Accelerating LTV Based Homomorphic Encryption in Reconfigurable Hardware

Yarkın Doröz[1]([✉]), Erdinç Öztürk[2], Erkay Savaş[3], and Berk Sunar[1]

[1] Worcester Polytechnic Institute, Worcester, USA
{ydoroz,sunar}@wpi.edu
[2] Istanbul Commerce University, Istanbul, Turkey
eozturk@ticaret.edu.tr
[3] Sabanci University, Istanbul, Turkey
erkays@sabanciuniv.edu

Abstract. After being introduced in 2009, the first fully homomorphic encryption (FHE) scheme has created significant excitement in academia and industry. Despite rapid advances in the last 6 years, FHE schemes are still not ready for deployment due to an efficiency bottleneck. Here we introduce a custom hardware accelerator optimized for a class of reconfigurable logic to bring LTV based somewhat homomorphic encryption (SWHE) schemes one step closer to deployment in real-life applications. The accelerator we present is connected via a fast PCIe interface to a CPU platform to provide homomorphic evaluation services to any application that needs to support blinded computations. Specifically we introduce a number theoretical transform based multiplier architecture capable of efficiently handling very large polynomials. When synthesized for the Xilinx Virtex 7 family the presented architecture can compute the product of large polynomials in under 6.25 msec making it the fastest multiplier design of its kind currently available in the literature and is more than 102 times faster than a software implementation. Using this multiplier we can compute a relinearization operation in 526 msec. When used as an accelerator, for instance, to evaluate the AES block cipher, we estimate a per block homomorphic evaluation performance of 442 msec yielding performance gains of 28.5 and 17 times over similar CPU and GPU implementations, respectively.

Keywords: Somewhat homomorphic encryption · NTT multiplication · FPGA

1 Introduction

Fully homomorphic encryption (FHE) is a promising new technology that enables efficient blinded computations on semi-trusted servers. The introduction of the first plausible FHE construction by Gentry in 2009 [19,20], fueled the race to develop more efficient schemes. More specifically, lattice-based [21,22,32], integer-based [10,11,15] and learning-with-errors (LWE) or (ring) learning with

errors ((R)LWE) based encryption [6,23,24] schemes were introduced in just a few years. Despite the rapid progression of new FHE optimization techniques such as ones developed to render expensive bootstrapping evaluations obsolete [5] and ones for more effective parallel processing through *batching* of multiple data bits into a ciphertext [4,9,33], FHE is still far from being ready for use in real-life applications. For instance, an implementation by Gentry et al. [25] homomorphically evaluates the AES circuit in about 36 hours resulting in an amortized per block evaluation time of 5 minutes. Another NTRU based proposal by Doröz et al. [16] manages to evaluate AES roughly an order of magnitude faster than [25]. Still it does not come close to what is acceptable in practice. The main difficulty in developing efficient FHE schemes is to overcome the massive parameter sizes necessary to retain security while allowing evaluation of deep circuits.

Clearly the gap between what is currently achievable on a CPU and what is practical is too far to consider software only solutions. This led researchers to investigate the use of alternative platforms such as graphic processing units (GPUs), reconfigurable logic such as FPGAs, and even further domain specific ASIC designs to accelerate homomorphic evaluations. Using Nvidia GPUs, for instance, Wang et al. [35] managed to accelerate the earlier implementation of the recryption primitive of Gentry and Halevi [23] by roughly an order of magnitude. The GPU library is capable to evaluate AES under 8 seconds. On the hardware side, Cousins et al. report the first reconfigurable logic implementations in [12,13], in which Matlab Simulink was used to design the FHE primitives. This was followed by further investigation in this direction [7,29,36,37]. Specifically, in [36], Wang et al. present an optimized version of their result [37], which achieves speed–up factors of 174, 7.6 and 13.5 for encryption, decryption and the recryption operations on an NVIDIA GTX 690, respectively, when compared to results of the implementation of Gentry and Halevi's FHE scheme [22] that runs on an Intel Core i7 3770 K machine. Cao et al. [7] proposed a number theoretical transform (NTT)-based large integer multiplier combined with Barrett reduction to alleviate the multiplication and modular reduction bottlenecks required in many FHE schemes. The encryption step in the proposed integer based FHE schemes by Coron et al. [10,11] were designed and implemented on a Xilinx Virtex-7 FPGA. The synthesis results show speed up factors of over 40 over existing software implementations of this encryption step [7]. A more recent work by Dai et al. [14] reports GPU acceleration for NTRU based FHE evaluating Prince and AES block ciphers, with 2.57 times and 7.6 times speedup values, respectively, over an Intel Xeon software implementation. Finally, in [17] and later in [18] Doröz et al. present an architecture for ASIC that implements a full set of FHE primitives including bootstrapping.

In Table 1, we present an overview of previous FHE implementations on various platforms. Clearly, since the platforms vary greatly according to available memory, clock speed, area/price of the hardware a side-by-side comparison is not possible and therefore this information is only meant to give an idea of what is achievable on various platforms. As evident from Table 1, significant gains are possible by developing custom tailored designs for FPGA and ASIC platforms.

Table 1. Overview of specialized FHE Implementations. GH-FHE: Gentry & Halevi's FHE scheme; CMNT-FHE: Coron et al.'s FHE schemes [10,11,22]; NTRU based FHE, e.g. [27,34]

Design	Scheme	Platform	Performance
CPU			
AES [25]	BGV-FHE	2.0 GHz Intel Xeon	5 min / AES block
AES [16]	NTRU-FHE	2.9 GHz Intel Xeon	55 sec / AES block
Full FHE [31]	NTRU-FHE	2.1 GHz Intel Xeon	275 sec / per bootst.
GPU			
NTT mul / reduction [35]	GH-FHE	NVIDIA C250 GPU	0.765 ms
NTT mul [35]	GH-FHE	NVIDIA GTX 690	0.583 ms
AES [14]	NTRU-FHE	NVIDIA GTX 690	7 sec / AES block
FPGA			
NTT transform [37]	GH-FHE	Stratix V FPGA	0.125 ms
NTT modmul / enc. [7]	CMNT-FHE	Xilinx Virtex7 FPGA	13 msec / enc.
ASIC			
NTT modmul [17]	GH-FHE	90 nm TSMC	2.09 sec
Full FHE [18]	GH-FHE	90 nm TSMC	3.1 sec / recrypt

Much of the development so far has focused on the Gentry-Halevi FHE [22], which intrinsically works with very large integers (million bit range). Therefore, a good number of works focused on developing FFT/NTT based large integer multipliers [17,18,35,35]. Currently, the only full-fledged (with bootstrapping) FHE hardware implementation is the one reported by Doröz et al. [18], which also implements the Gentry-Halevi FHE. At this time, there is a lack of hardware implementations of the more recently proposed FHE schemes, i.e. Coron et al.'s FHE schemes [10,11], BGV-style FHE schemes [5], [22] and NTRU based FHE, e.g. [27,34]. We, therefore, focus on providing hardware acceleration support for one particular family of FHE's: NTRU-based FHE schemes, where arithmetic with very large polynomials (both in degree and coefficient size) is crucial for performance.

Our Contribution. In this work, we present an FPGA architecture to accelerate NTRU based FHE schemes. Our architecture may be considered as a proof-of-concept implementation of an external FHE accelerator that will speed up homomorphic evaluations taking place on a CPU. Specifically, the architecture we present manages to evaluate a full large degree, e.g. 2^{15}, polynomial multiplication efficiently by utilizing a number theoretical transform (NTT) based approach. Using this FPGA core we can evaluate polynomial multiplication 28 times faster than on a similar CPU and 17 times faster than a similar GPU implementations. Furthermore, by facilitating efficient exchange using a PCI Express connection, we evaluate the overhead incurred in a sustained homomorphic computations of deep circuits. For instance, also taking into account the cycles lost

in data transfer our hardware can evaluate a full 10 round AES circuit in under 440 msec per block.

2 Background

In this section we briefly outline the primitives of the LTV-based fully homomorphic encryption scheme, and later discuss the arithmetic operations that will be necessary in its hardware realization.

2.1 LTV-Based Fully Homomorphic Encryption

While the arithmetic and homomorphic properties of NTRU have been long known by the research community, a full-fledged fully homomorphic version was proposed only very recently in 2012 by López-Alt, Tromer and Vaikuntanathan (LTV) [27]. The LTV scheme is based on a variant of NTRU introduced earlier by Stehlé and Steinfeld [34]. The LTV scheme uses a new operation called relinearization as well as existing techniques such as modulus switching for noise control. While the LTV scheme can support homomorphic evaluation in a multi-key setting where each participant is issued their own keys, here we focus only on the single user case for brevity.

The primitives of the LTV scheme operate on polynomials in $R_q = \mathbb{Z}_q[x]/\langle x^N + 1 \rangle$, i.e. with degree N, where the coefficients are processed using a prime modulus q. In the scheme an error distribution function χ – a truncated discrete Gaussian distribution – is used to sample random, small B-bounded polynomials. The scheme consists of four primitive functions:

KeyGen. We select decreasing sequence of primes $q_0 > q_1 > \cdots > q_d$ for each level. We sample $g^{(i)}$ and $u^{(i)}$ from χ, compute secret keys $f^{(i)} = 2u^{(i)} + 1$ and public keys $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ for each level. Later we create evaluation keys for each level: $\zeta_\tau^{(i)}(x) = h^{(i)} s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau (f^{(i-1)})^2$, where $\{s_\tau^{(i)}, e_\tau^{(i)}\} \in \chi$ and $\tau = [0, \lfloor \log q_i \rfloor]$.

Encrypt. To encrypt a bit b for the ith level we compute: $c^{(i)} = h^{(i)} s + 2e + b$ where $\{s, e\} \in \chi$.

Decrypt. In order to compute the decryption of a value for specific level i we compute: $m = c^{(i)} f^{(i)} \pmod 2$.

Evaluate. The multiplication and addition of ciphertexts correspond to XOR and AND operations, respectively. The multiplication operation creates a significant noise, which is handled with using relinearization and modulus switching. The relinearization computes $\tilde{c}^{(i)}(x) = \sum_\tau \zeta_\tau^{(i)}(x) \tilde{c}_\tau^{(i-1)}(x)$, where $\tilde{c}_\tau^{(i-1)}(x)$ are 1-bounded polynomials that are equal to $\tilde{c}^{(i-1)}(x) = \sum_\tau 2^\tau \tilde{c}_\tau^{(i-1)}(x)$. In case of modulus switching, we do the computation $\tilde{c}^{(i)}(x) = \lfloor \frac{q_i}{q_{i-1}} \tilde{c}^{(i)}(x) \rceil_2$ to cut the noise level by $\log(q_i/q_{i-1})$ bits. The operation $\lfloor \cdot \rceil_2$ is matching the parity bits.

2.2 Arithmetic Operations

To implement the costly large polynomial multiplication and relinearization operations we follow the strategy of Dai et al. [14]. For instance, in the case of polynomial multiplication we first convert the input polynomials using the Chinese Remainder Theorem (CRT) into a series of polynomials of the same degree, but with much smaller word-sized coefficients. Then, pairwise product of these polynomials is computed efficiently using Number Theoretical Transform (NTT)-based multiplier as explained in subsequent sections. Finally, the resulting polynomial is recovered from the partial products by the application of the inverse CRT (ICRT) operation. For relinearization we follow a similar route; however, we do not compute the ICRT until the final relinearization result is obtained in the residue space.

CRT Conversion. As an initial optimization we convert all operand polynomials with large coefficients into many polynomials with small coefficients by a direct application of the Chinese Remainder Theorem (CRT) on the coefficients of the polynomials: $\mathsf{CRT} : \mathsf{A}_j \longrightarrow \{\mathsf{A}_j \bmod p_0, \mathsf{A}_j \bmod p_1, \cdots, \mathsf{A}_j \bmod p_{l-1}\}$, where p_i's are selected small primes, l is the number of these small primes, and A_j is a coefficient of the original polynomial. Through CRT conversion we obtain a set of polynomials $\{A^{(0)}(x), A^{(1)}(x), \cdots, A^{(l-1)}(x)\}$ where $A^{(i)}(x) \in \mathrm{R}_{p_i} = \mathbb{Z}_{p_i}[x]/\Phi(x)$. These small coefficient polynomials provide us with the advantage of performing arithmetic operations on polynomials in a faster and efficient manner. Any arithmetic operation is performed between the reduced polynomials with the same superscripts, e.g. the product of $A(x) \cdot B(x)$ is going to be $\{A^{(0)}(x) \cdot B^{(0)}(x), A^{(1)}(x) \cdot B^{(1)}(x), \cdots, A^{(l-1)}(x) \cdot B^{(l-1)}\}$. A side benefit of using the CRT is that it allows us to accommodate the change in the coefficient size during the levels of evaluation, thereby yielding more flexibility. When the circuit evaluation level increases, since q_i gets smaller, we can simply decrease the number of primes l. Therefore, both multiplication and relinearization become faster as we proceed through the levels of evaluation. After the operations are completed, a coefficient of the resulting polynomial, $\mathsf{C}(x)$ is computed by the Inverse CRT (ICRT):

$$\mathsf{ICRT}(\mathsf{C}_j) = \sum_{i=0}^{l-1} \left(\frac{q}{p_i}\right) \cdot \left(\left(\frac{q}{p_i}\right)^{-1} \cdot C_j^{(i)} \mod p_i\right) \mod q,$$

where $q = \prod_{i=0}^{i=l-1} p_i$. Note that we will drop the superscript notation used for the reduced polynomials by the CRT for clarity of writing since we will deal with mostly the reduced polynomials henceforth in this paper.

Polynomial Multiplication. The fundamental operation in the LTV scheme, during which the majority of execution time is spent, is the multiplication of two polynomials of very large degrees. More specifically, we need to multiply two polynomials, $A(x)$ and $B(x)$ over the ring of polynomials $\mathbb{Z}_p[x]/(\Phi(x))$, where

p is an odd integer and degree of $\Phi(x)$ is $N = 2^n$. Namely, we have $A(x) = \sum_{i=0}^{N-1} A_i x^i$ and $B(x) = \sum_{i=0}^{N-1} B_i x^i$. The classical multiplication techniques such as the schoolbook algorithm have quadratic complexity in the asymptotic case, namely $\mathcal{O}(N^2)$. In general, the polynomial multiplication requires about N^2 multiplications and additions and subtractions of similar numbers in $\mathbb{Z}_p$. Other classical techniques such as Karatsuba algorithm [26] can be utilized to reduce the complexity of the polynomial multiplication to $\mathcal{O}(N^{\log_2 3})$. Nevertheless, the classical polynomial multiplication techniques do not yield feasible solutions for SWHE implementations, where we would need to perform billions of arithmetic operations in $\mathbb{Z}_p$ since N is a large number. The number theoretic transform (NTT)-based multiplication achieves a quasi-linear complexity for polynomial multiplication, which is especially beneficial for large values of N.

The NTT can essentially be considered as a Discrete Fourier Transform defined over the ring of polynomials $\mathbb{Z}_p[x]/(\Phi(x))$. Simply speaking, the forward NTT takes a polynomial $A(x)$ of degree $N - 1$ over $\mathbb{Z}_p[x]/(\Phi(x))$ and yields another polynomial of the form $\mathcal{A}(x) = \sum_{i=0}^{N-1} \mathcal{A}_i x^i$. The coefficients $\mathcal{A}_i \in \mathbb{Z}_p$ are defined as $\mathcal{A}_i = \sum_{j=0}^{N-1} A_j \cdot w^{ij} \mod p$, where $w \in \mathbb{Z}_p$ is referred as the twiddle factor. For the twiddle factor we have $w^N = \mod p$ and $\forall i < N$ $w^i \neq 1 \mod p$. The inverse transform can be computed in a similar manner $A_i = N^{-1} \cdot \sum_{j=0}^{N-1} \mathcal{A}_j \cdot w^{-ij} \mod p$. Once the NTT is applied to two polynomials, $A(x)$ and $B(x)$, their multiplication can be performed using coefficient-wise multiplication over $\mathcal{A}_i$ and $\mathcal{B}_i$ in $\mathbb{Z}_p$; namely we compute $\mathcal{A}_i \times \mathcal{B}_i \mod p$ for $i = 0, 1, \ldots N - 1$. Then, the inverse NTT (INTT) is used to retrieve the resulting polynomial $C(x) = INTT(NTT(A(x) \odot NTT(B(x)))$, where the symbol $\odot$ denotes the coefficient-wise multiplication of $\mathcal{A}(x)$ and $\mathcal{B}(x)$ in $\mathbb{Z}_p$. Note that the polynomial multiplication yields a polynomial $C(x)$ of degree $2N - 1$. Therefore, before applying the forward NTT, $A(x)$ and $B(x)$ should be padded with N zeros to have exactly $2N$ coefficients. Consequently, for the twiddle factor we should have $w^{2N} = 1 \mod p$ and $\forall i < 2N \; w^i \neq 1 \mod p$.

Cooley–Tukey algorithm [8], described in Algorithm 1, is a very efficient method of computing forward and inverse NTT. The permutation in Step 2 of Algorithm 1 is implemented by simply reversing the indexes of the coefficients of A_i. The new position of the coefficient A_i where $i = (i_n, i_{n-1}, \ldots, i_1, i_0)$ is determined by reversing the bits of i, namely $(i_0, i_1, \ldots, i_{n-1}, i_n)$. For example, the new position of A_{12} when $N = 16$ is 3. The inverse NTT can also be computed with Algorithm 1, using the inverse of the twiddle factor, i.e. $w^{-1} \mod p$. Therefore, we can use the same circuit for both forward and inverse NTT. Note that the NTT-based multiplication technique returns a polynomial of degree $2N - 1$, which should be reduced to a polynomial of degree $N - 1$ by diving it by $\Phi(x)$ and keeping the remainder of the division operation. When the reduction polynomial $\Phi(x)$ is of a special form such as $x^N + 1$, the NTT is known as Fermat Theoretic Transform (FTT) [1] and the polynomial reduction can be performed easily as described in [30] and [2]. However, for efficient SWHE implementation, we need to use reduction polynomials of general form.

Algorithm 1. Iterative Version of Number Theoretic Transformation

---

    input : $A(x) = A_0 + A_1 x + \ldots + A_{N-1} x^{N-1}$, $N = 2^n$, and w
    output: $\mathcal{A}(x) = \mathcal{A}_0 + \mathcal{A}_1 x + \ldots + \mathcal{A}_{N-1} x^{N-1}$

1 for $i = N$ **to** $2N - 1$ **do**
    | $A_i = 0$;
    end
2 $(\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_{2N-1}) \leftarrow \text{Permutation}(A_0, A_1, \ldots, A_{2N-1})$;
3 for $M = 2$ **to** $2N$ **do**
4   **for** $j = 0$ **to** $2N - 1$ **do**
5   **for** $i = 0$ **to** $\dfrac{M}{2} - 1$ **do**
6 $x \leftarrow i \times \dfrac{2N}{M}$;
7 $\mathcal{I} \leftarrow j + i$;
8 $\mathcal{J} \leftarrow j + i + \dfrac{M}{2}$;
9 $\mathcal{A}[\mathcal{I}] \leftarrow \mathcal{A}[\mathcal{I}] + w^{x \bmod 2N} \times \mathcal{A}[\mathcal{J}] \bmod p$;
10 $\mathcal{A}[\mathcal{J}] \leftarrow \mathcal{A}[\mathcal{I}] - w^{x \bmod 2N} \times \mathcal{A}[\mathcal{J}] \bmod p$;
         $i \leftarrow i + 1$;
      end
      $j \leftarrow j + M$;
   end
   $M \leftarrow M \times 2$;
end

Relinearization. Relinearization takes a ciphertext and set of evaluation keys $(EK_{i,j})$ as inputs, where $i \in [0, l-1]$ and $j \in [0, \lceil \log(q)/r \rceil - 1]$, l is the number of small prime numbers and r is the level index. Algorithm 2 describes relinearization as implemented in this work. We pre-compute the CRT and NTT of the evaluations keys (since they are fixed) and in the computations we perform the multiplications and additions in the NTT domain. The result is evaluated by taking l INTT and one ICRT at the end. An r-bit windowed relinearization involves $\lceil \log(q)/r \rceil$ polynomial multiplications and additions, which are performed again in the NTT domain. Since operand coefficients are kept in residue form, before relinearization we need to compute the inverse CRT of $\tilde{c}_\tau$.

3 Architecture Overview

3.1 Software/Hardware Interface

The performance of the NTRU based FHE scheme heavily depends on the speed of the large degree polynomial multiplication and relinearization operations. Since the relinearization operation is reduced to the computation of many polynomial multiplications, a fast large degree polynomial multiplication is the key to

---

Algorithm 2. Relinearization with r bit windows

> **input** : Polynomial c with $(n, \log(q))$
> **output**: Polynomial d with $(2n, \log(nq\log(q)))$

1 $\{\tilde{C}_\tau\} = \mathsf{NTT}(\{\tilde{c}_\tau\})$;
  for $i = 0$ **to** $l - 1$ **do**
2 | load $EK_{i,0}, EK_{i,1}, \cdots, EK_{i,\lceil \log(q)/r \rceil - 1}$;
3 | $\{D_i\} = \{\sum_{\tau=0}^{\lceil \log(q)/r \rceil - 1} \tilde{C}_\tau \cdot EK_{i,\tau}\}$;
  end
4 $\{d_i\} = \mathsf{INTT}(\{D_i\})$;
5 $d = \mathsf{ICRT}(\{d_i\})$;

---

achieve a high performance in the NTRU-FHE scheme. The complexity of NTT-based polynomial multiplication operation is quasi-linear $O(N \log N \log \log N)$, and the security levels require the degree of the polynomials to be $N = 2^{15}$ for applications such as LTV-AES in [16]. Having a large degree N increases the computation requirements significantly, therefore a standalone software implementation on a general-purpose computing platform fails to provide a sufficient performance level for polynomial multiplications. The NTT-based polynomial multiplication algorithm is highly suitable for parallelization, which can lead to performance boost when implemented in hardware. On the other hand, the overall scheme is a complex design demanding prohibitively huge memory requirements (e.g., in LTV-AES [16] key requirements exceed 64-GB of memory). Therefore, a standalone architecture for SWHE fully implemented in hardware is not feasible to meet the requirements of the scheme.

In order to cope with the performance issues we designed the core NTT-based polynomial multiplication in hardware, where the polynomials have relatively small coefficients (i.e., 32-bit integers) to use it in more complicated polynomial multiplications and relinearization evaluations. The designed hardware is implemented in an FPGA device, which is connected to a PC with a high speed interface, e.g. PCI Express (PCIe). The PC handles simple and non-costly computations such as memory transactions, polynomial additions and etc. In case of a large polynomial multiplication or NTT conversion (in case of relinearization), the PC using the CRT technique, computes an array of polynomials whose coefficients are 32-bit integers from the input polynomials of much larger coefficients. The array of polynomials with small coefficients are sent in chucks to the FPGA via the high-speed PCIe bus. The FPGA computes the desired operation: polynomial multiplications or only NTT conversion. Later, the PC receives the resulting polynomials from the FPGA and if necessary, i.e. before modulus switching or relinearization, evaluates the inverse-CRT to compute the result.

3.2   PCIe Interface

The PCIe is a serial bus standard used for high speed communication between devices which in our case are PC and the FPGA board. As the target FPGA

board, we use Virtex-7 FPGA VC709 Connectivity Kit and can operate at 8 GT/s, per lane, per direction with each board having 8 lanes. The system is capable of sending the data packets in bursts. This allows us to achieve real time data transaction rate close to the given theoretical transaction rate as the packet sizes become larger.

3.3   Arithmetic Core Units

In order to achieve multiplication of two polynomials of degree 2^{15}, we first designed hardware implementations for basic arithmetic building blocks to perform operations on the polynomial coefficients such as modular addition, modular subtraction and modular multiplication. We base our design on an architecture to perform modular arithmetic operations for 32-bit numbers.

32-Bit Modular Addition/Subtraction. The modular addition circuit, takes one clock cycle to perform one modular addition operation where operands A, B and the modulus p are all 32-bit integers and $A, B < p$. Since the largest value of $A+B$ can be at most $2p-2$, at most one final subtraction of the modulus p from $A + B$ will be sufficient to achieve full modular reduction after addition operation. Similarly the subtraction unit is optimized to take one clock cycle to finish one modular subtraction operation on a target device.

Integer Multiplication. The target FPGA device features many DSP units that are capable of performing very fast multiply and accumulate operations. Since these DSP units are highly optimized, it is particularly beneficial to utilize them in our core modular multiplier design. A DSP unit takes three inputs A, B and C, which are 18 bits, 25 bits and 48 bits, respectively. A and B are multiplicand inputs, and C is the accumulate input. The output is a 48–bit integer, which can be defined as $D = A \times B + C$. Therefore, we can accumulate the results of many 18×25–bit multiplications without overflow. Since our operands are 32 bits in length, first we need to perform a full multiplication operation of 32–bit numbers. The operand lengths of the DSP units dictate that we need to perform four 16×16–bit multiplication operations to achieve a 32–bit multiplication operation. Utilizing four separate DSP slices, we could perform a 32–bit multiplication with 1 clock cycle throughput. However, this brings additional complexity to the hardware and because of the overall structure of the polynomial multiplication algorithm, 1–cycle throughput is not crucial for our design. Therefore, we decided to utilize a single DSP unit and perform the four required 16×16–bit multiplication operations to achieve a 32–bit multiplication operation on the same DSP unit. This results in a 4–cycle throughput. In our design, we use Barrett's algorithm [3] for modular reduction, which requires 33×33–bit multiplication operations. Therefore, we use DSP slices to perform 17×17–bit integer multiplications at a time, instead of 16×16–bit multiplications, where both operations have exactly the same complexity. To minimize critical path delays, we utilize the optional registers for the multiplicand inputs

and the accumulate output ports of the DSP unit. These registers increase the latency of a single 33×33-bit multiplication to 6 clock cycles. On the other hand, the throughput is still four clock cycles, which allows the multiplier unit to start a new multiplication every four clock cycles.

32-Bit Modular Multiplication. We use Barrett's modular reduction algorithm [3] to perform modular multiplication operations. The Montgomery reduction algorithm [28], which is a plausible alternative to the Barrett reduction, can also be used for modular multiplication of 32-bit integers. Indeed, integer multiplications during the Montgomery reduction are slightly less complicated and can result in area efficiency. On the other hand, using the Montgomery reduction would not change the throughput, which is four clock cycles for a single modular multiplication in our design. Furthermore, the Montgomery arithmetic requires transformations to and from the residue domain, which can lead to complications in the design. Therefore, we prefer using the Barrett's algorithm in our implementation to alleviate the mentioned complications in the design.

4 $2^{15} \times 2^{15}$ Polynomial Multiplier

We implemented a $2^{15} \times 2^{15}$ polynomial multiplier, with 32–bit coefficients. Throughout the paper, we will use the term $32K$ to denote the $2^{15} \times 2^{15}$ polynomial multiplier. We do not utilize any special modulus, to achieve a generic and robust polynomial multiplier as we use Barrett's reduction algorithm for coefficient arithmetic. Instead of the classical schoolbook method for polynomial multiplication, we utilized the NTT–based multiplication algorithm, as explained in Sect. 2.2 and described in Algorithm 3. It should be noted that step 5 of Algorithm 3 is implemented by coefficient–wise 32–bit modular multiplications.

Algorithm 3. NTT–based $32K$ polynomial multiplication

---

    input : $A(x) = A_0 + A_1 x + \cdots + A_{32767} x^{32767}$,
           $B(x) = B_0 + B_1 x + \cdots + B_{32767} x^{32767}$, p
    output: $C(x) = A(x) \times B(x)$

1 $NTT_A(x) \leftarrow$ NTT of polynomial $A(x)$;
2 $NTT_B(x) \leftarrow$ NTT of polynomial $B(x)$;
3 $NTT_C(x) \leftarrow$ Inner products of polynomials $NTT_A(x)$ and $NTT_B(x)$;
4 $T(x) \leftarrow$ Inverse NTT of polynomial $NTT_C(x)$;
5 $C(x) \leftarrow T(x) \times (32768^{-1} \bmod p)$;

4.1 NTT Operation

NTT Algorithm. We apply the NTT operation on a polynomial $A(x)$ of degree $32K - 1$ over $\mathbb{Z}_p[x]/(\Phi(x))$. Since the result of the NTT–based multiplication will

be of degree $64\,K$, we need to zero–pad the polynomial $A(x)$ to make it also a polynomial of degree $64\,K$ as follows $A(x) = \sum_{j=0}^{32K-1} A_j \cdot x^j + \sum_{j=32K}^{64K-1} 0 \cdot x^j$. When we apply the NTT transform on $A(x)$, the resulting polynomial is $\mathcal{A}(x) = \sum_{i=0}^{64K-1} \mathcal{A}_i \cdot x^i$, where the coefficients $\mathcal{A}_i \in \mathbb{Z}_p$ are defined as $\mathcal{A}_i = \sum_{j=0}^{64K-1} A_j \cdot w^{ij} \bmod p$, and $w \in \mathbb{Z}_p$ is referred as the twiddle factor. Since the size of the NTT operation is actually $64\,K$, we need to choose a twiddle factor w which satisfies the property $w^{64K} \equiv 1 \bmod p$ and $\forall i < 64\,K\ w^i \neq 1 \bmod p$. As we are utilizing generic modular multipliers, no special form of w is required to achieve more efficient multiplications.

To achieve fast NTT operations, we utilize the Cooley–Tukey approach, as explained in Sect. 2.2. Cooley–Tukey approach works by splitting up the NTT–transform into two parts, performing the NTT operation on the smaller parts, and performing a final reconstruction to combine the results of the two half–size NTT transform results into a full–sized NTT operation. For the coefficients of NTT, we have $\mathcal{A}_i = \sum_{j=0}^{32K-1} A_{2j} \cdot w^{i(2j)} \bmod p + w^i \sum_{j=0}^{32K-1} A_{2j+1} \cdot w^{i(2j)} \bmod p$ and denote this expression as $\mathcal{A}_i = E_i + w^i O_i$, where E_i and O_i represent the ith coefficients of the $32\,K$ NTT operation on the even and odd coefficients of the polynomial $A(x)$, respectively. It is important to note that if the twiddle factor of the $64\,K$ NTT operation is w, the twiddle factor of the smaller $32\,K$ operation will be w^2. Because of the periodicity of the NTT operation, we know that $E_{i+32K} = E_i$ and $O_{i+32K} = O_i$. Therefore, we have $\mathcal{A}_i = E_i + w^i O_i$ for $0 \leq i < 32K$ and $\mathcal{A}_i = E_{i-32K} + w^i O_{i-32K}$ for $32K \leq i < 64K$. For the twiddle factor, it holds that $w^{i+32K} = w^i \cdot w^{32K} = -w^i$. Consequently, we can achieve a full $64K$ NTT operation with two small $32\,K$ NTT operations utilizing the following reconstruction operation

$$\mathcal{A}_i = E_i + w^i O_i,$$
$$\mathcal{A}_{i+32K} = E_i - w^i O_i. \tag{1}$$

The reconstruction operation is performed iteratively over very large number of coefficients. An 8×8 NTT circuit is illustrated in Fig. 1. Note that, in a full $64\,K$ NTT circuit, the twiddle factor w^{16484} is used in 8×8 NTT circuits.

Coefficient Multiplication and Accumulation. Since our target FPGA has multiple number of DSP units and Block RAMs, we are able to parallelize the multiplication and accumulation operations at each level of the iterative NTT operation. We can utilize $3 \cdot K$ DSP units to achieve K modular multiplications in parallel, with a 4–cycle throughput, where K is a design parameter that depends on the number of available DSP units in the target architecture. In our design, K is chosen as a power of 2.

To be able to feed the DSP units with correct polynomial coefficients during multiplication cycles, we utilize K separate Block RAMs (BRAM)) to store the polynomial coefficients. The algorithm used to access the polynomial coefficients in parallel is described in Algorithm 4. The algorithm takes the BRAM content (i.e., the coefficients of $A(x)$), the degree $N = 2^n$, the current level m, and the
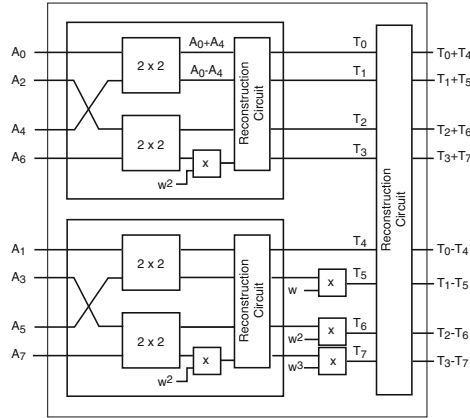
Fig. 1. Construction of the 8×8 NTT circuit iteratively.

number of modular multipliers $K = 2^\kappa$ as input, and generates the indexes in a parallel manner. Every four clock cycles, we try to feed modular multipliers the number of coefficients which is as close to K as possible. Ideally, it is desirable to perform exactly K modular multiplications in parallel, which is not possible due to the access pattern to the powers of w. Algorithm 4, on the other hand, achieves a good utilization of modular multiplication units.

For level m, we use the $2^m \times 2^m$ NTT circuit. The coefficients are arranged in $2^m \times 2^m$ blocks. For example when $K = 256$, for the first level of the NTT operation, where $m = 2$, we need to multiply every $4th$ coefficient of the polynomial with $w_2 = w^{16384}$. Since the coefficients are perfectly dispersed, we can read 256 coefficients to feed the 256 multipliers in four clock cycles. This is perfect as the throughput of our multipliers are also four cycles. When the multiplication operations are complete, with an offset of 19 cycles (four clock cycles are for the warm up of the pipeline whereas 15 clock cycles are the tail cycles necessary in a pipelined design to finish the last operation), the results are written back to the same address of the RAM block as the one the coefficients are read from. Since we are utilizing dual port RAM structures, and we guarantee different read and write addresses on each block, collisions never occur with this organization.

We provide formulae for the number of multiplications in each level and an estimate of the number of clock cycles needed for their computation in our architecture. Suppose $N = 2^n$ and $K = 2^\kappa$ $(n > \kappa)$ are the number of coefficients in our polynomial and the number of modulo multipliers in our target device, respectively. The coefficients are stored in block RAMS (BRAMs), with a word size of 32 bits and an address length of 10 bits (1024 coefficients per BRAM). For ideal case, the number of modular multipliers should be 4 times the number of BRAMS required to store a single polynomial. The formula for the number of multiplications for the level $m > 1$ can be given as $\mathcal{M} = 2^{n+1-m} \cdot (2^{m-1} - 1)$. Also, using $K = 2^\kappa$ multipliers, the number of clock cycles to compute all multiplications in a given level $1 < m \leq n + 1$ can be formulated as

---

Algorithm 4. Parallel access to polynomial coefficients

input : $A(x) = A_0 + A_1 x + \ldots + A_{2N-1} x^{2N-1}$, n, m, and $\kappa < n$
output: $B_i[j]$

1   $mCnt \leftarrow 2^{m-1} - 1$;   /* number of multiplications in a block */
2   $bSize \leftarrow 2^m$;   /* size of a block */
3   $BRAMCnt \leftarrow 2^{\kappa-2}$;   /* number of BRAMs */
4   **if** $bSize \leq 2^{\kappa-2}$ **then**
> **for** $t = 0$ **to** 1024 **do**
>> **for** $i = 0$ **to** $BRAMCnt$ **do in parallel**
>>> **for** $j = i + bSize - mCnt$ **to** $i + bSize$ **do**
>>>> **for** $k = 0$ **to** 3 **do**
5   Access $BRAM_j[t + 2k]$;
6   Access $BRAM_j[t + 2k + 1]$;
>>>> $k \leftarrow k + 1$;
>>>> **end**
>>> $j \leftarrow j + 1$;
>>> **end**
>> $i \leftarrow i + bSize$;
>> **end**
> $t \leftarrow t + 8$;
> **end**

  end
7   **else**
> **for** $i = 0$ **to** $BRAMCnt$ **do in parallel**
>> **for** $j = 0$ **to** 1024 **do**
>>> **for** $k = 2^{m-\kappa+1}$ **to** $2^{m-\kappa+2}$ **do**
8   Access $BRAM_i[k + j]$;
>>> $k \leftarrow k + 1$;
>>> **end**
>> $j \leftarrow j + 2^{m-\kappa+2}$;
>> **end**
> $i \leftarrow i + 1$;
> **end**

  end

$$
CC_m = \begin{cases} 4 + 4 \cdot \left\lfloor \dfrac{\mathcal{M}}{\alpha \cdot \lfloor K/\alpha \rfloor} \right\rfloor + 15 & \kappa \geq m \\[4mm] 4 + 4 \cdot \left(\dfrac{\beta}{K} + 1 \right) \cdot 2^{n+1-m} + 15 & \kappa < m, \end{cases}
$$

where $\alpha = 2^{\kappa-m} \cdot (2^{m-1} - 1)$ and $\beta = 2^{m-1} - 2^\kappa$. In the formula, the first (4) and the last terms (15) account for the warm up and the tail cycles.

As mentioned before, the modulo multipliers are not always fully utilized during the NTT computation. For example when $K = 2^8$ and $N = 2^{15}$, for $m = 2$, we have to read every 4th coefficient from the BRAMs. Because the

coefficients are perfectly dispersed throughout the 64 BRAMS, we can only read $16 \cdot 2 = 32$ coefficients every clock cycle, which yields a number of 128 concurrent multiplications every four clock cycles. Consequently, we can finish all the modular multiplications in the first level in $4 + 128 \cdot 4 + 15 = 531$ clock cycles. Since we can use half the modular multipliers, we achieve half utilization in the first level. However, when $m = 3$, we have to read every 6th, 7th and 8th out of every 8 coefficients. We can read $24 \cdot 2 = 48$ coefficients every clock cycle from the BRAMs. This means we can only utilize 192 out of 25 modular multipliers since the irregularity of the access to the polynomial coefficients. This, naturally, results in a slightly low utilization. However, since we can read 2 coefficients from each BRAM every clock cycle, we are at almost perfect utilization, resulting in $4 + 128 \cdot 4 + 15 = 531$ clock cycles for this and the rest of the stages.

Since the operands of the both operations are accessed in a regular manner, the number of clock cycles spent on modular additions and subtractions are calculated as $\frac{2^{n+1} \cdot (n+1)}{2^{\tau}}$, when there are 2^{τ} modular adders and 2^{τ} subtractors.

Reconstruction. Once we are done with the multiplications, we utilize 64 modular adders and 64 modular subtractors to realize the addition and subtraction operations as shown in Eq. 1.

4.2   Inner Multiplication

Inner multiplication of two 64 K polynomials is trivial for our hardware design. We can load 256 coefficients from each polynomial every 4 cycles and feed the multipliers, without increasing the 4–cycle throughput. For a 64 K polynomial inner multiplication we spend $1024 + 15 = 1039$ clock cycles.

4.3   Inverse NTT

The Inverse NTT operation is identical to the NTT operation, except that instead of the twiddle factor w, we use the twiddle factor $w_i = w^{-1} \bmod p$. The precomputed twiddle factors of the inverse NTT are stored in the same block RAMs as the forward NTT twiddle factors, with an address offset. Therefore, the same control block can be utilized with a simple address change for the w coefficients for the inverse NTT operation.

4.4   Final Scaling

Final scaling is similar to the inner multiplication phase. We load each coefficient of the resulting polynomial, and multiply them with the precomputed scaling factor. Similar to the inner multiplication phase, we can load 256 coefficients from the resulting polynomial in 4 cycles cycle and feed the multipliers, without increasing the 4–cycle throughput. For a 64 K polynomial final scaling operation, we spend 1039 clock cycles.

Table 2. Virtex-7 XC7VX690T device utilization of the multiplier

	Total	Used	Used (%)
Slice LUTs	433,200	219,192	50.59
Slice registers	866,400	90,789	10.47
RAMB36E1	1470	193	13.12
DSP48E1	3600	768	21.33

5   Implementation Results

We developed the architecture described in the previous section into Verilog modules and synthesized it using Xilinx Vivado tool for the Virtex 7 XC7VX690T FPGA family. The synthesis results are summarized in Table 2. We synthesized the design and achieved an operating frequency of 250 MHz for multiplication of polynomials of degree $n = 32,768$ with a small word size of $\log p = 32$ bit. The FPGA multiplier is used to process each component of the CRT representation of our large coefficient ciphertexts with $\log q = 1271$ bits. In fact we keep all ciphertexts in CRT representation and only compute the polynomial form when absolutely necessary, e.g. for parity correction during modulus switching and before relinearization. We assume any data sent from the PC through the PCIe interface to the FPGA is stored in onboard BRAM units.

CRT Computation Cost. To facilitate efficient computation of multiplication and relinearization operations we use a series of equal sized prime numbers to construct a CRT conversion. In fact, we chose the primes p_is such that $q = \prod_{i=0}^{l} p_i$. During the levels of homomorphic evaluation, this representation allows us to easily switch modulus by simply dropping the last p_i following by a parity correction. Also, since we have an RNS representation on the coefficients we no longer need to reduce by q. This also eliminates the need to consider any overflow conditions. Thus, $l = \log(q)/31 = 41$. We efficiently compute the CRT residue in software on the CPU for each polynomial coefficient as follows:

- Precompute and store $t_k = 2^{64 \cdot k} \pmod{p_i}$ where $k \in [0, \lceil \log(q/64) - 1 \rceil]$.
- Given a coefficient of c, we divide it into 64-bit blocks as $c = \{\ldots, w_k, \ldots, w_0\}$.
- We compute the CRT result by evaluating $\sum t_k \cdot w_k \pmod{p_i}$ iteratively.

The CRT computation cost for 41 primes p_i per ciphertext polynomial is in the order of 89 msec on the CPU. The CRT inverse is similarly computed (with the addition of a word carry) before each modulus switching operation at essentially the same cost. Note that this high latency is a significant contributor of our choice to keep the operands in the CRT representation.

Communication Cost. The PCIe bus is only used for transactions of input/output values, NTT constants and transport of evaluation keys to the FPGA board. With 8 lanes each capable of supporting 8 Gbit/sec transport speed the PCIe is capable to transmit a 5 MB ciphertext in about 0.65 msec.

Note that the NTT parameters used during multiplication also need to be transported since we do not have enough room in the BRAM components to keep them permanently. We have two cases to consider:

– Multiplication: We transport two polynomials of 5 MB each along with the NTT parameters of 5 MB and receive a polynomial of 10 MB, which costs about 3.25 msec per multiplication.
– Relinearization: We need to transport the ciphertext we want to relinearize, the NTT parameters and a set of $\log(q)/16 \approx 80$ evaluation keys (ciphertexts), where a window size of 16-bit is used, resulting in a 52 msec delay.

Multiplication Cost. We compute the product of two polynomials with coefficients of size $\log(p) = 32$ bits using 256 modular multipliers in 12720 cycles, which translates to 152 μsec. This figure is comprised of two NTT and one inverse NTT operations and one inner product computation. The addition of I/O transactions will increase the timing by 79 μsec. Using the multiplication time, the latency of large polynomial multiplication may be broken down as follows:

– Cost of small coefficient polynomial multiplications 41.152 μsec $= 6.25$ msec.
– The PCIe transaction of the two input polynomials, the NTT coefficients and the double sized output polynomial is 3.25 msec.

Thus, the total latency for large polynomial multiplication in the CRT representation is computed in 9.51 msec.

Polynomial Modular Reduction. Since all operations are computed in a polynomial ring with a characteristic polynomial as modulus without any special structure, we use Barrett's reduction technique to perform the reductions. Note that precomputing the constant polynomial $x^{2N}/\Phi(x)$ (truncated division) in the CRT representation we do not need to compute any CRT or inverse CRT operations during modular reduction. Thus we can compute the reduction using two product operations in about 19 msec.

Modulus Switching. We realize the modulus switching operation by dropping the last CRT coefficient followed by parity correction. To compute the parity of the cut polynomial we need to compute an inverse CRT operation. The following parity matching and correction step takes negligible time. Note that the parities are single bit and therefore we do not need to compute another CRT operation. Therefore, modulus switching can be realized using one inverse CRT computation in 89 msec.

Relinearization Cost. To realinearize a ciphertext polynomial

– We need to convert the ciphertext polynomial coefficients into integer representation using one inverse CRT operation, which takes 89 msec.
– The evaluation keys are kept in NTT representation, therefore we only need to compute two NTT operations for one operand and the result. For $l = 41$ primes and $\log(q)/16 \approx 80$ products the NTT operations take 331 msec.

Table 3. Primitive operation timings including I/O transactions.

	Timings (msec)		Timings (msec)
CRT	**89**	**Modulus switch**	**89**
Multiplication	**9.51**	**Relinearization**	**526**
NTT conversions	6.25	CRT conversions	89
PCIe cost	3.26	NTT conversions	331
Modular reduction	**19**	PCIe cost	52

– We need to transport the ciphertext, the NTT parameters and 80 evaluation
  keys (ciphertexts) resulting in a 52 msec delay.
– The summation of the partial products takes negligible time compared to the
  multiplications and the PCIe communication cost.

Then, the total relinearization operation takes 526 msec. With the current implementation, the actual NTT computations still dominate over the other sources of latency such as PCIe communication latency and the CRT computations. However, if the design is further optimized, e.g. by increasing the number of processing units on the FPGA or by building custom support for CRT operations on the FPGA, then the PCIe communication overhead will become more dominant. The timing results are summarized in Table 3.

6   Comparison

To understand the improvement gained by adding custom hardware support in leveled homomorphic evaluation of a deep circuit, we *estimate* the homomorphic evaluation time for the AES circuit and compare it with a similar software implementation by Doröz et al. [16].

Homomorphic AES evaluation. Using the NTRU primitives we implemented the depth 40 AES circuit following the approach in [16]. The tower field based AES SBox evaluation is completed using 18 Relinearization operations and thus 2,880 Relinearizations are needed for the full AES. The AES circuit evaluation requires 5760 modular multiplications. During the evaluation we also compute 6080 modulus switching operations. This results in a total AES evaluation time of 15 min. Note that during the homomorphic evaluation with each new level the operands shrink linearly with the levels thereby increasing the speed. We conservatively account for this effect by dividing the evaluation time by half. With 2048 message slots, the amortized AES evaluation time becomes 439 msec.

   We have also modified Doröz et al.'s homomorphic AES evaluation code to compute relinearization with 16-bits windows (originally single bit). This simple optimization dramatically reduces the evaluation key size and speeds up the relinearization. The results are given in Table 4. We also included the GPU optimized implementation by Dai et al. [14] on an NVIDIA GeForce GTX 680. With

Table 4. Comparison of multiplication, relinearization times and AES estimate

	Mul (msec)	Speedup	Relin (sec)	Speedup	AES (sec)	Speedup
CPU [16]	970	1×	103	1×	55	1×
GPU [14]	340	2.8×	8.97	11.5×	7.3	7.5×
CPU (16-bit)	970	1×	6.5	16×	12.6	4.4×
FPGA (ours)	**9.5**	**102×**	**0.53**	**195×**	**0.44**	**125×**

custom hardware assistance we obtain a significant speedups in both multiplication and relinearization operations. The estimated AES block evaluation is also improved significantly where some of the efficiency is lost to the PC to FPGA communication and CRT computation latencies.

7   Conclusions

We presented a custom hardware design to address the performance bottleneck in leveled SWHE evaluations. Given the large parameters used in such systems we design a large NTT based multiplier capable of multiplying very large degree polynomials. With the implementation of a CRT representation on the coefficients we managed to build a custom core capable of supporting polynomial multiplications with very large degree and very large coefficient polynomials. The design is highly optimized using numerous techniques to speedup the NTT computations, and to reduce the burden on the PC/FPGA interface. The resulting architecture dramatically improves the modular multiplication and relinearization speeds of the LTV SWHE scheme over comparable software implementations. To demonstrate the effectiveness of the accelerator, we estimated the AES evaluation performance and determined a speedup of about 28 times.

References

1. Agarwal, R.C., Burrus, C.S.: Fast convolution using fermat number transforms with applications to digital filtering. IEEE Trans. Acoust. Speech Signal Process. **22**(2), 87–97 (1974)
2. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient fpga implementations of lattice-based cryptography. In: HOST, pp. 81–86. IEEE (2013)
3. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987)
4. Brakerski, Z., Gentry, C., Halevi, S.: Packed ciphertexts in LWE-based homomorphic encryption. IACR Cryptology ePrint Archive 2012/565 (2012)

5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. Electron. Colloquium Comput. Complex. (ECCC) **18**, 111 (2011)
6. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: FOCS, pp. 97–106 (2011)
7. Cao, X., Moore, C., O'Neill, M., O'Sullivan, E., Hanley, N.: Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction. IACR Cryptology ePrint Archive 2013/616 (2013)
8. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. comput. **19**(90), 297–301 (1965)
9. Coron, J.S., Lepoint, T., Tibouchi, M.: Batch fully homomorphic encryption over the integers. IACR Cryptology ePrint Archive 2013/36 (2013)
10. Coron, J.-S., Mandal, A., Naccache, D., Tibouchi, M.: Fully homomorphic encryption over the integers with shorter public keys. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 487–504. Springer, Heidelberg (2011)
11. Coron, J.-S., Naccache, D., Tibouchi, M.: Public key compression and modulus switching for fully homomorphic encryption over the integers. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 446–464. Springer, Heidelberg (2012)
12. Cousins, D., Rohloff, K., Schantz, R., Peikert, C.: SIPHER: Scalable implementation of primitives for homomorphic encrytion. Internet Source, September 2011
13. Cousins, D., Rohloff, K., Peikert, C., Schantz, R.E.: An update on SIPHER (scalable implementation of primitives for homomorphic encRyption) - FPGA implementation using simulink. In: HPEC, pp. 1–5 (2012)
14. Dai, W., Doröz, Y., Sunar, B.: Accelerating NTRU based homomorphic encryption using GPUs. In: HPEC (2014)
15. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010)
16. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic AES evaluation using NTRU. IACR ePrint Archive (2014), https://eprint.iacr.org/2014/039.pdf
17. Doröz, Y., Öztürk, E., Sunar, B.: Evaluating the hardware performance of a million-bit multiplier. In: 2013 16th Euromicro Conference on Digital System Design (DSD) (2013)
18. Doröz, Y., Öztürk, E., Sunar, B.: Accelerating fully homomorphic encryption in hardware. IEEE Trans. Comput. **64**(6), 1509–1521 (2015)
19. Gentry, C.: A Fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)
20. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC, pp. 169–178 (2009)
21. Gentry, C., Halevi, S.: Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. IACR Cryptology ePrint Archive 2011/279 (2011)
22. Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011)
23. Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. IACR Cryptology ePrint Archive 2011/680 2011 (2011)
24. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. IACR Cryptology ePrint Archive Report 2011/566 (2011). http://eprint.iacr.org/

25. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. IACR Cryptology ePrint Archive 2012 (2012)
26. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Doklady Akad. Nauk SSSR **145**(293–294), 85 (1962)
27. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC (2012)
28. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
29. 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.A., Brenner, M., Smith, M. (eds.) FC 2013. LNCS, vol. 7862, pp. 226–237. Springer, Heidelberg (2013)
30. Pöppelmann, T., Güneysu, T.: Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 139–158. Springer, Heidelberg (2012)
31. Rohloff, K., Cousins, D.B.: A scalable implementation of fully homomorphic encryption built on NTRU. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014 Workshops. LNCS, vol. 8438, pp. 221–234. Springer, Heidelberg (2014)
32. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 420–443. Springer, Heidelberg (2010)
33. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. IACR Cryptology ePrint Archive 2011/133 (2011)
34. Stehlé, D., Steinfeld, R.: Making NTRU as secure as worst-case problems over ideal lattices. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. LNCS, vol. 6632, pp. 27–47. Springer, Heidelberg (2011)
35. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. In: HPEC, pp. 1–5 (2012)
36. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Exploring the feasibility of fully homomorphic encryption. IEEE Trans. Comput. **99**, 1 (2013). (PrePrints)
37. Wang, W., Huang, X.: FPGA implementation of a large-number multiplier for fully homomorphic encryption. In: ISCAS, pp. 2589–2592 (2013)