of INTT [11] [21] to return to the time domain.

IV. HW/SW CODESIGN ARCHITECTURE

The derived hardware kernel is subsequently packed as a Vitis kernel that can be integrated into an application using the Vitis development environment. The application can then be compiled and executed on a host CPU, with the Vitis runtime library managing the offloading of kernel computations to the FPGA device.

Vitis compiler creates a binary container file (.xclbin) for our kernel, which stores the configuration bitstream of the desired design, including the PL relative to the design and the communication with the PS. This file is used by the OpenCL Framework in order to download the particular design on the specified device.

In order to create a full Dilithium DS application, we use the Vitis Unified Software Platform tool, which compiles and links the HLS codes to a Vitis Xilinx platform and produces a final executable for the HW/SW co-design. A Vitis hardware platform is a pre-configured design for a specific FPGA-device, defining the connections between the PL and the PS, including various device peripherals and AXI connections in order to deploy Vitis kernels, working seamlessly with the V++ compiler and linker.

A. OpenCL Framework

To provide generality and flexibility of the proposed application solution, an OpenCL framework is used to manage the communication between the hardware (openCL wrapped) kernel and its Dilithium software host. However, using the OpenCL framework in the Dilithium DS hardware-accelerated application introduces significant delays. To address this issue, several adjustments of the OpenCL framework usage are proposed.

In general, running an OpenCL kernel by a host program requires the following steps:

1) Choose a device: Use the OpenCL API to detect and select the device that will execute the kernel.
2) Create a context: Use the OpenCL API to create a context based on the chosen device, which is a container for all OpenCL objects such as devices, command queues, and memory buffers.
3) Create a command queue: Use the OpenCL API to create a command queue, which is a list of commands to be executed on the selected device.
4) Create a Program: Use the OpenCL API to create a program, based on the binary container (.xclbin file), the device, and the context.
5) Create the kernel: Use the OpenCL API to create a kernel object from the kernel source, based on program created and the name of the kernel to execute.
6) Create the I/O buffers: Use the OpenCL API to create the buffers that will be used for communication between the CPU and the kernel.
7) Set kernel arguments: Use the OpenCL API to set the arguments for the kernel. These arguments can be buffers, scalar values, or arrays.
8) Enqueue the kernel: Use the OpenCL API to enqueue the kernel for execution in the command queue.
9) Retrieve the results: After the kernel has finished executing, use the OpenCL API to retrieve the results from the device's memory buffer into the host memory buffer.

Fig. 3 shows the time allocation of these steps on Dilithium's matrix multiplication, with parameters of security mode 5 ($K = 8$ and $L = 7$), running the OpenCL framework. It can be observed in this figure that OpenCL steps regarding the buffer allocation are the most time consuming ones, so such operations need to be minimized within a host program. After various tests and measurements, it was observed that all steps till step 7 can be run once for each kernel in an application (in the current case once as we have one kernel setup), keeping the variables and objects that were initialized in memory (using persistent variables). This conclusion leads to a dramatic reduction in the execution time of the framework as the most time-consuming part of the execution comprises steps 6 and 7 (map_buffer and buffers_allocation), as shown in Fig. 3.

Another possible time-consuming part of the proposed approach is the copy of input vectors from variables to the memory-buffer objects of OpenCL, which use memory-aligned mapping relative to the buffer-size of the device. This can be bypassed by using persistent pointers on the input variables that are already aligned to the device's buffer size.

B. Application architecture

The Vitis hardware platform provides a set of AXI interfaces that can be used to communicate between the kernel and the host CPU. Such interfaces are the AXI-Lite, used for scalar values and control signals, the AXI-Stream that transfers data in a sequential streaming manner (used for communication between kernels), and the AXI Memory-Mapped interface which allows kernels to read and write data in global memory.

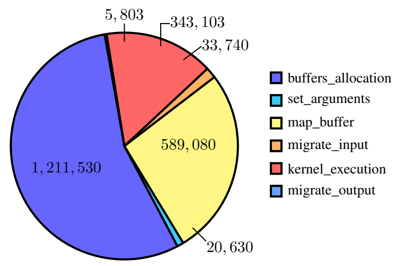

Fig. 3. OpenCL-framework time allocation break-down (in ns) for Matrix-multiplication kernel in Mode 5 setup

TABLE II
DILITHIUM MULTIPLIER BARE-METAL RESOURCES

Alg.	Lat. (cc)	LUT	FF	BRAM	DSP
Mode 2	18 775	6336	6400	26	41
Mode 3	27 977	7084	6555	48	41
Mode 5	41 933	8146	6747	62	41

These can be specified in the HLS source code by applying the corresponding HLS Pragmas. For the data transfer between the Dilithium software code (the kernel host) and the proposed Dilithium NTT multiplier hardware kernel, we employ the AXI Memory-Mapped Interface (using the interface HLS-pragma as shown on Table I), in order to utilize the Direct-Memory Access (DMA) of the processor through the operating system, copying the I/O data to the DDR memory of the platform, achieving the highest possible throughput rate.

V. RESULTS AND EVALUATION

The measurement approach that has been followed in the paper consists of two steps. Firstly, we analyze the kernel on a kernel level only testbench executing Dilithium's matrix-multiplication operations and then, secondly, we deploy a fully Dilithium DS application in a real embedded linux OS environment, utilizing the OpenCL (OCL) framework with the matrix-multiplication kernel. We compare each DS operation for the various Dilithium security levels with a purely software version of the application, i.e., single-thread CPU execution on the same device. For the evaluation of the proposed solution, we use a Xilinx ZynqZCU-104 MPSoC, running an ARM Cortex A-53 CPU, utilizing the FPGA running on a 150 MHz clock AXI bus using bare metal or Petalinux OS (with XRT support).

A. Hardware-Kernel results

To evaluate the overall performance of the proposed hardware-kernel individually, measurements of the execution time and utilized hardware resources have been collected when the only the kernel is deployed on a Xilinx ZynqZcu-104 device without utilizing the OpenCL framework as well as when the kernel is deployed on the same device with the OpenCL framework. The collected results are compared with the measurements of a CPU-only Dilithium NTT multiplier and are presented in Fig. 4 for time delay measurements and in Table II for utilized resources measurements.

TABLE III
DILITHIUM DS-APPLICATION CODESIGN *vs.* ARM-CPU EXECUTION TIME, FOR ALL SECURITY-LEVELS AND DIFFERENT CPU CLOCK FREQUENCY TIMES

Mode	CPU clock freq.: 1.2 GHz			CPU clock freq.: 0.6 GHz			CPU clock freq.: 0.4 GHz			CPU clock freq.: 0.3 GHz		
	PS (µs)	PS/PL (µs)	Gain (%)	PS (µs)	PS/PL (µs)	Gain (%)	PS (µs)	PS/PL (µs)	Gain (%)	PS (µs)	PS/PL (µs)	Gain (%)
Key-Gen 2	701	692	1.35	1408	1287	8.62	2115	1866	11.77	2824	2456	13.01
Key-Gen 3	1189	1116	6.10	2383	2031	14.77	3580	2955	17.45	4781	3891	18.62
Key-Gen 5	1936	1742	10.02	3880	3198	17.57	5825	4660	19.99	7779	6132	21.17
Sign 2	3481	3376	3.02	6974	6402	8.20	10 438	9356	10.36	13 932	12 290	11.79
Sign 3	5697	5320	6.60	11 461	9610	16.15	17 185	13 888	19.19	22 938	18 288	20.27
Sign 5	6930	6132	11.51	13 791	11 151	19.14	20 669	16 118	22.02	27 614	21 217	23.16
Verify 2	807	797	1.24	1620	1481	8.52	2432	2148	11.68	3248	2828	12.92
Verify 3	1290	1218	5.62	2586	2215	14.33	3886	3224	17.00	5189	4245	18.20
Verify 5	2097	1906	9.09	4203	3500	16.72	6310	5100	19.16	8426	6711	20.35

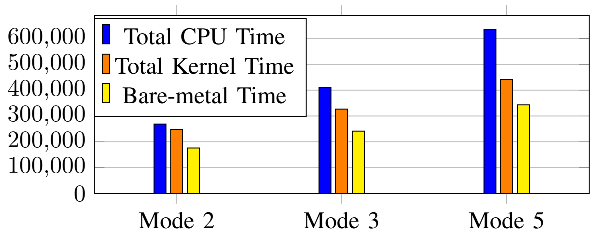

Fig. 4. Kernel *vs.* CPU times (ns) for the Dilithium multiplier

Table II reveals that for the all three Dilithium security levels the execution time of the design when using the OpenCL framework (OCL) is faster than the software Dilithium NTT multiplier (CPU version), the OCL hardware kernel versus software CPU version gain gets larger the higher the security-level gets, from 8% at level 2 to 30% at level 5. The same conclusions can be drawn for the non-OpenCL including implementation (HW*), but with higher gains relative to OpenCL (OCL) design, gains from 34% at level 2, 41% at level 3 and 46% at level 5. Fig. 4 shows more clearly the time delay overhead of the OCL and the AXI interfaces needed for the I/O communication between the kernel and the DDR memory. In a real-world application we need to include these overheads to properly evaluate a HW/SW codesign performance, concluding that for all security levels, it is possible to achieve a considerable gain in a HW/SW codesigned Dilithium DS, especially on Mode 3 and 5, compared to software designs.

B. Application results

Introducing the OCL framework and the proposed Dilithium NTT multiplier kernel within the overall Dilithium DS-application for all security levels and deploying such application on a Xilinx Zynq Zcu-104 device operating under a Petalinux OS, we are able to evaluate the proposed solution in a realistic embedded system setting. In this experimental setup we measure the execution time for each DS operation, i.e., Key generation, Signature, and Verification and the results are presented in Table III, for all security levels (columns PS/PL). We compare to the execution time of these operations observed when they are implemented on software on a single CPU core (columns PS) with clock frequencies ranging from 1.2 GHz to 300 MHz. In all scenarios, the hardware module has a clock of 150 MHz, utilizing AXI buses with a clock of 100 MHz.

TABLE IV
AREA CONSUMPTION AND PERFORMANCE OF DILITHIUM NTT MATRIX-MULTIPLIER MODULES

Submodule	LUT	FF	BRAMa	DSP	Lat. (cc)
NTT Radix-2	1344	766	0	12	1031
NTT Radix-4 [17]	444	421	32K	17	533
Matrix-Mul Mode-2	804	898	128K	15	1037
Matrix-Mul Mode-2 [17]	2129	59	0	0	2370
Matrix-Mul Mode-3	805	901	128K	15	1293
Matrix-Mul Mode-3 [17]	2774	49	0	0	3019
Matrix-Mul Mode-5	806	945	128K	15	1805
Matrix-Mul Mode-5 [17]	4591	46	0	0	4434
INTT Radix-2	1934	1082	16K	12	1096
INTT Radix-4 [17]	444	421	32K	17	536

Given that the DS Signature and Verification can be constantly used and operate in an on-line manner, those two operations have the greatest need to be performed fast. On the Verification operation we can see time-execution gains at 1.2–12.9% for level 2, 5.6–18.2% for level 3 and 9–20.4% for level 5, again for frequency values ranging from 300 MHz to 1.2 GHz.

Larger gains can be observed for the Sign operation, as our hardware kernel might be executed multiple times until the procedure concludes (this depends on the resampling process taking place within the Dilithium algorithm), having gains of 3–11.8% for level 2, 6.6–20.3% for level 3 and 11.5–23.2% for level 5, for CPU frequencies ranging from 300 MHz to 1.2 GHz. Table IV presents the resource utilization of the basic sub-modules NTT, INTT and matrix-pointwise multiplication on the DS application. Compared to the resources required for a single kernel implementation reported in Table II, it can be deduced that approximately 50% of the resources in the proposed NTT multiplier are bounded to the AXI interface and the pipelined I/O communication with the PS. Furthermore, the high usage of BRAMs is aiming to pipelining the design, placing them between the execution of different sub-modules. Comparing the resources of our design with other handwritten Hardware Description Language (HDL) implementations, we can see that the HLS designs have comparable performance characteristics and even outperform in some cases the handwritten HDL designs in terms of latency/resources. In Table IV the latency and utilized resources of the handwritten HDL multiplier by Land *et al.* [17] are also presented. It should be considered that the multiplier of [17] is a radix-4 NTT design and, by design, it has a different architecture compared to our radix-2 NTT solution and it is expected to have more resources and smaller latency (in clock cycles).

VI. Conclusion

In this paper, an efficient and realistic hardware-software codesign approach has been proposed for the Dilithium DS based on an HLS optimized NTT Dilithium matrix multiplier hardware kernel that efficiently handles the most computational intensive Dilithium DS internal operations. The implementation approach provide details on how to fully implement the hardware-software codesign and takes into account all implementation costs including the OpenCL framework for hardware-software intercommunication. The results of this paper demonstrate that the OpenCL Framework is capable of creating effective HW/SW co-designs when utilized correctly, as it utilizes a large amount of memory between different blocks to pipeline the design. Its use can be particularly beneficial in low-cost SoC devices, with a CPU clock frequency relatively close to the FPGA clock frequency, where utilization of the FPGA fabric can offer lower execution time. As future work, we could enhance the hardware acceleration of our Dilithium HW/SW co-design by adding a hardware kernel for the Keccak algorithm using HLS thus significantly speeding the whole Sign/Verify Dilithium operation, extending the execution chain of accelerator. Also, we believe that the proposed approach of integrating the NTT-matrix multiplier hardware kernel into a software PQC algorithm can be applicable on the FALCON PQC scheme and also on KEM schemes like Kyber.

Acknowledgment

This work has received funding from the European Union's Horizon Europe research and innovation programme SECURED under grant agreement No 10109571

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, p. 1484–1509, Oct. 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[2] D. J. Bernstein, *Introduction to post-quantum cryptography*. Springer, 2009.

[3] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber," *NIST, Tech. Rep*, 2017.

[4] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "Crystals-dilithium," *Algorithm Specifications and Supporting Documentation*, 2020.

[5] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, R. Karri, D. Soni, K. Basu, M. Nabeel, N. Aaraj *et al.*, "Falcon," *Hardware Architectures for Post-Quantum Digital Signature Schemes*, pp. 31–41, 2021.

[6] D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen *et al.*, "SPHINCS," 2017.

[7] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology – LATINCRYPT*, 2012, pp. 139–158.

[8] C. P. Rentería-Mejía and J. Velasco-Medina, "Hardware design of an NTT-based polynomial multiplier," in *2014 IX Southern Conference on Programmable Logic (SPL)*, 2014, pp. 1–5.

[9] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient FPGA implementations of lattice-based cryptography," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 81–86.

[10] S. Mondal, S. Patkar, and T. K. Pal, "A configurable and efficient implementation of Number Theoretic Transform (NTT) for lattice based Post-Quantum-Cryptography," in *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, 2022, pp. 1–6.

[11] A. El-Kady, A. P. Fournaris, T. Tsakoulis, E. Haleplidis, and V. Paliouras, "High-level synthesis design approach for number-theoretic transform implementations," in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2021, pp. 1–6.

[12] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, "Cfntt: Scalable radix-2/4 ntt multiplication architecture with an efficient conflict-free memory mapping scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, p. 94–126, Nov. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9291

[13] D. T. Nguyen, V. B. Dang, and K. Gaj, "A high-level synthesis approach to the software/hardware codesign of NTT-based post-quantum cryptography algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 371–374.

[14] K. Millar, *Design of a flexible schoenhage-strassen FFT polynomial multiplier with high-level synthesis*. Rochester Institute of Technology, 2019.

[15] K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast Number Theoretic Transform," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2018, pp. 106–111.

[16] K. Basu, D. Soni, M. Nabeel, and R. Karri, "NIST Post-Quantum Cryptography- A hardware evaluation study," Cryptology ePrint Archive, Paper 2019/047, 2019, https://eprint.iacr.org/2019/047. [Online]. Available: https://eprint.iacr.org/2019/047

[17] G. Land, P. Sasdrich, and T. Güneysu, "A hard crystal-implementing dilithium on reconfigurable hardware," in *Smart Card Research and Advanced Applications: 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11–12, 2021, Revised Selected Papers*. Springer, 2022, pp. 210–230.

[18] S. Ricci, L. Malina, P. Jedlicka, D. Smékal, J. Hajny, P. Cibik, P. Dzurenda, and P. Dobias, "Implementing CRYSTALS-Dilithium signature scheme on FPGAs," in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES 21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3465481.3465756

[19] G. Mao, D. Chen, G. Li, W. Dai, A. I. Sanka, c. K. Koç, and R. C. C. Cheung, "High-performance and configurable SW/HW co-design of Post-Quantum Signature CRYSTALS-Dilithium," *ACM Trans. Reconfigurable Technol. Syst.*, oct 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3569456

[20] D. T. Nguyen, V. B. Dang, and K. Gaj, "A high-level synthesis approach to the software/hardware codesign of NTT-based Post-Quantum Cryptography algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 371–374.

[21] A. El-Kady, A. P. Fournaris, E. Haleplidis, and V. Paliouras, "High-level synthesis design approach for number-theoretic multiplier," in *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2022, pp. 1–6.