A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm

VINCENT MIGLIORE, CÉDRIC SEGUIN, MARIA MÉNDEZ REAL, VIANNEY LAPOTRE, ARNAUD TISSERAND, CAROLINE FONTAINE, GUY GOGNIAT, Univ. Bretagne-Sud, UMR CNRS 6285, Lab-STICC, F-56100 Lorient, France
RUSSELL TESSIER, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, USA

Somewhat Homomorphic Encryption (SHE) schemes can be used to carry out operations on ciphered data. In a cloud computing scenario, personal information can be processed secretly, inferring a high level of confidentiality. The principle limitation of SHE is the size of ciphertext compared to the size of the message. This issue can be addressed by using a batching technique that "packs" several messages into one ciphertext. However, this method leads to important drawbacks in standard implementations. This paper presents a fast hardware/software co-design implementation of an encryption procedure using the Karatsuba algorithm. Our hardware accelerator is 1.5 times faster than the state of the art for 1 encryption and 4 times faster for 4 encryptions.

ACM Reference format:
Vincent Migliore, Cédric Seguin, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat and Russell Tessier. 2017. A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (October 2017), 17 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Homomorphic Encryption (HE) is a recent promising tool in modern cryptography that supports operations on encrypted data. This property allows for the protection of private and sensitive data in a cloud computing scenario. Figure 1 provides a flowchart of a basic Homomorphic cloud service. Historically speaking, early cryptographic schemes presented partial homomorphic properties, for multiplication [1] and addition [2]. Only after the approaches in [3] and [4] were presented was it possible to support both types of operations at the same time. These schemes have been followed by many other related contributions.
Most promising Fully Homomorphic Encryption (FHE) schemes base their arithmetic on a ring of polynomials with integer coefficients [5][6][7][8][9][10]. Each operation requires a double reduction: a modular reduction by an irreducible polynomial, typically required after each polynomial multiplication, and an integer reduction on each polynomial coefficient.

A key aspect of Homomorphic Encryption is the representation of messages. A message can be seen as a binary value, or an integer if it has more than one bit. If messages are represented as integers, only integer additions, subtractions and multiplications are possible. This list excludes

1539-9087/2017/10-ART1 $15.00
https://doi.org/0000001.0000001

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

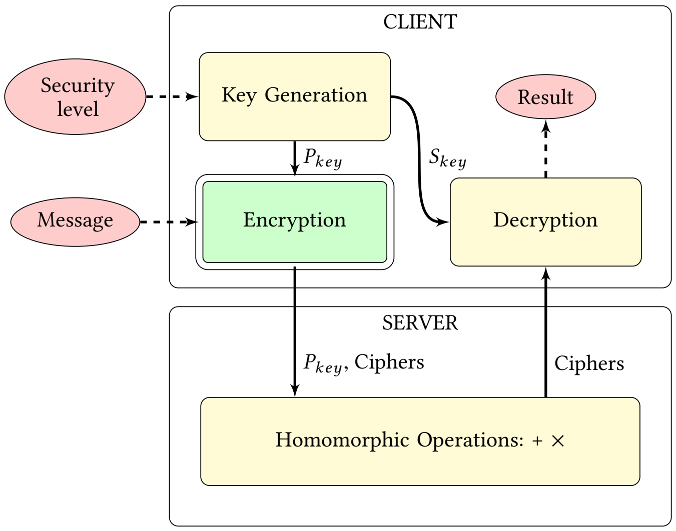

Fig. 1. Flow of an homomorphic cloud service.

standard operators like comparison. To enable such operations, it is necessary to switch to a binary message representation. The maximum value of a message is called the message space.

A second aspect of Homomorphic Encryption is the size of the encrypted data. Depending on the complexity of the cloud service, the cipher size can vary from a few KB to a few MB for 1 message (which can be binary). To address this penalty, two main solutions exist: transciphering and batching. With transciphering, data is sent to the server using standard symmetric encryption and then is decrypted on the server-side with homomorphic encryption. This operation requires a symmetric secret key, which is sent to the server encrypted with homomorphic encryption. This technique is not in the scope of this paper, but the reader can refer to [11] for further information. With batching, several messages are “packed" within one ciphertext using the Chinese Remainder Theorem (CRT) [12]. When homomorphic operations are computed on the server side, operations are executed for each message in parallel. Thus, the size of encrypted data per bit of information is reduced by the number of messages packed.

In practice, actual implementations of homomorphic encryption cannot perform batching efficiently due to the limitations of the standard algorithm used for such computation, i.e the NTT algorithm [13]. To perform a batching operation, the irreducible polynomial chosen for the FHE scheme must be factorizable in the message space. In particular, $x^n + 1$, the most efficient choice for NTT, is only factorizable for integer messages and not binary ones, and to our knowledge there is no efficient alternative. Without batching, NTT is a very powerful tool because once polynomials are converted to their NTT form, all computations are performed modulo $x^n + 1$ and thus, one can perform all required computations in this form.

In the batching case, the drawback is very significant. First, one needs to double the number of NTT points to avoid performing polynomial reduction during computations. Second, each polynomial multiplication must perform one NTT, one component-wise multiplication, and then one inverted NTT. This process is in contrast to the no-batching case where polynomials remain in NTT form. On the client side, because computing capacity is limited, this drawback can become very costly. As a consequence, the well known SEAL library only implements batching for non binary messages, which reduces the interest of the implementation for many algorithms. In practice, few

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

approaches use hardware to target homomorphic encryption with batching for binary messages due to the incompatibility with NTT. To our knowledge, only [14] and [15] provide complete accelerators with batching, although the former focuses on the server side only and the latter only targets complex homomorphic algorithms. In this paper, we provide the first batching compliant implementation of homomorphic encryption on the client side using the Karatsuba algorithm [16], and compare our results to the latest homomorphic libraries. The implementation greatly extends the work in [15] with important modifications to adapt the computation to the encryption both in terms of software and hardware. The main contributions of this work are as follows:

- Encryption step acceleration using a hardware/software co-design approach that leverages the Karatsuba algorithm (up to 4 ciphers in parallel).
- High performance software computations using vector programming (AVX2/SSE4.2 and NEON).

Compared to the server-side accelerator [15] in which parallel operations could not be performed due to the complexity of the homomorphic multiplication, we have exploited the polynomial arithmetic simplicity of the encryption operation to parallelize our design. Thus, the main challenge of this work was simultaneously dealing with a larger hardware design and much larger transfers between hardware and software. This limitation has been addressed by exploiting the structure of polynomials in Homomorphic Encryption.

This paper is organized as follows. Section 2 provides notation and basic mathematical knowledge about encryption and the Karatsuba algorithm. Section 3 describes the hardware and software accelerator architecture. Section 4 presents implementation results and compares them with a state of the art software implementation. Section 5 summarizes and concludes the paper.

2 THEORETICAL BACKGROUND

2.1 Notation

In the following, a polynomial is represented in uppercase and its coefficients in lowercase. For polynomial A, a_i represents its i^{th} coefficient. A vector of polynomials is noted in bold. For vector $\mathbf{A}$, $\mathbf{A}[i]$ is the i^{th} polynomial of the vector. For set R and polynomial A, $A \leftarrow U_R$ represents a uniformly sampled polynomial in R, $A \leftarrow \chi_\sigma$ is a polynomial sampled in a discrete Gaussian distribution with standard deviation σ and B_R a very narrow discrete Gaussian distribution in which polynomials have binary coefficients. For coefficient a_i of polynomial A, $a_{i,(j..k)}$ corresponds to the binary string extraction of a_i between bits j and k. This notation is extended to polynomial A where $A_{(j..k)}$ is the sub-polynomial in which the binary string extraction is applied to each coefficient. A modular reduction by an integer q is $[\cdot]_q$. For integer a, $\lfloor a \rfloor$, $\lceil a \rceil$ and $\lfloor a \rceil$ operators are floor, ceil and nearest rounding operations, respectively. This notation is extended to polynomials by applying the operation on each coefficient. For vectors $\mathbf{A}$ and $\mathbf{B}$, $\langle \mathbf{A}, \mathbf{B} \rangle$ represents $\sum \mathbf{A}[i]\mathbf{B}[i]$. In the following, polynomials have coefficients in $\mathbb{Z}_q$, i.e. integer coefficients in $[0, q[$.

2.2 Ciphering

This paper focuses on the encryption operation for the Ring-Learning With Error (R-LWE) [17] based schemes, and in particular the Fan Vercauteren (FV) [7] scheme. The basic idea of R-LWE is to hide a secret by using a noisy distribution. For a secret polynomial $S \leftarrow B_R$ and noisy polynomials $A \leftarrow U_R$ and $E \leftarrow \chi_R$, a R-LWE sample is the couple $(-A \cdot S + E, A)$. In the case of FV, $S_{key} = S$ is the secret key and $\mathbf{P_{key}} = (-A \cdot S + E, A)$ is the public key. For an integer t such as a message $m \in [0, t[$ (integer message), the encryption operation is performed as follows:

$$\mathbf{C} = \left(\left[\Delta m + \mathbf{P_{key}}[1]U + E_1 \right]_q, \left[\mathbf{P_{key}}[2]U + E_2 \right]_q \right) \tag{1}$$

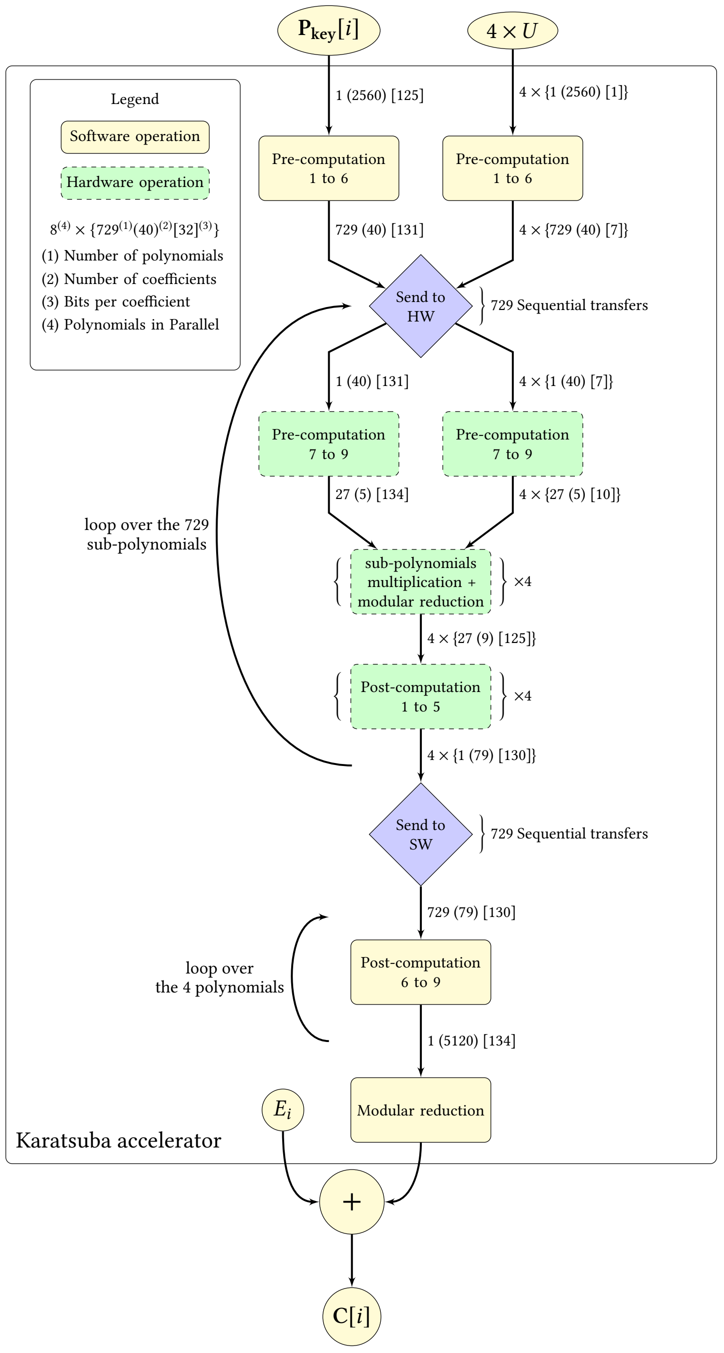
$\mathbf{P}_{\mathbf{key}}[i]$
$4 \times U$
Legend
Software operation
Hardware operation
$8^{(4)} \times \{729^{(1)}(40)^{(2)}[32]^{(3)}\}$
(1) Number of polynomials
(2) Number of coefficients
(3) Bits per coefficient
(4) Polynomials in Parallel
1 (2560) [125]
$4 \times \{1$ (2560) [1]$\}$
Pre-computation 1 to 6
Pre-computation 1 to 6
729 (40) [131]
$4 \times \{729$ (40) [7]$\}$
Send to HW
729 Sequential transfers
1 (40) [131]
$4 \times \{1$ (40) [7]$\}$
Pre-computation 7 to 9
Pre-computation 7 to 9
27 (5) [134]
$4 \times \{27$ (5) [10]$\}$
loop over the 729 sub-polynomials
sub-polynomials multiplication + modular reduction
×4
$4 \times \{27$ (9) [125]$\}$
Post-computation 1 to 5
×4
$4 \times \{1$ (79) [130]$\}$
Send to SW
729 Sequential transfers
729 (79) [130]
loop over the 4 polynomials
Post-computation 6 to 9
1 (5120) [134]
E_i
Modular reduction
Karatsuba accelerator
+
$\mathbf{C}[i]$

Fig. 2. Flow of the encryption operation in our architecture, where $i \in \{1, 2\}$ represents the i^{th} member of the ciphertext C. Values $\mathbf{P}_{\mathbf{key}}[i]$, U, and E_i are, respectively, the public key, a binary sampled polynomial and a Gaussian sampled polynomial.

with $U \leftarrow B_R$, $(E_1, E_2) \leftarrow \chi_R^2$ and $\Delta = \lfloor q/t \rfloor$. If the noise term $EU + E_1 + S_{key}E_2$ is below $\Delta/2$, the message can be decrypted without error. In the following, we set $t = 2$ which is a common choice in the literature. In particular, it allows a wide range of operations (such as comparison) instead of just integer addition, subtraction and multiplication (i.e. when $t > 2$).

To accelerate this operation, we use a software/hardware co-design implementation of a polynomial multiplication algorithm which is based on the Karatsuba algorithm. For ciphering, Karatsuba efficiently performs $\mathbf{P}_{\mathbf{key}}[1]U$ and $\mathbf{P}_{\mathbf{key}}[2]U$ operations. In addition, high-speed binary polynomial generation and a discrete Gaussian sampler are required to generate U, E_1 and E_2. However, we decided to do not include these primitives in the scope of this work as we believe a more mature background is required.

2.3 The Batching Technique

The arithmetic of R-LWE schemes must perform polynomial operations modulo an irreducible polynomial in $\mathbb{Z}_q$ (usually chosen in the literature as a cyclotomic polynomial for security concerns). Some cyclotomic polynomials have an additional property, they are reducible in $\mathbb{Z}_2$. If each factor is unique and has a multiplicative order of 1, then the batching technique is possible. Formally, for a vector of k messages $\boldsymbol{m}$ and a given irreducible polynomial Φ in $\mathbb{Z}_q$ compatible with batching, the batching polynomial M can be expressed as:

$$M = \sum_{i=1}^{k} \boldsymbol{m}_i \cdot \boldsymbol{S}_i \cdot \boldsymbol{\Phi}_i \mod \Phi, \text{with} \begin{cases} \Phi \equiv \prod_{i=1}^{k} \boldsymbol{\varphi}_i \mod 2 \\ \boldsymbol{\Phi}_i = \dfrac{\Phi}{\boldsymbol{\varphi}_i} \\ \boldsymbol{S}_i \equiv \boldsymbol{\Phi}_i^{-1} \mod \varphi_i \end{cases} \tag{2}$$

To recover the i^{th} message, it suffices to perform

$$\boldsymbol{m}_i \equiv M \mod \varphi_i \tag{3}$$

$\boldsymbol{m}_i$ is called the residue polynomial. Then, for two batching polynomials M_a and M_b, polynomial additions, subtractions and multiplications, perform the same operation between residue polynomials in parallel.

As stated in Section 1, the best choice for NTT is the cyclotomic polynomial $x^n + 1$. With such parameters, NTT can be adapted to compute polynomial modular reduction during computations (called Negative Wrapped Convolution). However, the factorization of $x^n + 1$ in $\mathbb{Z}_2$ is $(x + 1)^n$. So this polynomial is not compatible with batching due to the unique factor and the multiplicative order. NTT Positive Wrapped Convolution is the current alternative. It performs polynomial arithmetic modulo $x^n - 1$ during computations. However, $x^n - 1$ is not compatible with homomorphic encryption because it is clearly reducible (1 is an obvious root). This greatly penalizes NTT, which requires several adaptations. First, the number of points of the NTT algorithm must be twice as large versus the Negative Wrapped Convolution case to not perform polynomial reduction. This issue is quite critical when the degree is slightly higher than a power of two. For example, for degree-3000 polynomial multiplication, the required NTT has $2 \times 4096 = 8192$ points. Second, modular reduction by a cyclotomic polynomial Φ must be carried out. This implies a need to perform an inverted-NTT, a modular reduction, and then an NTT.

These limitations show that NTT has important issues dealing with batching. This issue motivates our architecture based on the concurrent polynomial multiplication algorithm called Karatsuba. Karatsuba does not suffer from batching limitations and it is possible to adapt the algorithm to various polynomial sizes.

2.4 Karatsuba Algorithm

The Karatsuba algorithm is an improvement on the standard polynomial multiplication algorithm which reduces the number of sub-products. In SHE, polynomials have the same number of coefficients, and our setup always provides an even number of coefficients. Thus, in the following, we only discuss the Karatsuba algorithm with these constraints. Input polynomials A and B of degree $n-1$ are split into two parts of equivalent size, $\frac{n}{2}$ coefficients. Let A_H and A_L be two polynomials composed of the coefficients of the highest degree of A and the lowest degree of A, respectively. B_H and B_L are constructed using the same approach. Input polynomials can now be expressed as $A = A_L + A_H x^{n/2}$ and $B = B_L + B_H x^{n/2}$.

When A and B are multiplied using the standard approach, the resulting decomposition is given by:

$$\begin{aligned} A \times B &= (A_L + A_H x^{n/2})(B_L + B_H x^{n/2}) \\ &= A_L B_L + (A_L B_H + A_H B_L) x^{n/2} + A_H B_H x^n \end{aligned} \tag{4}$$

Karatsuba optimization exploits the fact that the middle factor $(A_L B_H + A_H B_L)$ can be cleverly computed as $(A_L + A_H)(B_L + B_H) - A_L B_L - A_H B_H$. $A_L B_L$ and $A_H B_H$ are already computed and do not require additional multiplications.

In total, Karatsuba requires 3 sub-polynomial multiplications instead of 4, at a cost of two pre-computations, $(A_H + A_L)$ and $(B_H + B_L)$, and two post-computations for the reconstruction of the middle factor. These pre- and post-computations only require additions and subtractions. To further reduce the complexity of the polynomial multiplication, one can apply recursively the Karatsuba algorithm to each sub-polynomial multiplication, $A_L B_L$, $A_H B_H$ and $(A_H + A_L)(B_H + B_L)$. The number of times that the Karatsuba algorithm is applied is called the number of Karatsuba recursions. After several Karatsuba recursions, one has to perform many low degree polynomial multiplications instead of a large polynomial multiplication. This recursiveness allows computation sharing between software and hardware. For example, several recursions can be performed in software and the remaining ones in hardware.

Because each Karatsuba recursion halves the size of sub-polynomials, Karatsuba can achieve polynomial multiplication of degree $2^r(p+1) - 1$, where r is the number of Karatsuba recursions and p the degree of the smallest sub-polynomial.

3 ACCELERATOR ARCHITECTURE

3.1 High-Level Overview

We based our architecture on the design in [15] which uses Karatsuba algorithm to accelerate server-side operations. This work uses the same Homomorphic Encryption setup (e.g. degree-2559 polynomials with 125-bit coefficients) to speed-up client-side operations, in particular the ciphering. Figure 2 presents the flow of the proposed accelerator that supports 4 parallel encryptions. The accelerator operates as follows: Six Karatsuba pre-recursions are computed in software and three are performed in hardware. After software pre-computations for each input polynomial, 729 sub-polynomials with 40 coefficients are generated and sent sequentially to the hardware. The hardware is fully pipelined and operations on sub-polynomials are executed as soon as polynomials arrive. At the end of Karatsuba pre-computations, 19,683 degree-4 polynomials are generated on each input polynomial that must be multiplied term by term. These multiplications are computed in hardware by four parallel polynomial multiplier units using the standard polynomial multiplication algorithm. Because the accelerator computes up to four encryptions in parallel, four post-computation units are implemented in parallel in hardware. These post-computations are computed sequentially in software, although multi-threading can possibly be used. The distribution of Karatsuba pre- and

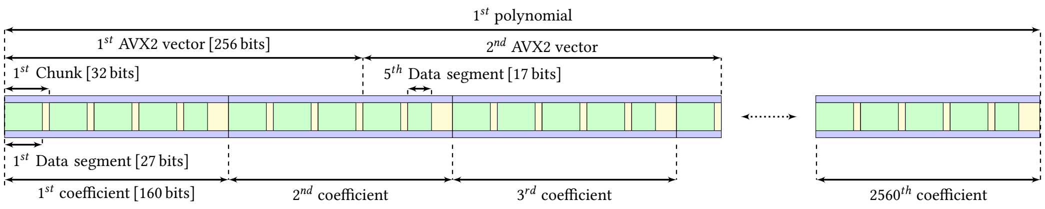
1st polynomial
1st AVX2 vector [256 bits]
2nd AVX2 vector
1st Chunk [32 bits]
5th Data segment [17 bits]
1st Data segment [27 bits]
1st coefficient [160 bits]
2nd coefficient
3rd coefficient
2560th coefficient

Fig. 3. Software representation of one polynomial of the public key.

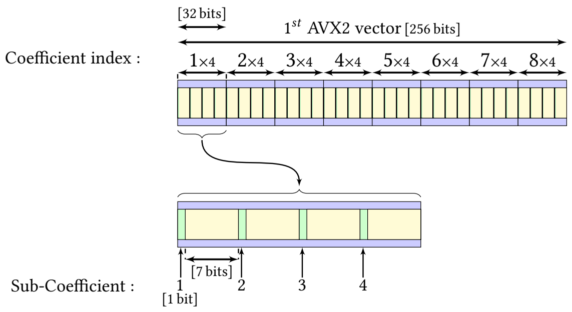

Fig. 4. Software representation of binary polynomials.

post-recursions between hardware and software is a key element of our architecture. Further details are provided in Section 3.3.5.

3.2 Software Implementation

Our Karatsuba software design is implemented using contemporary vector programming (AVX2, SSE4.2, NEON, ...). For simplicity, we describe our work for the AVX2 instruction set, but the approach remains valid for SSE4.2 and NEON with the exception that their vectors have smaller length. AVX2 Single Instruction Multiple Data (SIMD) instructions are performed on 256-bit vectors. The vector can be seen as 4 doubles (64-bit operands), 8 floats (32-bit operands) or 8 integers (32-bit operands). For each elementary operation, the computation is performed on each element of the vector in parallel. AVX2 supports additions, subtractions, multiplications, maskings and various methods to speed-up specific algorithms.

3.2.1 Representation of polynomials in memory. An efficient representation of polynomials in memory has been made to enhance the efficiency of vector programming with Karatsuba. As mentioned in Section 2.2, Karatsuba multiplies the public key $\mathbf{P_{key}}$ with a randomly-generated binary polynomial. Thus, two different kinds of storage are required: a full size polynomial with 125 bits per coefficient for the public key, and a small polynomial with 1 bit per coefficient for the binary polynomial.

Figures 3 and 4 provide data representations of a full size polynomial and a binary polynomial, respectively. For the full size polynomial, coefficients are split into five 32-bit chunks. Because AVX2 operations do not support addition with carry, guard bits are required for carry propagation between operations. In our case, this choice was quite simple because with 125-bit coefficients, it is not possible to have less than five chunks to be able to have at least one guard bit per chunk. This setup has the benefit of providing multiple guard bits per chunk, which allows for the computation of successive operations before carry propagation. Section 3.2.2 provides a further explanation on

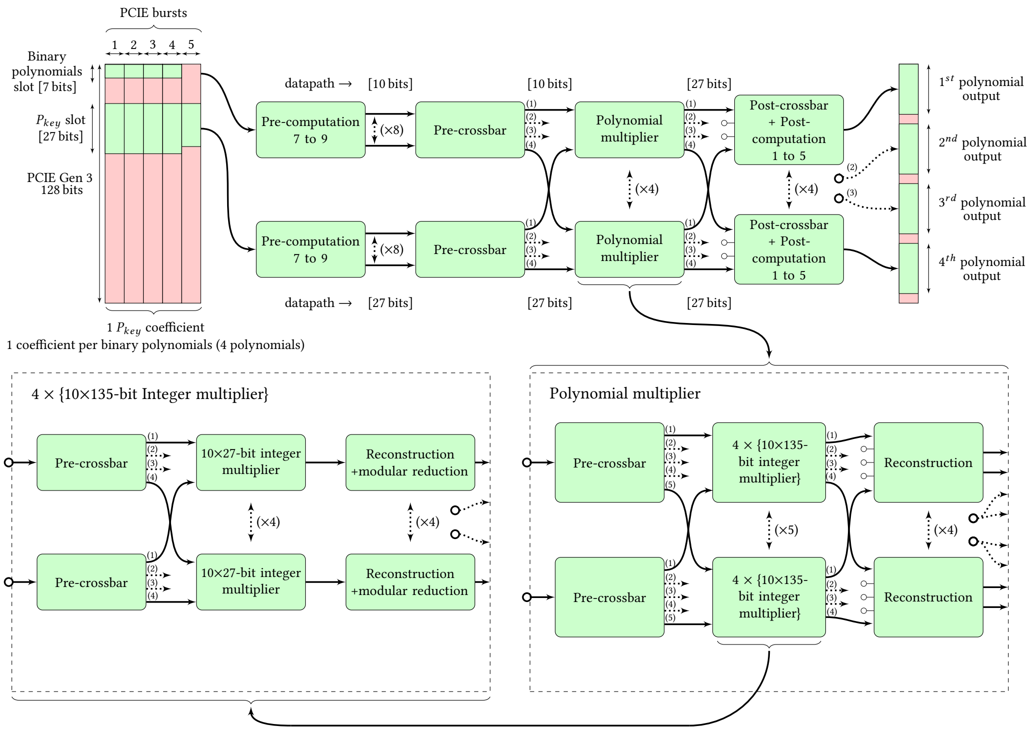

Fig. 5. Hardware accelerator architecture.

the impact of guard bits. For binary polynomials, only 1 bit per coefficient is needed, implying an inefficient use of memory and unnecessary computation overhead if the previous memory scheme is followed. Figure 4 provides an optimized representation with 20 times less memory consumption. First, the number of chunks per coefficient is reduced to one. Second, multiple coefficients are stored per chunk. Because our Karatsuba implementation has six recursions in software, at least six guard bits are required. Thus, one 32-bit chunk can store up to four coefficients. For efficiency considerations, we decided to store four polynomials using 2,560 chunks instead of one using 640 chunks.

3.2.2 Elementary polynomial arithmetic. Karatsuba pre- and post-computations are quite simple and only require polynomial additions and subtractions. That is why, our software implementation focuses on polynomial addition and subtraction. As a reminder, our setup provides five guard bits per chunk, and fifteen guard bits at the coefficient level. For addition, operations can be easily implemented. With five guard bits per chunk, five additions at the chunk level are allowed before an overlap. This effect is a consequence of the fact that an addition can increase the result by one bit in the worst case. Then, a coefficient reconstruction is required to restore the guard bits. This operation consists of taking the guard bits of one chunk, and adding them to the next one like a standard carry propagation. This operation, apart from restoring the guard bits, has the consequence of consuming the guard bits of the last chunk of each coefficient. As the addition consumes one guard bit per chunk, one guard bit at the coefficient level is consumed per addition. Therefore, our data representation supports fifteen successive polynomial additions before requiring a modular reduction of each coefficient.

The subtraction operation uses standard two's complement arithmetic. For integer subtraction of two integers A and B with $\log_2 q$ bits, two's complement subtraction can be expressed as :

$$A + \overline{B} + 1 = A - B \tag{5}$$

with $\overline{B}$ the binary inverse of B. In the standard case, carry propagation is performed during the subtraction computation. In our case, due to the guard bits, we cannot invert chunks directly. If we note t_{a_i} the guard bits of a chunk a_i and m_{a_i} the other bits, we must in fact compute the inverse of $t_{a_i} + m_{a_{i+1}}$. This operation creates data dependencies between chunks, breaking the parallelism of computations. This can be easily addressed using the following property:

$$\begin{aligned} \overline{t_{a_i} + m_{a_{i+1}}} &= -t_{a_i} - m_{a_{i+1}} - 1 \\ &= -t_{a_i} - 1 - m_{a_{i+1}} - 1 + 1 \\ &= \overline{t_{a_i}} + \overline{m_{a_{i+1}}} + 1 \end{aligned} \tag{6}$$

We just keep in mind that now the leading bit of a given chunk provides the sign. Thus, it reduces by 1 bit the number of guard bits during subtractions compared to additions.

The asymmetry between AVX2 vector length, polynomial length, and coefficient length must also be considered. An AVX2 vector has 256 bits which covers eight chunks. At the coefficient level, the fact that a AVX2 vector is longer than a coefficient is not a problem. Because coefficients are reconstructed later, a point-wise addition on the chunks is sufficient. At the polynomial level, the situation is not as simple. The computation process, and especially Karatsuba computations, leads to the creation of polynomials of various sizes. If the polynomial size is not a multiple of the AVX2 vector, a data overlap is possible. In this case, a masking operation on the latest operation is required to prevent operations and results outside of the valid data range.

3.2.3 Polynomial modular reduction. Polynomial modular reduction is performed after each polynomial multiplication. Without batching, this operation is quite simple. For example, with the cyclotomic polynomial $x^n + 1$, the reduction of a polynomial $A = A_L + A_H \cdot x^n$ is $A_L - A_H$. With batching, the cyclotomic polynomial may have numerous non-zero coefficients (Hamming weight), leading to a possible complex reduction. The standard algorithm for such an operation is the Barrett reduction [18]. However, this algorithm requires two polynomial multiplications and one polynomial subtraction which is undesirable for performance. To address the reduction in our design, we have made an exhaustive search on cyclotomic polynomials to extract good candidates compatible with batching, i.e. with the smallest Hamming weight. Then, we have exploited the structure of these polynomials to perform polynomial modular reduction as an addition/subtraction of sub-polynomials. A degree-n cyclotomic polynomial can be written as:

$$\Phi = \sum_{i=0}^{m} \alpha_i \cdot X^{\beta \cdot i}, \text{ with } \begin{cases} \alpha \in \{-1, 0, 1\} \\ (m, \beta) \in \mathbb{Z} \text{ such as } n = m \cdot \beta \end{cases} \tag{7}$$

Because α_i is an integer and Φ has been chosen to maximize β, the underlying polynomial has numerous empty coefficients which greatly simplifies the polynomial reduction.

The polynomial reduction algorithm is performed by solving a system of equations. We note A, a polynomial to be reduced by a cyclotomic polynomial Φ, B the quotient polynomial and R the residue. First, we split polynomials A, B and R into several degree-β polynomials.

$$A = \sum_{i=0}^{2 \cdot m-1} A_i \cdot X^{\beta i}, B = \sum_{i=0}^{m-1} B_i \cdot X^{\beta i}, R = \sum_{i=0}^{m-1} R_i \cdot X^{\beta i} \tag{8}$$

Second, we extract the equations system by exploiting the relationship between A, B and R:

$$A = B \cdot \Phi + R \tag{9}$$

Because $\deg R < n$, we have the following system:

$$\begin{cases} \forall i \in \{m, 2m\},\ A_i = \sum_{j=0}^{i} B_j \cdot \alpha_{i-j} & \text{(a)} \\ \forall i \in \{0, m-1\},\ A_i = \sum_{j=0}^{i} B_j \cdot \alpha_{i-j} + R_i & \text{(b)} \end{cases} \tag{10}$$

With equation 10(a), we can calculate the different B_i, and equation 10(b) determines the residue polynomial R. We have implemented a script to automatically determine the different R_i. For degree-2560 polynomials with 125-bit coefficients (22 batches), our software library performs the polynomial reduction in 114 µs on average (1000 runs). This value is competitive with Barrett reduction, because a simple polynomial multiplication using NFLlib for the same parameters costs 1.7 ms on average. To improve the performance of the polynomial modular reduction, we first compute sub-polynomial additions to maximize the number of successive operations before reconstruction. Then subtractions are performed with the limitation explained in Section 3.2.2.

3.2.4 *High-speed Batching Implementation.* One of the key elements for batching to be a good alternative to the standard approach is the batching cost. In equation 2, for k batches, possibly $2 \cdot k$ polynomial multiplications, k polynomial additions and 1 polynomial reduction are required. In our architecture, we need to avoid polynomial multiplications as much as possible for efficiency. We performed several optimizations to greatly reduce the complexity of this step. First, $S_i \cdot \Phi_i$ in equation 2 can be precomputed as they are constant for a given cyclotomic polynomial Φ. Second, messages are binary so the product $m_i \cdot S_i \cdot \Phi_i$ can be replaced by a simple test. Third, $\deg m_i = 0$, $\deg S_i = \deg \varphi_i - 1$ and $\deg \Phi_i = \deg \Phi - \deg \phi_i$, so $\deg m_i \cdot S_i \cdot \Phi_i = \deg \Phi - 1$. Thus the final polynomial reduction can be avoided. Finally, the complete process is reduced to k conditional polynomial additions, which can be quickly implemented.

3.2.5 *Karatsuba pre- and post-computation details.* Karatsuba pre- and post-computations have been implemented with a recursive algorithm which follows the approach presented in [15]. The pre-computation is quite easy to implement because operations are polynomials additions only. It is only necessary to reconstruct each coefficient after at most five successive polynomial additions, as explained in Section 3.2.2. When this operation is performed it is optimized to reduce performance overhead. The Karatsuba algorithm is composed of several successive recursions. At the i^{th} recursion, sub-polynomials are the results of at most i sub-polynomial additions. We experimented with several approaches to determine the best moment to implement the reconstruction. In our case, the best results were achieved when the reconstruction is implemented during the first Karatsuba recursion. A simple reasoning leads to this result. Between each Karatsuba recursion, the number of sub-polynomials increases but each elementary sub-polynomial has lower coefficients. However, each recursion increases the total size of sub-polynomials by 1.5. Thus, the earlier the reconstruction is performed, the smaller the amount of data that must be considered. The last recursion also requires reconstruction to support compatibility with the hardware accelerator. A minor modification to the hardware that performs this last reconstruction results in an accelerator performance improvement. Because the AVX2 vector performs addition on eight chunks in parallel, it is important to have a chunk count that is a multiple of eight to avoid masking operations. For a polynomial of 2,560 coefficients, this criterion is met during pre-computation since the smallest sub-polynomial addition has 80 coefficients after the fifth recursion. To prevent masking, four binary polynomials are stored in 2,560 chunks, as explained in Figure 3.

More issues are apparent for post-computation than for pre-computation. First, the number of coefficients is not a multiple of eight, so operations require masking. Second, data is not always aligned on 32-byte boundaries, so computation is penalized during non-aligned data loads. Third, post-computation requires successive polynomial subtractions which are subject to the limitations explained in Section 3.2.2. The final recursion of the pre-recursion and the first post-computation in software have been adapted to automatically deal with sub-polynomials in such a way that data is compatible with the PCI-E driver. The main benefit of this approach is avoiding data type conversion as much as possible.

3.3 Hardware implementation

3.3.1 Architecture overview. Software and hardware components in our system communicate via the PCI-E bus. The hardware accelerator is implemented on an FPGA (further details are provided in Section 4). A RIFFA [19] interface implemented for PCI-E Gen 3 with four lanes is used. On the FPGA side, the interface provides data bursts of 128 bits at 250 MHz. Figure 5 shows the hardware accelerator architecture. Computations are pipelined so computation operations are executed during transfers. The pre-computation units perform the remaining pre-computations and consist of three smaller units in cascade (one per recursion). For a sub-polynomial A, one smaller unit produces sub-polynomials A_L, A_H and $A_L + A_H$ (according to notation provided in Section 2.4), and pushes these polynomials to the following unit to perform one pre-computation operation on A_L, A_H and $A_L + A_H$, respectively. After three recursions, the design has eight lines of sub-polynomials but many lanes are not fully fed by sub-polynomials. Thus a crossbar is implemented to better schedule sub-polynomials and reduce the number of sub-polynomial lanes. A sub-polynomial multiplier is implemented to multiply the different sub-polynomials with the standard polynomial algorithm. The post-crossbar and the post-computation units are represented in the same box because they are implemented successively but at the recursion level to reduce the storage overhead.

Many improvements have been made to the preliminary work in [15] to support encryption operations. First, the pre-computation was adapted to efficiently pre-compute four binary polynomials in parallel during public key pre-computation. Second, the polynomial multiplier core was modified to support the multiplication of four polynomials with unbalanced coefficient sizes (10×135-bit integer multipliers). Third, five post-computations were implemented in hardware instead of three to reduce bandwidth. These new aspects of the architecture are discussed in the following sections of the paper.

3.3.2 Adaptation of the pre-computation. Two lane types are supported for the computations: one lane for the public key and a second for the binary polynomials. For the public key, coefficients are split into five 27-bit chunks. Elementary coefficient addition/subtraction is performed in five steps, which corresponds to a simple chunk-wise addition/subtraction with carry propagation. This setup is sufficient to store each coefficient and matches the software representation of polynomials, simplifying the connection with software. The binary polynomials are much smaller than standard ones and only 32 bits are required to send one coefficient of the four binary polynomials. To maximize resource utilization, one coefficient of one binary polynomial is stored per PCI-E burst. This approach allows for the pre-computation of five binary polynomials during P_{key} pre-computation. In practice, processing is only performed in four out of the five available slots because the software only pre-computes four binary polynomials.

At the beginning of the pre-computation process, each binary polynomial coefficient has at most 7-bit width. Because three pre-computations are performed in hardware, in the worst case, output

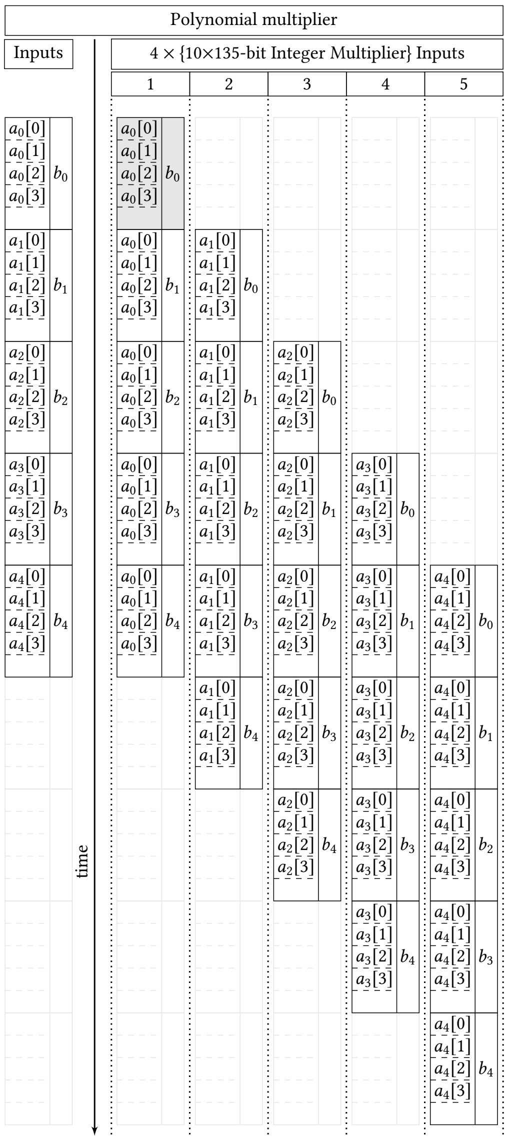
Polynomial multiplier
Inputs
4 × {10×135-bit Integer Multiplier} Inputs
1
2
3
4
5
time
$a_0[0]$ $a_0[1]$ $a_0[2]$ $a_0[3]$ b_0
$a_1[0]$ $a_1[1]$ $a_1[2]$ $a_1[3]$ b_1
$a_2[0]$ $a_2[1]$ $a_2[2]$ $a_2[3]$ b_2
$a_3[0]$ $a_3[1]$ $a_3[2]$ $a_3[3]$ b_3
$a_4[0]$ $a_4[1]$ $a_4[2]$ $a_4[3]$ b_4

Fig. 6. Polynomial multiplier schedule. $a_i[j]$ represents the i^{th} coefficient of the j^{th} binary polynomial, and b_i the i^{th} coefficient of the public key. The gray block represents an element scheduled by the 4×{10×135-bit Integer Multiplier}.

coefficients of the pre-computation unit have 10-bit widths. The addition with carry units in the previous pre-computation unit can be replaced by a simple adder and the datapath can be adjusted to 10 bits to avoid carry propagation.

3.3.3 Adaptation of the Polynomial Multiplier. The polynomial multiplier performs sub-polynomial multiplication of polynomials generated after the pre-computation process. These sub-polynomials have a low degree, four in our case, and are implemented with the standard multiplier algorithm.

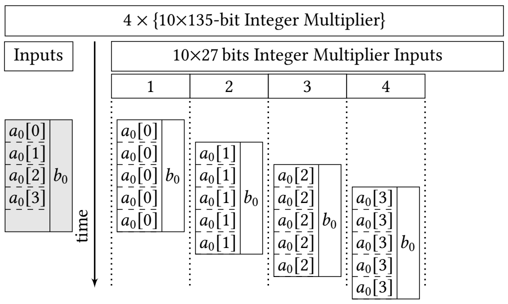
4 × {10×135-bit Integer Multiplier}
Inputs
10×27 bits Integer Multiplier Inputs
1
2
3
4
time
$a_0[0]$
$a_0[1]$
$a_0[2]$
$a_0[3]$
b_0

Fig. 7. 4×{10×135-bit Integer Multiplier} schedule. $a_i[j]$ represents the i^{th} coefficient of the j^{th} binary polynomial, and b_i the i^{th} coefficient of the public key.

For efficiency, the multiplier is fully parallelized, and as shown in Figure 5, requires five integer multiplier units. These integer multiplier units have been designed to support four 10×135-bit integer multipliers in parallel to support four encryptions. Because binary polynomial operations are scheduled to improve resource utilization, the polynomial multiplication schedule has been adapted. Both polynomial multiplier and 4×{10×135-bit Integer Multiplier} units follow the same approach: A pre-crossbar to schedule incoming data, multiplier units, and a reconstruction unit. Figures 6 and 7 provide the schedule of the polynomial multiplier and internal 4×{10×135-bit integer multiplier} units. The block of data represented in gray is an elementary block of data processed by the internal integer multipliers. From a high level point of view, the polynomial multiplier pre-crossbar is responsible for supporting the convolution operation as the integer multiplier pre-crossbar separates the coefficients of the four binary polynomials into four different lanes. In Figure 7, it is apparent that five lanes of binary polynomials are possible. However, to be compliant with our software architecture, we only use four lanes instead of five.

3.3.4 Adaptation of Post-Computation Operations. Due to limited PCI-E bandwidth, it was necessary to implement two additional post-computations in hardware beyond the standard case. In the standard case of three pre- and post-computations, input polynomials have 40 coefficients with a 131-bit width and output polynomials have 79 coefficients with a 128-bit width due to integer modular reduction. Thus, twice the bandwidth is needed for the output compared to the input. Because chunks have 27 bits, for four polynomial multiplications, $2 \times 4 \times 27 = 216$ bits are needed for the output, which is larger than the 128 bits provided by RIFFA. Depending on the number of post-computations, it is possible to have fewer output coefficients than the number of required input coefficients. This assertion becomes true when implementing the two additional post-computations in hardware in our case. Output polynomials now have 319 coefficients, and the number of input polynomials for two post-recursions is nine, leading to 360 input coefficients. This architectural modification has both pros and cons. Implementing additional post-computations in hardware speeds-up the software post-computation process and reduces the size of the transfer between the FPGA and the software. However, one needs to ensure that nine successive input polynomials can be sent to the hardware accelerator without significant latency between them. As will be explained in the next section, RIFFA input buffer management must be carefully implemented.

3.3.5 Selection of the distribution ratio between hardware and software for Karatsuba recursions. The distribution ratio of the Karatsuba recursions between hardware and software is a critical design choice. Although software can implement Karatsuba pre- and post-recursions, implementing numerous recursions in software has several limitations. First, software recursions increase the transfer size through the PCI-E (as a reminder, each recursion increases the size of the transfer by 1.5x). Since the software computation time increases, the overall computation time increases as well. Second, post-computations in software are costly compared to pre-computation. Our experiments show that post-computation can be eight times slower than pre-computation. Moreover, for four parallel encryptions, post-computation must be performed four times in software.

Implementing additional post-recursions in hardware (i.e. more than the number of pre-recursions) is not a complex task because additional post-crossbars are unneeded. The main issue is the implementation of the pre-crossbar. For fewer than three pre-recursions, there is not enough parallelism to efficiently feed hardware multipliers (multipliers are unused 25% of the time). For more than three, parallel implementation requires a complex sub-polynomial schedule which can penalize the maximum design frequency. As a result, three pre-recursions are performed in hardware to limit pre-crossbar complexity. For the post-computation, only five post-recursions are performed due to PCI-E bandwidth limitations. This implementation greatly improves upon the software performance by reducing computation time.

3.3.6 Limitations of the PCI-E interface. As mentioned earlier, a RIFFA 2.1 interface is used to make the connection between software and hardware components. The transfer bandwidth mostly depends on the size of the transfer [19]. For our system, instead of initiating one transfer per set of sub-polynomials, a large transfer with all pre-computed polynomials is initiated. As explained in the discussion in Section 3.3.1 regarding PCI-E bursts, a transfer requires the transmission of 729 degree-39 sub-polynomials, about 2.2 MB of data. With such a transfer, RIFFA should achieve a bandwidth of 3,000 MB/s which corresponds to a transfer time of 741 µs. In practice, we achieved a bandwidth of 2,000 MB, since our hardware accelerator shares the PCI-E bus with several components.

This limitation leads to two consequences. First, although the total hardware computation time (including transfers) is lengthened, the Karatsuba hardware computation can partially compensate since it can be performed during transfers. However, the transfer latency impact is not negligible. Compared to the best possible case of 583 µs to perform all hardware computations, performance is slowed down by 47%. Second, two extra components were designed to interface RIFFA to our Karatsuba accelerator. The *packager* component manages the input stream and the *buffer* component manages the output stream. The packager temporarily stores input coefficients so they can be sent to the accelerator without interruption. This approach compensates for the risk of pipeline bursts. The buffer is responsible for improving upload transfers. It stores output coefficients from the Karatsuba algorithm until it is able to perform a complete transfer to the software without interruption. This approach addresses the issue of the input stream length being larger than the output stream length. Two more post-computations than pre-computations were implemented to reduce the number of output lanes, resulting in a reduction in the size of the output stream. For 729 input sub-polynomials with 40 coefficients (29,160 coefficients in total), our Karatsuba accelerator produces 81 sub-polynomials with 319 coefficients (25,839 coefficients in total). The bandwidth issue in this case is limiting because it requires a long wait before transfer initiation, leading to the use of a large FIFO.

4 IMPLEMENTATION RESULTS

Our design has been implemented on a DE5-net platform from Terasic. The platform includes a Stratix V (GXEA7N2F45C2) FPGA, several embedded memories (SRAM, Flash), and 8GB of DDR3 memory. For communication, the DE5-net provides four SPI+ connectors, a PCI-E interface (up to 8 lanes) and one RS422. Our co-design architecture also includes a desktop computer which runs the Microsoft Windows 7 operating system on an Intel Core i7-4790. The DE5-net board was plugged into one of the PCI-E slots.

Algorithm 1 Product/accumulation using NFLlib

```
Require: $P_{key}$ (in RNS and NTT form), $U$ and $E$
1: $\widetilde{U} \leftarrow$ NTT(RNS($U$))
2: $\widetilde{E} \leftarrow$ NTT(RNS($E$))
3: $\widetilde{R} \leftarrow P_{key} \cdot \widetilde{U} + \widetilde{E}$
4: $R \leftarrow$ invRNS(invNTT($\widetilde{R}$))
5: return $R$
```

Table 1 shows the FPGA hardware resources consumption for the co-design accelerator. The contribution of RIFFA, the interface between RIFFA and the Karatsuba accelerator, and the accelerator itself are noted. The Karatsuba algorithm implementation is responsible for a substantial consumption of FPGA arithmetic logic modules (ALMs). Half of these ALMs are used as memory. This is the consequence of the pipeline cost and the temporary storage of coefficients required by the architecture. Pipelining is important to reach the minimum 250 MHz frequency imposed by the PCI-E. The substantial memory requirement of the design (more than 1 MB) is due to the interface between the multiplier and RIFFA, as described in Section 3.3.6. The buffer is responsible of more than 99% of the memory consumption. This issue reveals that the main Karatsuba limitation is the length of data transfers. For the output stream, we need to send data without interruption to maximize the bandwidth, and so a large amount of data is buffered for that purpose. To reduce the memory impact, the bandwidth itself would need to be improved, possibly by reducing the load on the PCI-E bus. It is also possible to implement additional post-computations in hardware, reducing the software post-computation time. Several issues complicate the comparison of our work with the state of the art. Because a large majority of implementations target the Negative Wrapped Convolution NTT for efficiency, which is not compatible with batching for binary messages, direct comparisons are not possible. As a result, we compared our design with software algorithms from the NFLlib library. NFLlib is also based on NTT, but it is enough flexible to be adapted for batching. To speed-up computations, NFLlib splits polynomial coefficients into small numbers using the RNS system. RNS provides an efficient strategy to parallelize computations and works similarly to CRT. Table 2 compares the total computation time of this work with NFLlib and the software memory requirements. The results are given for the computation of $P_{key} \cdot U + E$ (Algorithm 1 provides more details about this computation in NFLlib). As random number and Gaussian noise generation are not in the scope of the implementation, they are not included. Our accelerator is approximatively 1.5 times faster for 1 encryption and 4 times faster for 4 encryptions. The important drawback in NFLlib is the several computations of RNS and NTT. Karatsuba does not require such transformations.
For software memory requirements, because Karatsuba has several data dependencies between recursions, pre-allocation is required for efficiency. NFLlib also requires pre-allocated memory, but has been deported to the polynomial itself as they are mostly polynomial-dependents (NTT intermediate coefficients, RNS and NTT pre-computed constants, ...). However, the overall cost benefits to NFLlib as very few polynomials are required during encryption.
We must also notice that a pure software implementation of Karatsuba is clearly not competitive compared to NFLlib (more than 7 ms are spent just for pre- and post-computations). Thus, the hardware accelerator significantly improves computation time. This work also greatly improves the implementation in [15], the basis for our architecture. Because the architecture in [15] does not exploit the specific structure of polynomials, the computation time of the same operation costs 2.44 ms per encryption, without any asymptotic gain for several successive encryptions.

Table 1. Hardware resource consumption for the FPGA-based co-design accelerator.

		RIFFA	RIFFA/ Karatsuba interface	Karatsuba
Setup (n, $\log_2 q$)		(2560, 135)		
ALM	for logic	8,207	286	30,566
	for memory	110	0	30,030
	total	8,317	286	60,596
Registers		11,334	203	79,440
Memory bits		697,720	8,464,384	25,164
DSPs		0	0	80
f_{max}		250 MHz (PCI-E limitation)		

Table 2. Computation times and software memory requirements to perform the product/accumulation operation required during the ciphering of FV (n=2560, $\log_2 q$ = 135).
Software computations are performed on an Intel Core i7-4790.

Computation time Notation: average time (standard deviation)		
Encryptions	This work	NFLlib [20]
1	1,935 μs (220μs)	3,078 μs (98μs)
2	2,191 μs (138μs)	5,646 μs (148μs)
3	2,410 μs (176μs)	8,218 μs (191μs)
4	2,525 μs (184μs)	10,814 μs (237μs)

Software memory requirements		
	This work	NFLlib [20]
P_{key}, U, E	150 KB	480 KB
Pre-allocation	570 KB	0 KB
Total	720 KB	480 KB

5 CONCLUSION

In this paper, we described a high speed hardware/software accelerator to speed-up the ciphering operation of lattice-based cryptography and, in particular, the promising FV homomorphic scheme. This implementation focuses on batching techniques which pack several messages inside one ciphertext to reduce the ratio between encrypted data length and message length. We target polynomial arithmetic and compare the approach speed-up to a state of the art Lattice-Based arithmetic software library, NFLlib. Our accelerator is approximatively 1.5 times faster for 1 encryption and 4 times faster for 4 encryptions. To achieve such performance, a high speed software library using AVX2 has been implemented, coupled with a fully pipelined hardware accelerator to reduce the transfer latency impact between software and hardware. This paper also demonstrates the interesting use of FPGAs as peripherals to improve computation times in homomorphic operations.

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

Future work will consist of implementing the architecture on an FPGA/processor platform targeted to embedded applications.

ACKNOWLEDGMENT

This study has been partially funded by the french Direction Générale de l'Armement (DGA).

REFERENCES

[1] T. E. Gamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," in *Proc. of CRYPTO*, 1984.

[2] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *Proc. of EUROCRYPT*, 1999.

[3] C. Aguilar Melchor, P. Gaborit, and J. Herranz, "Additively Homomorphic Encryption with D-Operand Multiplications," in *Proc. of CRYPTO*, 2010.

[4] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, Stanford University, 2009.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption Without Bootstrapping," in *Proc. of ITCS*, 2012.

[6] Z. Brakerski, "Fully Homomorphic Encryption Without Modulus Switching from Classical GapSVP," in *Proc. of CRYPTO*, 2012.

[7] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," Cryptology ePrint Archive, Report 2012/144, 2012.

[8] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *Proc. of CRYPTO*, 2013.

[9] Z. Brakerski and V. Vaikuntanathan, "Lattice-Based FHE as Secure as PKE," in *Proc. of ITCS*, 2014.

[10] A. Khedr, G. Gulak, and V. Vaikuntanathan, "SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers," *IEEE Transactions on Computers*, 2015.

[11] T. Lepoint and M. Naehrig, "A Comparison of the Homomorphic Encryption Schemes FV and YASHE," in *Proc. of AFRICACRYPT*, 2014.

[12] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun, "Batch Fully Homomorphic Encryption over the Integers," in *Proc. of EUROCRYPT*, 2013.

[13] J. Pollard, "The Fast Fourier Transform in a Finite Field," in *Mathematics of Computation*, 1971.

[14] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation," in *Proc. of CHES*, 2015.

[15] V. Migliore, M. Mendez Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/Software co-Design of an Accelerator for FV Homomorphic Encryption Scheme using Karatsuba Algorithm," *IEEE Transactions on Computers*, 2016.

[16] A. Karatsuba and Y. Ofman, "Multiplication of Multi-Digit Numbers on Automata (in Russian)," *Doklady Akad. Nauk SSSR*, 1962, translation in Soviet Physics-Doklady.

[17] O. Regev, "On Lattices, Learning With Errors, Random Linear Codes, and Cryptography," *Journal of the ACM*, 2009.

[18] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Proc. of CRYPTO*, 1986.

[19] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, 2015.

[20] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, and L. T. Killijian, MArc-Olivier, "NFLlib: NTT-Based Fast Lattice Library," 2016.