# HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation

Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede

**Abstract**—In this paper, we present an FPGA based hardware accelerator 'HEPCloud' for homomorphic evaluations of medium depth functions which has applications in cloud computing. Our HEPCloud architecture supports the polynomial ring based homomorphic encryption scheme FV for a ring-LWE parameter set of dimension $2^{15}$, modulus size 1,228-bit, and a standard deviation 50. This parameter-set offers a multiplicative depth 36 and at least 85 bit security. The processor of HEPCloud is composed of multiple parallel cores. To achieve fast computation time for such a large parameter-set, various optimizations in both algorithm and architecture levels are performed. For fast polynomial multiplications, we use CRT with NTT and achieve two dimensional parallelism in HEPCloud. We optimize the BRAM access, use a fast Barrett like polynomial reduction method, optimize the cost of CRT, and design a fast divide-and-round unit. Beside parallel processing, we apply pipelining strategy in several of the sequential building blocks to reduce the impact of sequential computations. Finally, we implement HEPCloud on a medium-size Xilinx Virtex 6 FPGA board ML605 board and measure its on-board performance. To store the ciphertexts during a homomorphic function evaluation, we use the large DDR3 memory of the ML605 board. Our FPGA-based implementation of HEPCloud computes a homomorphic multiplication in 26.67 s, of which the actual computation takes only 3.36 s and the rest is spent for off-chip memory access. It requires about 37,551 s to evaluate the SIMON-64/128 block cipher, but the per-block timing is only about 18 s because HEPCloud processes 2,048 blocks simultaneously. The results show that FPGA-based acceleration of homomorphic function evaluations is feasible, but fast memory interface is crucial for the performance.

**Index Terms**—Homomorphic encryption, FV, lattice-based cryptography, ring-LWE, polynomial multiplication, number theoretic transform, hardware implementation

## 1 Introduction

The concept of fully homomorphic encryption (FHE), a form of encryption that allows evaluating arbitrary functions on encrypted data, was introduced by Rivest, Adleman, and Dertouzos [32] already in 1978. Constructing FHE schemes proved to be a difficult problem that remained unsolved until 2009 when Gentry [17] proposed the first FHE scheme by using ideal lattices. Despite its groundbreaking nature, Gentry's proposal did not provide a practical solution because of its low performance. Since then, many researchers have developed more efficient schemes to improve the performance of FHE [6], [7], [12], [16], [18], [19], [27], [37]. Despite these major advances, FHE schemes are too slow to be used in practical applications. Even somewhat homomorphic encryption (SHE) schemes, which can perform a limited number of operations on encrypted data, are also very slow. Software implementations require minutes or hours to evaluate even rather simple functions. For e.g., evaluating the decryption of a lightweight block cipher SIMON-64/128 (block/key size 64/128 bits) [3] requires 4,193 s (an hour and 10 minutes) on a 4-core Intel Core-i7 processor [23]. If FHE could achieve performance levels that would permit large-scale practical use, it would have a drastic effect on cloud computing: users could outsource computations to the cloud without the need to trust service providers and their mechanisms for protecting users' data from outsiders.

Hardware accelerators have been successfully used for accelerating performance-critical computations in cryptology already for several decades. Hence, it is not surprising that during the recent years several publications [8], [13], [14], [15], [26], [28], [31], [33], [40], [41] have reported results on hardware-based acceleration of different FHE and SHE schemes or their central operations. Technical maturity and suitability for practical deployment differ significantly between the published implementations. The parts of the schemes that are implemented in the publications range from only central operations such as large integer multiplications (see, e.g., [8]) through full implementations of the


- *S.S. Roy, J. Vliegen, and I. Verbauwhede are with KU Leuven ESAT/COSIC and IMEC, Kasteelpark Arenberg 10, Leuven-Heverlee B-3001, Belgium. E-mail: {sujoy.sinharoy, jo.vliegen, Ingrid.Verbauwhede}@esat.kuleuven.be.*
- *K. Järvinen is with the Department of Computer Science, University of Helsinki, Gustaf Hällströmin katu 2b, Helsinki FI-00014, Finland. E-mail: kimmo.u.jarvinen@helsinki.fi.*
- *F. Vercauteren is with KU Leuven ESAT/COSIC and IMEC, Kasteelpark Arenberg 10, Leuven-Heverlee B-3001, Belgium, and Open Security Research, FangDa Building 704, Kejinan-12th, Nanshan, Shenzhen 518000, China. E-mail: frederik.vercauteren@esat.kuleuven.be.*


*Manuscript received 31 May 2017; revised 11 Oct. 2017; accepted 26 Oct. 2017. Date of publication 15 Mar. 2018; date of current version 16 Oct. 2018. (Corresponding author: Sujoy Sinha Roy.)*
*Recommended for acceptance by Ç. K. Koç, Z. Liu, and P. Longa.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TC.2018.2816640*

computational parts of the schemes (see, e.g., [33]) to complete implementations in real hardware that include also all memory handling, interfacing with a host processor, etc. (see, e.g., [31]).

The published implementations focus on many different FHE and SHE schemes. In this paper, we focus on the Fan-Vercauteren (FV) SHE scheme [16], which is based on the Ring Learning with Errors (Ring-LWE) problem. To the best of our knowledge, there are no published hardware implementations of FV prior to this paper; a very recent paper by Cathébras et al. [9] analyses parameter selection of FV in the light of hardware efficiency but does not provide any actual implementation results. From the implementation point-of-view, the FV scheme is close to the Yet-Another-Somewhat-Homomorphic-Encryption (YASHE) scheme by Bos et al. [6] because the FV scheme can be implemented with certain minor modifications and additions to a YASHE architecture. Hence, the closest counterparts in the literature are the two implementations of the YASHE scheme that were published by Pöppelmann et al. in [31] and by us in [33]. Recently in 2016, Albrecht et al. [25] published a subfield lattice attack that runs in sub-exponential time on overstretched NTRU assumptions. Since the key generation part of the YASHE scheme relies on a mildly overstretched NTRU assumption, the subfield attack makes YASHE insecure. However, the FV scheme remains secure as this attack does not apply to it.

In this paper, we present HEPCloud, an FPGA-based hardware accelerator for homomorphic evaluations of medium depth functions, which supports the FV SHE scheme. It is designed primarily for speeding up homomorphic evaluations in cloud computing: the cloud service provider installs our HEPCloud accelerator on its servers and delegates the heavy homomorphic function evaluations to this FPGA-based accelerator. Our goal in designing HEPCloud is to provide a proof-of-concept implementation of a solution that is complete and mature enough for practical use. We implement all required components for homomorphic function evaluations in real hardware. The architecture of HEPCloud is based on the SHE accelerator architecture for the YASHE scheme that we introduced in our earlier work [33]. The contributions of this paper compared to previous works and, especially, [33] can be summarized as follows:

- We introduce an efficient hardware implementation of the FV SHE scheme that, to the best of our knowledge, is the first FPGA-based accelerator for FV.
- We improve the architecture from [33] to support FV by designing a unified architecture for two lifting operations and an architecture for the residue polynomial computation, and by fine-tuning the design decisions.
- We implement HEPCloud using a Xilinx Virtex-6 ML605 evaluation board and verify its operation and performance via on-board performance measurements; [33] presented only after place-and-route results and did not provide any evaluation results on real hardware.
- We discuss the feasibility of FPGA-based acceleration of homomorphic function evaluation. We conclude that despite certain obstacles, in particular, with the speed of the memory interface, FPGA-based acceleration and HEPCloud are feasible solutions for reducing the large overhead of homomorphic function evaluations in cloud computing environments.

The paper is structured as follows. Section 2 describes the FV SHE scheme as well as the system setup and the parameter set that we use. Section 3 contains a high level description of known optimization techniques to speed-up computations in modular polynomial rings and describes how we represent polynomials using the Chinese Remainder Theorem (CRT) in order to parallelize computations. We present our hardware architecture for FV in Section 4. Section 5 shows the performance results and we end the paper with the summary in Section 6.

## 2 The FV Homomorphic Encryption Scheme

In this section we briefly describe the FV somewhat homomorphic encryption scheme. The FV scheme was introduced by Fan and Vercauteren [16] in 2012. It uses a basic ring-LWE public-key encryption scheme and two additional functions Add and Mult to perform arithmetic operation on encrypted data. The polynomial ring is $R = \mathbb{Z}[x]/\langle f(x)\rangle$ with $f(x) = \Phi_d(x)$, the $d$th cyclotomic polynomial of degree $n = \varphi(d)$. The key generation and the encryption operations in FV require sampling from two probability distributions defined on $R$, namely $\chi_{key}$ and $\chi_{err}$ respectively. The security is determined by the degree $n$ of $f$, the size of the ciphertext modulus $q$, and by the probability distributions. Following [24] one may sample the key and the error polynomials from a common distribution $\chi$. Typically $\chi$ is a discrete Gaussian distribution $\chi_\sigma$ with a small standard deviation $\sigma$. However in practice the authors of FV took the private key as a polynomial with coefficients from a narrow set like $\{-1, 0, 1\}$. In the following we introduce two functions that are used to describe the FV scheme.

**Definition 2.1 (WordDecomp$_{w,q}(a)$).** *This function is used to decompose a ring element $a \in R_q$ in base $w$ by slicing each coefficient of $a$. For $v = \lceil \log_w(q) \rceil$, this function returns $a_i \in R$ with coefficients in $(-w/2, w/2]$, where $a = \sum_{i=0}^{v-1} a_i w^i$.*

**Definition 2.2 (PowersOf$_{w,q}(a)$).** *This function scales an element $a \in R_q$ by the different powers of $w$. It is defined as* PowersOf$_{w,q}(a) = (aw^i)_{i=0}^{v-1}$. *The two functions can be used to perform a polynomial multiplication in $R_q$ as*

$$\langle \mathsf{WordDecomp}_{w,q}(a), \mathsf{PowersOf}_{w,q}(b)\rangle = a \cdot b \bmod q\,.$$

*This expression has the advantage of reducing the noise during homomorphic multiplications, as the first vector contains small elements (in base $w$).*

Now we enumerate the functions used in the FV scheme. For details of the functions, interested readers may follow the original paper [16] or the presentation [38].

1) FV.ParamsGen($\lambda$): For a given security parameter $\lambda$, choose a polynomial $\Phi_d(x)$, ciphertext modulus $q$ and plaintext modulus $t$, and distributions $\chi_{err}$ and $\chi_{key}$. Also choose the base $w$ for WordDecomp$_{w,q}(\cdot)$. Return the system parameters $(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$.

Following [16] we use a uniform signed binary distribution for $\chi_{key}$.

2) FV.KeyGen($\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w$): Sample polynomial $s$ from $\chi_{key}$, sample $a \leftarrow R_q$ uniformly at random, and sample $e \leftarrow \chi_{err}$. Compute $b = [-(as+e)]_q$. The public key consists of two polynomials $pk = \{b, a\}$ and the secret key is $sk = s$. The scheme uses another key called *relinearisation key* or **rlk** in the function ReLin. This key is computed as follows: first sample $\mathbf{a} \leftarrow R_q^l$ uniformly, then sample $\mathbf{e} \leftarrow \chi_{err}^l$, and then compute $\mathbf{rlk} = \{\mathbf{rlk}_0,\ \mathbf{rlk}_1\} = \{[\mathsf{PowersOf}_{w,q}(s^2) - (\mathbf{e} + \mathbf{a} \cdot s)]_q, \mathbf{a}\} \in \{R_q^l, R_q^l\}$.
3) FV.Encrypt($pk, m$): First encode the input message $m \in R_t$ into a polynomial $\Delta m \in R_q$ with $\Delta = \lfloor q/t \rfloor$. Next sample the error polynomials $e_1, e_2 \leftarrow \chi_{err}$, sample $u$ uniformly from the signed binary distribution, and, compute the two polynomials $c_0 = [\Delta m + bu + e_1]_q \in R_q$ and $c_1 = [au + e_2]_q \in R_q$. The ciphertext is the pair of polynomials $\mathbf{c} = \{c_0, c_1\}$.
4) FV.Decrypt($sk, \mathbf{c}$): Recover the message $m = \left[\lfloor \frac{t}{q} \cdot [c_0 + sc_1]_q \rceil\right]_t$.
5) FV.Add($\mathbf{c_1}, \mathbf{c_2}$): For two ciphertexts $\mathbf{c_1} = \{c_{1,0}, c_{1,1}\}$ and $\mathbf{c_2} = \{c_{2,0}, c_{2,1}\}$, return $\tilde{\mathbf{c}}_{add} = \{c_{1,0} + c_{2,0}, c_{1,1} + c_{2,1}\}$.
6) FV.Mult($\mathbf{c}_1, \mathbf{c}_2, \mathbf{rlk}$): Compute $\tilde{\mathbf{c}}_{mult} = \{\tilde{c}_0, \tilde{c}_1, \tilde{c}_2\}$ where $\tilde{c}_0 = [\lfloor \frac{t}{q} \cdot c_{1,0} \cdot c_{2,0} \rceil]_q$, $\tilde{c}_1 = [\lfloor \frac{t}{q} \cdot (c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}) \rceil]_q$, and $\tilde{c}_2 = [\lfloor \frac{t}{q} \cdot c_{1,1} \cdot c_{2,1} \rceil]_q$. Next call the function ReLin($\tilde{\mathbf{c}}_{mult}, \mathbf{rlk}$).
7) FV.Relin($\tilde{\mathbf{c}}_{mult}, \mathbf{rlk}$): Compute a relinearised ciphertext is $\mathbf{c}' = \{c_0', c_1'\}$, where $c_0' = [\tilde{c}_0 + \langle \mathsf{WordDecomp}_{w,q}(\tilde{c}_2), \mathbf{rlk}_0 \rangle]_q$ and $c_1' = [\tilde{c}_1 + \langle \mathsf{WordDecomp}_{w,q}(\tilde{c}_2), \mathbf{rlk}_1 \rangle]_q$.

### 2.1 System Setup and Parameter Set

The polynomial ring used in the FV scheme is of the form $R = \mathbb{Z}[x]/\langle f(x) \rangle$ where $f(x)$ is a monic irreducible polynomial of degree $n$. We put no restriction on $f(x)$, which allows us to deal with any cyclotomic polynomial $\Phi_d(x)$ and thus to utilize single instruction multiple data (SIMD) operations [35], [36]. The SIMD feature embeds multiple plaintexts into different "slots" in a single ciphertext and allows evaluating a function on all of them in parallel with a single execution. Indeed to exploit the SIMD feature, we choose an irreducible polynomial $f(x)$ such that $f(x) \bmod 2$ splits into many different irreducible factors, each factor corresponding to "one slot" in the SIMD representation. It is easy to see that this excludes $f(x) = x^n + 1$ with $n$ a power of two, since it results in only one irreducible factor modulo 2. For ring-LWE based public-key encryption and signature schemes [29], [30], [34] it is very common to take $f(x) = x^n + 1$ with $n$ a power of two. This particular choice enables polynomial multiplications without reductions modulo $f(x)$. With our current choice, we achieve SIMD, but we pay in modular reductions by $f(x)$.

We use a parameter set with $d = 65535$ (and thus the degree of $f(x)$ is $32768 = 2^{15}$), $\log_2(q) = 1228$ and $\chi_{err}$ a discrete Gaussian distribution with parameter $\sigma = 50$. We choose the plaintext modulus $t = 2$, i.e., we evaluate bit-level operations. The irreducible polynomial $f(x)$ splits modulo 2 in 2,048 different irreducible polynomials, which implies that we can work on 2,048 bits in parallel using the SIMD method first outlined in [35]. Following the recent work [5] the multiplicative depth of the parameter set is 36. To estimate the security of the parameter set we took help of the tool developed by Albrecht [1]. The run time of the tool increases with the size of the parameter set, and for the chosen parameter set (which is very large) we were not able to calculate the security directly in a reasonable time from the tool. The tool can calculate the security of smaller LWE instances with $(n, \log_2(q), \sigma) = \{(2{,}048, 100, 50), (4{,}096, 188, 50), (8{,}192, 352, 50)\}$. These values are 79, 81, and 85-bits respectively. If we extrapolate in the same way, then for the parameter set of HEPCloud we get at least 85-bit security.

## 3 High Level Optimizations

To efficiently implement FV we have to analyze the two main operations in detail, namely homomorphic addition and homomorphic multiplication. Homomorphic addition is easy to deal with since this simply corresponds to polynomial addition in $R_q$. Homomorphic multiplication is much more involved and is the main focus of this paper. As can be seen from the definition of FV.Mult in Section 2, we first need to multiply the polynomials (i.e., $c_{1,0}$, $c_{2,0}$ etc.) of the input ciphertexts over integers (i.e., without reduction modulo $q$), then scale by $t/q$ and round, before mapping back into the ring $R_q$. The fact that one first has to compute the result over the integers (to allow for the scaling and rounding) has a major influence on how elements of $R_q$ are represented and on how the multiplication has to be computed.

**Algorithm 1.** Iterative NTT [11]

**Input:** Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $N-1$ and $N$th primitive root $\omega_N \in \mathbb{Z}_q$ of unity
**Output:** Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \text{NTT}(a)$

```
1  begin
2    A ← BitReverse(a)
     /* m doubles each iteration */
3    for m = 2 to N by m = 2m do
4      ω_m ← ω_N^{N/m}
5      ω ← 1
6      for j = 0 to m/2 − 1 do
7        for k = 0 to N − 1 by m do
8          /* Butterfly operation */
8          t ← ω · A[k + j + m/2]
9          u ← A[k + j]
10         A[k + j] ← u + t
11         A[k + j + m/2] ← u − t
12       ω ← ω · ω_m
13 end
```

Since each element in $R_q$ is a polynomial of degree $n-1$, the result of a polynomial multiplication (without reduction modulo $f(x)$) will have degree $2n-2$. As such we choose the smallest $N = 2^k > 2n-2$, and compute the product of the two polynomials in the ring $\mathbb{Z}_q[x]/(x^N - 1)$ by applying the $N$-point NTT (see Algorithm 1). The NTT requires the $N$th roots of unity to exist in $\mathbb{Z}_q$, so we either choose $q$ a prime with $q \equiv 1 \bmod N$ or $q$ a product of small primes $q_i$ with each $q_i \equiv 1 \bmod N$. It is the latter choice that will be used throughout this work. The product of two elements $a, b \in R_q$ is then computed in two steps: first, the product modulo $x^N - 1$ (note that there will be no reduction, since the degree of the product is small enough) is computed

using two NTT's, $N$ pointwise multiplications modulo $q$ and then finally, one inverse NTT. To recover the result in $R_q$, we need a reduction modulo $f(x)$. For this purpose, we use the Newton iteration method [39].

Note that the polynomial multiplication in FV.Mult are performed over integers. To get the benefit of NTT, we perform these multiplications in a larger ring $R_Q$ where $Q$ is a sufficiently large modulus of size $\sim 2\log_2(q)$ such that the coefficients of the result polynomials are in $\mathbb{Z}$.

### 3.1 CRT Representation of Polynomials

The biggest challenge while designing a homomorphic processor is the complexity of computation. During a homomorphic operation, computations are performed on polynomials of degree $2^{15}$ or $2^{16}$ and coefficients of size ~1,200 or ~2,500 bits. To tackle the problem of long integer arithmetic, we take inspiration from the application of the CRT [4] in the RSA cryptosystems. We choose the moduli $q$ and $Q$ as products of many small prime moduli $q_i$, such that $q = \prod_0^{l-1} q_i$ and $Q = \prod_0^{L-1} q_i$, where $l < L$. Thus any operation modulo $q$ or $Q$ maps into small computations moduli $q_i$. We use the term *small residue* to represent coefficients modulo $q_i$ and the term *large residue* to represent coefficients modulo $q$ or $Q$.

### 3.2 FV.Mult in Residue Domain

Let the two input ciphertexts be $\mathbf{c}_1 = \{c_{1,0}, c_{1,1}\}$ and $\mathbf{c}_2 = \{c_{2,0}, c_{2,1}\}$. The homomorphic multiplication steps are described below.

1) $\mathsf{Lift}_{q\to Q}$: Lift $c_{1,0}$, $c_{1,1}$, $c_{2,0}$ and $c_{2,1}$ to $R_Q$ from $R_q$, i.e., compute the additional residue polynomials moduli $q_j$ for $j \in [l, L-1]$. Since the ciphertexts are represented as residue polynomials moduli $q_i$ for $i \in [0, l-1]$ in $R_q$, we first need to compute the coefficients modulo $q$ in $(-q/2, q/2)$ by applying the CRT, and then compute the additional residue polynomials.
2) $\mathsf{PolyArithmetic}_Q$: Compute the product polynomials $\tilde{c_0} = c_{1,0} \cdot c_{2,0}$, $\tilde{c_1} = c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}$ and $\tilde{c_2} = c_{1,1} \cdot c_{2,1}$ by computing multiplications and additions of the residue polynomials moduli $q_j$ for $j \in [0, L-1]$.
3) $\mathsf{Lift}_{Q\to q}$: Apply CRT on the coefficients of the residue polynomials of $\tilde{c_0}, \tilde{c_1}$ and $\tilde{c_2}$ to get the coefficients modulo $Q$ in $(-Q/2, Q/2)$. Now compute the division-and-rounding operations to $\tilde{c_0} = \lfloor \frac{t \cdot \tilde{c_0}}{q} \rceil$, $\tilde{c_1} = \lfloor \frac{t \cdot \tilde{c_1}}{q} \rceil$ and $\tilde{c_2} = \lfloor \frac{t \cdot \tilde{c_2}}{q} \rceil$. Next, reduce the coefficients modulo $q$ in $(-q/2, q/2)$.
4) $\mathsf{ResPol}_{q\to q_i}$: Compute the residue polynomials of $\tilde{c_0}$ and $\tilde{c_1}$ modulo $q_i$ for $i \in [0, l-1]$.
5) WordDecomp: Split the coefficients of $\tilde{c_2}$ into $w$-bit words to get the vector $\tilde{\mathbf{c}}_2$ of $\lceil \lg q/w \rceil$ polynomials.
6) FV.Relin: Compute the residue polynomials for each member of $\tilde{c_2}$ and then compute $c_0' = c_0 + \langle \tilde{\mathbf{c}}_2, \mathbf{rlk}_0 \rangle$ and $c_1' = c_1 + \langle \tilde{\mathbf{c}}_2, \mathbf{rlk}_1 \rangle$ by performing arithmetic on the residue polynomials modulo $q_i$ for $i \in [0, l-1]$. This step outputs the result of the homomorphic multiplication $\mathbf{c}' = \{c_0', c_1'\}$ as a set of residue polynomials in $R_q^2$.

The polynomial arithmetic on the residue polynomials can be performed in parallel. The size of the moduli $q_i$ is an important design decision and depends on the underlying platform. We implement the hardware accelerator on the Xilinx ML605 board, which has a Virtex-6 FPGA. The FPGA provides $24 \times 17$-bit unsigned DSP multipliers to perform integer multiplications. We could implement a slightly larger integer multiplier by combining a DSP multiplier with LUT-based logic. In this work we choose 30-bit prime $q_i$ that satisfy $q_i \equiv 1 \bmod N$. The reasons for selecting only 30-bit of primes are: 1) there are sufficiently many primes of size 30-bit to compose 1,228-bit $q$ and 2,517-bit $Q$, 2) the data-paths for performing computations modulo $q_i$ become symmetric, and 3) the basic computation blocks, such as adders and multipliers of size 30-bit can be implemented efficiently using the available DSP slices and a few LUTs.

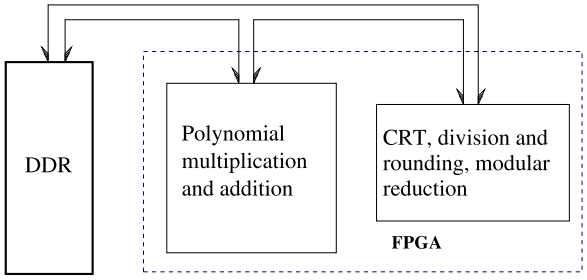



Fig. 1. High-level view of HEPCloud architecture.

## 4 Architecture

In this section we design a hardware architecture of HEPCloud to accelerate FV.Add and FV.Mult. In Fig. 1a high-level view of our architecture is shown. The computation core of HEPCloud is hosted on a medium size Xilinx Virtex-6 XC6VLX240T FPGA. The design decisions take account of the resources available on the board. Since the ciphertexts are large, of size $2 \times 4.8$ MB, we use the DDR memory of the board to store the ciphertexts. During a computation, portions of the ciphertext(s) are read from the DDR memory and stored in the on-FPGA BRAMs. After the computation, the result is written back in the DDR memory. The speed of the communication between the DDR memory and the FPGA has a major impact on the performance. In this work we restrict the data-size to 256 bits per DDR memory access. In the remaining part of this section, we describe our design decisions and optimization tricks.

### 4.1 Architecture for Polynomial Arithmetic

As discussed in Section 3.2, FV.Mult requires arithmetic with polynomials that have high degrees and large coefficient sizes. For faster multiplications of the residue polynomials in the steps $\mathsf{PolyArithmetic}_Q$ and FV.Relin of Section 3.2, we use the NTT-based polynomial multiplication algorithm. However a sequential implementation of Algorithm 1 will not be enough to accelerate FV.Mult since the parameter set (Section 2.1) that we use in this paper is very large. E.g., if we use the sequential NTT core of [34], then one $N$-point NTT of a residue polynomial will consume more than 524K cycles. Note that the NTT algorithm (Algorithm 1) is amicable to parallelism. Hence we use parallel cores to reduce the number of cycles.

#### 4.1.1 Optimization in the Routing

Using $v$ parallel cores, where $v|N$, we can split an $N$-point NTT into $v$ parallel butterfly computation threads. Let the $N$ coefficients be stored in $b$ BRAMs and $v|b$. There are two main technical issues related to the memory access that affect the performance of the NTT computation. The first one is: all the parallel cores access the BRAMs simultaneously. Since a simple dual port BRAM has one port for reading and one port for writing, it can support only one read and write in a cycle. This puts the restriction that a BRAM should be read (or written) by one core in a cycle, i.e., the generation of the BRAM-addresses by the parallel cores should be free from conflicts.

The second issue is related to the routing complexity. A residue polynomial is stored in many BRAMs, and hence, if a core needs to access a BRAM that is far from it, then the routing of wires will be very long. Note that in the basic NTT (Algorithm 1) we see that the maximum difference between the indexes of the two coefficients is $N/2$. In our parameter-set $N = 2^{16}$ and this results in a long critical path.

We address these two technical issues by connecting the read ports of a group of BRAMs to only one butterfly core. This *dedicated read* prevents any sort of conflict during the memory read operations. Additionally this helps the design tool to place the butterfly core adjacent to the proper group of BRAMs in the FPGA, thus reducing routing complexity. Memory access by a butterfly core with core-index $c$ during NTT is shown in Algorithm 2.

In Algorithm 2 $MEMORY_c$ is the group of BRAMs that are connected to the input ports of the $c$th butterfly core. Each memory word contains two residue ($2 \times 30$-bit) coefficients. In lines 13-14 two coefficient-pairs (hence four coefficients) are read from the memory and then the 'butterfly steps' (lines 16-18) are computed on them. This gives four new coefficients $s_{1,c}$, $s_{2,c}$, $s_{3,c}$, and $s_{4,c}$.

The twiddle factor $\omega$, that is used in the butterfly steps, is initialized to a constant value $\omega_{m,c}$ in line 7. The value of $\omega_{m,c}$ is actually an exponent of $\omega_m$, where the exponentiation-magnitude depends on the core index $c$. The values of $\omega_{m,c}$s can be stored in a small table or a ROM. The counter $I_{twiddle}$ denotes the interval at which $\omega$ should be updated with a new value (line 21). Whenever the number of butterfly operations ($N_{butterfly}$) becomes a multiple of $I_{twiddle}$, a new $\omega$ is computed in line 21. The addresses registers $address_1$ and $address_2$ are computed from the counters: $base$, $increment$, and $offset$, that represent the starting memory address, the increment value, and the current difference between $address_1$ and $address_2$ respectively.

The *memory-write* module in line 31 collects the coefficients generated by several of the parallel butterfly cores and writes the 'proper' coefficients in the $c$th group of BRAMs. By 'proper' we mean the coefficients that will be read by the $c$th butterfly core in the next iteration of the $m$-loop in Algorithm 2. The proper coefficients are filtered by observing the value of a variable $gap$ that depends on $c$, $m$ and $v$.

*Discussion.* Since a butterfly core is attached to a fixed set of BRAMs, our Algorithm 2 minimizes the critical paths (and hence routing) that lie between the output ports of the BRAMs and the input ports of the butterfly core. However the input ports of the BRAMs still remain connected to multiple butterfly cores. The routing delay at the input ports might be reduced by inserting pipeline-registers in the critical paths that start from the output ports of the butterfly cores and end at the input ports of the memories. We consider this as a future work. Note that if Algorithm 1 is used to implement HEPCloud then long critical paths would appear at both the input and output ports of the BRAMs.

**Algorithm 2.** Routing Efficient Parallel NTT [33]

```
    /* Core with index c for butterfly computations */
 1  module butterfly-core(parameter c; output: m, address_1,
    address_2, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c})
 2  begin
 3    (I_twiddle, offset) ← (N/2, 1)
 4    for m = 0 to log N − 1 do
 5        ω_m ← 2^m th primitiveroot(1)
 6        N_butterfly ← 0 /* Counts butterfly op. in m-loop */
 7        ω ← ω_{m,c}    /* Power of ω_m for a core-index c */
 8        for base = 0 to base < offset do
 9          increment ← 0
10          while base + offset + increment < N/2v do
11            address_1 ← base + increment
12            address_2 ← base + offset + increment
              /* Reads only cth group of BRAMs              */
13            (t_1, u_1) ← MEMORY_c[address_1]
14            (t_2, u_2) ← MEMORY_c[address_2]
15            if m < log N − 1 then
              /* Comp. two butterfly op.                    */
16            (t_1, t_2) ← (ω · t_1, ω · t_2)
17            (s_{1,c}, s_{2,c}) ← (u_1 + t_1, u_1 − t_1)
18            (s_{3,c}, s_{4,c}) ← (u_2 + t_2, u_2 − t_2)
19            N_butterfly ← N_butterfly + 2
20            increment = increment + 2 · offset
21            if I_twiddle | N_butterfly then ω ← ω · ω_m^{v/2}
22          else
23            t_1 ← ω · t_1; ω ← ω · ω_m^{v/2}
24            t_2 ← ω · t_2; ω ← ω · ω_m^{v/2}
25            (s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}) ← (u_1 + t_1, u_1 − t_1, u_2 + t_2, u_2 − t_2)
26            N_butterfly ← N_butterfly + 2
27            increment = increment + 2 · offset
28      I_twiddle ← I_twiddle/2
29      if offset < v/2 then offset ← 2 · offset
30  end
    /* Assembles output coefficients from two butter-
       fly cores and writes in the cth group of BRAMs  */
31  module memory-write(parameter c; input: m, address_1,
    address_2, s_{1,0}, ... s_{4,v−1})
32  begin
33    if 2^m < v/2 then gap ← 2^m
34    else gap ← v/2        /* Index gap between two cores */
35    if c < v/2 then
36      MEMORY_c[address_1] ← (s_{2,c}, s_{1,c})
37      MEMORY_c[address_2] ← (s_{2,c+gap}, s_{1,c+gap})
38    else
39      MEMORY_c[address_1] ← (s_{4,c}, s_{3,c})
40      MEMORY_c[address_2] ← (s_{4,c+gap}, s_{3,c+gap})
41  end
```

#### 4.1.2 Architecture of Polynomial Arithmetic Unit (PAU)

In Fig. 2 we show the internal architecture of the cores that we use to perform arithmetic on the residue polynomials. The cores have been designed following the footprints of the polynomial arithmetic core of [34]. The input register

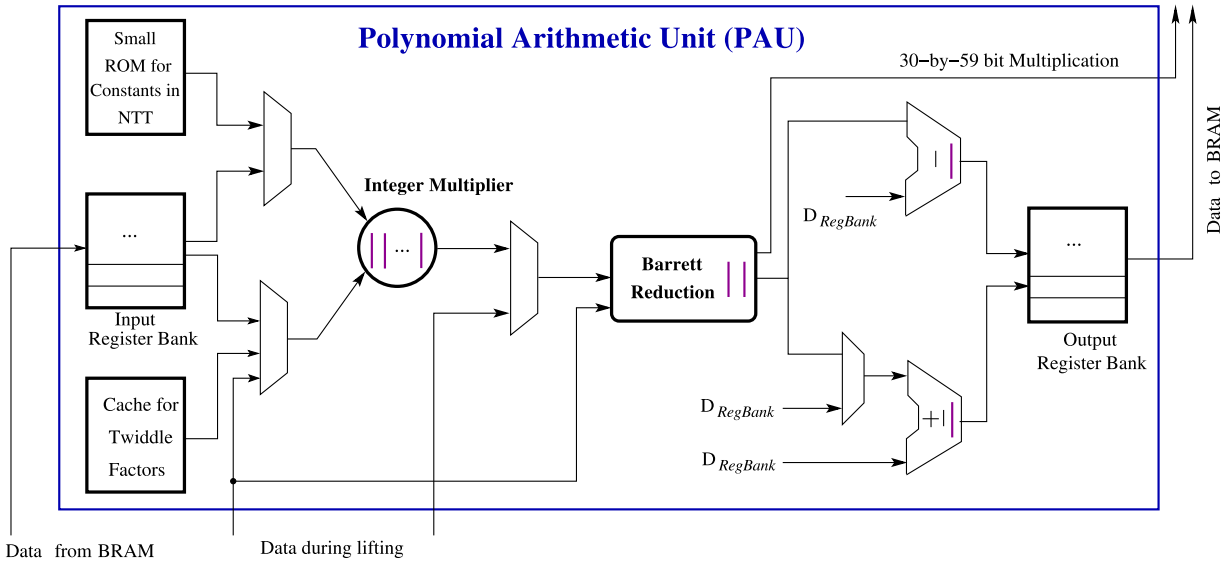



Fig. 2. Architecture for the vertical cores [33].

bank contains registers to store data from the BRAMs. In addition, the register bank also contains shift registers to delay the input coefficients in a pipeline during an NTT computation. The register bank has several ports to provide data to several other components present in the core. We use the common name $D_{regbank}$ to represent all data-outputs from the register bank. The small ROM block in Fig. 2 contains the twiddle factors and the value of $N^{-1}$ to support the computation of NTT and INTT.

The integer multiplier (shown as a circle in Fig. 2) is a $30 \times 30$-bit multiplier. We maintain a balance between area and speed by combining two DSP multipliers and additional LUT based small multipliers to form this multiplier. After an integer multiplication, the result is reduced using the Barrett reduction circuit [2] shown in Fig. 2. We use the Barrett reduction technique due to two reasons. The first reason is that the primes used in this implementation are not of pseudo-Mersenne type which support fast modular reduction technique [20]. The second reason is that the cores are shared by all the prime moduli, and hence, a generic reduction circuit is more preferable than several dedicated reduction circuits. The Barrett reduction circuit is bit parallel to process the outputs from the bit-parallel multiplier in a flow. The reduction consists of three $31 \times 31$-bit multipliers and additional adders and subtractors. The multipliers are implemented by combining two DSP multipliers with additional LUTs. Thus in total, the Barrett reduction block consumes six DSP multipliers. Beside performing the modular reduction operations, the multipliers present in the Barrett reduction circuit can be reused to perform $30 \times 59$-bit multiplications during the CRT computations.

The adder/subtracter circuits after the Barrett reduction block in Fig. 2 are used to compute the butterfly operations during an NTT computation and to perform coefficient-wise additions and subtractions of polynomials. Finally, the results of a computation are stored in the output register bank and then the registers are written back in the memory. To achieve high operating frequency, we put pipeline registers (shown as magenta colored lines) in the data paths of the computation circuits.

**Algorithm 3.** Calculation of the Reverse of an Index. A Polynomial of $2^{16}$ Coefficients is Stored as $2^{15}$ Coefficient Pairs in 16 Memory Elements. BiasTable Contains 16 Bias Values {0, 16, 8192, 8208, 4096, 4112, 12288, 12304, 2048, 2064, 10240, 10256, 6144, 6160, 14336, 14352} Corresponding to the 16 Memory Elements

**Input:** An index of a coefficient pair where $index \in [0, 2^{15} - 1]$
**Output:** Reverse of the index $reverse\_index \in [0, 2^{15} - 1]$

**begin**
  $c \leftarrow index \gg 11$ /* index of the memory element */
  $low \leftarrow index \& 31$ /* least five bits */
  $high \leftarrow (i \gg 5) \& 63$ /* next six bits */
  $lsb \leftarrow low[0]$
  $low \leftarrow low - lsb$
  $low\_reverse \leftarrow bitreverse(low)$ /* bits in reverse order */
  $high\_reverse \leftarrow bitreverse(high)$
  $bias \leftarrow BiasTable[c]$
  $reverse\_index \leftarrow bias + (high\_reverse \ll 5) + low\_reverse + (lsb \ll 14)$
**end**

### 4.1.3 External Memory Access During NTT

During an NTT, the coefficients of the residue polynomial are read sequentially from the DDR memory and then loaded in the 16 internal memory blocks. For this purpose the 256-bit DDR3 interface is used to receive four coefficient pairs (i.e., eight coefficients) in a burst. However Algorithm 2 generates the output coefficient pairs in a permutation that is different from their initial arrangement. The coefficients pairs are written back in the DDR memory in the right arrangement using Algorithm 3. For a write address $index$, the coefficient pair from the address $reverse\_index$ should be read from the internal memory. Note that we perform this rearrangement of the coefficients after the completion of an NTT following Algorithm 2; whereas in the traditional NTT Algorithm 1 this rearrangement is performed in the beginning using the *bitreverse* function.

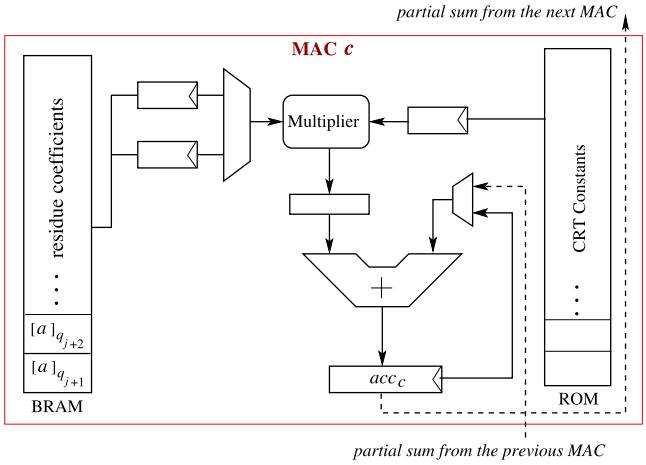



Fig. 3. Architecture for computing sum of products.

## 4.2 Architecture for Lifting Back and Forth in $R_q \leftrightarrow R_Q$

In Section 3 we described the lifting operations that we need to perform during FV.Mult. In this section we describe the computational steps that we follow to implement the lifting operations, and then we describe the hardware architectures of the building blocks. In the end we design a unified architecture for computing the two lifting operations.

### 4.2.1 Computation Steps for $\mathsf{Lift}_{q\to Q}$

Let for an integer $a \bmod q$, the residues be $[a]_{q_i}$ for $i \in [0, l-1]$. So we are interested in computing $[a]_{q_j}$ for $j \in [l, L-1]$. We first compute the sum of products for $i \in [0, l-1]$ as follows:

$$a_{sp} = \sum [a]_{q_i} \cdot \left(\frac{q}{q_i}\right) \cdot \left[\left(\frac{q}{q_i}\right)^{-1}\right]_{q_i} = \sum [a]_{q_i} \cdot b_i. \quad (1)$$

Next we compute $[a']_{q_j}$ for $j \in [l, L-1]$ using

$$[a']_{q_j} = \left[\sum [a]_{q_i} \cdot [b_i]_{q_j}\right]_{q_j}. \quad (2)$$

Note that $[b_i]_{q_j}$ are 30-bit integers. Finally, we compute the residues $[a]_{q_j}$ for $j \in [l, L-1]$ using the following equation:

$$[a]_{q_j} = \left[[a']_{q_j} - [\lfloor a_{sp}/q \rfloor]_{q_j} \cdot [q]_{q_j} - sign \cdot [q]_{q_j}\right]_{q_j}. \quad (3)$$

This computation involves a division of $a_{sp}$ by $q$. The $sign$ takes a value 0 or 1 depending on $a_{sp} - \lfloor a_{sp}/q \rfloor \cdot q$ is smaller than $q/2$ or not.

### 4.2.2 Computation Steps for $\mathsf{Lift}_{Q\to q}$

We compute the sum of products $a_{sp}$ from the residue polynomials moduli $q_j$ for $j \in [0, L-1]$

$$a_{sp} = \sum [a]_{q_j} \cdot \left[\left(\frac{Q}{q_j}\right)^{-1}\right]_{q_j} \cdot \left(\frac{Q}{q_j}\right) = \sum [a']_{q_j} \cdot b_j. \quad (4)$$

Here the values $[(\frac{Q}{q_j})^{-1}]_{q_j}$ are 30-bit integers and hence the computation $[a']_{q_j} = [a]_{q_j} \cdot [(\frac{Q}{q_j})^{-1}]_{q_j}$ is a 30-bit modular multiplication. Next we reduce $a_{sp}$ by $Q$ and get $a_Q$ in $(-Q/2, Q/2)$. Then the division and rounding operation is performed on $a_Q$ and the result is reduced modulo $q$ to a value in $(-q/2, q/2)$.

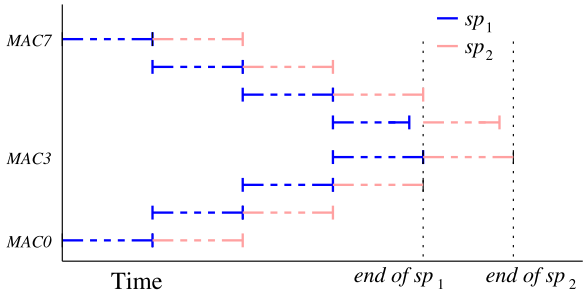



Fig. 4. Timing diagram for pipeline processing of two consecutive sum-of-products ($sp$) by the first MAC-group.

## 4.3 Unified Architecture

Note that $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$ operations involve similar computation steps such as sum of products in Eqs. (1), (2) and (4), and divisions by $q$. Hence we design a unified architecture to compute both $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$. The architecture is composed of four blocks: 1) sum of products, 2) reduction modulo $Q$, 3) division-and-rounding, and 4) reduction modulo $q$. The blocks are described as follows.

### 4.3.1 Sum of Products Block

Fig. 3 shows a multiply-and-accumulate (MAC) core to compute the sum of products in Eqs. (1), (2) and (4). In the figure, the 'multiplier' block is borrowed from the PAU (Fig. 2). Since there are 16 PAU cores in the HE-processor, we instantiate 16 MAC cores. These cores are divided into two parallel MAC-groups: MAC-0 to MAC-7 form the first group, and MAC-8 to MAC-15 form the second group. Each MAC-group is responsible for computing one sum of products.

The ROM block in the MAC core is a loadable memory and is used to store the constants for $\mathsf{Lift}_{q\to Q}$ or $\mathsf{Lift}_{Q\to q}$. We set the word size of the ROM to 59 bits. Note that Eqs. (1) and (4) require multiplications of 30-bit coefficients $[a]_{q_i}$ by long $b_i$. The MAC cores compute these long multiplications word-serially using the $31 \times 59$-bit integer multipliers that are present inside the multiplier blocks. Algorithm 4 shows the word-serial computation of the sum of products by the 0th MAC core. In the algorithm we have assumed that the MAC core is responsible for the accumulation of first $m$ products and each $b_i$ has $w$ 59-bit words in the ROM. The $k$-loop computes the $k$th 59-bit word of the partial result in $sum_0$. Whenever a word is computed in $sum_0$, it is forwarded to MAC-1. Now MAC-1 computes Algorithm 4 with the initialization of $acc_1$ to $sum_0$ and computes the words of the partial sum-of-products in $sum_1$. Following the same sequence MAC-2 computes $sum_2$ and then MAC-3 computes $sum_3$. In parallel to this computation-flow, MAC-7 downto MAC-4 compute $sum_4$. Finally $sum_4$ is added with $sum_3$ in MAC-3 to get a word of the final sum of products. In Fig. 4 we show the timing diagram for the pipeline processing.

The computation of Eq. (2) requires sum of modular multiplications. For this purpose the modular-multiplier circuit from PAU is used.

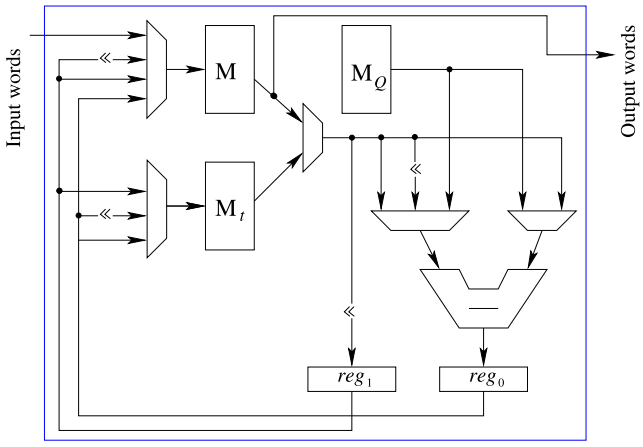



Fig. 5. Architecture for reduction modulo $Q$.

**Algorithm 4.** Partial Sum in Eq. (1) by MAC Core-0

**Input:** Residue coefficients $[a]_{q_j}$ and constants $b_j$
**Output:** Partially accumulated sum of products

**begin**
  $acc_0 \leftarrow 0$
  **for** $k = 0$ *to* $w - 1$ **do**
    **for** $j = 0$ *to* $m - 1$ **do**
      $acc_0 \leftarrow acc_0 + [a]_{q_j} \cdot b_j[k]$
    $sum_0 \leftarrow least59bits(acc_0)$
    $acc_0 \leftarrow acc_0 \gg 59$
  $sum_0 \leftarrow least59bits(acc_0)$
**end**

### 4.3.2 Reduction Modulo $Q$ Block

Let $a_{sp}$ be the sum of the products in Eq. (4). For the chosen parameter set, $a_{sp}$ is 7 bits larger than $Q$. We first reduce $a_{sp}$ to a value in $[0, Q-1]$ and then central-lift the result to a value in the range $(-Q/2, Q/2)$. To reduce $a_{sp}$ in $[0, Q-1]$, we sequentially reduce the extra bits of $a_{sp}$ from the most significant side. The steps are shown in Algorithm 5.

**Algorithm 5.** Reduction Modulo $Q$

**Input:** An integer $a_{sp}$ that is 7-bit larger than $Q$
**Output:** Integer $a_{sp}$ in $[0, Q-1]$

**begin**
  $Q_n \leftarrow (Q \ll 7)$
  **for** $i = 0$ *to* 6 **do**
    **if** $msb(a_{sp}) > 0$ **then**
      $temp \leftarrow a_{sp} - Q_n$
      **if** $temp \geq 0$ **then**
        $a_{sp} \leftarrow temp$
    $a_{sp} \leftarrow (a_{sp} \ll 1)$
  $a_{sp} \leftarrow (a_{sp} \gg 7)$
  **if** $a_{sp} \geq Q$ **then**
    $a_{sp} \leftarrow a_{sp} - Q$
**end**

A word-serial architecture for computing the reduction modulo $Q$ is shown in Fig. 5. The architecture has three addressable memory components: $M$ and $M_t$ are used to store the computational data and $M_Q$ is used to keep the modulus $Q$. These memory components are distributed RAMs of word size 59-bits and depth 64. At the beginning of a computation, the input number $a_{sp}$ is loaded in $M$. Then within the *for*-loop of Algorithm 5, the words of $a_{sp}$ are left-shifted by one position and then stored in $M_t$. Note that, line 5 has a conditional subtraction operation. In our implementation, the subtraction is performed word-serially using the subtraction circuit, and the result words are left-shifted by one position and then stored in $M$. Based on the sign of the subtraction, either $M$ or $M_t$ is used as the source of $a_{sp}$ for the next computations.

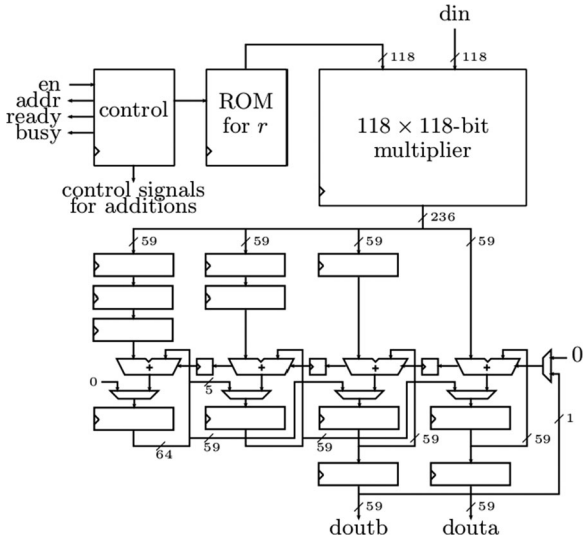



Fig. 6. Division and rounding unit (DRU) [33].

### 4.3.3 Division and Rounding Unit (DRU)

The DRU computes $\lfloor tc/q \rceil$ during $\mathsf{Lift}_{Q \to q}$ where $t = 2$, $c$ is a coefficient computed from the reduction modulo $Q$, and $\lfloor \cdot \rceil$ denotes rounding towards the nearest integer. The division is carried out by precomputing the reciprocal $r = 2/q$ and then computing $r \times c$. The word size of the DRU is 118 bits $(2 \times 59)$ as a compromise between area and latency.

To round a division of two $k$-bit integers correctly to $k$-bits, the quotient must be computed correctly to $2k + 1$ bits [22, Theorem 5.2]. In our case, the computation of $\lfloor tc/q \rceil$ requires a division of a $k_1$-bit dividend by a $k_2$-bit divisor. The precision that we will need in this case to guarantee correct rounding, based on the above, is $k_1 + k_2 + 1$ bits. The divisor $q$ is a 1,228-bit constant integer and the dividend $c$ is an at most 2,517-bit integer, which gives a bound of 3,746 bits. Hence, the reciprocal $r$ is computed up to a precision of 32 118-bit words, of which 22 words are nonzero.

Fig. 6 shows the architecture of the DRU. The multiplication $r \times c$ is computed by using a $118 \times 118$-bit multiplier that computes $22^2 = 484$ partial multiplications. This multiplier performs a 118-bit Karatsuba multiplication by using three $59 \times 59$-bit multipliers generated with the Xilinx IP Core tool (which supports only up to 64-bit multipliers). The 59-bit multipliers each require 16 DSP blocks giving the total number of 48 DSP blocks. In order to achieve a high clock frequency, the 118-bit multiplier utilizes a 23-stage pipeline, of which 18 stages are in the 59-bit multipliers (the optimal number according to the tool).

The partial products from the 118-bit multiplier are accumulated into a 241-bit $(2 \times 118 + 5)$ register using the Comba Alg [10]. These additions are computed in a 4-stage

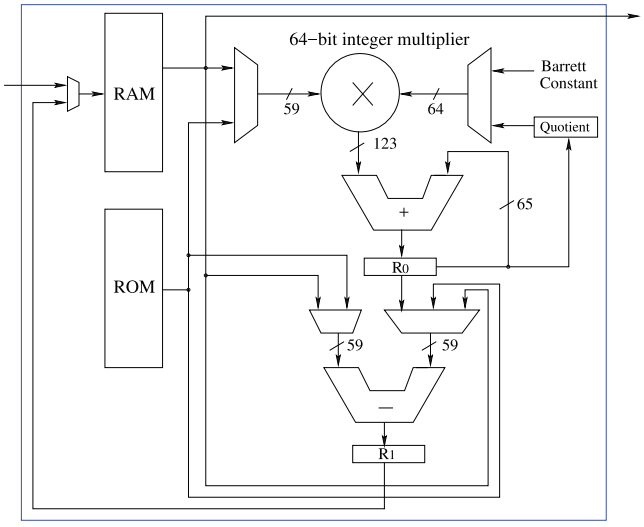



Fig. 7. Architecture for reduction modulo $q$.

pipeline with three 59-bit adders and one 64-bit adder, which are all implemented with LUTs. Whenever all partial products of an output word have been computed, the register is shifted to the right by 118 bits and the overflowing bits are given at the output of the DRU. Once the computation proceeds to the first word after the fractional point, then the MSB of the fractional part is added to the register in order to perform the rounding. The DRU has a constant latency of 687 clock cycles per coefficient.

The DRU is reused for computing $\lfloor a_{sp}/q \rfloor$ during the $\mathsf{Lift}_{q\to Q}$. The computation proceeds analogously to the above. The differences are that the reciprocal is now $r = 1/q$ and it needs to be computed only to a precision of 2,493 bits (12 nonzero words) because $c$ can be only 36 bits longer than $q$. The computation has a latency of 246 clock cycles.

### 4.3.4 Reduction Modulo $q$ Block

This block reduces 1,291-bit output from the DRU by 1,228-bit modulus $q$. Since the input data is 63-bit larger than the modulus, a bit-by-bit modular reduction architecture similar to Fig. 5 will be slow. Hence, we use a word-serial Barrett reduction algorithm to perform the reduction in $[0, q-1]$ and then we center-lift the reduced data to $(-q/2, q/2)$. The architecture of this block is shown in Fig. 7.

The input data is stored in the *RAM* and the modulus $q$ is kept in the *ROM*. In the first step, the quotient is computed in the *Quotient* register by multiplying the RAM content by the Barrett constant. For this purpose the 64-bit multiplier (Fig. 7) is used. To save area, we designed this 64-bit multiplier circuit using a 32-bit multiplier. The 64-bit multiplication is computed in four cycles. The accumulation of the word-serial 64-bit multiplication results is done by the adder block which uses a 64-bit adder circuit to compute the addition in two cycles.

After the quotient is computed in the *Quotient* register, it is multiplied with the words of the modulus $q$ from the *ROM* block and then the result words (in the $R_0$ register) are subtracted from the words of the input data (from the *RAM* block). The result of the subtraction, which is the partially reduced result, is then written back in the *RAM* module.

After the partially reduced result is compared with the modulus $q$ by performing word-serial subtractions. Based on the comparison, the conditional subtraction of the modulus $q$ is computed. With this, we get the modulus $q$-reduced result in the RAM block. Next the result is center-lifted to $(-q/2, q/2)$.

### 4.3.5 Integration of the Building Blocks

Now we describe how we integrate of the building blocks to compute the $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$ operations. Note that in the first MAC-group, MAC-3 computes the final sum of products. So the reduction and division blocks are attached to the MAC-3 core. In Fig. 8 we show the connection of MAC-3 core with the remaining three blocks. Similarly in the second MAC-group, MAC-11 core is accompanied by the reduction and division blocks. The four blocks are in a pipeline during $\mathsf{Lift}_{Q\to q}$ to achieve optimum computation time. The division block takes the maximum cycles and hence determines the throughput of the entire pipeline. Every block contains additional memory elements to enable the pipeline processing: while one memory element is read by the next block in the pipeline, the other memory element is used to store the new results.

During $\mathsf{Lift}_{q\to Q}$ operation, the sum of products $a_{sp}$ in Eq. (1) is computed by a MAC-group, and then it is passed to the DRU for the computation of $\lfloor a_{sp}/q \rfloor$. In parallel to this division, the MAC-group computes $[a']_{q_j}$ in Eq. (2). For the computation of Eq. (3), a small computation block (consisting of a multiplier, subtracter and some small memory components) is used in the pipeline. The $sign$ is computed by performing arithmetic on the most significant words of $a_{sp}$ and $q$. This block is common to both the MAC-groups as the amount of computation in Eq. (3) is small. The throughput of the pipeline during $\mathsf{Lift}_{q\to Q}$ is determined by the 'computation of $a_{sp}$ followed by the division $\lfloor a_{sp}/q \rfloor$'.

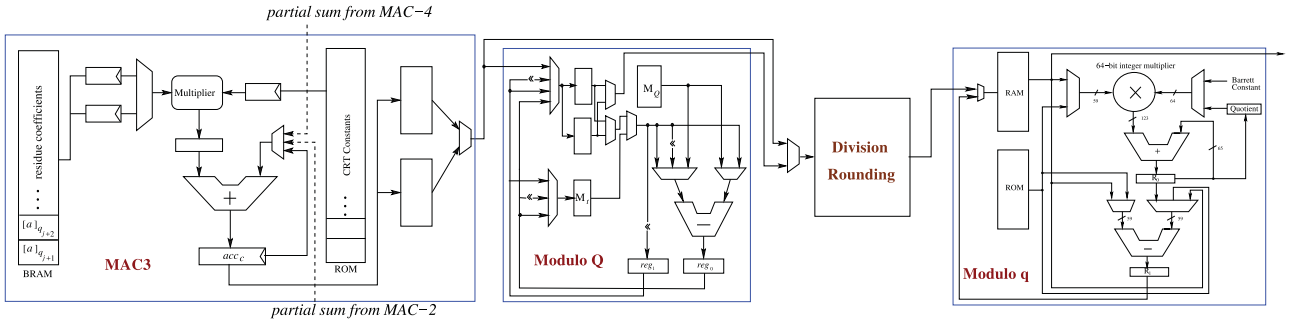



Fig. 8. Unified architecture for $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$.

#### 4.3.6 External Memory Access

The DDR memory access during the $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$ is more complicated than the memory access during NTT. Here we need to fetch the residue coefficients for different moduli, whereas during an NTT we fetch coefficients from a single moduli. So we design a customized DDR memory access interface for the lifting operations. Since the DDR-burst data length is 256 bits, at a time we read eight coefficients for a single residue from the DDR memory and copy them in the BRAM. Eight lifting operations are computed by the two MAC-groups, i.e., four lifting operations per MAC-group, before writing back the result in the DDR memory.

After eight $\mathsf{Lift}_{q\to Q}$, the result is a collection of $43 \times 8$ coefficients. This is because there are extra 43 moduli in $Q$ and for each moduli there are eight coefficients. Hence 43 DDR-write operations (each copying eight coefficients) are performed to copy the result to the memory. After every eight $\mathsf{Lift}_{Q\to q}$ during the computation of $\tilde{c}_2$, the result is a collection of eight coefficients, each of size 1,228 bits. Now WordDecomp slices each 1,228-bit coefficient into 21 59-bit coefficients. Since a single DDR-write operation copies four sliced coefficients, a total of 42 DDR-write operations are performed.

### 4.4 Architecture of the $\mathsf{ResPol}_{q\to q_i}$ Block

Step 4 in Section 3.2 computes the residue polynomials for $\tilde{c}_0$ and $\tilde{c}_1$. This step is performed by reducing the 1,228-bit coefficients of $\tilde{c}_0$ and $\tilde{c}_1$ by the 30-bit moduli $q_i$ for $i \in [0, l-1]$. In Algorithm 6 we show the steps that we follow to reduce a 1,228-bit coefficient by a 30-bit $q_i$.

**Algorithm 6.** Reduction of 1,228-Bit Coefficient by 30-Bit $q_i$

**Input:** 1228-bit coefficient $a \bmod q$
**Output:** $a \bmod q_i$
1 **begin**
2 $A \leftarrow [a_0, \ldots, a_{40}]$ /* 30-bit words of $a$ */
3 $R \leftarrow 0$
4 **for** $j = 0$ *to* 40 **do**
5 $R \leftarrow R + A[j] \cdot (2^{30 \cdot j} \bmod q_i)$
/* $R$ contains a 66-bit value */
6 $R \leftarrow R[57:0] + R[65:58] \cdot (2^{58} \bmod q_i)$
/* Now $R$ contains a 59-bit value */
7 $R \leftarrow Barrett\ Reduction59bit(R, q_i)$
8 **return** $R$
9 **end**

In line 2 of Algorithm 6 the input 1,228-bit coefficient $a$ is split into 30-bit words. The 41 words of $a$ are stored in the array $A$ with the least significant word in the index position 0 and the most significant word in the index position 40. Next the *for*-loop in line 4 multiplies the words $A[j]$ with the constants $2^{30 \cdot j} \bmod q_i$ and the multiplication results are accumulated in the register $R$. After the completion of the *for*-loop, $R$ contains a partially reduced result which is of size 66 bits. Next the most significant 8 bits of $R$ is multiplied with $2^{58} \bmod q_i$ and the result is added with the least significant 58 bits of $R$ to get a 59-bit partially reduced value in $R$. This value is reduced by a Barrett reduction circuit of 59-bit input size to get the final 30-bit result modulo $q_i$.

The architecture of the $\mathsf{ResPol}_{q\to q_i}$ block that computes Algorithm 6 is shown in Fig. 9. The input to the $\mathsf{ResPol}_{q\to q_i}$ block is the output of the *Reduction modulo q* block in Fig. 8.

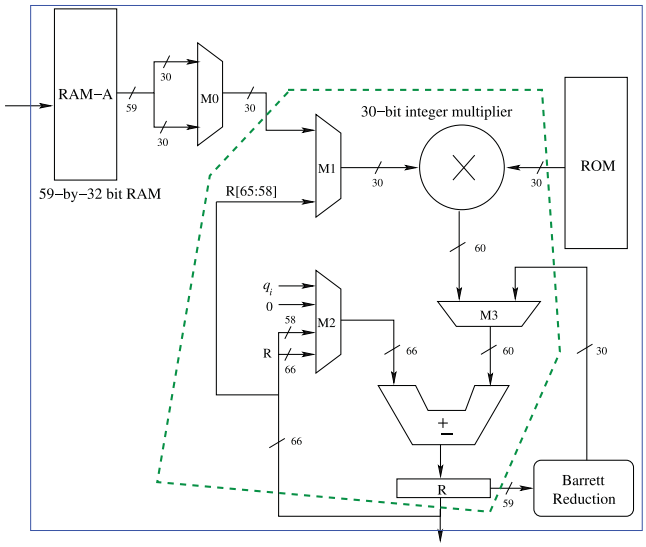



Fig. 9. Architecture of the $\mathsf{ResPol}_{q\to q_i}$ block to compute Algorithm 6.

Since the *Reduction modulo q block* outputs in 59-bit words, we use a BRAM *RAM-A* of word size 59 and depth 32 to store the 59-bit words of the input. The constants that are used in Algorithm 6 are kept in *ROM*. During the execution of the *for*-loop in Algorithm 6, a 59-bit word is fetched from *RAM-A*, then split into two 30-bit chunks. The first chunk is then multiplied with a 30-bit constants from *ROM* using the 30-by-30 bit integer multiplier and result is accumulated in *R*. Next the second chunk is multiplied by a constant and then accumulated in *R*. Following a similar way, all the words of the 1,228-bit coefficient are processed. This gives a 66-bit partially reduced result in *R*. Now the most significant 8 bits of *R* are multiplied by $2^{58} \bmod q_i$ and then added with the least significant 58-bits of *R*. The output of the addition is then reduced using the *Barrett Reduction* circuit to obtain the final 30-bit modulo $q_i$ reduced result. Since the *Barrett Reduction* circuit is used only once in Algorithm 6, the 30-bit integer multiplier in the figure is actually borrowed from the *Barrett Reduction* circuit (which contains three such multipliers).

Note that $\mathsf{ResPol}_{q\to q_i}$ gets its input from the *Reduction modulo q*. Hence the input has a sign. To keep the description simple, we do not sign of the input in Algorithm 6. But in the actual implementation the sign of the input is taken care of: depending on the sign bit, the architecture either adds the output of the *Barrett Reduction* circuit to 0 or subtracts it from $q_i$.

Since the $\mathsf{ResPol}_{q\to q_i}$ block processes the output coefficients from the architecture of Fig. 8, the best computation time can be achieved if it is kept in a pipeline. In that case, the $\mathsf{ResPol}_{q\to q_i}$ block should be fast enough to reduce the 1,228-bit input coefficient by all of the 41 moduli $q_i$ for $i \in [0, l-1]$ before a new 1,228-bit coefficient arrives in the input. We observed that a single instance of the architecture shown in Fig. 9 is not fast enough to meet the throughput of the pipeline. So we replicate the part of the architecture that is present inside the green dashed-block of Fig. 9 for four times. Since we borrow three 30-bit multipliers from the *Barrett Reduction* circuit, we instantiate one extra multiplier. The four instances run in parallel and distribute the computation job: the first one reduces the input coefficient by first

TABLE 1
Area Results on Xilinx Virtex-6 XC6VLX240T-1FF1156 FPGA

| Resource | Used | Avail. | Percentage |
|---|---|---|---|
| Slice Registers | 63,086 | 301,440 | 20.9% |
| Slice LUTs | 72,613 | 150,720 | 48.2% |
| BlockRAM | 84 BRAM36, 22 BRAM18 | 416 | 22.8% |
| DSP48 | 250 | 768 | 32.5% |

11 moduli, and the remaining three instances reduce the input coefficient by 10 moduli each. We keep two dual-port ROM blocks in the architecture: the first (second) ROM block stores the constants required by the first (last) two instances.

## 5 Results

We compiled the processor for the ML605 board which has a Virtex-6 FPGA XC6VLX240T-1FF1156. Different clock domains are used in the design: communication with the DDR memory is performed at 200 MHz, whereas computations are performed using a 100 MHz clock. The HEPCloud has $v = 16$ parallel cores for performing polynomial arithmetic, and two cores for computing the lifting operations. The area counts of our HEPCloud, including the DDR interface, are shown in Table 1.

Table 2 gives the latencies of the building blocks excluding the cost of DDR memory access. NTT and INTT computations are performed on polynomials of $N = 2^{16}$ coefficients. To save memory requirement, we compute the twiddle factors on the fly at the cost of $N$ integer multiplications. One NTT computation using $v = 16$ cores requires $(N + \frac{N}{2}\log_2(N))/16 = 36{,}864$ multiplications. However the computation of the twiddle factors in the pipelined data path of the PAU (Fig. 2) has data dependencies and thus causes bubbles in the pipeline. We use a small register-file that stores four consecutive twiddle factors, and reduce the cycles spent in the pipeline bubbles to around 10,000. In the case of an INTT, the additional cycles are spent during scaling operation by $N^{-1}$. To compute $N$-point-wise addition/subtraction/multiplication we need slightly more than 4,096 cycles.

### 5.1 Computation Cost of the Lifting Operations

The cycle requirement for $\mathsf{Lift}_{Q\to q}$ is determined by the division-and-rounding operation, since it is the costliest computation in the pipeline of Fig. 8. If we assume that many $\mathsf{Lift}_{Q\to q}$ operations are performed in pipeline, then the cycle requirement per coefficient will be 687. However, due to the restrictions put by the DDR interface, we process only four $\mathsf{Lift}_{Q\to q}$ in pipeline (see Section 4.2). As a consequence 4,744 cycles are needed to process four coefficients by a single $\mathsf{Lift}_{Q\to q}$ core. Similarly when $\mathsf{ResPol}_{q\to q_i}$ is computed in pipeline with $\mathsf{Lift}_{Q\to q}$, 5,387 cycles are spent per four coefficients. Similarly, when we assume that many $\mathsf{Lift}_{q\to Q}$ operations are performed in pipeline, cycle requirement per coefficient is 401. In practice, we can compute only four $\mathsf{Lift}_{q\to Q}$ in pipeline, and thus it takes total 2,016 cycles for computing four $\mathsf{Lift}_{q\to Q}$ operations.

### 5.2 Computation Cost of the Residue Polynomial Multiplication

To multiply two residue polynomials modulo $q_j$, we compute two NTTs, then $N$-point-wise multiplications, and one INTT. The reduction of the result modulo $f(x)$ follows the Newton iteration method Newton iteration method [39]. In this step, two NTTs, two $N$-point-wise multiplications, one $N/2$-point-wise subtraction and two INTTs are computed. Hence the computation of a polynomial multiplication in $R_{q_j}$ requires four NTTs, three $N$-point-wise multiplications, one $N/2$-point-wise subtraction and three INTTs. This translates into 361,376 cycles.

TABLE 2
Latencies of the Building Blocks without DDR Access Overhead

| Operation | Clocks |
|---|---|
| $N$-point NTT | 47,795 |
| $N$-point INTT | 51,909 |
| $N$-point-wise add/sub/mult | 4,096 |
| $\mathsf{Lift}_{Q\to q}$ (per coeff)† | 687 |
| $\mathsf{Lift}_{Q\to q}$ followed by $\mathsf{ResPol}_{q\to q_i}$ (per coeff)† | 687 |
| $\mathsf{Lift}_{q\to Q}$ (per coeff)† | 401 |
| Poly mult in $R_{q_j}$ | 361,376 |

† *Assuming pipeline processing of many coefficients.*

### 5.3 Computation Cost of FV.Mult

The cycle counts for the steps (see Section 3.2) are as follows.

1) $\mathsf{Lift}_{q\to Q}$: To lift $c_{1,0}$, $c_{1,1}$, $c_{2,0}$ and $c_{2,1}$ (each having $N/2$ coefficients) from $R_q$ to $R_Q$, we compute $\mathsf{Lift}_{q\to Q}$ operations on $2N$ coefficients. This takes total 33,030,144 cycles using the two lifting cores, as each takes 2016 cycles to process four coefficients.
2) $\mathsf{PolyArithmetic}_Q$: Here we compute $\tilde{c}_0$, $\tilde{c}_1$ and $\tilde{c}_2$ by performing four multiplications and one addition over $R_Q$. Since an element in $R_Q$ consists of 84 residue polynomials, the four multiplications require 121,422,336 cycles and the addition requires 172,032 cycles. Hence the total computation cost of $\mathsf{PolyArithmetic}_Q$ is 121,594,368 cycles.
3) $\mathsf{Lift}_{Q\to q}$ and $\mathsf{ResPol}_{q\to q_i}$: To bring $\tilde{c}_0$ and $\tilde{c}_1$ back to the residue representation over $R_q$, we apply $\mathsf{Lift}_{Q\to q}$ followed by $\mathsf{ResPol}_{q\to q_i}$ (in a pipeline) on the coefficients. Using two lifting cores, the total number of cycles required is 44,130,304. For $\tilde{c}_2$ we do not need to compute $\mathsf{ResPol}_{q\to q_i}$. Hence only $\mathsf{Lift}_{Q\to q}$ is performed on the $N/2$ coefficients of $\tilde{c}_2$. Since each lifting core takes 4,744 cycles to process four coefficients, it takes 19,431,424 cycles.
4) WordDecomp: This only splits the large coefficients into words and does not have any computation cost. In HEPCloud, this happens automatically when data is copied from the FPGA to the DDR memory.
5) FV.Relin: Each of $\langle \tilde{\mathbf{c}}_2, \mathbf{rlk}_0 \rangle$ and $\langle \tilde{\mathbf{c}}_2, \mathbf{rlk}_1 \rangle$ requires summation of 21 polynomial multiplications in $R_q$. Note that $\mathbf{rlk_0}$ and $\mathbf{rlk_1}$ are constant, and hence they can be kept in the NTT domain to reduce the computation cost. For a single moduli it requires 21 NTTs, 21 $N$-point-wise multiplications and 20 $N$-point-wise additions to compute the summation $\langle \tilde{\mathbf{c}}_2, \mathbf{rlk}_0 \rangle$ in the NTT domain. An INTT is needed to get the $f(x)$-unreduced result. Next to reduce modulo $f(x)$, two NTTs, two $N$-point-wise multiplications, one $N/2$-point-wise subtraction and two INTTs are computed. Finally

TABLE 3
Latencies and Timings at 100/200 MHz Computation/DDR Clock

| Operation | Computation cycles | DDR cycles | Total time |
|---|---|---|---|
| FV.Add | 83,968 | 9,740,288 | 0.050 s |
| FV.Mult | 335,978,912 | 4,663,738,368 | 26.67 s |

TABLE 4
Comparison of Homomorphic Multiplication Timings

| Implementation | Scheme | Dimension | Coeff-size | Time |
|---|---|---|---|---|
| [31] HW | YASHE | 16,384 | 512 | 48.67 ms |
| [23] SW | YASHE | 32,768 | 1,225 | $\approx$ 2.98 s |
| Our HW | FV | 32,768 | 1,228 | 26.67 s |

the computation of $c_0'$ for a single moduli requires one $N/2$-point-wise addition. Hence in total 23 NTTs, 23 $N$-point-wise multiplications, 20 $N$-point-wise additions, one $N/2$-point-wise subtraction, 3 INTTs and one $N/2$-point-wise addition are performed to compute $c_0'$ for a single moduli. The computation cost is same for computing $c_1'$ for a single moduli. In this way the total cost of computing $c_0'$ and $c_1'$ for all the 41 moduli translates into 117,792,672 cycles.

Overall 335,978,912 cycles are spent in FV.Mult. At 100 MHz clock frequency, this corresponds to 3.36 seconds.

### 5.4 Overhead of the DDR Memory Access

To evaluate our proof of concept implementation, we use a DDR interface that reads or writes 256 bits in a burst. For FV.Mult, DDR memory accesses take around 4,663,738,368 cycles at 200 MHz. For FV.Add, the number of cycles for the memory access is around 9,740,288.

Table 3 shows the timing requirement for computing FV.Add and FV.Mult operations including the overhead of DDR memory access. Based on the timing of FV.Mult, we see that the designed architecture would take roughly 37551 s (11 h and 26 min) to evaluate SIMON-64/128 (44 rounds with 32 ANDs). Since the SIMD feature processes 2,048 slots, the per-block timing will be roughly 18.34 s.

### 5.5 Comparison and Discussion

Since implementations of the FV scheme are largely missing from the literature, in Table 4, we compare HEPCloud to YASHE implementations on both hardware and software, which are the closest counterparts.

Pöppelmann et al. [31] presented an FPGA implementation of YASHE, which computes a homomorphic multiplication in just 48.67 ms. However, these timings cannot be compared one-to-one because YASHE is computationally lighter than FV, but also insecure. Their parameter set also offers lower security, supports only multiplicative depth up to 9, and cannot take advantage of the SIMD feature. They implemented their accelerator using Catapult, an FPGA-based datacenter accelerator with very fast memory access. Consequently, they were able to solve the problem of slow memory access, which is the main problem of HEPCloud that was implemented on a generic ML605 FPGA development board. Indeed, if one observes only the latency of computation, then HEPCloud is faster than Pöppelmann et al.'s design (taking SIMD feature into account) despite implementing FV with a larger parameter set.

Lepoint and Naehrig [23] presented C++ implementations of YASHE for homomorphic evaluations of SIMON 64/128 with YASHE running on a 4-core Intel Core i7-2600 at 3.4 GHz. They reported computation times of 4193 s for SIMON-64/128 using all 4 cores. If we use HEPCloud to compute YASHE (which is lighter than FV), then it would take roughly 12000s to evaluate SIMON-64/128. With respect to their implementation, HEPCloud is 2.8 times slower. Again, the difference is caused by the memory access.

In this work our focus was on designing the computation core of the FV; the DDR memory interface is a proof of concept implementation. With 256-bit burst data width, the DDR interface offers a only 1.97 Gb/s read speed and hence becomes the main bottleneck in our implementation. Desktop computers have industry-optimized DDR interface, and the Intel Core i7-2600 processor has 8 MB cache memory [21]. Since a polynomial in $R_q$ is of size 4.8 MB, the overhead of memory access in [23] would be much lower than ours.

Using a faster DDR interface with 2,048-bit burst data length, one can achieve 10 Gb/s read and 27 Gb/s write speed. With this interface, the overhead of memory access would become roughly equal to the computation cost. Hence the time for a homomorphic multiplication could be reduced significantly by performing the memory access and computation in parallel using two sets of BRAMs: when one set is used for the computation, the other set is used for the memory access. We consider integration of a faster DDR interface in the HEPCloud as a future work. This would make HEPCloud a practical solution for accelerating SHE function evaluations in cloud computing.

## 6 Summary

In this work we designed the hardware building blocks for homomorphic evaluation of medium depth functions using the FV scheme. We showed that FPGAs can accelerate the computation intensive operations of homomorphic function evaluations. Despite this, we found that a massive amount of data exchange takes place between the FPGA and the external DDR memory because only a part of the ciphertext can be fit in the internal memory of the FPGA. The interface with the DDR memory plays a very important role in the performance and becomes a bottleneck unless it is implemented with special care.

We introduced HEPCloud, a single-FPGA design of homomorphic evaluation with FV. We presented a proof-of-concept implementation of HEPCloud that implements all hardware components required for homomorphic function evaluations in cloud computing environments. We demonstrated that HEPCloud is a feasible solution for accelerating the very expensive homomorphic function evaluations, particularly, when fast external memory is available.

Even with the state-of-the-art FPGA acceleration, homomorphic function evaluations remain very expensive and further improvements are still needed. We see several parallelization approaches to accelerate HEPCloud. An obvious way to improve the performance would be to use a multi-FPGA design (a cluster) where each FPGA computes different homomorphic evaluations independently of each other.

This approach improves throughput, but the latency of an individual evaluation remains the same. The second approach is to reduce latency by using parallel FPGAs for independent FV.Add and FV.Mult inside a single homomorphic evaluation. While this is conceptually simple, it may still face difficulties because data needs to be transferred between multiple FPGAs. The third option is to distribute the residue polynomial arithmetic into several FPGAs since they can be computed independently. However, the lifting operations need coefficients from different residue polynomials and require inter-FPGA communication. The fourth option is to divide different parts of a homomorphic multiplication to different FPGAs and perform them in a pipelined fashion in order to increase throughput. The fifth option is to mix the other options which may lead to good tradeoffs that avoid the disadvantages. The techniques represented in this paper can be extrapolated to support these options.

## Acknowledgments

This work was supported in part by the Research Council KU Leuven: C16/15/058. In addition, this work was supported in part by the Flemish Government, by the Hercules Foundation AKUL/11/19, and by the European Commission through ICT Programme under contract through the Horizon 2020 Research and Innovation Programme under contract No H2020-ICT-2014-644371 WITDOM, H2020-ICT-2014-644209 HEAT, H2020-ICT-2014-645622 PQCRYPTO, and Cathedral ERC Advanced Grant 695305. The work of K. Järvinen was supported in part by the Academy of Finland with the projects 283250 (while working at Aalto University) and 303578 (while working at the University of Helsinki).

## References

[1] M. R. Albrecht, "Complexity estimates for solving LWE." [Online]. Available: https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py, Accessed in May 2017.

[2] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Conf. Theory Appl. Cryptographic Techn.*, 1987, pp. 311–323.

[3] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK families of lightweight block ciphers," Cryptology ePrint Archive, Report 2013/404, 2013. [Online]. Available: http://eprint.iacr.org/

[4] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," *Notices Amer. Math. Soc.*, vol. 46, pp. 203–213, 1999.

[5] J. W. Bos, W. Castryck, I. Iliashenko, and F. Vercauteren, "Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling," in *Proc. 9th Int. Conf. Cryptology Africa*, 2017, pp. 187–201.

[6] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *Proc. 14th IMA Int. Conf. Cryptography*, 2013, pp. 45–64.

[7] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Proc. 32nd Annu. Cryptology Conf. Adv. Cryptology—CRYPTO*, 2012, pp. 868–886.

[8] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Optimised multiplication architectures for accelerating fully homomorphic encryption," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2794–2806, Sep. 2016.

[9] J. Cathébras, A. Carbon, R. Sirdey, and N. Ventroux, "An analysis of FV parameters impact towards its hardware acceleration," Cryptology ePrint Archive, Report 2017/246, 2017. [Online]. Available: http://eprint.iacr.org/2017/246

[10] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Syst. J.*, vol. 29, no. 4, pp. 526–538, 1990.

[11] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. New York, NY, USA: McGraw-Hill, 2001.

[12] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Scale-invariant fully homomorphic encryption over the integers," in *Proc. Int. Workshop Public-Key Cryptography*, 2014, pp. 311–328.

[13] D. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Trans. Emerging Topics Comput.*, vol. 5, no. 2, pp. 193–206, Apr.–Jun. 2017.

[14] Y. Doröz, E. Öztürk, E. Savas, and B. Sunar, "Accelerating LTV based homomorphic encryption in reconfigurable hardware," in *Proc. 17th Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 185–204.

[15] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *Proc. 16th Euromicro Conf. Digit. Syst. Des.*, 2013, pp. 955–962.

[16] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012. [Online]. Available: http://eprint.iacr.org/

[17] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st ACM Symp. Theory Comput.*, 2009, pp. 169–178.

[18] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Proc. 32nd Annu. Cryptology Conf. Adv. Cryptology—CRYPTO*, 2012, pp. 850–867.

[19] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. Adv. Cryptology—CRYPTO*, 2013, pp. 75–92.

[20] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. New York, NY, USA: Springer-Verlag, 2003.

[21] Intel, "Core i7–2600 processor." [Online]. Available: https://ark.intel.com/products/52213/Intel-Core-i7–2600-Processor-8M-Cache-up-to-3_80-GHz, Accessed in May 2017.

[22] A. H. Karp and P. Markstein, "High-precision division and square root," *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 561–589, 1997.

[23] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes FV and YASHE," in *Proc. Int. Conf. Cryptology Africa*, 2014, pp. 318–335.

[24] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2010, pp. 1–23.

[25] L. D. M. Albrecht and S. Bai, "A subfield lattice attack on overstretched NTRU assumptions: Cryptanalysis of some FHE and graded encoding schemes," Cryptology ePrint Archive, Report 2016/127, 2016. [Online]. Available: http://eprint.iacr.org/2016/127

[26] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting FPGA DSP slices for a large integer multiplier for integer based FHE," in *Proc. Int. Conf. Financial Cryptography Data Secur. Workshops*, 2013, pp. 226–237.

[27] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proc. 3rd ACM Workshop Cloud Comput. Secur. Workshop*, 2011, pp. 113–124.

[28] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 3–16, Jan. 2017.

[29] T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced lattice-based signatures on reconfigurable hardware," Cryptology ePrint Archive, Report 2014/254, 2014. [Online]. Available: http://eprint.iacr.org/

[30] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Proc. Int. Conf. Sel. Areas Cryptography*, 2014, pp. 68–85.

[31] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Proc. 17th Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 143–163.

[32] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.

[33] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," in *Proc. 17th Int. Workshop Cryptographic Hardware Embedded Syst.*, 2015, pp. 164–184. Revised version. [Online]. Available: https://eprint.iacr.org/2015/337.pdf

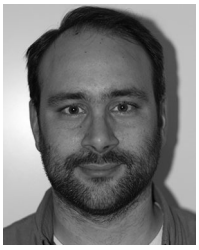
[34] S. Sinha Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2014, pp. 371–391.

[35] N. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Proc. 13th Int. Conf. Practice Theory Public Key Cryptography*, 2010, pp. 420–443.

[36] N. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.
[37] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2010, pp. 24–43.
[38] F. Vercauteren, "Fully homomorphic encryption." Presentation at Katholieke Universiteit Leuven, Leuven, Belgium, 2014. [Online]. Available: https://www.esat.kuleuven.be/cosic/publications/talk-320.pdf
[39] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*. New York, NY, USA: Cambridge Univ. Press, 1999.
[40] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2013, pp. 2589–2592.
[41] W. Wang and X. Huang, "VLSI design of a large-number multiplier for fully homomorphic encryption," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 9, pp. 1879–1887, Sep. 2014.

**Sujoy Sinha Roy** received the MS degree in computer science and engineering from the Indian Institute of Technology Kharagpur and the PhD degree in electrical engineering from the Katholieke Universiteit Leuven, Belgium. His research area has been broadly in the field of efficient implementation of public key cryptography.

**Kimmo Järvinen** received the MSc (Tech.) and DSc (Tech.) degrees in electrical engineering from the Helsinki University of Technology (TKK), Finland, in 2003 and 2008, respectively. From 2008 to 2013 and from 2015 to 2016, he was a postdoctoral researcher with the Department of (Information and) Computer Science, Aalto University in Finland. From 2014 to 2015, he was with the COSIC Group, KU Leuven ESAT in Belgium. Since 2016, he has been a senior researcher with the Department of Computer Science, University of Helsinki in Finland. His research interests lie in the domains of security and cryptography and, especially, in developing efficient and secure hardware implementations of cryptosystems.



**Jo Vliegen** received the master's degree in engineering technology from Catholic University College Limburg, Diepenbeek, Belgium, in 2005 and the PhD degree in engineering technology from the KU Leuven, Leuven, Belgium, in 2014. After three years in industry, he returned to the university college in 2008 and started his research on the reconfigurability of FPGAs. Since 2014, he has been a postdoctoral researcher with the COSIC Research Group, KU Leuven. His main research activities focus both on the implementation of cryptographic primitives on FPGAs, and on the use of (fine-grained) reconfigurability of FPGAs.

**Frederik Vercauteren** received the MSc degree in computer science, the MSc degree in pure mathematics, and the PhD degree in electrical engineering from Katholieke Universiteit Leuven, Belgium. He is a professor with the research group COSIC, Electrical Engineering Department, KU Leuven, Belgium. Previously, he was a lecturer with the Department of Computer Science, University of Bristol, United Kingdom. His research interests include applications of computational number theory and arithmetic geometry in cryptography, in particular post-quantum cryptography and homomorphic encryption.

**Ingrid Verbauwhede** is a professor with the research group COSIC, Electrical Engineering Department, KU Leuven, Belgium. At COSIC, she leads the embedded systems and hardware group. She is also an adjunct professor with the EE Department, UCLA, Los Angeles, California. She is as member of the Royal Academy of Belgium for science and the arts. She is a recipient of an ERC Advanced Grant in 2016. She will received the IEEE 2017 Computer Society Technical Achievement Award. She is a pioneer in the field of efficient and secure implementations of cryptographic algorithms in embedded context on ASIC, FPGA, and embedded SW. She is the author and co-author of more than 300 publications at conferences, journals, book chapters, and books.