Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation

Sujoy Sinha Roy[1], Kimmo Järvinen[1], Frederik Vercauteren[1], Vassil Dimitrov[2], and Ingrid Verbauwhede[1]

[1] KU Leuven ESAT/COSIC and iMinds
Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
[2] University of Calgary, Department of Electrical and Computer Engineering
2500 University Dr. NW, Calgary, AB, Canada, T2N 1N4

Abstract. We present a hardware architecture for all building blocks required in polynomial ring based fully homomorphic schemes and use it to instantiate the somewhat homomorphic encryption scheme YASHE. Our implementation is the first FPGA implementation that is designed for evaluating functions on homomorphically encrypted data (up to a certain multiplicative depth) and we illustrate this capability by evaluating the SIMON-64/128 symmetric key cipher in the encrypted domain. Our implementation provides a fast polynomial operations unit using CRT and NTT for multiplication combined with an optimized memory access scheme; a fast Barrett like polynomial reduction method that allows all possible polynomial moduli; an efficient divide and round unit required in the multiplication of ciphertexts and an efficient CRT unit. These building blocks can be easily reused to instantiate any other polynomial ring based fully homomorphic scheme, including the ones designed for SIMD operations, since no restricting assumptions have been made. These building blocks are integrated in a coprocessor with instructions to execute YASHE, which can be controlled by a computer for evaluating arbitrary functions (up to the multiplicative depth 44 and 128-bit security level). Our architecture was compiled (place-and-route analysis) for a single Xilinx Virtex-7 XC7V1140T FPGA, where it consumes 23 % of registers, 50 % of LUTs, 53 % of DSP slices, and 38 % of BlockRAM memory. The implementation evaluates SIMON-64/128 in approximately 157.7 s (at 143 MHz) and it processes 2048 ciphertexts at once giving a relative time of only 77 ms per block. This is 26.6 times faster than the leading software implementation on a 4-core Intel Core-i7 processor running at 3.4 GHz.

1 Introduction

The concept of fully homomorphic encryption (FHE) was introduced by Rivest, Adleman, and Dertouzos [34] already in 1978 and allows evaluating arbitrary functions on encrypted data. Constructing FHE schemes proved to be a difficult problem that remained unsolved until 2009 when Gentry [22] 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 followed the blueprints set out by Gentry's proposal with an objective to improve the performance of FHE [6, 7, 14, 17, 20, 24, 25, 32]. Most schemes are either based on (ring) learning with errors ((R)LWE) or N-th degree truncated polynomial ring (NTRU) and thus manipulate elements in modular polynomial rings, or on the approximate greatest common divisor (GCD) problem which manipulates very large integers. In this paper, we focus on the former category. Although major advances have been made, we are still lacking FHE schemes with performance levels that would allow large-scale practical use. Software implementations still require minutes or hours to evaluate even rather simple functions. For instance, evaluating the lightweight block cipher SIMON-64/128 [4] requires 4193 s (an hour and 10 minutes) on a 4-core Intel Core-i7 processor [29]. 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.

Application-specific integrated circuits (ASIC) and field-programmable gate arrays (FPGA) have been successfully used for accelerating performance-critical computations in cryptology (see, e.g, [28]). Hence, it is somewhat surprising that, so far, mainly standard software implementations of FHE have been published, because hardware acceleration could bring FHE significantly closer to practical feasibility. Only few publications have reported results on (hardware) acceleration of FHE and most are dealing with manipulation of very large integers. Wang et al. [38] implemented primitives for the Gentry-Halevi (GH) FHE scheme [23] using Graphics Processing Unit (GPU) and observed speedups up to 7.68. They focused particularly on the multi-million-bit modular multiplication by using Strassen multiplication and Barrett reduction. Wang and Huang [39, 40] later showed that this multiplication can be further accelerated by a factor of approximately two together with significant reductions in power consumption by using FPGA and ASIC. Doröz et al. [18] presented a million-bit multiplier for the GH scheme on ASIC. They reported that the performance remains roughly the same as on Intel Xeon CPU, but with a significantly smaller area than the CPU. Moore et al. [31] studied the use of hardwired multipliers inside FPGAs for accelerating large integer multiplication. The same researchers later presented the first implementation of the full FHE encryption in [10]. They reported a speedup factor of 44 compared to a corresponding software implementation. Cousins et al. [15, 16] drafted an architecture of an FPGA accelerator for FHE using Simulink extension to Matlab but they did not provide any implementation results for their architecture. To conclude, only few works are available on hardware acceleration of FHE schemes. So far, no results are available on hardware acceleration of function evaluation on homomorphically encrypted data, although this is the most crucial part of FHE schemes in application scenarios.

We present the first efficient FPGA implementation of the building blocks required in modular polynomial ring based fully homomorphic schemes such as those built on RLWE [8] or NTRU [30]. These building blocks are sufficiently generic to allow implementation of such FHE schemes, and to illustrate this, we integrate these building blocks into a coprocessor architecture that can evaluate functions encrypted with the FHE scheme called Yet Another Somewhat Homomorphic Encryption (YASHE) [6]. To the best of our knowledge, it is the first FPGA implementation that supports function evaluation of homomorphically encrypted data. We use several standard optimization techniques such as the Chinese remainder theorem (CRT) representation, the number theoretic transformation (NTT) and fast modular polynomial reduction, but introduce several optimizations specific for the FPGA platform such as a specific memory access scheme for the NTT. We compile the architecture for Xilinx Virtex-7 XC7V1140T FPGA. We show that a single FPGA achieves speedups up to factor 26.6 in executing SIMON-64/128 compared to a corresponding software implementation running on a 4-core Intel Core i7 processor from [29].

The paper is structured as follows. Section 2 describes the mathematical objects underlying FHE and recaps the YASHE scheme. 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 CRT in order to split computations to small parallel suboperations. We present our hardware architecture for the primitives of YASHE in Section 4. We provide implementation results on a Xilinx Virtex-7 FPGA and compare them to existing software results in Section 5. We end with conclusions and future work in Section 6.

2 System Setup

2.1 Modular Polynomial Rings

The FHE schemes based on RLWE [8] or NTRU [30] compute in modular polynomial rings of the form $R = \mathbb{Z}[x]/(f(x))$ where $f(x)$ is a monic irreducible polynomial of degree n. A very popular choice is to take $f(x) = x^n + 1$ with $n = 2^k$, since this is compatible with a $2n$-degree NTT and reduction modulo $f(x)$ comes for free due to the NTT. However, 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 [36, 37].

For an integer q, we denote by $R_q = R/qR$, i.e. the polynomial ring where the coefficients are reduced modulo q. The plaintext space in FHE schemes typically will be R_2, and if one wants to utilize SIMD operations the polynomial $f(x)$ should be chosen 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 the most popular choice x^n+1 with $n = 2^k$, since it results in only one irreducible factor modulo 2. In most polynomial ring based FHE schemes, a ciphertext consists of one or two elements in R_q. However, not all operations take place in the ring R_q; sometimes (see below for an illustration

with YASHE) one is required to temporarily work in R itself before mapping down into R_q again using (typically) a divide and round operation.

2.2 YASHE

The YASHE scheme was introduced by Bos et al. in [6] in 2013. The scheme works in the ring $R = \mathbb{Z}[x]/(f(x))$, with $f(x) = \Phi_d(x)$ the d-th cyclotomic polynomial. The plaintext space is chosen as R_t for some small t (typically $t = 2$) and a ciphertext consists of only one element in the ring R_q for a large integer q. The main security parameters of the scheme are the degree of $f(x)$ and the size of q. We note that q is not required to be a prime and can be chosen as a product of small primes to speed-up computations (see Section 3). To define the YASHE scheme we also require two probability distributions defined on R, namely χ_{key} and χ_{err}. In practice one often takes χ_{err} to be a discrete Gaussian distribution, whereas χ_{key} can be simply sampling each coefficient from a narrow set like $\{-1, 0, 1\}$. Given an element $a \in R_q$ and a base w, we can write a in base w by splicing each of its coefficients, i.e. write $a = \sum_{i=0}^{u} a_i w^i$ with each $a_i \in R$ and coefficients in $(-w/2, w/2]$ and $u = \lfloor \log_w(q) \rfloor$. Decomposing an element $a \in R_q$ into its base w components $(a_i)_{i=0}^{u}$ is denoted by $\mathsf{WordDecomp}_{w,q}(a)$. For an element $a \in R_q$, we define $\mathsf{PowersOf}_{w,q}(a) = (aw^i)_{i=0}^{u}$, the vector that consists of the element a scaled by the different powers of w. Both operations can be used to provide an alternative description of multiplication in R_q, namely:

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

The advantage of the above expression is that the first vector contains small elements, which limits error expansion in the homomorphic multiplication.

An FHE scheme is an augemented encryption scheme that defines two additional operations on ciphertexts, $\mathsf{YASHE.Add}$ and $\mathsf{YASHE.Mult}$ that result in a ciphertext encrypting the sum (respectively the product) of the underlying plaintexts. The YASHE scheme is then defined as follows (full details can be found in the original paper [6]).

- $\mathsf{YASHE.ParamsGen}(\lambda)$: For security parameter λ, choose a polynomial $\Phi_d(x)$, moduli q and t and distributions χ_{err} and χ_{key} attaining security level λ. Also choose base w and return the system parameters $(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$.
- $\mathsf{YASHE.KeyGen}(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$: Sample $f', g \leftarrow \chi_{key}$ and set $f = (tf' + 1) \in R_q$. If f is not invertible in R_q choose a new f'. Define $h = tgf^{-1} \in R_q$. Sample two vectors $\mathbf{e}$, $\mathbf{s}$ of $u+1$ elements from χ_{err} and compute $\gamma = \mathsf{PowersOf}_{w,q}(f) + \mathbf{e} + h\mathbf{s} \in R_q^{u+1}$ and output $(pk, sk, evk) = (h, f, \gamma)$.
- $\mathsf{YASHE.Encrypt}(h, m)$: To encrypt a message $m \in R_t$ sample $s, e \leftarrow \chi_{err}$ and output the ciphertext $c = \Delta \cdot m + e + sh \in R_q$ with $\Delta = \lfloor q/t \rfloor$.
- $\mathsf{YASHE.Decrypt}(f, c)$: Recover m as $m = \lfloor \frac{t}{q} \cdot [f \cdot c]_q \rceil \in R_t$ with $[\cdot]_q$ reduction in the interval $(-q/2, q/2]$.
- $\mathsf{YASHE.Add}(c_1, c_2)$: Return $c_1 + c_2 \in R_q$.
- $\mathsf{YASHE.KeySwitch}(c, evk)$: Return $\langle \mathsf{WordDecomp}_{w,q}(c), evk \rangle \in R_q$

```
Input: Polynomial a(x) ∈ Z_q[x] of degree N − 1 and N-th primitive root ω_N ∈ Z_q of unity
Output: Polynomial A(x) ∈ Z_q[x] = NTT(a)
1  begin
2      A ← BitReverse(a);
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                  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 ;
```

Algorithm 1: Iterative NTT [13]

– $\mathsf{YASHE.Mult}(c_1, c_2, evk)$: Return $c = \mathsf{YASHE.KeySwitch}(c', evk)$ with $c' = \lfloor \frac{t}{q} c_1 c_2 \rceil \in R_q$.

YASHE Paramater Set: We use the parameter set Set-III from [29], in particular $d = 65535$ (and thus the degree of $f(x)$ is $32768 = 2^{15}$), $\log_2(q) = 1228$ and χ_{err} a discrete Gaussian distribution with parameter $\sigma = 8$. The paper [29] claims that this set has security level 128-bits, but this is an underestimate due to a small error in the security derivation.

3 High Level Optimizations

To efficiently implement YASHE 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 $\mathsf{YASHE.Mult}$ in Section 2.2, to multiply two ciphertexts c_1 and c_2 one first needs to compute $c_1 \cdot c_2$ over the integers, 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.

First we will consider polynomial multiplication in R_q where the modulus $f(x)$ is an arbitrary polynomial of degree n. Since each element in R_q therefore can be represented as a polynomial of degree $n-1$, the resulting product 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 using the N-fold NTT (see Alg. 1). Since N is taken a power of 2, the NTT can be computed very efficiently using simple butterfly operations in $O(k \cdot N)$ multiplications modulo q. 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 paper.

The product of two elements $a, b \in R_q$ is then computed in two steps: firstly, 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 general $f(x)$ this reduction does not come for free (unlike the choice $f(x) = x^n + 1$) and for the parameters used in the YASHE scheme the polynomial $f(x)$ is in fact quite dense (although almost all coefficients are ± 1). We have to consider general $f(x)$ because the most obvious choice $f(x) = x^n + 1$ does not allow SIMD operations, since $f(x)$ mod 2 has only one irreducible factor. The polynomial $\Phi_d(x)$ from the YASHE parameter set splits modulo 2 in 2048 different irreducible polynomials, which implies that we can work on 2048 bits in parallel using the SIMD method first outlined in [36].

To speed-up the reduction modulo $f(x)$ we rely on a polynomial version of Barrett reduction [21], where one precomputes the inverse of $x^n f(1/x)$ modulo x^n. The quotient and remainder can then be recovered at the cost of two polynomial multiplications.

Note that the multiplication of c_1 and c_2 in YASHE.Mult is performed over integers. To get the benefit of NTT based polynomial multiplication, we perform this multiplication in a ring R_Q where Q is a sufficiently large modulus of size $\sim 2 \log q$ such that the coefficients of the result polynomial are in $\mathbb{Z}$.

CRT Representation of Polynomials: In the cryptosystems based on the RLWE problem, computations are performed on the polynomials of a ring R_q. The reported FPGA-based architectures [3, 33, 35] of such cryptosystems use BRAM slices to store the polynomials and use arithmetic components made up of DSP multipliers and LUTs. To utilize the available resources efficiently, we envision a similar design outline for our homomorphic processor. However, 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. Now we take an example where two coefficients are multiplied. If we use a bit-parallel multiplier, then a $2,500 \times 2,500$-bit multiplier will not only result in an enormous area, but will also result in a very low operating frequency. On the other side, a word-serial multiplier is too slow for homomorphic computations.

To tackle the problem of long integer arithmetic, we take inspiration from the application of the CRT 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 $l < L$. We map any long integer operation modulo q or Q into small computations moduli q_i, and apply CRT whenever a reverse mapping is required. We use the term *small residue* to represent coefficients modulo q_i and the term *large residue* to represent coefficients modulo q or Q.

Parallel Processing: Beside making the long integer operations easy, such small-residue representation of the coefficients have a tremendous effect on the computation time. Since the computations in the small-residue representations are independent of each other, we can exploit this parallelism and speedup the computations using several parallel cores.

The size of the moduli q_i is an important design decision and depends on the underlying platform. We use the largest Xilinx Virtex-7 FPGA *XC7VX1140T* to implement our homomorphic processor. We note the following information [1] of the FPGA and tune the design decisions accordingly. A Xilinx Virtex-7 FPGA XC7VX1140T has 1,880 36Kb-BRAMs, each having 1,024 words of size 36 bits. Beside the BRAMs, the LUTs present in the FPGA fabric can be configured as distributed RAMs. The number of LUTs available in the FPGA is 712,000. The FPGA has 3,360 25×18-bit DSP multipliers. One could implement a slightly larger multiplier by combining DSP multipliers with LUTs. For the set of moduli, we choose in total 84 primes (hence $l = 41$ and $L = 84$) of size 30 bits, (the primes from 1008795649 onwards) satisfying $q_i \equiv 1 \mod N$. 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.

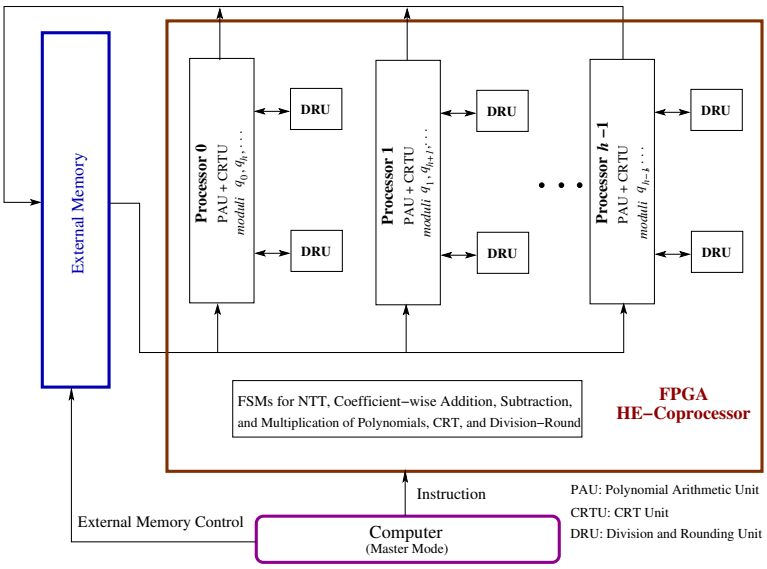

Fig. 1. Overview of the HE Architecture

4 Architecture

We propose an architecture (Fig. 1) to perform the operations in the YASHE scheme. The central part of the architecture is an FPGA based accelerator that works as a coprocessor of a computer and executes the computationally intensive operations. We call this coprocessor the *HE-coprocessor*. The HE-coprocessor supports the following operations: NTT of a polynomial, coefficient wise addition/subtraction/multiplication of two polynomials, computation of the residues using the CRT, computation of the coefficients modulo Q from the residues, and the scaling of the coefficients. The external memory in this architecture is implemented using high-speed RAMs and is used to store the polynomials during a homomorphic computation. The computer in Fig. 1 works in master-mode and instructs the HE-coprocessor and the controller of the external memory. Since the execution of the homomorphic scheme is controlled by a software program in the computer, a level of flexibility to implement other homomorphic schemes based on a similar set of elementary operations is offered.

The HE-coprocessor comprises of three main components: the polynomial arithmetic unit (PAU), the CRT unit (CRTU), and the division and rounding unit (DRU). We do not implement the discrete Gaussian sampling in the HE-coprocessor due to the reason that a sampling is required only during an encryption, which is a less frequent operation in comparison to the operations that are performed in the encrypted domain. Since the samples from a narrow discrete Gaussian distribution can be generated very efficiently using lookup tables in software [11], we use the master-computer (Fig. 1) for this purpose. This design decision also keeps the FPGA available for the expensive homomorphic computations.

4.1 Polynomial Arithmetic Unit

To exploit the parallelism provided by the small-residue representation of the polynomials, the PAU has h parallel processors to perform computations on h residue polynomials simultaneously. We call this *horizontal parallelism*. Since the targeted FPGA does not provide sufficient resources to process all the small-residue polynomials in parallel, we design the processors in a generic way such that a processor can be time-shared by a group of $\lceil L/h \rceil$ prime moduli. However, this parallelism is not enough to speed up the computations in the encrypted domain, since each polynomial has 2^{15} or 2^{16} coefficients. To add an additional degree of acceleration, we add v parallel cores in each processor. We call this *vertical parallelism*. The cores in a processor are connected to a group of BRAMs through a switching matrix.

Optimization in the Routing: During an NTT computation (Alg. 1), coefficients are fetched from the memory and then butterfly operations are performed on the coefficients. Let us assume that a residue polynomial of $N = 2^{16}$ coefficients is stored in b BRAMs and then processed using v butterfly cores. If v is a

divisor of b, then we can split the NTT computation in equal parts among the v parallel butterfly cores. However there are two main technical issues related to the memory access that would affect the performance of the NTT computation. The first one is: all the parallel cores read and write simultaneously in the memory blocks. 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 clock cycle. Hence a memory block can be read (or written) by one butterfly core in a cycle and thus address generation by the parallel butterfly cores should be free from conflicts. The second technical issue is related to the routing complexity. Since a residue polynomial is stored in many BRAMs, access of data from a BRAM that is very far from a butterfly core will result in a very long routing distance. Now in Alg. 1 we see that the maximum difference between the indexes of the two coefficients is $N/2$. In the FPGA, fetching data from memory locations at a relative distance of 2^{15} will result in a very long routing of the data, and thus could drastically reduce the operating frequency.

We propose a memory access scheme that addressess these two technical challenges. The memory access scheme has been developed by analysing the address generation during different iterations of the loops in the NTT (Alg. 1). We segment the set of b BRAMs into b/v groups. The read ports of a group are accessed by only one butterfly core. This *dedicated read* prevents any sort of conflict during the memory read operations. Moreover, in the FPGA the group of BRAMs can be placed close to the corresponding butterfly core and thus the routing complexity can be reduced.

We describe the proposed memory access scheme during an execution of the NTT by parallel cores in Alg. 2. The module *butterfly-core* performs butterfly operations on two coefficent pairs following the optimization technique in [35]. In the algorithm the v parallel butterfly cores of a processor are indexed by c where $c \in [0, v-1]$. During the m-th loop of a NTT, the twiddle factor in the c-th core is initialized to a constant value $\omega_{m,c}$. In the hardware, these constants are stored in a ROM. The counter $I_{twiddle}$ denotes the interval between two consecutive calculations of the twiddle factors. Whenever the number of butterfly operations ($N_{butterfly}$) becomes a multiple of $I_{twiddle}$, a new twiddle factor is computed. The c-th butterfly core reads the $c-$th group of BRAMs $MEMORY_c$ using two addresses $address_1$ and $address_2$. The addresses are computed from the counters: $base$, $increment$, and $offset$, that represent the starting memory address, the increment value, and the difference between $address_1$ and $address_2$ respectively. A butterfly core outputs the two addresses and the four coefficients $s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}$. These output signals from the parallel butterfly cores are collected by a set of parallel modules *memory-write* that are responsible for writing the groups of BRAMs. The input coefficients that will be read by the adjacent butterfly core in the next iteration of the m-th loop, are selected for the writing operation in $MEMORY_c$ by the c-th memory-write module. The top module *Parallel-NTT* instantiates v butterfly cores and memory write blocks. These instances run in parallel and exchange signals.

```
   /* This module computes butterfly operations                                          */
 1 module butterfly-core(input 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 the number of butterfly operation in a m-loop */
 7         ω ← ω_{m,c}              /* Initialization to a 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, address_2) ← (base + increment, base + offset + increment)
12                 (t_1, u_1) ← MEMORY_c[address_1]      /* Read from c-th group of RAMs */
13                 (t_2, u_2) ← MEMORY_c[address_2]
14                 if m < log N − 1 then
15                     (t_1, t_2) ← (ω · t_1, ω · t_2)
16                     (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)
17                     N_butterfly ← N_butterfly + 2
18                     increment = increment + 2 · offset
19                     if N_butterfly ≡ I_twiddle then ω ← ω · ω_m^{v/2}
20                 else
21                     t_1 ← ω · t_1; ω ← ω · ω_m^{v/2}
22                     t_2 ← ω · t_2; ω ← ω · ω_m^{v/2}
23                     (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)
24                     N_butterfly ← N_butterfly + 2
25                     increment = increment + 2 · offset
26         I_twiddle ← I_twiddle/2
27         if offset < v/2 then offset ← 2 · offset

   /* This module writes the coefficients computed by two butterfly-cores                */
28 module memory-write(input c, m, address_1, address_2, s_{1,0}, ... s_{4,v−1})
29 begin
30     if 2^m < v/2 then gap ← 2^m
31     else gap ← v/2               /* This represents the index gap between the two cores */
32     if c < v/2 then
33         MEMORY_c[address_1] ← (s_{2,c}, s_{1,c})
34         MEMORY_c[address_2] ← (s_{2,c+gap}, s_{1,c+gap})
35     else
36         MEMORY_c[address_1] ← (s_{4,c}, s_{3,c})
37         MEMORY_c[address_2] ← (s_{4,c+gap}, s_{3,c+gap})

   /* This is the top module that executes butterfly-core in parallel                    */
38 module Parallel-NTT()
39 begin
40     butterfly-core bc_0(0, m, address_1, address_2, s_{1,0}, s_{2,0}, s_{3,0}, s_{4,0})
41     memory-write mw_0(0, m, address_1, address_2, s_{1,0}, ... s_{4,v−1})
42     ...
43     butterfly-core bc_{v−1}(v − 1, m, address_1, address_2, s_{1,v−1}, s_{2,v−1}, s_{3,v−1}, s_{4,v−1})
44     memory-write mw_{v−1}(v − 1, m, address_1, address_2, s_{1,0}, ... s_{4,v−1})
```

Algorithm 2: Routing Efficient Parallel NTT using v cores

Internal Architecture of the PAU: In Fig. 2 we show the internal architecture of the vertical cores that we use inside the horizontal processors in Fig. 1. We follow the pipelined RLWE encryption architecture presented in Fig. 2 of [35] and design our cores to support additional computations due to the increased complexity of the YASHE homomorphic scheme in comparison to the basic RLWE encryption scheme. Moreover we design the cores in a more generic

way such that a single core can perform computations with respect to several moduli.

The input register bank in Fig. 2 contains registers to store data from the BRAMs and data from the CRTUs. In addition, the register bank also contains shift registers to delay the input coefficients in a pipeline during a NTT computation (see [35] for more details). 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. Though the figure shows only one such ROM block, there are actually $\lceil \frac{L}{h} \rceil$ such ROM blocks, since a core is shared by $\lceil \frac{L}{h} \rceil$ primes. The integer multiplier (shown as a circle in Fig. 2) is a is a 30×30-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 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 [26]. The second reason is that the cores are shared by several prime moduli, and hence, a generic reduction circuit is more preferable than several dedicated reduction circuits. We design the Barrett reduction circuit in a bit parallel way such that it can process the outputs from the bit-parallel multiplier in a flow. The reduction consists of three 31×31-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 30×59-bit multiplications during the CRT computations. The adder/subtracter and the subtracter circuits after the Barrett reduction block in Fig. 2 are used to compute the butterfly operations during a NTT computation and to perform

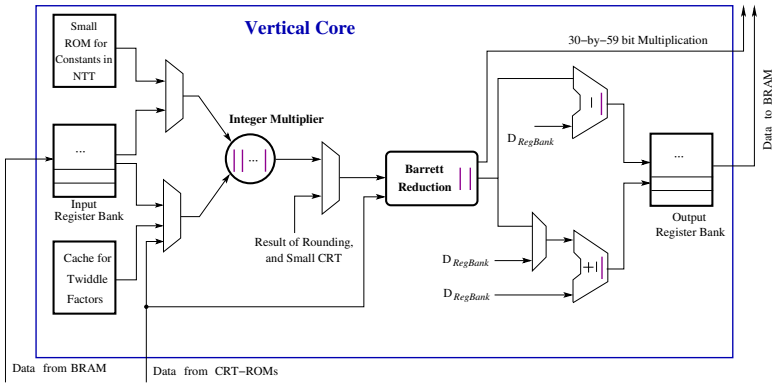

Fig. 2. Architecture for the Vertical Cores

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 follow the pipelining technique from [35] and put pipeline registers in the data paths of the computation circuits. In Fig. 2, the pipeline registers are shown as magenta colored lines.

4.2 CRT Unit

We accelerate polynomial arithmetic by representing the polynomials of R_q as smaller residue polynomials moduli q_j, $j \in [0, l-1]$. However, this representation also has the following overhead:

- The multiplication of the input polynomials c_1 and c_2 in YASHE.Mult is performed in the larger ring R_Q (see Section 2.2). Since c_1 and c_2 are in R_q, we need to first lift the polynomials from R_q to R_Q. This lifting operation essentially computes the residue polynomials moduli q_i, $i \in [l, L-1]$ from the residue polynomials moduli q_j, $j \in [0, l-1]$ by applying the CRT. We call this operation the *small-CRT*.
- After the multiplication of c_1 and c_2, the result is a set of residue polynomials moduli q_j, $j \in [0, L-1]$. The scaling operation in YASHE.Mult requires the coefficients of the result in the form of modulo Q. Hence, we apply the CRT to get back the coefficients modulo Q from the small residue polynomials moduli q_j, $j \in [0, L-1]$. We call this operation the *large-CRT*.

We now present fast architectures for the two types of CRT operations.

Architecture for the Small-CRT Unit: In the case of CRT, we solve a set of congruences $[a]_{q_i}$ where $i \in [0, l-1]$, and compute a simultaneous solution $[a]_q$, using the following equation:

$$[a]_q = [\sum [a]_{q_i} \cdot (\frac{q}{q_i}) \cdot [(\frac{q}{q_i})^{-1}]_{q_i}]_q \tag{1}$$

When the moduli are fixed, the computation cost is reduced by storing the fixed values $b_i = (\frac{q}{q_i}) \cdot [(\frac{q}{q_i})^{-1}]_{q_i}$ in a table. Still, the computation of $[a]_q$ involves long multiplications as the b_i values are around $\log_2(q)$ bits long. In the case of small-CRT, we actually do not need to compute the solution $[a]_q$, but a solution $[a]_{q_j}$ where q_j is a small 30-bit prime moduli. We can avoid long multiplications if we compute the solution in the following way:

$$[a]_{q_j} = [\sum [a]_{q_i} \cdot b_i]_{q_j} \tag{2}$$

$$= [\sum [a]_{q_i} \cdot [b_i]_{q_j}]_{q_j} \tag{3}$$

In the above equation, the constant values $[b_i]_{q_j}$ are 30-bit integers, and hence we need only 30-bit multiplications for all the l residues to accumulate the results.

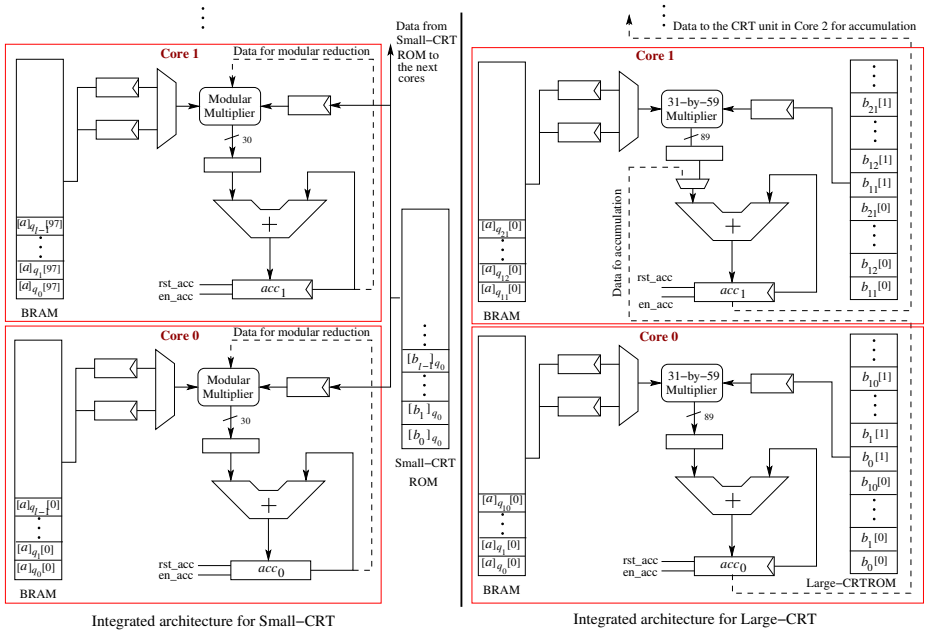

Fig. 3. Architecture for the Small and Large CRT. The computation blocks are aligned along a horizontal processor. Exchange of data between the cores occur during a Large-CRT computation.

Finally, the result is reduced modulo q_j to get $[a]_{q_j}$. We show our hardware architecture for the small-CRT computations in the left half of Fig. 3. The red boxes represent the vertical cores present in a processor of our HE-coprocessor. During a small-CRT computation, the modular multiplier and some registers that are present inside a core are reused. The figure shows how the residue polynomials are stored in the BRAM. Here $[a]_{q_i}[k]$ represents the kth coefficient of a residue polynomial $[a]_{q_i}$. The Small-CRT ROM is a BRAM that contains the constants required during a small-CRT computation. The result of a modular multiplication $\left[[a]_{q_i}[k] \cdot [b_i]_{q_j}\right]_{q_j}$ is accumulated in a register acc_c, where c represents the core-index. In the end of the accumulation process, the final result is reduced with respect to the modulus q_j using the Barrett reduction circuit present in the modular multiplier.

Architecture for the Large-CRT Unit: In this case, we compute simultaneous solution with respect to the large modulus Q. Hence, this CRT is more costly than the small-CRT. Our large-CRT architecture is shown in the right half of Fig. 3. The constant values b_i are stored in a ROM and then multiplied word by word with the coefficients from the BRAMs. To speedup the computation, we set the word size of the ROMs to 59 bits. Since the multiplications in this case are 30-by-59 bits, we reuse the 31 $\times$ 59-bit multipliers present in the

Barrett reduction circuits (Fig. 2). The computation is distributed among the vertical cores of a processor. In our HE-coprocessor, there are 16 vertical cores in a processor. These cores divided into two groups: Core-0 to Core-7 form the first group, whereas Core-8 to Core-15 form the second group. Each group computes one large-CRT in parallel. Since there are 84 b_i constants (Equation 1), each core in a group computes multiplications with a maximum of 11 b_i constants. The results of the multiplications are accumulated in the accumulation registers. After that, the partially accumulated results are then added together as follows: the register acc_0 is added with acc_1 and the result is stored in acc_1. Then acc_1 is added with acc_2 and finally acc_2 is added with acc_3. In parallel, acc_7 downto acc_4 are added together and the result is stored in acc_4. Finally acc_4 is added with acc_3 and the result is stored in acc_3. This final result is then stored in a small distributed RAM (not shown in the figure). This distributed RAM is read by the DRU. Similar computations are performed in the second group consisting of Core-8 to Core-15.

4.3 Division and Rounding Unit

The DRU computes $\lfloor tc/q \rceil$ where $t = 2$, c is a coefficient from the Large-CRTU, 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 DRU outputs 59-bit words that can be directly reduced modulo the 30-bit q_i using the existing Barrett reduction circuitries in the PAU that operate on inputs of size $< q_i^2$. The word size of the DRU was selected to be 118 bits (2×59) 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 [27, 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 1228-bit constant integer and the dividend c is an at most 2492-bit integer[3], which gives a bound of 3721 bits. Hence, the reciprocal r is computed up to a precision of 32 118-bit words, of which 22 words are nonzero.

Figure 4 shows the architectural diagram of the DRU. The multiplication $r \times c$ is computed by using a 118×118-bit multiplier that computes $22^2 = 484$ partial multiplications. This multiplier performs a 118-bit Karatsuba multiplication by using three 59×59-bit multipliers generated with the Xilinx IP Core Generator (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 IP Core Generator).

[3] The dividend c is a sum of 2^{16} integers, each in $[0, (q-1)^2]$ (2455-bit integers). Further growth by 14 bits is introduced by the polynomial modular reduction and 7 bits by the large-CRT computation.

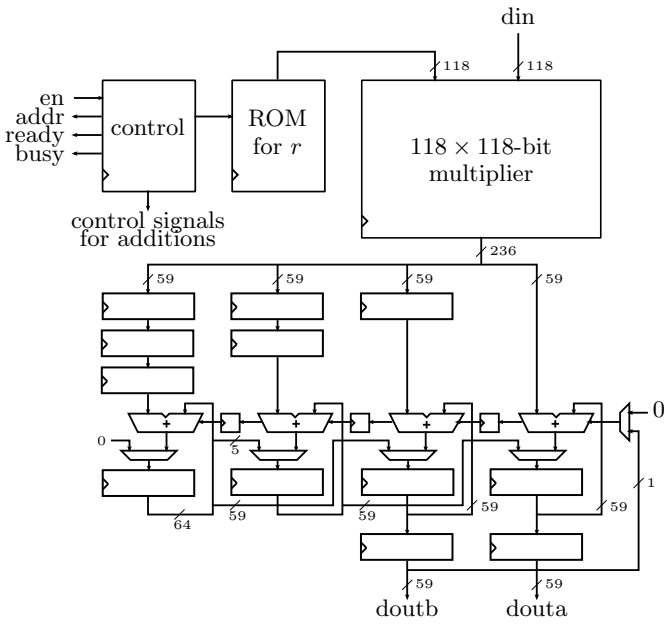

Fig. 4. The Division and Rounding Unit (DRU)

The partial products from the 118-bit multiplier are accumulated into a 241-bit $(2 \times 118 + 5)$ register using the Comba algorithm [12]. These additions are computed in a 4-stage 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.

5 Results

The HE-coprocessor proposed in Sect. 4 was described using mixed Verilog and VHDL. We compiled the processor for Xilinx Virtex-7 XC7V1140T-2, the largest device of the Virtex-7 FPGA family, by using the Xilinx ISE 13.4 Design Suite. We set the optimization goal in the ISE tool to *speed*. All results reported in the section have been obtained after place-and-route analysis.

The HE-coprocessor has $h = 8$ horizontal processors, each having $v = 16$ parallel vertical cores for performing polynomial arithmetic, 16 small-CRTUs, two large-CRT computation groups and two DRUs. The area counts of our HE-coprocessor are shown in Table 1. As seen from the table, our HE-coprocessor consumes nearly 50% of the computational elements (LUTs and DSP multipliers) available in the FPGA. To know the maximum number of processors that we

Table 1. The area results on Xilinx Virtex-7 XCV1140TFLG1930-2 FPGA

Resource	Used	Avail.	Percentage
Slice Registers	339,086	1,424,000	23 %
Slice LUTs	360,353	712,000	50 %
BlockRAM	640 BRAM36, 152 BRAM18	1,880	38 %
DSP48	1,792	3,360	53 %

Table 2. Latencies and timings at 143 MHz after place-and-route

Operation	Clocks	Time	Rel. time†
NTT	47,795	0.334 ms	0.163 μs
INTT	51,909	0.363 ms	0.177 μs
Poly-add/sub/mult	4,096	0.029 ms	0.014 μs
Small-CRT	118,784	0.831 ms	0.405 μs
Large-CRT	2,752,512	19.248 ms	9.398 μs
Divide-and-round	2,813,952	19.678 ms	9.559 μs
YASHE.Add⋆	24,576	0.172 ms	0.083 μs
YASHE.Mult⋆	16,019,635	112.025 ms	54.699 μs
SIMON-32/64⋆	8,202,053,120	57.357 s	28.006 ms
SIMON-64/128⋆	22,555,646,080	157.731 s	77.017 ms

† Time per slot in SIMD operations (in total 2048 slots)

⋆ Excluding interfacing with the computer and the external memory

can implement in a single FPGA, we performed a design exploration using the Xilinx PlanAhead tool. This tool allows the designer to manually place different components in the FPGA. The Virtex-7 XCV1140 FPGA consists of four super large logic regions (SLRs). From the design exploration we have found that we can put two processors in one SLR. If we put three processors in one SLR (occupies around 70% of the SLR-area), then the Xilinx tool reports congestion of routing channels and the place-and-route analysis fails. From this design exploration we conclude that we can implement two processors per SLR, and hence eight processors in one Virtex-7 XCV1140 FPGA.

Table 2 gives the latencies of different operations supported by our HE-coprocessor. The operating frequency of our HE-coprocessor is 143 MHz after place-and-route analysis. 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 $(N + \frac{N}{2}\log_2(N))/16$, i.e. 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. Following [35], 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 computation, the additional cycles are spent

during scaling operation by N^{-1}. Cycle count per coefficient for the small-CRT is 58. It is computed for $2^{15} = 32,768$ coefficient using 16 vertical cores and accumulators and, hence, the total cost is 118,784 cycles. Similarly, the costs of large-CRT and the division-and-rounding are 672 and 687 cycles per coefficient respectively. However, the cycles spent for the large-CRT operations are not a part of the actual cost, as the large-CRT operation runs in parallel with the division-and-rounding operations during a homomorphic multiplication. The division-and-rounding operations are computed 2^{16} times using 16 DRUs (two from each horizontal processor) at the cost of 2,813,952 cycles.

Table 2 includes estimates for the latencies of YASHE.Add and YASHE.Mult as well as the evaluations of SIMON-32/64 and SIMON-64/128. The cycle count for a single YASHE.Mult operation is derived as follows. First the input polynomials c_1 and c_2 are lifted from R_q to R_Q using two small-CRTs; then a polynomial multiplication (including polynomial reduction) is performed using 4 NTTs, 3 INTTs, 3 Poly-mul and 1 Poly-sub. Since there are 84 moduli, we compute these operations in 11 batches using $h = 8$ processors. After the polynomial multiplication, a scaling operation by t/q is performed to compute c'. The YASHE.KeySwitch uses NTT of evk and computes the result of YASHE.Mult by performing 24 NTTs, 3 INTTs, 24 Poly-mul and 23 Poly-add. Again these computations are performed in six batches. Estimates for SIMON consider only YASHE.Mult, which dominate the costs of function evaluations. SIMON-32/64 and SIMON-64/128 require 512 (32 rounds with 16 ANDs) and 1,408 (44 rounds with 32 ANDs) YASHE.Mult, respectively. In addition to the latencies and timings for a single execution of the operations, we also provide the relative timings which represent the timings per slot (in total 2048 slots).

Lepoint and Naehrig [29] presented C++ implementations for homomorphic evaluations of SIMON-32/64 and SIMON 64/128 with YASHE running on a 4-core Intel Core i7-2600 at 3.4 GHz. They reported computation times of 1029 s (17.2 min) and 4193 s (69.9 min) for SIMON-32/64 and SIMON-64/128 using all 4 cores, respectively. The implementations included 1800 and 2048 slots and, hence, the relative times per slot were 0.57 s and 2.04 s, respectively. With one core, they achieved 16500 s (275 min) for SIMON-64/128. The homomorphic evaluation of SIMON-64/128 on our FPGA implementation takes 158 s (2.6 min) and it also allows 2048 slots giving a relative time of only 77 ms per slot. Hence, our single FPGA implementation offers a speedup of up to 26.6 (or 104.6) times compared to the 4-core (or 1-core) software implementation.

6 Conclusions and Future Work

We showed that modern high-end FPGAs (such as Virtex-7) have sufficient logic, hardwired multipliers, and internal memory resources for efficient computation of primitives required for evaluating functions on FHE encrypted data. Despite this, memory requirements and the speed of memory access is critical. Only ciphertexts that are currently being processed fit into the internal memory of the FPGA and other ciphertexts must be stored in external memory. Interface

with the external memory may become a bottleneck unless it is done with special care. Sufficiently fast memory access can be achieved by using high bandwidth memory and/or parallel memory chips. Moreover, most of the memory access can be performed in parallel with computation which reduces the overhead. Many FPGAs include dedicated transceivers for fast interfacing that could be used for fast data access in FHE computations.

We presented a single-FPGA design of homomorphic evaluation with YASHE. An obvious way to improve the performance would be to use a multi-FPGA design (a cluster). We see four parallelization approaches. The first and simplest option is to instantiate parallel FPGAs so that each of them 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 divide independent homomorphic operations inside a single homomorphic evaluation to parallel FPGAs. Thanks to the numerous independent computations included, e.g., in homomorphic evaluations of block ciphers, this approach is likely to improve both throughput and latency. While this is conceptually a simple approach, it may still face difficulties because data needs to be transferred between multiple FPGAs. The third 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 fourth option is to mix the other three options and it may lead to good tradeoffs that avoid the disadvantages of the other options. The techniques represented in this paper can be extrapolated to support these options.

The SIMD approach achieves high throughput, but it has been argued that low latency can be more important in practice [29]. The leading software implementation [29] requires as much as 200 s for a single slot evaluation of SIMON-32/64 with YASHE. Our FPGA-based implementation achieves smaller latencies even for SIMON-64/128 with an implementation that allows 2048 slots. Reducing the number of slots to one would allow more efficient parameters (see Sect. 2). We plan to investigate such schemes in the future.

We evaluated the performance of our architecture for YASHE by providing performance values for SIMON in order to provide straightforward comparisons to the leading software implementation from [29]. However, SIMON is not necessarily an optimal cipher for FHE purposes. For instance, the low-latency Prince cipher [5] may lead to better performance [19]. FHE-friendly symmetric encryption that is designed to minimize the multiplicative size and depth can offer significant improvements over SIMON (and Prince) [2, 9]. Our architecture is able to evaluate arbitrary functions (up to a certain multiplicative depth) and, hence, these options will be studied in the future. Performance can be further increased by tuning the parameters of the architecture for these specific functions. Such changes can be easily done because our architecture is highly flexible.

We presented the first FPGA implementation for function evaluation on homomorphically encrypted data. Although it already achieves competitive performance compared to leading software implementations, we see several ways to optimize the architecture and these issues will be explored further in the

future. For instance, we can increase throughput of NTT by precomputing certain values. We used Barrett reduction for the primes but choosing 'nice' primes and optimizing reduction circuitries by hand could offer speedups. However, in that case the performance would be bounded by the worst prime and finding suitably many 'nice' primes may be challenging. The architecture uses simple data-flow and pipelining in order to keep high clock frequency, but it results in pipeline bubbles that increase the latencies of operations. A more elaborated data-flow could allow removing the pipeline bubbles and reducing the latencies. More research is required also in balancing the pipelines so that even higher clock frequencies can be achieved for the architecture. The architecture utilizes parallelism on various levels and degrees of parallelism in different parts of the architecture should be fine-tuned to achieve optimal resource utilization and performance. Our source code is highly flexible and significant parts of it were generated with scripts. This allows us to perform parameter space explorations that will enable us to find more optimal parameters for the architecture.

References

1. Xilinx 7 Series FPGAs Overview, DS180 (v1.16.1) December 17, 2014, http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf
2. Albrecht, M., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Advances in Cryptology — EUROCRYPT 2015. Lecture Notes in Computer Science, Springer (to appear)
3. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and Area-efficient FPGA Implementations of Lattice-based Cryptography. In: HOST. pp. 81–86. IEEE (2013)
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404 (2013), http://eprint.iacr.org/
5. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T.: PRINCE — a low-latency block cipher for pervasive computing applications. In: Wang, X., Sako, K. (eds.) Advances in Cryptology — ASIACRYPT 2012. Lecture Notes in Computer Science, vol. 7658, pp. 208–225. Springer (2012)
6. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) Proceedings of the 14th IMA International Conference on Cryptography and Coding (IMACC 2013). Lecture Notes in Computer Science, vol. 8308, pp. 45–64. Springer (2013)
7. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology — CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 868–886. Springer (2012)
8. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed.) Advances in Cryptology — CRYPTO 2011, Lecture Notes in Computer Science, vol. 6841, pp. 505–524. Springer Berlin Heidelberg (2011)
9. Canteaut, A., Carpov, S., Fontaine, C., Lepoint, T., Naya-Plasencia, M., Paillier, P., Sirdey, R.: How to compress homomorphic ciphertexts. Cryptology ePrint Archive, Report 2015/113 (2015), http://eprint.iacr.org/

10. Cao, X., Moore, C., O'Neill, M., Hanley, N., O'Sullivan, E.: High-speed fully homomorphic encryption over the integers. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) Financial Cryptography and Data Security Workshops, the 2nd Workshop on Applied Homomorphic Cryptography and Encrypted Computing (WAHC 2014). Lecture Notes in Computer Science, vol. 8438, pp. 169–180. Springer (2014)
11. de Clercq, R., Sinha Roy, S., Vercauteren, F., Verbauwhede, I.: Efficient software implementation of ring-LWE encryption. Cryptology ePrint Archive, Report 2014/725 (2014), http://eprint.iacr.org/
12. Comba, P.G.: Exponentiation cryptosystems on the IBM PC. IBM Systems Journal 29(4), 526–538 (1990)
13. Cormen, T., Leiserson, C., Rivest, R.: Introduction To Algorithms. http://staff.ustc.edu.cn/$\sim$csli/graduate/algorithms/book6/toc.htm
14. Coron, J.S., Lepoint, T., Tibouchi, M.: Scale-invariant fully homomorphic encryption over the integers. In: Krawczyk, H. (ed.) Public-Key Cryptography — PKC 2014, Lecture Notes in Computer Science, vol. 8383, pp. 311–328. Springer (2014)
15. Cousins, D.B., Rohloff, K., Peikert, C., Schantz, R.: SIPHER: Scalable implementation of primitives for homomorphic encryption — FPGA implementation using Simulink. In: Proceedings of the 15th Annual Workshop on High Performance Embedded Computing (HPEC 2011) (2011)
16. Cousins, D.B., Rohloff, K., Peikert, C., Schantz, R.: An update on SIPHER (scalable implementation of primitives for homomorphic encryption) — FPGA implementation using Simulink. In: Proceedings of the 2012 IEEE High Performance Extreme Computing Conference (HPEC '12). pp. 1–5 (2012)
17. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) Advances in Cryptology — EUROCRYPT 2010. Lecture Notes in Computer Science, vol. 6110, pp. 24–43. Springer (2010)
18. Doröz, Y., Öztürk, E., Sunar, B.: Evaluating the hardware performance of a million-bit multiplier. In: Proceedings of the 16th Euromicro Conference on Digital System Design (DSD 2013). pp. 955–962 (2013)
19. Doröz, Y., Shahverdi, A., Eisenbarth, T., Sunar, B.: Toward practical homomorphic evaluation of block ciphers using Prince. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) Financial Cryptography and Data Security Workshops, the 2nd Workshop on Applied Homomorphic Cryptography and Encrypted Computing (WAHC 2014). Lecture Notes in Computer Science, vol. 8438, pp. 208–220. Springer (2014)
20. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), http://eprint.iacr.org/
21. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, New York, NY, USA (1999)
22. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st ACM Symposium on Theory of Computing (STOC 2009). pp. 169–178 (2009)
23. Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) Advances in Cryptology — EUROCRYPT 2011. Lecture Notes in Computer Science, vol. 6632, pp. 129–148. Springer (2011)
24. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology — CRYPTO 2012, Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer (2012)

25. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology — CRYPTO 2013, Lecture Notes in Computer Science, vol. 8042, pp. 75–92. Springer (2013)
26. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
27. Karp, A.H., Markstein, P.: High-precision division and square root. ACM Transactions on Mathematical Software 23(4), 561–589 (1997)
28. Koç, C.K. (ed.): Cryptographic Engineering. Springer (2009)
29. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes FV and YASHE. In: Pointcheval, D., Vergnaud, D. (eds.) Progress in Cryptology — AFRICACRYPT 2014. Lecture Notes in Computer Science, vol. 8469, pp. 318–335. Springer (2014)
30. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing. pp. 1219–1234. ACM, New York, NY, USA (2012)
31. Moore, C., Hanley, N., McAllister, J., O'Neill, M., O'Sullivan, E., Cao, X.: Targeting FPGA DSP slices for a large integer multiplier for integer based FHE. In: Adams, A., Brenner, M., Smith, M. (eds.) Financial Cryptography and Data Security Workshops, the 1st Workshop on Applied Homomorphic Cryptography and Encrypted Computing (WAHC 2013). Lecture Notes in Computer Science, vol. 7862, pp. 226–237. Springer (2013)
32. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW 2011). pp. 113–124. ACM (2011)
33. Pöppelmann, T., Güneysu, T.: Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In: Selected Areas in Cryptography – SAC 2013, pp. 68–85. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2014)
34. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of secure computation 4(11), 169–180 (1978)
35. Sinha Roy, S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact ring-lwe cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems CHES 2014, Lecture Notes in Computer Science, vol. 8731, pp. 371–391. Springer Berlin Heidelberg (2014)
36. Smart, N., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography PKC 2010, Lecture Notes in Computer Science, vol. 6056, pp. 420–443. Springer Berlin Heidelberg (2010)
37. Smart, N., Vercauteren, F.: Fully homomorphic SIMD operations. Designs, Codes and Cryptography 71(1), 57–81 (2014)
38. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. In: IEEE Conference on High Performance Extreme Computing (HPEC 2012). pp. 1–5 (2012)
39. Wang, W., Huang, X.: FPGA implementation of a large-number multiplier for fully homomorphic encryption. In: IEEE International Symposium on Circuits and Systems (ISCAS 2013). pp. 2589–2592 (2013)
40. Wang, W., Huang, X.: VLSI design of a large-number multiplier for fully homomorphic encryption. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 22(9), 1879–1887 (2014)