Compact domain-specific co-processor for accelerating module lattice-based KEM

Jose Maria Bermudo Mera*, Furkan Turan*, Angshuman Karmakar*, Sujoy Sinha Roy†, Ingrid Verbauwhede*
*imec-COSIC, KU Leuven, Belgium †University of Birmingham, United Kingdom
*{Jose.Bermudo,Furkan.Turan,Anghusman.Karmakar,Ingrid.Verbauwhede}@esat.kuleuven.be †s.sinharoy@cs.bham.ac.uk

***Abstract*—We present a domain-specific co-processor to speed up Saber, a post-quantum key encapsulation mechanism competing on the NIST Post-Quantum Cryptography standardization process. Contrary to most lattice-based schemes, Saber doesn't use NTT-based polynomial multiplication. We follow a hardware-software co-design approach: the execution is performed on an ARM core and only the most computationally expensive operation, i.e., polynomial multiplication, is offloaded to the co-processor to obtain a compact design. We exploit the idea of distributed computing at micro-architectural level together with novel algorithmic optimizations to achieve approximately a 6 times speedup with respect to optimized software at a small area cost.**

***Index Terms*—Domain-specific co-processor, post-quantum cryptography, lattice-based cryptography, Saber**

I. Introduction

Currently deployed public key cryptography is based on number theoretic problems that can be easily solved by a quantum computer using Shor's algorithm [15] thus putting our privacy at risk. Fortunately, there are computational problems that will remain hard to solve even by a quantum computer and, therefore, they can be used to construct secure post-quantum cryptography (PQC). In order to anticipate the threat of a quantum computer powerful enough to break the existing protocols, the National Institute of Standards and Technology (NIST) has launched a standardization process for quantum-resistant public-key cryptography. Among the submissions that have advanced to the second round of the standardization process [1], cryptosystems based on lattice problems are a popular solution for KEMs.

State of the art: Over the last decade, the theory of lattice-based cryptography has shown significant developments [12]. In addition to the theoretical developments, significant effort has been devoted for efficient implementations [11]. In the most recent literature we can find hardware implementations of FrodoKEM [6], NewHope [8] and Kyber [3]. All of them are KEMs in the second round of the standardization contest run by NIST. However, we can observe a certain bias towards implementing schemes that can perform the polynomial multiplication using the Number Theoretic Transform (NTT) due to its efficiency. To the best of our knowledge, there are no hardware implementations of module Learning With Rounding (module-LWR) KEM Saber.

Our contributions in this paper can be summarized as follows:

- We provide the first hardware implementation of a polynomial multiplier using Toom-Cook algorithm. In the existing literature, only NTT-based multipliers and systolic array implementations are considered.
- We provide the first hardware implementation of Saber, which is a lattice-based KEM that could become the most widespread public key protocol if NIST chooses it for their standards. Benchmarking Saber on hardware platforms contributes to the standardization effort.
- We show that polynomial multipliers based on generic algorithms can be competitive with NTT-based polynomial multiplication when implemented on hardware platforms. This can impact the design considerations of lattice-based schemes.

II. Preliminaries

In this section, we provide the necessary background for understanding the rest of the paper.

A. From LWE to module-LWR

The LWE problem [14] states that, given a randomly selected $a \in \mathbb{Z}_q^n$, it is hard to distinguish between n uniformly random samples drawn from $\mathbb{Z}_q^n \times \mathbb{Z}_q$ and the same number of samples drawn as $b = \langle a, s\rangle + e$, where $s \in \mathbb{Z}_q^n$ is the secret and $e \in \mathbb{Z}_q$ is a freshly generated error term. Notwithstanding distinct parameter choices, the most complex operation of LWE schemes remains the matrix vector multiplication. In its algebraic version, ring-LWE [9], the samples are drawn as polynomials in the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with the form $t = a * s + e$. Thus, the core operation is a convolution, which can be implemented efficiently utilizing the NTT [13], a variant of the Fast Fourier Transform (FFT) that requires exclusively integer arithmetic. However, ring-LWE raises some security concerns when compared to LWE.

To benefit from the increased efficiency of ring-LWE while providing a higher confidence in the security level, module-LWE [4] drafts a small matrix of dimension $l \times l$ composed by ring polynomials. These polynomials have a lower degree than in the ring-LWE setting and, additionally, the convolutions can be parallelized. Lastly, the error term can be introduced by a rounding operation instead of drawing it from a random distribution. This variant is called Learning With Rounding [2]. The samples for LWR problem are generated as in (1). Modulo-LWR combines this technique with the module matrices.

$$\left(a, b = \left\lfloor \frac{p}{q}\langle a, s\rangle \right\rceil_p\right) \in \mathbb{Z}_q^n \times \mathbb{Z}_p \tag{1}$$

Algorithm 1: `Saber.KeyGen()`

1 $seed_A \leftarrow \mathcal{U}(\{0,1\}^{256})$
2 $A \leftarrow \texttt{gen}(\text{seed}_A) \in R_q^{l\times l}$
3 $s \leftarrow \beta_\mu(R_q^{l\times 1})$
4 $b = \texttt{bits}(As + h, \epsilon_q, \epsilon_p) \in R_p^{l\times 1}$
5 **return** $(pk := (b, seed_A), sk := s)$

Algorithm 2: `Saber.Encaps`$(pk = (b, seed_A))$

1 $m \leftarrow \mathcal{U}(\{0,1\}^{256})$; $(\hat{K}, r) = \mathcal{G}(pk, m)$
2 $A \leftarrow \texttt{gen}(\text{seed}_A) \in R_q^{l\times l}$
3 $s' \leftarrow \beta_\mu(R_q^{l\times 1})$
4 $b' = \texttt{bits}(A^T s' + h, \epsilon_q, \epsilon_p) \in R_p^{l\times 1}$
5 $v' = b^T \texttt{bits}(s', \epsilon_p, \epsilon_p) + h_1 \in R_p$
6 $c_m = \texttt{bits}(v' + 2^{\epsilon_p - 1} m, \epsilon_p, \epsilon_t + 1) \in R_{2t}$
7 **return** $(c := (c_m, b'), K := \mathcal{H}(\hat{K}, c))$

Algorithm 3: `Saber.Decaps`$(sk = s, c, K, pk)$

1 $v = b'^T \texttt{bits}(s, \epsilon_p, \epsilon_p) + h_1 \in R_p$
2 $m' = \texttt{bits}(v - 2^{\epsilon_p - \epsilon_t - 1} c_m + h_2, \epsilon_p, 1) \in R_2$
3 $(\hat{K}', r') = \mathcal{G}(pk, m')$
4 $c' = \texttt{Saber.Enc}(pk, m'; r')$
5 **if** $c = c'$ **then**
6 | **return** $K := \mathcal{H}(\hat{K}', c)$ **else** $K := \mathcal{H}(z, c)$

B. Parameter choices

Regardless the underlying problem, system parameters can be grouped into two classes. (1) NTT-friendly parameters. Straightforward polynomial multiplication algorithm has a complexity $\mathcal{O}(n^2)$ while the complexity of NTT is $\mathcal{O}(n \log n)$. (2) Parameters that do not allow the use of the NTT for the polynomial multiplication. The main reason to select parameters belonging to this set is to avoid the expensive reduction modulo a prime by choosing the modulus q as a power of 2 instead. Modular reduction by 2^k is equivalent to keep only the k least significant bits.

Software implementations [7] have proven that NTT can be outperformed by theoretically costlier multiplication algorithms due to the penalization suffered by non-uniform memory accesses and non-trivial modular reduction. However, these performance penalties can be overcome with custom memory accesses and dedicated circuitry, so it is an open problem if schemes that pick parameters from this set can be competitive with NTT-friendly schemes on hardware platforms.

C. Saber

Saber is an IND-CCA KEM with three operations: key generation, encapsulation and decapsulation [5]. During key generation, summarized in Alg. 1, the public matrix A is constructed from a 256-bit seed using the extendable output function `SHAKE-128`. The seed is publicly known but it must be randomly sampled from an uniform distribution to preserve the security. The secret vector s is sampled from a centered binomial distribution β_μ with parameter $\mu = 8$. Matrix-vector multiplication $A \cdot s$ is followed by the rounding operation to generate the vector b. The seed used to generate A and the vector b constitute the public key, while s is the secret key.

Encapsulation is described in Alg. 2. $\mathcal{G}$ and $\mathcal{H}$ are two secure hash functions implemented using Keccak. First, $\mathcal{G}$ is used to generate the session key from a random seed. Then, similarly to key generation, A is regenerated from the public seed, another secret vector s' is sampled and the vector b' is calculated using $A^T \cdot s'$. Subsequently, a vector-vector multiplication is performed between b^T and s'. The ciphertext consists of b',reconciliation information c_m [5] and the hash of the session key and the ciphertext.

Decapsulation is described in Alg. 3. Vector-vector multiplication is followed by the bit selection function named `bits` to recover the message. Then, this message is encrypted again. If the resulting ciphertext is the same as the received one, the key will be established correctly. Otherwise, a random value is output without disclosing information about the failure.

In Saber, the ring dimension $n = 256$ and the two moduli $p = 2^{10}$ and $q = 2^{13}$ are fixed. The size of the matrices and vectors, l, is used to tune the security level. The specifications define $l = 2$ for lightweight cryptography, $l = 3$ as the standard security level and $l = 4$ for long-term high security. Our co-processor supports all security levels, but for concretion we refer to the case $l = 3$ in the rest of the paper. The number of polynomial multiplications is l^2 for key generation, $l^2 + l$ for encapsulation and $l^2 + 2l$ for decapsulation.

III. Algorithmic optimizations

In this section, we describe the rationale behind the partition between software and hardware in our system as well as the selected polynomial multiplication algorithm and the optimizations to achieve our goal.

A. HW/SW boundaries

The domain-specific accelerator is designed following a hardware-software co-design approach in order to: (1) take advantage of the custom logic that can be implemented in an FPGA to accelerate the scheme, (2) maintain the flexibility offered by a micro-controller for controlling the execution flow and (3) keep the resource utilization low by offloading to hardware only the most computationally expensive operations. As explained in Section II-C, the most expensive operation is the multiplication of polynomials with 256 coefficients, which has to be executed l^2 times during matrix-vector multiplication, so its computation is offloaded to hardware. Polynomials are generated from a seed using a hash function, which could also be accelerated in hardware. However, Keccak is costly in terms of area. Instead, we exploit parallelism at system level by generating the polynomials needed for the next multiplication in software while the hardware performs the arithmetic on the previous operands. This approach pipelines the generation of the polynomials with the arithmetic operations, improving the performance as well as the utilization of the available resources.

Algorithm 4: Evaluation for Toom-Cook-4 with vertical scanning [7]

Input: $a(x)$ with $n = 256$ coefficients
Output: $\{aw_1, ..., aw_7\}$ with 64 coefficients each

```
for i = 0 to 63 do
    r_0 = a_0[i]; r_1 = a_1[i]; r_2 = a_2[i]; r_3 = a_3[i];
    r_4 = r_0 + r_2;   r_5 = r_1 + r_3;
    r_6 = r_4 + r_5;   r_7 = r_4 - r_5;
    aw_3[i] = r_6; aw_4[i] = r_7;
    r_4 = 2 * (r_0 * 4 + r_2);   r_5 = r_1 * 4 + r_3;
    r_6 = r_4 + r_5;   r_7 = r_4 - r_5;
    aw_5[i] = r_6; aw_6[i] = r_7;
    r_4 = 8 * r_3 + 4 * r_2 + 2 * r_1 + r_0;
    aw_2[i] = r_4; aw_7[j] = r_0; aw_1[i] = r_3;
```

B. Polynomial multiplication

The polynomial arithmetic in Saber is performed in the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ where $n = 256$ and $q = 2^{13}$. This choice does not allow the use of the NTT to perform polynomial multiplication which makes accelerating this operation a challenging task. Instead, we apply Toom-Cook 4-way to divide a multiplication of polynomials with 256 coefficients into seven multiplications of polynomials with 64 coefficients. These seven multiplications are independent and, hence, can be run in parallel by small multipliers in a distributed computing fashion.

Toom-Cook k-way is a generalization of Karatsuba where a polynomial $a(x)$ with n coefficients is split into k polynomials $a_1 \cdots a_{k-1}$ each with n/k coefficients such that $a(y) = a_0 + a_1 y + ... + a_{k-1} y^{k-1}$ where $y = x^{n/k}$. It works in three steps: evaluation, multiplication and interpolation. First, these k polynomials are used to generate the so-called weighted polynomials, which represent the evaluation of the original polynomial in $2k-1$ different points. Then, point-wise multiplication is computed as the product between the weighted polynomials. Lastly, interpolation is opposite of evaluation step which combines the results from these multiplications to get the final result. During evaluation and interpolation steps, a number of additions and subtractions are required creating a trade-off between the reduction obtained in the number of multiplications that must be performed and the overhead introduced by these operations.

Polynomials are evaluated on $\left\{0, 1, -1, \frac{1}{2}, -\frac{1}{2}, 2, \infty\right\}$. This choice will simplify the hardware for the evaluation since scaling a coefficient by a power of 2 with positive or negative exponent means shifting the bits to the left or to the right that many positions, respectively. To improve the memory access pattern of the evaluation, we use a vertical coefficient scanning to generate all weighted polynomials in-place as shown in Alg. 4.

Both evaluation and interpolation are linear transformations that are inverse of each other. Hence, they are additively homomorphic. Let's assume we want to compute $s = s_1 + s_2$ where $s_1 = a_1 * b_1$ and $s_2 = a_2 * b_2$ and a_i,b_i are polynomials of a certain degree. Denoting evaluation as TC and interpolation TC^{-1}, we can write $s_i = TC^{-1}(TC(a_i) * TC(b_i))$. Using the additive homomorphic property of linear transformations we can also write s as $s = TC^{-1}((TC(a_1) * TC_(b_1)) + (TC(a_2) * TC_(b_2)))$. In Saber, due to module structure, we need to add results of multiple polynomial multiplications during matrix-vector and vector-vector multiplication. Hence, we can apply the lazy interpolation that has been shown in [10] to reduce the number of interpolations. The matrix equation for interpolation is shown in (2). Every division by an odd number is equivalent to a multiplication by the inverse modulo $q = 2^{13}$. However, divisions by powers of 2 become shift operations that could cause a loss of precision that leads to a wrong result. Since the highest division is by $8 = 2^3$, three extra bits of precision are required and, therefore, the data width of the co-processor must be of at least 16 bits.

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & \frac{2}{45} & -\frac{2}{3} & -\frac{2}{9} & \frac{1}{36} & \frac{1}{60} & -2 \\ -\frac{5}{4} & 0 & \frac{2}{3} & \frac{2}{3} & -\frac{1}{24} & -\frac{1}{24} & 4 \\ \frac{5}{2} & -\frac{1}{18} & \frac{3}{2} & -\frac{7}{18} & -\frac{1}{18} & 0 & \frac{5}{2} \\ \frac{1}{4} & 0 & -\frac{1}{6} & -\frac{1}{6} & \frac{1}{24} & \frac{1}{24} & -5 \\ -\frac{1}{2} & \frac{1}{90} & -\frac{1}{3} & \frac{1}{9} & \frac{1}{36} & -\frac{1}{60} & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c(\infty) \\ c(2) \\ c(1) \\ c(-1) \\ c(\frac{1}{2}) \\ c(\frac{-1}{2}) \\ c(0) \end{pmatrix} \quad (2)$$

IV. Hardware architecture

In this section, we describe the hardware architecture following a bottom-up approach.

A. 64-coefficient polynomial multiplier

This unit is responsible for computing the 64-coefficients polynomial multiplications during Toom-Cook point-wise product. It will be instantiated seven times in parallel, so it is necessary to keep it simple to lower the area requirements of the overall design. For this reason, we choose straightforward schoolbook polynomial multiplication. We exploit parallelism once again and propose the generic architecture depicted in Fig. 1. First, there is a loading stage where $n_m = 4$ coefficients from b are loaded into the rightmost inputs of the multipliers, implemented with fabric DSPs. Then, all 64 coefficients in a are loaded consecutively into the other register. During this phase, one coefficient of the result is produced each clock cycle in the leftmost output register, while the rest of the accumulated intermediate values shift to the left. After this, n_m additional coefficients are produced while the datapath is flushed. Then, the next n_m coefficients in b are loaded and the process repeats until the full multiplication has been computed. The pipeline strategy is not trivial due to data dependencies between the accumulation and the previous result generated in the immediate right MAC. The critical path is shown in Fig. 1 with a red dashed line. To break it down without altering the dataflow, pipeline registers are included only in the multiplier. These pipeline registers are represented as green lines.

Since fabric LUTs have a depth of 64 bits, which matches the length of the polynomials, the distributed memory is implemented as LUT-based memory. Operand a is accessed

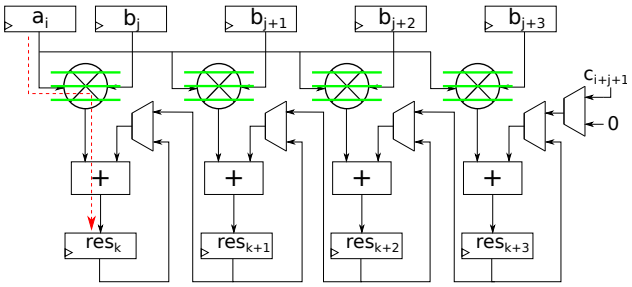

Fig. 1. Architecture for the 64×64 polynomial multiplier utilizing 4 DSP units, including the critical path and the pipeline registers that break it down

sequentially, so a single port RAM is enough to store it. To halve the latency of the loading stage, operand b is stored in a dual port RAM. To allow simultaneous read and write operations, the result is stored in a dual port RAM. The data width is 16 bits as explained in Sec. III-B.

To conclude the design-space exploration of this module, we show the impact of changing the number of DSPs, n_m. In our design, n_m affects the latency as in (3). The four terms in the equation correspond to (1) loading the n_m coefficients from b, (2) filling up the datapath, (3) performing the computation and (4) flushing the datapath between iterations. For a compact design we have chosen $n_m = 4$, i.e., 1168 clock cycles. It is the smallest option that allows our multiplier to be competitive with NTT, e.g., 1289 clock cycles [3]. For a high-performance implementation $n_m = 8, 16$ can be considered.

$$d = \frac{64}{n_m}\left(\frac{n_m}{2} + 4 + 64 + (n_m - 1)\right) = \frac{4288}{n_m} + 96 \qquad (3)$$

B. Toom-Cook multiplier

The three steps of Toom-Cook are implemented on different datapaths with independent control. The memory requirements of this module depend exclusively on how evaluation and interpolation are implemented. In particular, the reading throughput of the system memory imposes a performance limitation on the evaluation hardware while the interpolation hardware must accommodate to the writing pattern. Sec. IV-C details the memory requirements, the access pattern, the address generation and the memory layout. In the following we focus on the evaluation and interpolation circuits.

1) Evaluation hardware: The evaluation datapath is derived directly from Alg. 4 as shown in Fig. 2. The same datapath is used to perform the evaluation for both operands, a and b, one immediately after the other. The weighted polynomials are directly stored in the distributed memory of the seven small polynomial multipliers. The delay introduced by two 16-bit adders is not big enough to require pipeline. The latency of the entire evaluation step is 128 clock cycles, which corresponds to reading the two 256-coefficient multiplicands in chunks of four coefficients per clock cycle.

2) Interpolation hardware: Building the hardware to execute the interpolation step is a challenging task because a direct mapping of (2) as for evaluation would result in a very asymmetric datapath with a long critical path. Instead, we identify certain symmetries in the interpolation matrix

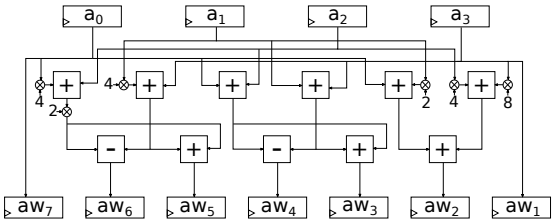

Fig. 2. Datapath for the evaluation step

together with a trial and error approach to derive the circuit in Fig. 3. The critical path, indicated with a red dashed line, is broken down with pipeline registers, represented as horizontal green lines to allow a higher clock frequency. The interpolation hardware can read seven coefficients in parallel coming from the seven small polynomial multipliers but write operations can only be done at the clock rate due to the irregular memory accesses. Thus, memory operations become the bottleneck for interpolation. Irregular memory accesses are caused by the polynomial indexing. Interpolation outputs seven coefficients that must be written with offsets equal to $\{0, 64, 128, 192, 256, 320, 384\}$. The six least significant bits of the iteration counter can be used to generate the base address of the corresponding iteration while the offsets set the most significant bits of the writing address. However, the 127 iterations will end up mismatching the offsets and making inefficient a possible memory alignment to increase the writing throughput.

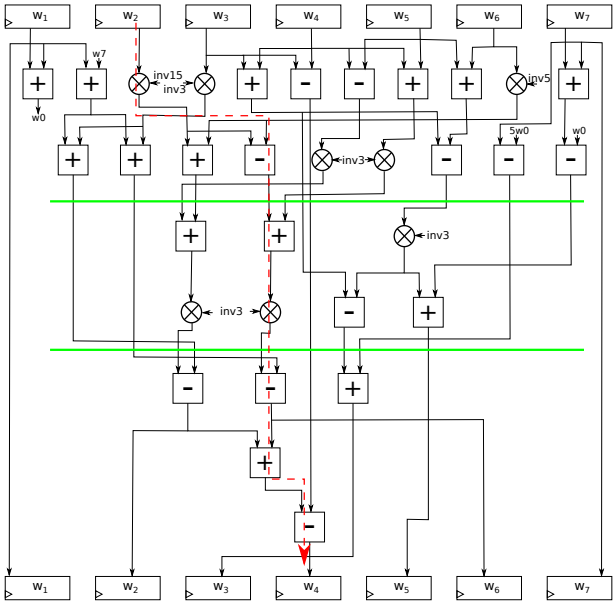

Fig. 3. Datapath for the interpolation step including the critical path and the pipeline registers that break it down

C. On-chip memory

System memory is implemented using dedicated block RAM primitives called BRAM36K which can store up to 1024 words of 36 bits. For 64-bit words required required by the evaluation stage, 2 BRAMs are needed. The memory is configured as dual port to allow simultaneous read and write

operations, and with asymmetric read and write operations for the same port since read operations are performed on 64-bit words while write operations are performed on 16-bit words. One of the ports is also multiplexed between the HW/SW interfacing and the accelerator. Regarding the memory layout, the four coefficients that evaluation reads every clock cycle are not consecutive but offset with $0, 64, 128$ and 192. Then, coefficients must be aligned as in Fig. 4. Besides the coefficient alignment, polynomials are also aligned to 64 words, which is the natural alignment for 256-coefficient polynomials but not for 512-coefficient polynomials as shown in the same figure.

This memory layout can be created with almost no overhead when realizing the data transfer from software as it just requires a fixed offset on the indexing of the array. Since the memory is accessed asymmetrically for read and write operations, address translation is needed for computing the real writing address. When addressing the memory as a 16-bit word RAM, the least significant word of figure's address 0 corresponds to address 0, the next 16-bit word of figure's address 0 corresponds to address 1, etc. until the least significant 16-bit of figure's address 1, which corresponds to address 4 and so on. Since the memory is aligned to 256-coefficient polynomials, only the eight least significant bits of address need to be translated. Rewiring the two most significant bits from the coefficient index, which has a length of eight bits, to the two least significant bits of the address and shifting the other six two positions to the left gives the corresponding writing address for the coefficient.

address 0	a_0	a_{64}	a_{128}	a_{192}
address 1	a_1	a_{65}	a_{129}	a_{193}
address 2	a_2	a_{66}	a_{130}	a_{194}
⋮	•••			
address 64	b_0	b_{64}	b_{128}	b_{192}
address 65	b_1	b_{65}	b_{129}	b_{193}
⋮	•••			
address 128	c_0	c_{64}	c_{128}	c_{192}
address 129	c_1	c_{65}	c_{129}	c_{193}
⋮	•••			
address 255	c_{319}	c_{383}	c_{447}	c_{511}

Fig. 4. Coefficient alignment used in the system memory

D. HW/SW Interfacing

Fig. 5 shows an overview of the system architecture. We implemented our hardware co-processor on a Xilinx Zynq device which integrates FPGA to ARM processors. Zynq devices support an AXI based communication interface for the interaction of ARM cores and any hardware module in FPGA. Additionally, Xilinx DMA offers the highest performance for bulky data transfers between memory and the modules. Hence, we tailored our interfacing mechanism for proficient use of them. We kept the data word size 64-bit for handling the coefficients both in the ARM side software and in the BRAM. As a result, the software stores the polynomials as array of 64-bit data words. The BRAM uses also the same data word length, hence we configured the DMA for transferring polynomial

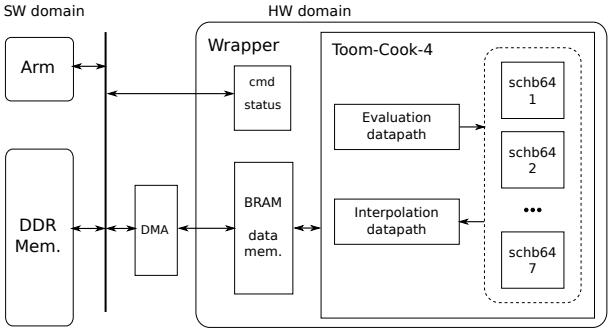

Fig. 5. High-level architecture and interfacing of hardware and software.

arrays from memory to BRAM directly. When performing these transfers, the DMA accesses the array with its memory address, and delivers it over (or receives from) the wrapper as a stream, i.e., one data word at each clock cycle. This stream is free from address information, hence our wrapper associates data words with an address in the BRAM. For associating the right address, the wrapper is informed with the transfer's base address by a command prior to the transfer's start. A register unit is used to support the interfacing with command and status registers. This approach makes the software side the master of our architecture, responsible of sending commands to the co-processor, observing its execution status and handling data transfers. Currently, commands for data transfers, evaluation, multiplication, MAC and interpolation are supported. The instruction-set architecture (ISA) is quite flexible leaving room for the inclusion of new commands or even the integration of more modules to accelerate other operations utilizing the same co-processor.

V. Results

We have implemented our domain-specific co-processor in the Xilinx ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. Software has been adapted from [7] by substituting their custom assembly optimizations by C code compiled with the GCC version available in the Xilinx SDK development tool. Hardware has been synthesized, placed and routed using Vivado 2018.1. Although different synthesis and implementation strategies can be explored for a fine-grained optimized design, all results reported correspond to default configurations where the hardware co-processor runs at 125 MHz and the ARM processor runs at 666 MHz.

A. Performance results

Table I presents the performance of key generation, encapsulation, decapsulation and polynomial multiplication measured from software. Columns show the execution time when using only software and the full co-processor. Saber becomes between 5.4 and 7 times faster while polynomial multiplication is almost 27 times faster. The execution time of the multiplication includes the overhead due to data transfers. In practice, many of these transfers are not necessary because we are performing matrix-vector multiplication instead of a standalone polynomial multiplication. Arithmetic operations only take 11835 clock cycles, which is 91 times faster than

TABLE I
EXECUTION TIME (MEASURED IN MILLION CPU CYCLES)

	only SW	SW/HW	Improvement
Key Generation	11.761	2.180	5.4
Encapsulation	14.944	2.762	5.4
Decapsulation	17.983	2.560	7.0
Polynomial Mult.	1.097	0.041	26.7

software even though the hardware is clocked more than five times slower than the software.

The overhead introduced by the commands sent from software to hardware is negligible due to the parallel transfer. However, this is not the case for the data transfers to the BRAM. Sending a polynomial, i.e., 512 bytes, from ARM to the co-processor takes 1816 clock cycles. Sending two polynomials, i.e., 1024 bytes, takes 2908 clock cycles. Larger data transfers are not of interest in our use case since polynomials are generated just-in-time on the CPU while the hardware runs the multiplication with the previous operands. Transfers in the other direction have almost the same execution times.

B. Resource utilization and comparisons with other works

The utilization of a single 64-coefficient polynomial multiplier including the LUT-based memory is of 342 LUTs, 155 FFs and 4 DSPs. This module is instantiated seven times and constitutes the core of arithmetic operations. The full hardware co-processor, including Toom-Cook multiplier, system memory and command decoding is implemented using 2927 LUTs, 1279 FFs, 2 BRAMs and 38 DSPs, which is quite a compact design.

Table II shows a comparison of our co-processor with other hardware implementations of NIST PQC second round candidates. For our work, we report the utilization of the full system including the processing system and the HW/SW interfacing. We can observ that module-LWR offers a trade-off between LWE, e.g., Frodo, and ring-LWE, e.g., NewHope. Comparison to an ASIC implementation is less fair but Kyber is a more similar scheme to Saber.

TABLE II
COMPARISON WITH STATE-OF-THE-ART.

Scheme	Platform	Time [μs] KeyGen/Encaps/ Decaps	Freq [MHz]	BRAM/ DSP	FF/ LUT
Kyber [3]	ASIC	1548/2465/ 1646	72	-	-
Frodo [6]	FPGA	45454/45454/ 47619	167	24 / 1	3559/ 7773
NewHope [8]†	FPGA	51.9/78.6/ 21.1	133	14 / 8	9975/ 20826
Saber [This]	FPGA	3273/4147/ 3844	125	2 / 28	7331/ 7400

†Implements only CPA secure NewHope

VI. CONCLUSIONS

Domain-specific accelerators and hardware-software co-design approaches are becoming more important nowadays. In this paper, we have presented a compact domain-specific accelerator for Saber. Moreover, efficiency on hardware platforms is a crucial evaluation criteria for NIST PQC standardization. NTT-friendly lattice-based cryptography has been studied well in the past, but there exists less research on alternative polynomial multiplication algorithms for hardware acceleration. We believe our design will rekindle interest in such designs.

ACKNOWLEDGMENT

This work was partly supported by the Research Council KU Leuven: C16/15/058, and also by the European Commission through the Horizon 2020 research and innovation programme under agreement Cathedral ERC Advanced Grant 695305 and by EU H2020 project FENTEC Grant 780108.

REFERENCES

[1] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone. (2019, jan) Nistir 8240 - status report on the first round of the nist post-quantum cryptography standardization process. [Online]. Available: https://doi.org/10.6028/NIST.IR.8240

[2] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs, "Learning with rounding, revisited: New reduction, properties and applications," *IACR Cryptology ePrint Archive*, vol. 2013, p. 98, 2013. [Online]. Available: http://eprint.iacr.org/2013/098

[3] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 4, pp. 17–61, 2019.

[4] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé, "CRYSTALS - kyber: a cca-secure module-lattice-based KEM," *IACR Cryptology ePrint Archive*, vol. 2017, p. 634, 2017. [Online]. Available: http://eprint.iacr.org/2017/634

[5] J. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure KEM," *IACR Cryptology ePrint Archive*, vol. 2018, p. 230, 2018. [Online]. Available: http://eprint.iacr.org/2018/230

[6] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard lattice-based key encapsulation on embedded devices," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 372–393, 2018.

[7] A. Karmakar, J. M. B. Mera, S. S. Roy, and I. Verbauwhede, "Saber on ARM cca-secure module lattice-based key encapsulation on ARM," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 243–266, 2018.

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

[9] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, pp. 43:1–43:35, 2013.

[10] J. M. B. Mera, A. Karmakar, and I. Verbauwhede, "Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography," Cryptology ePrint Archive, Report 2020/268, 2020, https://eprint.iacr.org/2020/268.

[11] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 129:1–129:41, 2019.

[12] C. Peikert, "A decade of lattice cryptography," *Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 4, pp. 283–424, 2016.

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

[14] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '05, 2005, pp. 84–93.

[15] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.