HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA

Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede

Abstract—Homomorphic Encryption makes privacy preserving computing possible in a third party owned cloud by enabling computation on the encrypted data of users. However, software implementations of homomorphic encryption are very slow on general purpose processors. With the emergence of 'FPGAs as a service', hardware-acceleration of computationally heavy workloads in the cloud are getting popular. In this article we propose HEAWS, a domain-specific coprocessor architecture for accelerating homomorphic function evaluation on the encrypted data using high-performance FPGAs available in the Amazon AWS cloud. To the best of our knowledge, we are the first to report hardware acceleration of homomorphic encryption using Amazon AWS FPGAs. Utilizing the massive size of the AWS FPGAs, we design a high-performance and parallel coprocessor architecture for the FV homomorphic encryption scheme which has become popular for computing exact arithmetic on the encrypted data. We design parallel building blocks and apply pipeline processing at different levels of the implementation hierarchy, and on top of such optimizations we instantiate multiple parallel coprocessors in the FPGA to execute several homomorphic computations simultaneously. While the absolute computation time can be reduced by deploying more computational resources, efficiency of the HW/SW communication interface plays an important role in homomorphic encryption as it is computation as well as data intensive. Our implementation utilizes state of the art 512-bit XDMA feature of high bandwidth communication available in the AWS Shell to reduce the overhead of HW/SW data transfer. Moreover, we explore the design-space to identify optimal off-chip data transfer strategy for feeding the parallel coprocessors in a time-shared manner. As a result of these optimizations, our AWS-based accelerator can perform 613 homomorphic multiplications per second for a parameter set that enables homomorphic computations of depth 4. Finally, we benchmark an artificial neural network for privacy-preserving forecasting of energy consumption in a Smart Grid application and observe five times speed up.

Index Terms—Homomorphic encryption, FPGA-accelerated cloud computing

1 Introduction

Since the beginning of this decade, the world has seen a rapid transition towards cloud-based computing. We are already in the cloud in our everyday life as we follow social networks, watch and upload videos on Youtube, use internet search engines, play online games etc. Cloud services are becoming more popular in spreading and expanding education and health care and thus play a major role in the upliftment of our society. Similarly, business applications are using 'cloud as a platform' and 'cloud as a service' for storing large volume of data, as well as analyzing the data.

Despite the numerous advantages, cloud computing raises privacy issues. As data is being stored in the cloud which is owned by third parties, the owners of the cloud could see, use or abuse the data. Even if the cloud-owners are honest, data breaches caused by the hackers are causing concerns. For example, a group of hackers breached the security of Linkedin and posted 167 million email addresses and passwords for sale on the dark web in May 2016 [1]. Homomorphic Encryption (HE) enables privacy-preserving, confidential cloud computing and thus can be used as a tool to protect privacy of users' data while keeping the conveniences offered by the cloud services. A homomorphic encryption scheme is an augmented public-key encryption scheme with extra procedures to evaluate functions on the encrypted data. A user could upload its *homomorphic-encrypted* data to a cloud computer and can evaluate a required functionality directly on the encrypted data. As the data remains encrypted in the cloud platform, the owner of the cloud could not learn any useful information from it. In a similar way, any data breach from a cloud just exposes the encrypted data, not the plaintext. Although homomorphic encryption was conceptualized four decades ago in 1978 by Rivest, Adleman and Dertouzos [2], all homomorphic encryption schemes constructed until 2009 were partially homomorphic, i.e., they could perform a single type of operation (either homomorphic addition or multiplication) on the encrypted data. The first homomorphic encryption scheme that can compute both homomorphic additions and multiplications and thus evaluate generic applications on the encrypted data was constructed by Gentry in 2009 [3]. These first generation schemes were extremely slow and suffered from a massive ciphertext expansion problem. The present-generation homomorphic encryption schemes are an order of magnitude faster compared to the early schemes, and their ciphertext size ranges from kilobytes to megabytes.

- *Furkan Turan and Ingrid Verbauwhede are with COSIC, Department of Electrical Engineering (ESAT), KU Leuven, 3000 Leuven, Belgium. E-mail: {furkan.turan, ingrid.verbauwhede}@esat.kuleuven.be.*
- *Sujoy Sinha Roy is with the School of Computer Science, University of Birmingham, B15 2TT Birmingham, United Kingdom. E-mail: s.sinharoy@cs.bham.ac.uk.*

Manuscript received 5 Jan. 2020; revised 14 Apr. 2020; accepted 15 Apr. 2020. Date of publication 20 Apr. 2020; date of current version 9 July 2020. (Corresponding author: Furkan Turan.) Recommended for acceptance by A. Louri. Digital Object Identifier no. 10.1109/TC.2020.2988765

Yet, homomorphic encryption is still very slow, as it is computationally very demanding. Hence, it naturally becomes a very interesting research target for the accelerator designer community, which propose speedup to performance-critical applications in many application-domains.

As homomorphic encryption itself is a rather new research topic, hardware acceleration of it is a less explored research topic compared to the other areas of cryptography such as symmetric-key and public-key cryptography. In a typical HE encryption with 80 bits of security level, a plaintext of 6 bits results in two ciphertext polynomials of 90 kB each with 4,096 coefficients of 180-bit size. On this encrypted form, homomorphic additions and multiplications can be performed. Besides huge data processing for these operations, they require several complicated modules for performing computations in a large polynomial ring. It might appear that hardware accelerators will naturally speedup computations; however, the data- as well as computation-intensive nature of homomorphic encryption make its hardware implementation very challenging. Literature [4] shows that even dedicated hardware could yield slower computation than a general purpose CPU.

A typical software implementation of our target HE scheme on an Intel CPU takes 33 ms for one multiplication [5]. It is our aim to accelerate the homomorphic operations using FPGAs, and use it for privacy preserving cloud applications. However, accelerating homomorphic encryption and building a working high-performance implementation on FPGA platforms is full of challenges. While a general-purpose desktop/laptop processor runs at 3 or 4 GHz clock frequency, achieving above 200 MHz clock frequency on an FPGA platform can be really challenging for a very large and complex circuit such as a processor for homomorphic computing on encrypted data. In this paper we start from the work by Roy, Turan, Järvinen, Vercauteren and Verbauwhede on accelerating homomorphic evaluation using FPGA, published at HPCA 2019 [6] by incorporating more parallel processing on FPGA-accelerated cloud computing.

We present HEAWS, a programmable and high-performance domain specific accelerator architecture for FPGA accelerated homomorphic computation in the cloud. Specifically, HEAWS targets Amazon Web Services (AWS), which offers the service named Elastic Compute Cloud (EC2) consisting of F1 instances. The F1 instances attach high-end Xilinx FPGAs to powerful Intel CPUs. Our work tailors the hardware acceleration of homomorphic computing to the large FPGAs of F1 instances and its hardware-software interface for achieving the best performance. To the best of our knowledge, our work is the first to use the large F1 instance FPGAs for accelerating homomorphic computing, and to evaluate the performance benchmarking on them. A list of our contributions is given below.

- We propose a massively parallel datapath architecture to accelerate the homomorphic computation on the F1 platform of Amazon AWS. Six parallel coprocessors fit on the F1 instance, and each coprocessor calculates with two parallel computation cores.
- Based on the Chinese Remainder Theorem (CRT) and the extension proposed by Halevi *et al.* [7] we represent the moduli of the used HE scheme in the Residue Number System (RNS). With RNS we decompose the moduli into products of small primes of size 30 bits such that residue arithmetic modulo these small primes fit well on the on-board DSP units and BRAMs. In addition, we assign the decomposed prime numbers with parallel execution datapaths. Furthermore, each coprocessor consists of a 2-core Lift/Scale unit, a 2-core NTT unit and seven 2-core Reside Polynomial Arithmetic Units (RPAUs), thus making the architecture massively parallel.
- We designed a distributed memory architecture and dedicated addressing scheme so that we can make maximum use of the FPGA specific distributed memory blocks, that include both BRAMs and a new type of memory called 'URAMs'. Each coprocessor is assigned with eight polynomial memories. Each memory stores a polynomial of 4,096 coefficients in a combination of two BRAMs and two URAMs. For the optimised polynomial multiplication, NTT algorithm is used. The algorithm is adapted to support data-dependent operations with an addressing scheme and provides guaranteed conflict-free dual read/write access.
- As our coprocessor deals with large ciphertexts, an optimized hardware-software interface is essential to avoid communication bottlenecks. We make maximum use of the communication interfaces on the Amazon AWS FPGA platforms to achieve low latency transfer of polynomials, to minimise the overhead of off-chip data transfers. Our efforts include the investigation of how the OS and corresponding kernel drivers handle the transfers and adapting the memory layout of polynomials to fit the page layouts of the drivers.
- Our coprocessor can be programmed with a set of dedicated instructions. We use them to demonstrate and measure elementary arithmetic operations of homomorphic computation. We also use the coprocessors to demonstrate a machine learning application for a smart grid forecasting application. We estimate that our FPGA accelerator would perform nearly 3 times more work per dollar compared to a GPU instance, and at a much lower power consumption.

2 Background

Homomorphic encryption converts a plaintext message into ciphertext for protecting its confidentiality, in addition to allowing computation on the ciphertext. With its mathematical homomorphism, any computation on the ciphertexts yields a new ciphertext, the decryption of which corresponds to the result of the same computation as if it had been performed on plaintext.

Existing homomorphic encryption schemes are augmented public-key encryption schemes with two extra procedures, namely homomorphic addition and multiplication, for performing homomorphic computations on the encrypted data. Plaintexts are encrypted using the public-key encryption procedure of the scheme and then uploaded to the cloud. For evaluating a function `f()` on the plaintexts, the function is first translated into a representation that consists only of addition and multiplication gates. In the cloud, the function is evaluated

on the ciphertexts by evaluating *homomorphic* additions and multiplications on the respective ciphertexts. Due to the underlying homomorphism property, all operations on the ciphertexts get mapped into the same operations on the plaintexts.

Homomorphic encryption schemes that can perform both homomorphic additions and multiplications on the encrypted data, have been constructed using mathematical lattice problems. These schemes use noise during the encryption of plaintext. Besides, each homomorphic arithmetic yields extra noise in the output. With the number of homomorphic evaluations, the 'signal to noise' ratio diminishes and there is a threshold beyond which recovery of the 'signal', i.e., the plaintext-result after a decryption becomes erroneous. Therefore, the existing homomorphic encryption schemes have a noise threshold which determine the maximum number of HE operations admissible on a ciphertext. This threshold is called the *depth* in the context of homomorphic encryption, and it can be tuned by tuning the parameter set of the encryption scheme. For example, a larger depth can be supported by choosing a larger parameter set at the cost of performance penalty. In lattice-based homomorphic encryption schemes, the most 'noisy' operation is the homomorphic multiplication between two ciphertexts, and thus the noise growth during the evaluation of an application is mostly dependent on the number of sequential homomorphic multiplication operations, i.e., the number of multiplication gates in the critical path. In practice, the depth of a lattice-based homomorphic encryption scheme is actually the *multiplicative depth* of the scheme. A HE scheme that can support a limited number of homomorphic computations is called a *Somewhat Homomorphic Encryption (SHE)*. A SHE scheme can be extended to a Fully Homomorphic Encryption (FHE) scheme by adding an extra procedure called *bootstrapping*. The bootstrapping operation evaluates the decryption circuit homomorphically on the encryption of a ciphertext (thus a double encryption of a plaintext) and results in a fresh re-encryption of the plaintext with a much lower amount of noise.

In the last 9 years, several lattice-based homomorphic encryption schemes have been proposed. The most efficient schemes are based on ideal lattices where polynomial arithmetic is at the centre of all computations. They use polynomial arithmetic with some variations in the parameter sets and high-level protocol executions.

In this work we targeted accelerating the Fan-Vercauteren (FV) somewhat homomorphic encryption scheme [8], as a case study. The FV scheme supports *exact* computations on the encrypted data. HEAWS is a modular design and our design methodology is quite generic. With some adaptations to the architecture, it might be possible to accelerate multiple homomorphic encryption schemes.

2.1 Fan-Vercauteren (FV) SHE Scheme

The FV is a popular SHE scheme [8] for performing calculations on the encrypted data. It is based on the Ring Learning with Errors (ring-LWE) problem [9] which uses special structured ideal lattices that correspond to ideals in polynomial rings $R = \mathbb{Z}[\mathbf{x}]/\langle f(x)\rangle$, where f is an irreducible polynomial of degree n. The ring is denoted as R_q when the polynomial-coefficients are reduced to integer modulo q. Let $s \in R_q$ be a secret polynomial with coefficients uniformly random modulo q. The ring-LWE distribution consists of tuples $(a_i, b_i) \in R_q^2$, where $a_i \in R_q$ is uniformly random and $b_i = a_i \cdot s + e_i \in R_q$. The 'error' polynomials e_i are sampled from a discrete error distribution, typically a discrete Gaussian, $\mathcal{X}$. The search ring-LWE problem states that, given many tuples $(a_i, b_i) \in (R_q, R_q)$, it is computationally infeasible to compute the secret s.

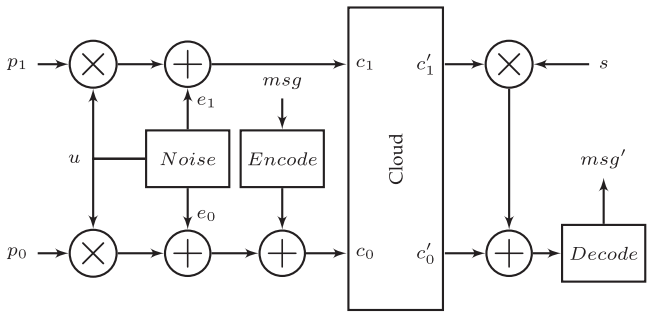

Fig. 1. FV encryption is a public-key encryption scheme, which receives a message m and encrypts it into two ciphertext polynomials c_0 and c_1 using the public key pair p_0 and p_1. The ciphertext is decrypted into a plaintext using the secret key s. For a cloud based homomorphic computation, the encryption and decryption operations are handled on local computer, but the ciphertext is processed in cloud.

The FV scheme uses the polynomial ring $R = \mathbb{Z}[x]/\langle f(x)\rangle$ with reduction polynomial $f(x) = \Phi_d(x)$, the dth cyclotomic polynomial of degree $n = \varphi(d)$. To reduce the noise during homomorphic computations, the authors of FV proposed the use of ternary secret polynomial s with coefficients from $\{-1, 0, 1\}$.

In a HE application, the encryption and decryption operations take place on the users' local computer only, but the homomorphic computations, namely additions and multiplications, on the encrypted data are performed in the cloud. In the following paragraphs we first introduce the encryption and decryption operations and then describe the homomorphic addition and multiplication operations. For a detailed mathematical description of the FV scheme, readers may follow the original paper [8] by Fan and Vercauteren.

The encryption and decryption operations are shown in Fig. 1. All variables in the figure are degree $n-1$ polynomials. The public key is the polynomial pair (p_0, p_1) and the private key is s. The encryption operation encodes the message m, generates three error polynomials (u, e_1, e_2) and computes polynomial additions and multiplications to generate the ciphertext $(c_0, c_1) \in R_q^2$. Following the recommendations by the authors of FV, we use a ternary polynomial for u. The decryption first performs a polynomial multiplication, and then an addition and finally coefficient-wise decoding. The challenge in HE schemes are the large key sizes and large polynomials subject to these operations.

The homomorphic addition and multiplication are the two operations calculated on ciphertext, and our FPGA acceleration aims at executing them fast in the cloud. Homomorphic addition is a simple coefficient-wise operation adding two polynomial pairs. For two ciphertext $c_0 = (c_{0,0}, c_{0,1})$ and $c_1 = (c_{1,0}, c_{1,1})$, their homomorphic addition yields the result ciphertext $c = (c_{0,0} + c_{1,0}, c_{0,1} + c_{1,1})$.

In contrast, a homomorphic multiplication is a rather complicated operation. It is calculated with the steps shown

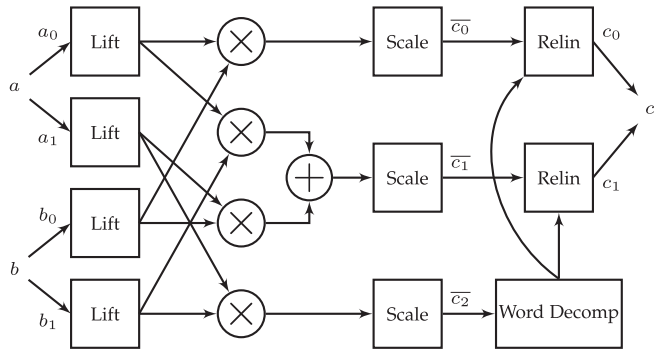

Fig. 2. FV homomorphic multiplication over the ciphertext $a = \{a_0, a_1\}$ and $b = \{b_0, b_1\}$, calculates the ciphertext $c = \{c_0, c_1\}$ in several steps. First, the lift operations move input polynomials from ring R_q to R_Q. Next, cross multiplications of the polynomials are computed. Later, the scale operation moves the polynomials back to R_q. The final ciphertext is calculated by reducing the resulting three polynomials to two. That reduction is made by decomposing one of the polynomials into two pieces, and relinearising the other two with these pieces.

in the block diagram of Fig. 2. It starts with a $\mathsf{Lift}_{q\to Q}$ operation which lifts the polynomials from R_q to R_Q, where Q is a much larger modulus than q and is in the order of $\mathcal{O}(n \cdot q^2)$. In our instance, the size of q and Q values are 180-bit and 390-bit respectively (see later in Section 2.4 for parameter set details). Following the $\mathsf{Lift}_{q\to Q}$, cross multiplications in R_Q are performed. The $\mathsf{Scale}_{Q\to q}$ operation later scales the polynomials from R_Q to R_q.

Algorithm 1. Iterative NTT [13]

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

1: $A \leftarrow \text{BitReverse}(a)$
2: **for** $m = 2$ to n by $m = 2m$ **do**
3: $\quad \omega_m \leftarrow \omega_n^{n/m}$
4: $\quad \omega \leftarrow 1$
5: $\quad$ **for** $j = 0$ to $m/2 - 1$ **do**
6: $\quad\quad$ **for** $k = 0$ to $n-1$ by m **do**
7: $\quad\quad\quad t \leftarrow \omega \cdot A[k+j+m/2]$
8: $\quad\quad\quad u \leftarrow A[k+j]$
9: $\quad\quad\quad A[k+j] \leftarrow u + t$
10: $\quad\quad A[k+j+m/2] \leftarrow u - t$ } Butterfly Operation
11: $\quad\quad$ **end for**
12: $\quad\quad \omega \leftarrow \omega \cdot \omega_m$
13: $\quad$ **end for**
14: **end for**

$\mathsf{WordDecomp}$ takes a polynomial with coefficients in base R_q, and decomposes it into base w by slicing each coefficient. Hence, it is a cheap operation requiring only bit-level manipulations. It outputs a vector of polynomials as shown in bold font in Fig. 2. ReLin is the relinearization operation, which takes these vectors and computes a relinearised ciphertext $\mathbf{c} = \{c_0, c_1\} \in \{R_q, R_q\}$, where $c_0 = \tilde{c}_0 + \mathtt{SoP}(\tilde{\mathbf{c}}_2, \mathbf{rlk}_0)$ and $c_1 = \tilde{c}_1 + \mathtt{SoP}(\tilde{\mathbf{c}}_2, \mathbf{rlk}_1)$. Here $\mathtt{SoP}$ stands for the summation of products. ReLin uses a special key $\mathbf{rlk} = (\mathbf{rlk}_0, \mathbf{rlk}_1)$, called the relinearization key, as a fixed vector of polynomials.

The high-level block diagram in Fig. 2 shows the main building blocks that are required for implementing the homomorphic multiplication operation in the FV scheme. In the remaining part of this section, we describe the computations that are performed by the building blocks in the figure.

2.2 Polynomial Multiplication

Homomorphic encryption computes multiplications of large-degree and large-coefficient polynomials, many number of times. In our case, the polynomial degree is 4,096, and coefficient size is 180-bit (see Section 2.4 for the parameter set details). For computing the multiplication of polynomials, the commonly used efficient algorithms are the Fast Fourier Transform (FFT) based multiplication [10], Karatsuba multiplication [11] and Toom-Cook multiplication [12]. The FFT-based polynomial multiplication has the lowest asymptotic time complexity of $O(n \log n)$ and hence it is the best candidate for implementing polynomial multipliers for large-degree polynomials.

The Number Theoretic Transform (NTT) is an *integer* generalization of the FFT where all arithmetic operations are performed on integers only. The NTT-based polynomial multiplication does not introduce any approximation error and is thus suitable for homomorphic encryption and cryptography in general as the accuracy of computation plays a critical role. However, the NTT requires the modulus q of the polynomial-coefficients to be prime number that satisfies the congruence relation $q \equiv 1 \bmod n$ where n is the degree of the polynomial. An NTT-based polynomial multiplication is computed in three major steps. First, two forward NTTs are computed for the two input polynomials. Next, their coefficients are multiplied element-wise producing an intermediate result in the NTT domain. Finally, an inverse NTT is applied to that result to transform it back to the original domain. An iterative version of the forward NTT algorithm is shown in Algorithm 1. In the algorithm, the constant ω_n is the nth primitive root of the unity. The inverse NTT operation is similar to the forward NTT and requires an additional scaling of the resulting coefficients by n^{-1}.

2.3 Residue Number System (RNS)

While large-degree polynomials can be multiplied using NTT, we need an additional optimization to deal with the large coefficient-size in homomorphic encryption. Coefficient size in homomorphic encryption can vary from hundreds of bits to thousands, depending on the parameter set. In our instance, the polynomial coefficient sizes are 180-bits (see Section 2.4 for the parameter set details). Arithmetic on such long integers requires implementation of a complicated multi-precision architecture and can severely slowdown computation time. For the FV homomorphic encryption scheme, which is our target scheme, we can use a Residue Number System (RNS) to avoid expensive long-integer arithmetic and additionally achieve parallel processing. The Residue Number System (RNS) relies on the Chinese Remainder Theorem (CRT) given below, hence sometimes referred to as CRT representation.

Theorem 1. *For pairwise coprime positive integers q_i and arbitrary integers a_i, the system of simultaneous congruences $x \equiv a_i \bmod q_i$ has a unique solution $x \bmod q$ with modulo $q = \prod q_i$.*

With RNS, a large modulus q is represented as a product of small primes q_i such that $q = \prod_{i=0}^{l-1} q_i$. The RNS basis

consists of the moduli $\{q_i\}$. A large coefficient, say $(a \bmod q)$, has an equivalent RNS representation $(a_i \bmod q_i)$ such that $a \equiv a_i \bmod q_i$ for all the prime factors of q. Using an RNS representation, any arithmetic operations on the large coefficient $(a \bmod q)$ gets mapped to the same arithmetic operations on all the shares $(a_i \bmod q_i)$. Since the involved numbers are much smaller, we do not require a an expensive data-path for processing these small numbers. On FPGAs arithmetic on small numbers can be computed using the on-chip specialised hardware, called DSP slices. Additionally, computations on these shares can be performed in parallel, thus resulting in a factor l parallel processing.

The FV homomorphic encryption requires moving polynomials between modulo q and RNS representations, back-and-forth. While computing the residues $\{a_i\}$ require modulo q_i reductions for all the bases in the RNS system, computing $(a \bmod q)$ from its residues require application of the CRT. Both operations are expensive, requiring long-integer arithmetic. In the following sections, we first explain the parameter set details explaining the data-widths used in the preferred RNS system, and then describe $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Scale}_{Q\to q}$ operations that require RNS basis conversions.

2.4 Parameter Set

The parameter set of a HE defines the orders of its polynomial and coefficient size, and has a proportional relationship to its multiplicative depth, and to the security level it achieves. In other words, if a large parameter set is chosen, a greater multiplicative depth can be achieved and more complex applications can be evaluated on the encrypted data. In this paper we design the HE accelerator architecture for a parameter set that offers a multiplicative depth of 4 and at least 80-bit security [14]. The ciphertext-modulus q is 180-bit, polynomial size is 4,096 coefficients and the standard deviation of the error distribution to 102. The larger modulus Q is at least 372-bit. With this multiplicative depth, one could homomorphically evaluate privacy-friendly forecasting for the smart grid [5], low-complexity block cipher such as Rasta [15] on ciphertext, private information retrieval or encrypted search in a table of 2^{16} entries, encrypted sorting.

As we use the RNS, the 180-bit modulus q is taken as a product of six 30-bit primes. The larger modulus Q is a product of q and additional seven 30-bit primes. Thus the RNS basis of Q extends the RNS basis of q by seven additional primes.

2.5 Lift

$\mathsf{Lift}_{q\to Q}$ is an operation that moves an input ciphertext from ring R_q to a ring R_Q with a larger modulus Q as shown in Fig. 2. Naturally, the $\mathsf{Lift}_{q\to Q}$ operation is free of cost provided the input polynomial is represented modulo q. However, the RNS representation, preferred for optimised polynomial multiplication, requires extending the RNS representation from base q to base Q and prevents a free $\mathsf{Lift}_{q\to Q}$ operation.

A naive method of extending the RNS representation of coefficients would be to construct the coefficients modulo q from their RNS representation by applying the CRT, and then compute the extended RNS representation for Q by computing modular reductions. However, such computations would require expensive long integer arithmetic, including modular division.

Recently, Halevi *et al.* [7] proposed a fast RNS extension that uses approximate arithmetic to avoid long-integer arithmetic completely. Their approximate method adds a small amount of extra noise to the result-ciphertext and can lead to a negligible increase in the decryption failure rate. In the context of HE, a negligible increase in the decryption failure rate is acceptable in many privacy-preserving applications. The details of their optimization, which is known as the 'HPS' optimization, is explained in the following paragraph.

The RNS representation of a coefficient x in the RNS of q is composed of six shares $\{x_0, \ldots, x_5\}$ in six moduli $\{q_0, \ldots, q_5\}$, as the parameter set is described in Section 2.4. The simultaneous solution to the coefficient x is

$$\begin{aligned} x &\equiv \sum_{0}^{5} x_i \cdot \tilde{q}_i \cdot q_i^* \bmod q \\ q_i^* &= \frac{q}{q_i} \\ \tilde{q}_i &= (q_i^*)^{-1} \bmod q_i. \end{aligned} \tag{1}$$

For a known quotient $v = \lfloor (\sum_0^5 x_i \cdot \tilde{q}_i \cdot q_i^*)/q \rfloor$, this relationship can be used for reconstructing x from x_i such that

$$x = \sum_{0}^{5} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* - v \cdot q. \tag{2}$$

The HPS optimized algorithm [7] offers a solution to calculate an approximation of the quotient v without computing any multi-precision arithmetic as

$$\begin{aligned} [x]_Q &= \Big[\sum_{0}^{5} [x_i \cdot \tilde{q}_i]_{q_i} \cdot q_i^* - v \cdot q \Big]_Q \\ v &= \Big\lceil \sum_{0}^{5} \frac{[x_i \cdot \tilde{q}_i]_{q_i}}{q_i} \Big\rfloor. \end{aligned} \tag{3}$$

Note that, all the required calculations are handled with 30-bit operands and moduli. In fact, it computes an approximated solution; however, the approximation errors could be bounded to 2^{-53} with IEEE 754 double floats, which is negligible in practice [7].

The details of the architecture designed for the described lift operation will be given in Section 3.2.

2.6 Scale

The $\mathsf{Scale}_{Q\to q}$ operation moves input ciphertext from ring R_Q to R_q, hence could be considered as the inverse of $\mathsf{Lift}_{q\to Q}$. In mathematical terms, it calculates y in R_q from a coefficient x in R_Q as $y = \lceil t/q \cdot x \rfloor$, where t is the plaintext modulus (e.g., 2 for a binary message). The expensive part of $\mathsf{Scale}_{Q\to q}$ is the division by q operation.

The HPS algorithm [7] approximates this expensive division too and makes $\mathsf{Scale}_{Q\to q}$ computation possible without the need to compute any multi-precision arithmetic. The scaling operation is divided into two steps. In the first step, the coefficients are moved from the RNS of Q, which consist of thirteen primes, to the RNS of modulus $p = Q/q$, which

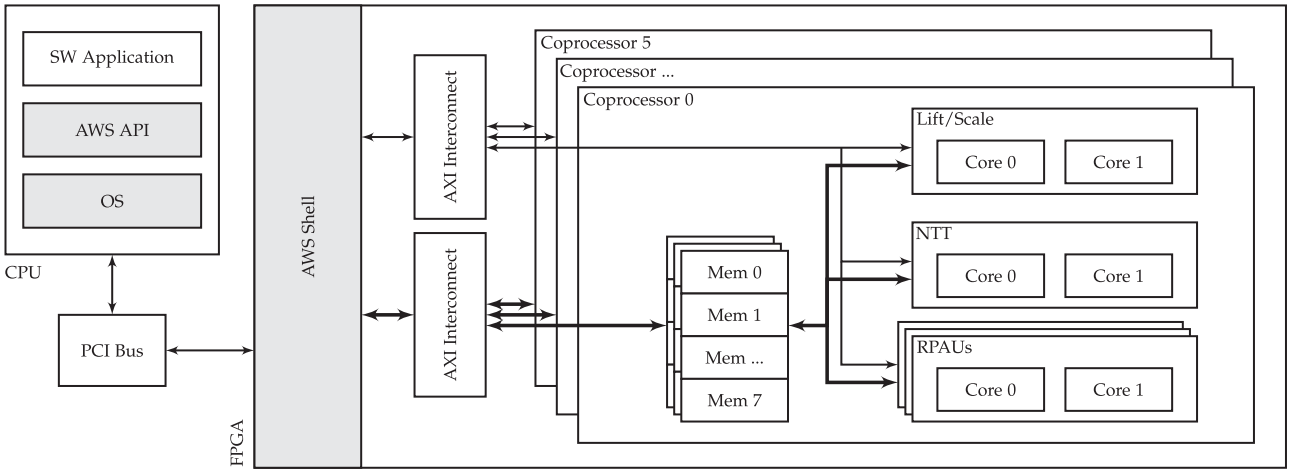

Fig. 3. High-level architecture of HEAWS. We instantiate six coprocessors in a single FPGA accelerator on a single F1 instance of Amazon AWS. The block diagram shows the connectivity between the memories and processing units of each coprocessor. The thin lines indicate 32-bit wires for wires for accessing control signal, while thick lines denote data transfers. The polynomial computation benefits from the Residue Number Arithmetic; hence, each coprocessor consist of seven parallel Residue Polynomial Arithmetic Unit (RPAU)s, each accessing the corresponding memories in parallel. The RPAUs, NTT and Lift/Scale units are present inside two parallel executing computation cores. The F1 instances connects the FPGA to CPU over the PCI bus; however, it provides the gray coloured blocks for a simplified communication. The AWS Shell offers an AXI-based interface to the accelerators, and applications use AWS API to communicate with their accelerators. The coprocessors connect to the shell through AXI interconnects.

consists of seven primes. The corresponding calculation is for the seven RNS coefficients $6 \leq j \leq 12$, such that

$$\lceil t/q \cdot x \rfloor_{q_j} = \left[\left\lceil \sum_{0}^{5} a_i \cdot \frac{t\tilde{Q}_i p}{q_i} \right\rfloor + a_j \cdot t\tilde{Q}_j q_j^* \right]_{q_j}, \tag{4}$$

where $\tilde{Q}_k = (Q/q_k)^{-1} \bmod q_k$ for both $k = i$ and $k = j$. This is a computation friendly approach, because all operands including the constants are 30-bit values. In the next step, the coefficients are moved from the RNS of p to q by following a basis conversion. How the $\mathsf{Scale}_{Q \to q}$ operations are implemented will be described in Section 3.3.

3 Architecture

During a homomorphic computing, different computationally expensive operations are performed on large-degree polynomials with large coefficient sizes. The section above described various techniques from the literature for fast computation over such polynomials. Turning these mathematical operations into a hardware building blocks and integrating them into a processor architecture that can compute on encrypted data *efficiently*, demands multiple optimizations, such as reducing on-chip and off-chip memory access cost, design of arithmetic blocks for large coefficients, turning serial computations into parallel computations, and achieving high clock frequency. In this section, we describe our architecture design with detailed explanation of its processing units, and how the mathematics are employed on them. A high level block diagram of the proposed HEAWS architecture is given in Fig. 3. The figure shows the basic processing units of a coprocessor, multiple coprocessor instances in a singe FPGA accelerator, and the interface connecting them to software.

3.1 Polynomial Arithmetic Unit

The basic arithmetic units of a general-purpose processor operate on integer (or floating point) data; however, the data processed with homomorphic encryption is of type 'polynomial'. Therefore, we make use of FPGA specific memory blocks, i.e., Block RAM (BRAM), to store the polynomials. Our arithmetic unit computes using nine memories. The eight of them are for storing polynomials. However, the ninth is not a physical memory, but imitates a memory filled with zeros, i.e., a polynomial with zero coefficients. The zero polynomial is useful for executing *move* operations as additions with zero. Besides the polynomial arithmetic, the other functional units (for lift and scale operations) also operate on these memories as shown in Fig. 3. The details of their architecture will be described in the following subsections.

Note that, for the chosen parameter set, q and Q are composed of six and thirteen primes respectively. This allows dividing the corresponding polynomials of q and Q into six or thirteen smaller-coefficient residue polynomials, and processing them in parallel with smaller arithmetic units in *Residue Polynomial Arithmetic Units* (RPAUs). Besides parallelism, the RNS offers various advantages in hardware, such as flexible placement of these units and avoiding long carry propagation. Both are helpful for achieving a high clock frequency. However, there is also a disadvantage of working with the RNS representation. It requires *merging* the coefficients when moving them between the modulo q and Q, required by the $\mathsf{Lift}_{q \to Q}$ and $\mathsf{Scale}_{Q \to q}$ operations in the FV scheme.

While the use of RNS enables processing the residue polynomials in parallel (algorithmic parallelism), an extra level of parallelism is introduced at the architecture-level by instantiating multiple computation-cores inside the RPAUs. The two basic operations performed by these RPAUs are addition and multiplication of the residue polynomials. The addition is already a coefficient-wise operation, which can be parallelized easily. The multiplication of two residue polynomials modulo q_i is performed using the NTT-based polynomial multiplication method which was described in Section 2.2.

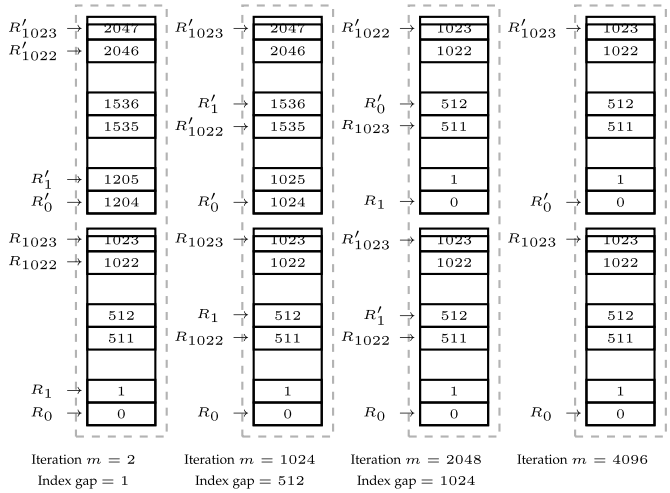

Fig. 4. Access pattern of memory addresses during an NTT using two cores are shown. The dashed lines show a memory block, the upper and lower halves of which are implemented with different BRAM blocks, and two coefficients are stored at each address. The columns indicate various loop iterations of the NTT. Each column shows the accessed coefficient pairs by two NTT cores respectively with R_i and R'_i.

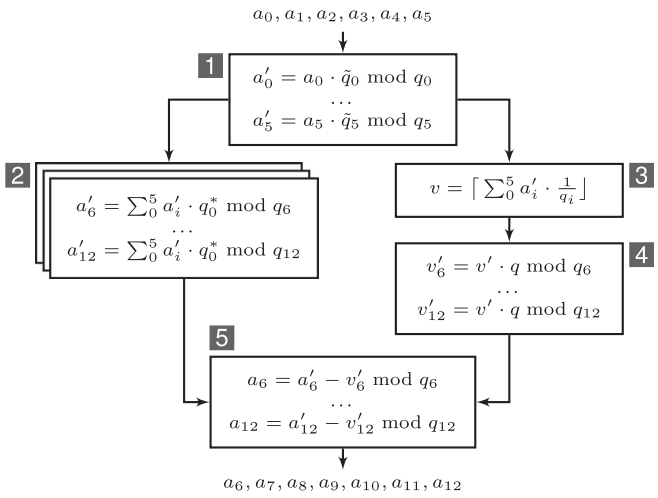

Fig. 5. The lift operation is divided into pipelined smaller blocks of computation. At every seventh clock cycle, an input polynomial coefficient is read (as a set of six RNS shares), and processed data progresses through the pipeline. For the the relatively expensive computation of Block 2, parallel executing Multiple-and-Accumulate (MAC) units are preferred.

3.1.1 The NTT Unit

The calculation of Number Theoric Transform (NTT) is also parallelized for further accelerating the computation. In fact, the mathematical cost of NTT operation is already low, but that does not capture the implementation cost. Its implementation is constrained with the cost of memory access operations, and data-dependencies between the corresponding calculations of accessed data. These data dependent operations are annotated as 'butterfly operation' in Algorithm 1. Therefore, achieving a parallelized NTT computation requires modification in the algorithm taking memory access into consideration.

Irregular memory access pattern during butterfly operations (annotated in Algorithm 1) makes the parallel computation difficult by introducing data dependency between the processed coefficients. The number of BRAM access ports is a factor to be considered for overcoming the data dependency problem. With two ports, a write and read operation can be executed every cycle, for two different memory locations. Therefore, we decided for two-fold parallelism by processing four polynomial coefficients every cycle. This requires a polynomial memory as a 2-by-2 BRAM construction; having 2,048 memory addresses each storing two coefficients. As a result, we read, compute, and write four coefficients at every cycle with two butterfly cores, each processing two coefficients. This is made possible by the designed memory access scheme, which enables the parallel computation with these two butterfly cores. This scheme is described in the following paragraphs.

The loop dependent indexes of NTT butterfly cores introduce a complex access pattern. A parallel execution of two cores complicates this pattern further. One difficulty is to avoid memory access conflicts, i.e., prevention of cores from simultaneous read or write accesses to same memory locations. Another difficulty is to pair the coefficients $A[k+j]$ and $A[k+j+m/2]$ (see the lines 9 and 10 of Algorithm 1) at the same memory location, so that they can be accessed together.

The loop variable m of the NTT (in Algorithm 1) creates an increasing index-gap between the two accessed coefficients at each loop iteration. The index-gap and accessed memory indexes at each loop iteration by the two NTT cores are shown in Fig. 4 for the different values of m. The gap doubles at each iteration, until it reaches 512 for m equal to 1,024. That allows to make two butterfly cores access to upper and lower memory blocks (of the above mentioned 2-by-2 memory construction) exclusively, i.e., with the memory address ranges [0,1023] and [1,024, 2,047] respectively. These ranges allow to make cores operate exclusively on different memory blocks. When m reaches to 2,048, the index-gap increases to 1,024, which means that both cores demand simultaneous access to the same memory block. The memory access conflicts caused by this demand is prevented by inverting the memory address requests of the second core. In other words, the first core accesses the lower memory block first and then the upper memory block, with the address sequence 0, 1,024, 1, 1,025, and so on. Whereas, the second core accesses the upper memory block first and then the lower memory block, with the address sequence 1,536, 512, 1,537, 513, and so on. As a result, the two cores access the upper and lower memory blocks exclusively, effectively avoiding any memory access conflicts. At the last loop iteration, m is 4,096, and the access pattern described by [16] as *one memory word at a time* is used, again for making two butterfly cores operate exclusively on the upper and lower memory blocks.

3.2 Lift Unit

The $\mathsf{Lift}_{q\to Q}$ operation described in Section 2.5 receives each of 4,096 polynomial coefficients as a set of six RNS shares in ring R_q, and calculates seven more shares for R_Q. The designed architecture divides the corresponding coefficient-wise computation into smaller pieces, which are shown in Fig. 5 as pipelined and parallel computation blocks, in combination with parallel computation within them. Each block processes at most seven RNS shares, hence a seven cycle pipeline is applied. At each seventh cycle the blocks hand over their outputs to the successors, and receive the inputs from the predecessors.

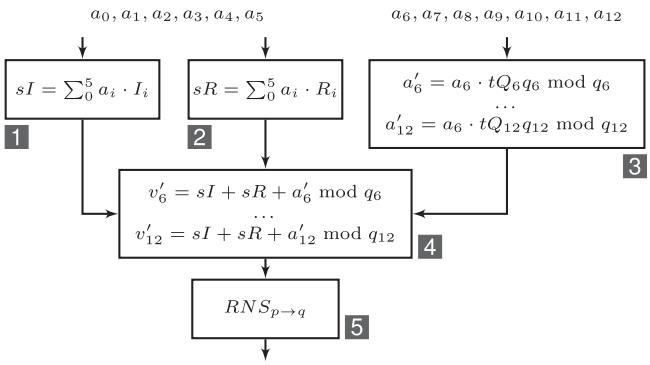

Fig. 6. The scale operation is divided into smaller blocks of computation. These blocks allow block level parallelisation and pipelining.

The used HPS optimization [7] is advantageous for lift in hardware, as it replaces large-integer operations with small-integer arithmetic, such that the operations can be executed efficiently and in parallel. Block 2 shown in Fig. 5 has the highest computation cost, thus it uses seven parallel MAC. On top of that, Block 2 is kept in parallel to Blocks 3 and 4 to implement block-level parallel processing. In our architecture, the division operation of the Block 3 is replaced with multiplications by the divisions' reciprocals. For these reciprocals, we consider 89-bit precision after the decimal point. However, the first 29 bits after the decimal point of each reciprocal are zero. Therefore, our architecture uses a 60-bit integer multiplier hardware. The 89-bit precision yields us an approximation error less than 2^{-80}, while the original implementation could only achieve 2^{-53} using the IEEE 754 double floats [7].

The described lift architecture only receives the six RNS shares for each coefficient in Ring R_q, and produces seven shares for R_Q. These input and output are shown in Fig. 5 as $a_0, \ldots a_5$ and $a_6, \ldots a_{12}$ respectively. While the q values of the figure are the computation constants, so hard-coded into the computation blocks.

3.3 Scale Unit

The architecture of the $\mathsf{Scale}_{Q\to q}$ operation (described in Section 2.6) divides the computation into blocks as shown in Fig. 6. Similar to the lift operation, these blocks are executed in block-level parallelism and in pipeline.

The Blocks 1 and 2 in the figure consist of MAC hardware. The R_i and I_i values on these two blocks represent the real and imaginary parts of the $t\tilde{Q}_i p/q_i$ in Equation (4). When Block 4's computation is completed, all the input shares in the RNS of Q are represented with six shares in the RNS of p. Then, Block 5 moves them from the RNS of p to q, by reusing the $\mathsf{Lift}_{q\to Q}$ operation described in the previous section.

In our architecture we preferred to have one processing unit unifying the lift and scale operations, as shown in Fig. 3. Furthermore, this processing unit consist of two computation cores for increased throughput.

Our architecture provides dedicated hardware for computation blocks shown in Fig. 2, except *word decomposition* and *relinearisation*. The former is based on a bit-select operation 2.1, and integrated into the scale unit as proposed by Halevi *et al.* [7]. In comparison, the latter can be implemented with basic polynomial arithmetic as described in Section 2.1, without requiring a special hardware. We opt to transfer its required relinearisation key from CPU to the coprocessor, instead of reserving on-chip memory elements for it. The efficiency of this decision depends on fast data transfer capability, which will be described in the next section.

4 Implementation

We implemented the above described HEAWS architecture as a homomorphic encryption coprocessor which has a dedicated instruction set. These instructions are used for coefficient-wise addition/multiplication, NTT, inverse-NTT, coefficient rearrangement, $\mathsf{Lift}_{q\to Q}$, and $\mathsf{Scale}_{Q\to q}$. A sequential execution of these instructions allows to build homomorphic operations, such as arithmetic over ciphertext. In this respect, our coprocessor has a domain specific processor architecture. In a single FPGA on the Amazon AWS, we instantiate six coprocessors in parallel. In this section, we describe the implementation details, and explain the trade offs between increasing the number of parallel coprocessors and clocking them at the maximum frequency possible.

4.1 Hardware-Software Interface

Since homomorphic encryption is computation- as well as communication-centric, the overhead of data transfer across the hardware-software interface plays a critical role in the performance and can even result in slower computation [4] than software! Hence, in this design we take special care to reduce the data transfer overhead while designing the hardware accelerator. For accelerating homomorphic computations in the cloud, we chose Amazon AWS *F1 instances* as the target platform. An F1 instance is a CPU-FPGA heterogeneous platform where a large Xilinx FPGA is tightly attached to an Intel CPU through a high-bandwidth PCIe interface.

A high-level block diagram of our architecture on an F1 instance is shown in Fig. 3. The F1 instance provides a wrapper, known as *AWS Shell* to encapsulate an accelerator module using partial reconfiguration. The shell also provides interfaces for the communications between hardware and software. In particular, various communication ports are given to hardware accelerators on the FPGA side, and corresponding APIs are used by their respective software applications on the CPU side.

The communication ports of AWS Shell are of two types. The first type is a 32-bit port for performing address-mapped communication between applications and accelerators. The other type is a 512-bit port for Direct Memory Access (DMA) of the accelerators. In our architecture, we use the 32-bit port for sending instructions to the FPGA-based accelerator, and checking its status from the software applications. The 512-bit data transfer port is used to send input operands to the accelerator and receiving outputs. For executing the data transfers fast, we used the *XDMA* facility provided by the Amazon.

To minimise the data transfer overhead, we opted for writing the residue polynomials to the memories of RPAUs (and reading from them) in parallel. We store each polynomial coefficient as a set of 64-bit aligned eight residue polynomial coefficient pairs. The input and output ciphertext polynomials are in modulo q, which consist of six RNS shares. Hence, the next two shares are spared in the

memory. This structure allows to have a new polynomial coefficient (i.e., a set of seven residue coefficient pairs) at the address of next 512th bit. As a result, a 512-bit wide XDMA transfer of length 2,048 is enough to move a polynomial between the system memory and the RPAU memories.

The memory management of software applications offer minor improvements over the data transfer overhead. The size of each 512-bit aligned polynomial in modulo q mentioned above is 128 KiB. Since the default page size in Linux is 4 KiB, a polynomial in modulo q consist of 32 pages in total. The XDMA mode of data transfer requires several bursts as the OS is expected to place the pages in arbitrary memory locations. Hence, the transfer is divided into chunks of page size at maximum. The DMA is configured in the scatter-gather mode, which allows to set up a link-list kind of a configuration for a sequence of DMA transfers. To keep the length of this list minimum, we used a page aligned memory allocation for the polynomials. As each polynomial requires exactly 32 pages, this structure prevents creation of any partially occupied first and last pages, and eliminates need for any extra transfer related to fragmentation. This simple consideration improves the data transfer time of a ciphertext by upto 4 percent.

4.2 Parallel Instantiation of Coprocessors

We add another level of parallel processing at the highest level of the design hierarchy by instantiating multiple parallel coprocessors, each capable of evaluating independent homomorphic operations, in the FPGA. The number of coprocessors that can be instantiated depends on the amount of available resources in the target FPGA.

The design heavily uses FPGAs memory resources for storing polynomials, and NTT coefficients. The implementation benefits from the new generation URAMs that offer extra storage on high-end Xilinx UltraScale(+) series FPGAs, preferred on our target Amazon F1 instances. URAMs are placed in the FPGA in columns parallel to BRAMs. As a result, the URAMs and BRAMs are not physically distant from each other. Therefore, an instantiation strategy of picking BRAMs for odd-indexed and URAMs for even-indexed coprocessors requires interleaving the coprocessors. However this kind of interleaving results in long critical paths which increase the routing delay. Hence, avoiding this kind of interleaving is required for achieving high clock frequency. We constructed the memory in an isometric manner using both URAMs and BRAMs within each coprocessor. For each RPAU memory described in Section 3.1, we implement the upper part of a 2-by-2 memory using BRAMs and lower part using URAMs. In Section 5 we discuss the impact of this strategy on both resource utilization and critical delay.

A big challenge in implementing a large and massively parallel hardware accelerator is the placement of modules that are distant from each other on the floor-plan. Note that, data transfer between distant modules require long wires which could easily turn into long critical paths due to high net delay. In our architecture the HE coprocessors are unconnected individual blocks and work independently. However, the problem of long routing appears in the implementation of the IO wires, which are basically AXI interconnect signals extending from the AXI port on the AWS Shell to the farthest coprocessors. In order to break these long interface wires, we put several layers of registers in the AXI interconnects and chose a high performance synthesis strategy.

A configuration users have when attaching multiple AXI modules to each other is the data-width of AXI Crossbar, connecting the modules to each other. In our instance, it is used to connect multiple coprocessor to the same AXI port on the AWS Shell. When the crossbar's data-width does not match with the width of input or output ports, either down or up conversion is performed. That conversion yields to an extra communication overhead. In the case of our implementation, the AWS Shell already has an AXI port of 512-bit data-width, and our parallel transfer scheme also prefers 512-bit wide data. According to our experiments, if a 32-bit crossbar is used instead of a 512-bit, the up-and-down conversions slows down the data transfers 4 times, and computation of a homomorphic multiplication 1.6 times. However, 512-bit crossbar is a costly hardware particular for a large design such as ours, requiring many wires to extend from one side of the FPGA to another. Sweeping over different widths, we found that 64-bit crossbar is the sweet spot of our implementation, that yields six coprocessors without decreasing the clock frequency.

Working with the FPGAs of AWS Cloud disallows using some features of the hardware development tool. For example, floorplanning is a technique for giving instructions to the software for the placement of hardware modules on the FPGA. Especially for big target FPGAs, manual floorplanning helps achieving better routing to meet a given clock constraint. In the case of AWS Cloud, the FPGA design tool performs implicit floorplanning and does not offer an option for manual floorplanning to the designers. We observed a tendency of the design-tool to place the hardware modules at the central region of the FPGA, leaving the resources residing near the edges unused. With manual placement, we think that one more coprocessor could fit on the FPGA; however, uncontrollable placement did not let realize that without reducing the clock frequency.

As multiple coprocessors are instantiated in a single FPGA, multiple applications can accelerate their computation simultaneously. Alternatively, a single application can use all these coprocessors. Although, the plurality of coprocessors is useful for acceleration of the *computation* only, it introduces limitations in the input/output data transfers to them. Since there is only a single instance of the XDMA to communicate with the coprocessors, it is used in a time-multiplexed manner. The overhead of time-multiplexed data transfer is provided in Section 5.

5 Results

In this section we describe the performance and utilisation of the designed accelerator architecture HEAWS. First, the computation times for homomorphic addition and multiplication are presented. Next, the accelerator is used to benchmark a homomorphic application, namely evaluation of privacy-preserving neural network that works on encrypted data for forecasting in the Smart Grid application. Resource utilisations on the target FPGA platform are also presented. Finally, comparisons of performances as well as estimated financial costs between the two cases: homomorphic computing using FPGAs or GPUs, are demonstrated.

TABLE 1
Execution Time of Homomorphic Operations on Amazon AWS F1 and Comparisons With Our Previous Implementation [6] on Zynq SoC

Operation	Milliseconds on Zynq [6]	on AWS F1	
Number of parallel coprocessors	1	1	6
Homomorphic Multiplication	4.46	4.34	9.78
Homomorphic Addition	0.03	0.01	0.01
Polynomial Write	0.18	0.11	0.18
Polynomial Read	0.18	0.11	0.18

Both Zynq and AWS F1 implementations are clocked at 200 MHz.

TABLE 2
FPGA Resource Utilisation of the Designed Hardware Accelerator With Six Homomorphic Coprocessors Instances on AWS F1 Instances

	LUT	REG	DSP	BRAM	URAM
Available					
on FPGA	1181768	2363536	6840	2160	960
for Accelerator	895100	1790400	5640	1680	800
Used by					
AWS Shell	213600	427200	1008	384	43
Accelerator	369049	207333	1248	1506	336
A Coprocessor	57877	25648	208	249	56

5.1 Performance of Arithmetic Operations

The maximum clock frequency our accelerator achieves is 200 MHz. However, the AWS Shell's main frequency is set to 250 Mhz and this frequency determines the speed of the AXI master interface. The execution time of homomorphic operations at these clock frequencies are shown in Table 1. When a single coprocessor is used, each homomorphic multiplication operation takes 4.34 ms, which corresponds to 230 multiplications per second. In addition, the cost of each ciphertext transfer is 0.11 ms.

The table also shows the measurements when six coprocessors are used simultaneously. As their simultaneous execution requires time-multiplexing the data transfer interface, a decrease in the per-multiplication performance is observed. When an application runs six parallel threads, each using a coprocessor for computing homomorphic multiplications in a loop, an overhead of 125 percent in the computation time is observed. Due to this overhead we could execute 613 multiplications per second using one FPGA on AWS F1.

Our design implements a cloud service on the Amazon AWS for computing on the encrypted data, and the software components of the service run on Linux. In comparison, our previous design [6] runs baremetal (i.e., without OS) on a Zynq SoC device. The availability of an OS offers flexibility as third party libraries can be installed, making application development easier. Moreover, applications can manage the jobs and resources using the available system calls. One disadvantage is that an application cannot directly access the hardware, but requires interactions with the OS or its kernel drivers. When these interactions take place, corresponding function calls require context switching between the user and kernel space. Although our design encapsulates these overheads, it still achieves a minor performance improvement over our previous design [6]. This improvement is mostly an outcome of a better HW/SW interface design.

5.2 Benchmarking Homomorphic Evaluation of a Simple Artificial Neural Network (ANN)

To see how far can we push homomorphic computation on encrypted data using our hardware accelerator, we benchmarked a simple privacy-friendly forecasting algorithm for the Smart Grid. The application offers users more control over the consumption, by forecasting energy consumption by measuring the users' consumption data in real-time. However, the smart grid raises privacy issues as it collects large amount of data from users. There have been several initiatives that try to protect the privacy of the users in the smart grid application scenario. Recently, Bos *et al.* designed a privacy-friendly forecasting for the smart grid using the FV homomorphic encryption scheme [5]. Since homomorphic multiplication is computationally very expensive, they designed [5] a low-complexity Neural Network by applying the group method of data handling [17] for simplifying the activation function of the neurons that are used in the network. They found that the required multiplicative depth for evaluating the targeted neural network is only four. They implemented a C++ library for the forecasting algorithm that uses NFLLib library [18] for the homomorphic procedures. On an Intel i5 laptop, their software takes half a minute to forecast energy consumption for an apartment of 10 houses.

We started from their software implementation of the high-level forecasting algorithm, and our coprocessor for accelerating the required homomorphic multiplications. The entire implementation is run on Amazon AWS where a single-thread software program executes on processor, and makes use of *one* coprocessor on the FPGA. First, we used the software-only implementation, and found that each forecasting requires 9.46 seconds. When we use the hardware-software codesign, where homomorphic multiplications are computed using one coprocessor, each forecasting takes only 1.73 seconds. These results show that more than five times acceleration is feasible with a single coprocessor. With a multithreaded software implementation, we could run six independent privacy-preserving forecasting in parallel using the six coprocessors that are available in the FPGA. We estimate that, using such application-level parallel processing, one F1 instance could achieve 15.7 privacy-preserving forecasting every minute.

5.3 FPGA Resource Utilisation

The FPGAs available on Amazon AWS are big devices of the Xilinx Virtex UltraScale+ family. As a result, they offer plenty of programmable hardware resources. Nearly 20 percent of these are reserved already for the AWS Shell, and hence cannot be used by the accelerators. The rest of the FPGA resources are available for implementing the architecture. Table 2 shows the amount of available resources in an F1 FPGA, the area consumption by the AWS Shell, our entire accelerator and a single coprocessor instance. The individual resource utilisation of sub-computation modules are given in Table 3, which will be explained later in the next section.

As shown in the tables, the coprocessor implementation has a heavy memory utilisation. That is an expected outcome

TABLE 3
Cost of a Single Coprocessor (Having Two Instances of Each Computation Core), Resource Overhead of Each Core, and the Estimated Execution Time of Homomorphic Multiplication Different Number of Cores

	Resources					Mult. Time in % w/ Core Count		
	LUT-REG-BRAM-URAM-DSP					1	2	4
Copro.	57,877	25,648	249	56	208			
NTT	483	80	69	0	0	81	100	137
Lift	13,807	6,157	0	0	76	94	100	112
RPAU	12,274	6,528	0	0	28	87	100	125

as the hardware is designed to process polynomials, the storage of which requires memory. In addition, memories are also used to store the NTT computation constants, e.g., to store precomputed inverse of the modulus. As a result, the number of parallel coprocessors is determined by the availability of memory slices in the FPGA. In our implementation the number of coprocessors is limited to six. Fitting the sixth coprocessor was made possible by making use of the large URAMs. In fact, another coprocessor (i.e., the seventh) could also be fit if more URAMs are used instead of BRAMs. We found that with seven coprocessors, the implementation requires lowering the clock frequency.

5.4 Different Design Choices

The above given results are for our architecture configured with two core RPAU, NTT, and Lift modules. Table 3 summarizes the resource cost of each core on it, and the estimated execution time of homomorphic multiplication using a coprocessor configured with one, two, and four instances of each computation cores.

A homomorphic multiplication is computed using less polynomial memories in this design, compared to the work of Roy *et al.* [6]. That is made possible with a revision on the set of instructions to execute a multiplication. This revision re-computes some intermediate values, instead of computing them once, and storing them in memory for long because they will be re-accessed in the later parts of the computation. With this revision, we reduced the number of polynomial memories (see Fig. 3) from 9 to 8 at the cost of only 6 percent increase in homomorphic multiplication time. It saves extra 5 percent BRAM and 12.5 percent URAM utilisation, helping us instantiate more coprocessors in the FPGA.

5.5 Estimates for Other Parameter Sets

We estimate the resource and performance results of a single coprocessor for different parameter sizes, assuming that they will scale proportionally with the parameter set. The number of RPAUs and Lift/Scale cores should double for every doubling of both polynomial degree n and coefficient size $logq$ of the parameter set. Their individual computation time should increase $\approx 4.34\times$, and resource utilisation by $\approx 2\times$. That yields an increase the net computation time by $\approx 2.17\times$, and off-chip data transfer time by $\approx 4\times$. Table 4 presents resource utilisation and computation time estimations for various extrapolated parameter-sets that have a security at least 80-bits [14].

TABLE 4
Estimations of Single Coprocessor Results for Different Parameter Sets

Parameter $(n, \log q)$	Resources LUT - REG - U+BRAM - DSP				Mult. Time (msec) Comp - Comm -Total		
$2^{12}, 180$	58K	34K	0.3K	0.2K	4.34	0.33	4.67
$2^{13}, 360$	116K	68K	0.6K	0.4K	9.42	0.72	10.13
$2^{14}, 720$	232K	136K	1.2K	0.8K	20.44	1.55	21.99
$2^{15}, 1,440$	464K	272K	2.4K	1.6K	44.35	3.37	47.72

5.6 FPGA- Versus GPU-Based Acceleration

Badawi *et al.* [19] implemented a high performance FV library targeting NVIDIA's Tesla V100 GPUs, designed for hyperscale data center workloads, and consist of 5,120 cores with 16 GB RAM and a system clock of 1.38 GHz. Their implementation for a parameter set with a 60-bit ciphertext modulus performs a homomorphic multiplication in 0.86 msec. We estimate that its computation time will increase at least by three times for a 180-bit modulus. With this scaling, a Tesla V100 GPU can perform around 388 multiplications per second. In comparison, our design with 6 coprocessors achieves 613 multiplication per second. Note that GPUs have much higher power consumption than FPGAs.

In addition to timing, we compare the cost of using GPUs and FPGAs in the Amazon AWS. According to the AWS's Cost Calculator[1] using a Tesla V100 GPU is 1.85 times expensive than an FPGA. When both timing and financial costs are taken into account, our FPGA-based accelerator would perform nearly 2.9 times more work per dollar than a GPU instance. We remark that these figures may change with time depending on different factors. However, to the best of our knowledge, comparative prices have not changed since February 2019.

5.7 Comparison to Other FPGA-Based Accelerators

The literature is scarce with FPGA-based accelerators of HE, and comparing them is hard when they target different schemes. For a fair comparison we focus on HEAX [20]. It is another FPGA accelerator, which implements many-fold parallel computation with low-level hardware optimisations. Performance comparing of homomorphic operations are tough, as it prefers approximate computing with CKKS scheme. Its resource utilisation (on Intel FPGAs) are much higher than ours, so does its performance. For example, it has dedicated NTT and Inverse-NTT modules, each requires more resources than our unified one. In return, its NTT computation is $\sim$8 times fast on same-sized polynomials.

The accelerator of Mert *et al.* [21] targets the FV scheme as ours. They focus on encryption/decryption operations handled on users' local computer, while we aim at accelerating the homomorphic operations executed on cloud. The most expensive operation in their design is NTT. Hence they specifically target accelerating it by using the four-step NTT algorithm [22]. For NTT only, they spend more resources than a single coprocessor does in our case, and achieve around 40 times faster NTT calculation.

1. Online. [Available]: https://calculator.s3.amazonaws.com/index.html

6 Conclusion

In this paper we presented a programmable and high-performance domain specific architecture, which we call HEAWS, and implemented it in an Amazon AWS F1 FPGA, for accelerating homomorphic computing on the encrypted data in the cloud. To achieve fast homomorphic evaluation time, we applied specific algorithmic and architectural optimization techniques for all the computation blocks. In addition, we benefit from parallel computation in many levels of the design hierarchy. In summary, we implemented an instruction-set coprocessor, and instantiated six copies of it on a single Amazon AWS F1 FPGA. In addition, we provided them with a custom-made HW/SW interface, for the fast transfer of huge polynomial data between our massively parallel architecture HEAWS and attached CPU.

HEAWS achieves 613 homomorphic multiplications per second resulting in 20 times performance improvement for the privacy-preserving forecasting in the Smart Meter application with respect to a software implementation. In comparison to a GPU acceleration on the Amazon AWS, it offers almost 3 times more computation per dollar cost.

Acknowledgments

This work was supported in part by the KU Leuven Research Council through C16/15/058, in part by the ERC Advanced Grant 695305 Cathedral, in part by the University of Birmingham and the Ramsay research support fund. Furkan Turan was supported by German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] Notice of data breach: May 2016, " *Linkedin*, 2016. [Online]. Available: https://www.linkedin.com/help/linkedin/answer/69603/notice-of-data-brea ch-may-2016?lang=en
[2] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 44, pp. 169–179, 1978.
[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 169–178.
[4] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.
[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. Cryptol.*, 2017, pp. 184–201.
[6] S. S. Roy, F. Turan, K. Järvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. 25th IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 387–398.
[7] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Proc. Cryptographers Track RSA Conf.*, 2019, pp. 83–105.
[8] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Archive*, vol. 2012, 2012, Art. no. 144.
[9] V. Lyubashevsky, A. Palacio, and G. Segev, "Public-key cryptographic primitives provably as secure as subset sum," in *Proc. Theory Cryptography Conf.*, 2010, pp. 382–400.
[10] D. J. Bernstein, "Fast multiplication and its applications," *Algorithmic Number Theory*, vol. 44 2008, pp. 325–344.
[11] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk*, vol. 145, no. 2, 1962, pp. 293–294.
[12] D. E. Knuth, Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd ed., Addison-Wesley, 1998.
[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The Cambridge, MA, USA and New York, NY, USA: MIT Press and McGraw-Hill Book Company, 2001.
[14] M. R. Albrecht, "Complexity estimates for solving LWE," [Online]. Available: https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py
[15] C. Dobraunig *et al.*, "Rasta: A cipher with low ANDdepth and few ANDs per bit," in *Proc. 38th Annu. Int. Cryptol. Conf.*, 2018, pp. 662–692.
[16] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *Proc. 16th Int. Workshop Cryptogr. Hardware Embedded Syst.*, 2014, pp. 371–391.
[17] A. Ivakhnenko, "Heuristic self-organization in problems of engineering cybernetics," *Automatica*, vol. 6, no. 2, pp. 207–219, 1970.
[18] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "NFLlib: NTT-based fast lattice library," in *Proc. Cryptographers Track RSA Conf.*, 2016, pp. 341–356.
[19] A. A. Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IACR Cryptol. ePrint Archive*, vol. 2018, 2018, Art. no. 589.
[20] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 1295–1309.
[21] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 353–362, Feb. 2020.
[22] J. W. Cooleyand J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

Furkan Turan is currently a research assistant at the COSIC research group at KU Leuven, Leuven, Belgium. His research interests include hardware-software co-design for FPGA-accelerated cloud and trusted computing, and the hardware implementations of cryptographic algorithms.

Sujoy Sinha Roy is currently an assistant professor in cyber security at the University of Birmingham's School of Computer Science, and a member of the Security and Privacy Group which is recognised by the UK National Cyber Security Centre (NCSC) in partnership with EPSRC as an Academic Centre of Excellence in Cyber Security Research (ACE-CSR). He is an expert in cryptographic engineering, especially with respect to lattice-based public-key cryptography and homomorphic encryption. His PhD thesis received the "IBM Innovation Award 2018", a scientific prize that acknowledges an outstanding doctoral thesis in informatics.

Ingrid Verbauwhede is currently 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, University of California, Los Angeles, California. She is a member of the Royal Academy of Belgium for science and the arts. She is a recipient of an ERC Advanced Grant, in 2016. She received the IEEE 2017 Computer Society Technical Achievement Award.