Tile-Based Modular Architecture for Accelerating Homomorphic Function Evaluation on FPGA

Mustafa Khairallah[1] and Maged Ghoneima[1]

Ain Shams University, Cairo, Egypt
khairallah@ieee.org,m_ghoneima@ieee.org

Abstract. Fully Homomorphic Encryption is a powerful cryptographic tool that enables performing arbitrary meaningful computations over encrypted data. Despite its evolution over the past 7 years, FHE schemes are still not suitable for practical use due to performance inefficiencies, where a simple operation can be performed in several seconds. In this paper, a new architecture for accelerating homomorphic function evaluation on FPGA is proposed. While ideas such as the small/large-CRT representation are reused from previous architectures, a modified version of the cached NTT algorithm is presented in this paper, allowing it to be efficiently computed in a multi-core environment. In order to compute an N-point NTT, the architecture consists of $\sqrt{N}$ cores, each capable of computing a $\sqrt{N}$-point NTT, with a special purpose Network-on-Chip (NoC) for coefficient reordering. The proposed NoC enables reordering coefficients in time $\mathcal{O}(\sqrt{N})$, leading to an overall parallel NTT algorithm of time complexity $\mathcal{O}(\sqrt{N}\log\sqrt{N})$. The architecture has been implemented on Xilinx Virtex 7 XC7V1140T FPGA. The design consumes 22% of the registers, 95% of the LUTs, 91% of the DSPs and 85% of the Block RAMs. The implementation performs 32-bit 2^{16}-point NTT algorithm in $23.8\mu s$, achieving speed-up of 14x over the state of the art architecture in this crucial operation. The architecture has been evaluated by computing a block of each of the AES and SIMON-64/128 on the LTV and YASHE schemes. The proposed architecture can evaluate the AES circuit using the LTV scheme in 4 minutes, processing 2048 blocks in parallel, which leads to an amortized performance of 117 ms/block, which is the fastest performance reported to the best of our knowledge.

Keywords: FHE, Homomorphic, FPGA, Virtex, NTT, CRT

1 Introduction

In 1978, the concept of Fully Homomorphic Encryption (FHE) was introduced by Rivest, Adleman and Dertouzos [1]. It is a form of encryption that allows meaningful computations to be performed on the ciphertext, so that when the results are decrypted, they match the results of performing the same computations on the plaintext. While cloud computing has become a wide-spread technology, a lot of applications that require FHE have emerged. Despite the need

for FHE in many applications, FHE remained an open problem until 2009, when Gentry [2] proposed the first FHE scheme based on ideal lattices. However, his solution was not practical and had very low performance. Since then, a lot of efforts have been directed towards constructing practical FHE schemes. Despite its drawbacks, Gentry's scheme represented the blueprint for all subsequent schemes [3][4][5][6][7][8][9][10][11][12]. The recently proposed FHE schemes are either based on the Learning with Errors (LWE)[6][8][9][10], Ring-LWE[11][5], N-th degree Truncated Polynomial Ring (NTRU) [3][7] or the Approximate Greatest Common Divisor (AGCD)[12] problems.

Due to the enormous cost of the bootstrapping procedure used in FHE schemes, it is preferred to use Somewhat Homomorphic Encryption (SHE) or leveled FHE, where the maximum number of levels of operations that can be performed on encrypted data has to be decided in advance. For many applications the definition of SHE is sufficient. Ring-based SHE schemes, such as BGV[11], FV[5], YASHE[3] and LTV[7], achieve the best performance results, due to the possibility to use the Single-Instruction-Multiple-Data (SIMD) technique[13], where you can process the same operation on multiple plaintexts using only one operation on a ciphertext that packs these plaintexts. Two comprehensive studies have been published comparing the different ring-based SHE schemes. In [14], a comparison between the YASHE scheme and the FV scheme has been presented, showing that the homomorphic evaluation of the low weight block cipher SIMON-64/128 [15] requires 12418 s using the FV scheme and 4193 s using YASHE scheme both on a 4-core Intel Core i7 CPU at 3.4 GHz. Although these results show that the YASHE scheme has much better performance, they also show that the software evaluation on CPUs is not sufficient for practical applications. A more recent study[16] has expanded the comparison to all the 4 ring-based schemes: BGV, LTV, YASHE and FV. It has shown the YASHE scheme has the best performance results for small plaintext moduli, while BGV is more efficient for large plaintext moduli.

In 2015, three implementations, [17], [18] and [19], have been published targeting the acceleration of the whole operation of homomorphic function evaluation using FPGA. The design proposed in [19] uses an efficient double-buffered memory access scheme and a polynomial multiplier based on the Number Theoretic Transform (NTT). For the parameter set ($n = 16384, \lceil log_2(q) \rceil = 512$) of YASHE scheme capable of evaluating 9 levels of multiplications, homomorphic addition can be performed in 0.94 ms and homomorphic multiplication can be performed in 48.67 ms. However, the authors failed to implement the design for larger parameter sets. Despite its drawbacks, it is the first design to use cached-NTT to enhance external memory access. In [18], a hardware/software implementation is designed, including a large NTT based multiplier capable of multiplying very large degree polynomials. With the implementation of a CRT representation on the coefficients, a custom core capable of supporting polynomial multiplications with very large degree and very large coefficient polynomials is implemented. 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 SHE scheme over a comparable software implementations. However, the architecture presented in[17] is the first complete hardware accelerator to be able to process the polynomial ring operations for degree $n = 32768$. A modular implementation for all building blocks required in polynomial ring based fully homomorphic schemes is presented and used to instantiate the somewhat homomorphic encryption scheme YASHE. The implementation provides a fast polynomial operations unit using CRT and NTT for multiplication combined with an optimized memory access scheme, a fast Barrett like polynomial reduction method, an efficient division and rounding unit required in the multiplication of cipher-texts and an efficient CRT unit. Despite its performance gain, the implementation of the architecture in[17] on Xilinx Virtex-7 XC7V1140T FPGA uses less than 50% of the FPGA resources available. The complexity of the NTT algorithm for very large polynomial degrees leads to routing congestion, limiting the butterfly cores that can be used.

The main contribution in this paper is a modified version of the cached-FFT algorithm suitable for multi-core environments with distributed memories, which enables executing the NTT algorithm with time complexity of $\mathcal{O}(\sqrt{N}\log\sqrt{N})$ (Section 3). This algorithm inspired the design of the multi-core processor architecture presented in Section 4, which has been implemented on FPGA. The AES-128 and SIMON-64/128 circuits have been homomorphically evaluated using the LTV and YASHE schemes (Section 5). The AES-128 circuit has been evaluated in 4 minutes using the LTV scheme, with parameters that enable processing 2048 blocks in parallel, leading to an amortized performance of 117 ms. To the best of our knowledge this is the fastest performance result for the homomorphic evaluation of the AES circuit.

2 Background

2.1 LTV

The LTV scheme (Lopez-Tromer-Vaikuntanathan Leveled Fully Homomorphic Encryption Scheme) [7] was introduced in 2012 as the first fully-fledged fully homomorphic encryption scheme based on NTRU. It works in the ring $R = \mathbb{Z}[x]/f(x)$, where $f(x)$ is the d-th cyclotomic polynomial. The plain-text space is R_t, where t is small (typically, $t = 2$) and the cipher-text space is R_q, where q is a large integer (a more than 1000-bit integer). The scheme is described below as an example of the ring-based SHE schemes. It is also used in the evaluation of the proposed architecture. In this paper, only the routines of the scheme related to homomorphic function evaluation are discussed[1], which are:

1. *LTV.Add*:
   To add two ciphertext $c_{add}^{(i)} = c_1^{(i)} + c_2^{(i)}$ is computed.

[1] For the complete list of routines included in the LTV scheme, refer to [7]

2. *LTV.Mult*:
   To multiply two ciphertext $c_{mult}^{(i-1)} = ModSwitch(Relinearize(c_1^{(i)} * c_2^{(i)}))$ is computed. To relinearize (switch the key of) a ciphertext, $c^{(i)} = \Sigma_\tau \zeta_\tau^{(i)} c_\tau^{(i-1)}$ is computed, where $c^{(i-1)} = \Sigma_\tau 2^\tau c_\tau^{(i-1)}$. To perform modulus switching, $c^{(i)} = \lfloor \frac{q_i}{q_{i-1}} c^{(i)} \rceil_2$ is computed, where $\lfloor \cdot \rceil_2$ means matching parity bits.

   Parameter Set The parameter set used in this paper to evaluate the LTV scheme is the same parameter set used in [18] and is presented in Table 1.

d	$n = \Phi(d)$	$log_2(q)$	r^a	l^b
65536	32768	1271	16	41

Table 1: The parameter set used to evaluate the LTV scheme

[a] The relinearization window[18]
[b] Number of levels

2.2 YASHE

The YASHE scheme (Yet Another Somewhat Homomorphic Encryption Scheme) [3] was introduced in 2013. It works in the ring $R = \mathbb{Z}[x]/f(x)$, where $f(x)$ is the d-th cyclotomic polynomial. The plain-text space is R_t, where t is small (typically, $t = 2$) and the cipher-text space is R_q, where q is a large integer (a more than 1000-bit integer). The scheme is described below as an example of the ring-based SHE scheme. It is also used in the evaluation of the proposed architecture. In this paper, only the routines of the scheme related to homomorphic function evaluation are discussed[2], which are:

1. *YASHE.Add*(c_1, c_2):
   Return $c_1 + c_2 \in R_q$.
2. *YASHE.Mult*(c_1, c_2, evk):
   Return $c =$YASHE.$KeySwitch(c_0, evk)$ with $c_0 = \lfloor \frac{t}{q} c_1 c_2 \rceil \in R_q$. The KeySwitch operation returns $\langle WordDecomp_{w,q}(c), evk \rangle \in R_q$, where $\langle \cdot, \cdot \rangle$ is the inner product of two vectors and $WordDecomp_{w,q}(a)$ means decomposing a into its base w components $(a_i)_{i=0}^u$) such that a $= \sum_{i=0}^u a_i w^i$.

Parameter Set In [14] a group of parameters for the YASHE scheme have been presented with the results of their software implementations and security analysis. In this paper, we use the same parameter set used in [17], which is parameter set III in [14]. Table 2 presents the values of parameters in this parameter set.

[2] For the complete list of routines included in the YASHE scheme, refer to [3]

d	$n = \Phi(d)$	$log_2(q)$	$log_2(w)$
65536	32768	1225	205

Table 2: The parameter set used to evaluate the YASHE scheme

2.3 Chinese Remainder Theorem

During a homomorphic operation, computations are performed on polynomials of degree 2^{15} or 2^{16}, and coefficients consisting of thousands of bits. In [19], the authors failed to synthesize a 1040-bit parallel multiplier. In addition, synthesizing such a large multiplier will lead to a large critical path and low operating frequency. In order to overcome this issue, the Chinese Remainder Theorem and Residue Number System are used. CRT has proven to be efficient in several FPGA implementations, such as [17], [18], [20], [21] and [22].

In [17], the authors showed how apply the CRT in a similar way to how it is used in RSA cryptosystems. Two moduli are chosen, q and Q as the product of many small prime moduli q_i, such that $q = \prod_{i=0}^{l-1} q_i$ and $Q = \prod_{i=0}^{L-1} q_i$, $l < L$. Any computation in R_q can be converted into l computations in R_{q_i}. Additionally, if $Q \geq q^2$, then an operation in R can be regarded as an operation in R_Q as long as the coefficients of the inputs are less than q. Hence, the polynomial multiplication step in the *YASHE.Mult* operation, which is performed in R, can be performed as L polynomial multiplications in R_{q_i}. The moduli q used in this paper for modular operations of both the YASHE and LTV schemes are the product of 41 30-bit and 32-bit primes, respectively. In addition, the polynomial multiplication in YASHE is performed in R instead of R_q. In order to make use of the NTT algorithm for polynomial multiplication, it is performed in R_Q where $\log(Q) \geq 2\log(q)$. In this paper, Q is a product of 84 30-bit primes.

2.4 Cached Number Theoretic Transform

Although the complexity of straight-forward implementation of polynomial multiplication is $\mathcal{O}(n^2)$, it can be decreased to $\mathcal{O}(n \log n)$ using the Number Theoretic Transform (NTT), which is the Fast Fourier Transform (FFT) defined over a finite field or ring. A detailed discussion of the NTT algorithm can be found in[20]. The bottleneck of NTT hardware design is represented in the external memory access frequency. It ranges from loading and storing coefficients in the external memory after each butterfly operation to loading the entire polynomial inside the on-chip data cache, and computing the whole operation at once. Although it is tempting to choose the latter option, it was shown in [17] that due to the increasing gap between addresses accessed simultaneously by the NTT algorithm, different cores need to access each others caches, which leads to routing congestion.

In order to overcome this problem, the cached-FFT algorithm presented in [23] is used. It was proposed to enhance FFT performance on devices with hierarchical memory systems. The idea behind the algorithm is to load as many

coefficients as possible into the data cache and perform as many butterflies as possible on this set of coefficients. The operations between 2 memory access operations are called an epoch. In this paper, the 2-epoch implementation of the cached-FFT algorithm [23] is chosen, where the N-point NTT is computed using a series of $\sqrt{N}$-point NTTs.

3 Parallel Cached $\mathbf{NTT}_N$

While the NTT algorithm can be used to speed up polynomial multiplication over rings from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$, it can still be time consuming for large values of N (e.g. for $N = 2^{16}, N \log N = 2^{19} = 524,288$). In order to further speed up the computations, a parallel NTT algorithm is needed. In [18] and [17], the authors tried to adapt the iterative NTT algorithm to a multi-core environment. However, the performance gain for 2^{16}-point NTT on a 128 or 256 cores environment was not as expected due to bottlenecks in memory access.

In order to overcome these bottlenecks, a parallel-NTT algorithm is proposed. The proposed algorithm targets a specific processing environment with configurations that match the requirements of the algorithm. For an application that uses an N-point NTT algorithm, the proposed environment consists of $\sqrt{N}$ processing tiles. Each tile includes, but not restricted to:

1. A $\frac{3}{2}\sqrt{N}$-location data cache.
2. A $\sqrt{N}$-location twiddle factors cache.
3. A butterfly data path (a modular multiplier, modular adder and modular subtractor).

The idea behind the proposed algorithm, Algorithm 1, is to treat each tile as a separate 2-epoch cached-NTT processing element (For a complete description of the cached NTT algorithm, refer to [19], [23] and [24]). However, instead of writing the coefficients back to the external memory, a simple $\sqrt[4]{N}$-channel ring network on chip with $2\sqrt[4]{N}$ switches is used to reorder the coefficients in time $\mathcal{O}(\sqrt{N})$.

3.1 $\sqrt{N}$-point NTT ($\mathbf{NTT}_{\sqrt{N}}$)

Algorithm 2 is a modified version of the compact memory efficient NTT algorithm proposed in [20]. The main contribution in [20] is an advanced memory addressing scheme that enables packing the two coefficients related to the same butterfly into one memory location, such that there is only one read and one write memory accesses during a butterfly. These two coefficients are $A[k+j]$ and $A[k+j+m/2]$. The idea behind the algorithm is simple; since in the outer-most loop $m_i + 1 = 2 * m_i$, the coefficient $A[k+j]$ should be packed with $A[k+j+m]$ for the next iteration. Therefore, the next butterfly executed is that between $A[k+j+m]$ and $A[k+j+3*m/2]$. While the algorithm in [20] is well-suited for memory constrained devices, it needs two modification to fit in our algorithm:

Algorithm 1 Parallel NTT_N

Input: Polynomial $a(x) \in Z_q[x]$ of degree $N-1$, Array $\omega[N/2 * \log(N) - 1 : 0]$ of pre-computed twiddle factors
Output: Polynomial $A(x) \in Z_q[x] = \text{NTT}(a)$

```
A(x) ← bit_reverse(a(x))
for i = 0 to 1 do
    for j = 0 to √N − 1 do
        NTT_√N(a[(j + 1) * √N − 1 : j * √N], w[i * √N + (j + 1) * √N/2 − 1 : i * √N + j * √N/2]);
    end for
    Reorder;
end for
```

1. In our algorithm, Algorithm 2 acts as the cached-NTT algorithm, which is used $2\sqrt{N}$ times, twice on each core. The twiddle factors for the first $\sqrt{N}$ times are the same, while they are different for each of the second $\sqrt{N}$ times. To avoid costly twiddle factors calculations, it is assumed that for each core, the corresponding $\sqrt{N} * \log(\sqrt{})$ twiddle factors are precomputed.
2. In [20], the final stage of the algorithm does not produce ordered coefficients. Additionally, it is hard to order them with the memory constraint in [20]. For example, $(A[\sqrt{N}/2], A[0])$ (stored in $Data\_Cache[0]$) should be swapped with $(A[\sqrt{N}/2+1], A[1])$ (stored in $Data\_Cache[1]$) and the outputs should be stored in $Data\_Cache[0]$ and $Data\_Cache[\sqrt{N}/2]$. However, $Data\_Cache[\sqrt{N}/2]$ holds a value that will be used later. In the proposed parallel-NTT algorithm, the final loop is not the final stage of the algorithm, so the coefficients need to be in the correct order. On the other hand, during coefficient reordering, a core may receive a pair of coefficients targetting a memory location that holds a pair that should yet be sent to another core. To solve these two problems, the final loop in Algorithm 2 stores coefficients in a group of memory locations that are not used by this algorithm, in the correct order.

3.2 Coefficient Reordering

In this section, the structure and operation of the NoC used for coefficient reordering is described. The 2-epoch N-point cached-NTT the reordering function works as follows: coefficient i in group j of the first epoch becomes coefficient j in group i, where $i, j \in [0 : \sqrt{N} - 1]$.

In order to implement this function, the $\sqrt{N}$ are divided into $2\sqrt{N}$ clusters, each contains $\frac{1}{2}\sqrt{N}$ tiles. Each cluster is responsible for performing reordering in two steps:

1. Reordering between tiles of the same cluster.
2. Reordering between different clusters.

Algorithm 2 Memory Efficient Iterative NTT$_{\sqrt{N}}$

Input: Polynomial $a(x) \in Z_q[x]$ of degree $\sqrt{N}-1$, Array $\omega[(\sqrt{N}/2)*(\log(\sqrt{N}))-1:0]$ of pre-computed twiddle factors
Output: Polynomial $A(x) \in Z_q[x] = \text{NTT}(a)$

$A \leftarrow bit\_reverse(a(x))$ or $a(x)$; /*According to whether the first of second epoch*/
$i = 0$ or $1/2 * \sqrt{N}$; /*According to whether the first of second epoch*/
for $m = 2$ to $\sqrt{N}/2$ by $m = 2m$ **do**
  for $j = 0$ to $m/2 - 1$ **do**
    for $k = 0$ to $\sqrt{N}/2 - 1$ **do**
      $(t_1, u_1) \leftarrow (A[k + j + m/2], A[k + j])$; /*Stored in $Data\_Cache[k + j]$*/
      $(t_2, u_2) \leftarrow (A[k+j+3*m/2], A[k+j+m])$; /*[$Data\_Cache[k+j+m/2]$*/
      $t_1 \leftarrow t_1 * \omega[i]$;
      $t_2 \leftarrow t_2 * \omega[i + 1]$;
      $(A[k + j + m/2], A[k + j]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
      $(A[k + j + 3 * m/2], A[k + j + m]) \leftarrow (u_2 - t_2, u_2 + t_2)$;
      $Data\_Cache[k + j] \leftarrow (A[k + j + m], A[k + j])$;
      $Data\_Cache[k + j + m/2] \leftarrow (A[k + j + 3 * m/], A[k + j + m/2])$;
      $i = i + 2$;
    end for
  end for
end for
$m \leftarrow \sqrt{N}$;
$k \leftarrow 0$;
for $j = 0$ to $m/2 - 1$ **do**
  $(t_1, u_1) \leftarrow (A[k + j + m/2], A[k + j])$; /*Stored in $Data\_Cache[k + j]$*/
  $(t_2, u_2) \leftarrow (A[k + j + m/2 + 1], A[k + j + 1])$; /*[$Data\_Cache[k + j + 1]$*/
  $t_1 \leftarrow t_1 * \omega[i]$;
  $t_2 \leftarrow t_2 * \omega[i + 1]$;
  $(A[k + j + m/2], A[k + j]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
  $(A[k + j + m/2 + 1], A[k + j + 1]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
  $Data\_Cache[k + j + \sqrt{N}/2] \leftarrow (A[k + j + 1], A[k + j])$;
  $Data\_Cache[k + j + m/2 + \sqrt{N}/2] \leftarrow (A[k + j + m/2 + 1], A[k + j + m/2])$;
  $i = i + 2$;
end for

The algorithm responsible for reordering coefficients between tiles of the same cluster should read the pair of coefficients stored Data_Cache$_T[\frac{1}{2}\sqrt[4]{I} + j]$, which are $A[\sqrt{N}(T + \frac{1}{2}I\sqrt[4]{N}) + 2 * j]$ and $A[\sqrt{N}(T + \frac{1}{2}I\sqrt[4]{N}) + 2 * j + 1]$ and store these coefficients in Data_Cache$_{(2j)}[\frac{1}{2}\sqrt[4]{I} + T]$ and Data_Cache$_{(2j+1)}[\frac{1}{2}\sqrt[4]{I} + T]$, where I is the cluster id $\in [0, 2\sqrt[4]{N} - 1]$ and T is the tile id $\in [0, \frac{1}{2}\sqrt[4]{N} - 1]$. If T is even, then the loaded coefficients are stored in the lower part of target memory location, and vice versa. Since the NoC consists of $\sqrt[4]{N}$ channels, $\sqrt[4]{N}$ can be ordered in parallel.

First, a reordering square matrix $O_{\frac{1}{4}\sqrt[4]{N} * \frac{1}{4}\sqrt[4]{N}}$ is precomputed. This matrix is not a unique matrix but it must satisfy the following properties:

1. $o_{ij} \in [0, \frac{1}{4}\sqrt[4]{N} - 1]$.

2. $o_{ij} \neq o_{ik} \forall j \neq k$.
3. $o_{ik} \neq o_{jk} \forall i \neq j$.
4. If $o_{ik} = j$, then $o_{ij} = k$.

The row index i represents the reordering step, while the column index $j = T/2$. Each values of j is associated, with two tiles: $2T$ and $2T + 1$. Algorithm 3 describes how this matrix is used to reorder coefficients within the same cluster I, where CH_x represents channel number x of the NoC. The properties of O ensure that reordering is performed correctly and that the maximum bandwidth of the NoC is used. After executing Algorithm 3 by all clusters in parallel, $\frac{1}{2}\sqrt{N}\sqrt[4]{N}$ coefficients are in their correct place, while the algorithm consumes time $\frac{1}{2}\sqrt[4]{N}$ (The outer loop consumes two steps, as the iterations of the inner loops are executed in parallel). Algorithm 4 describes how inter-cluster reordering is performed. Clusters need to exchange $(\frac{1}{2}\sqrt[4]{N})^2 = \frac{1}{4}\sqrt{N}$ coefficients between each two of them. However, in order to maximize the throughput of the NoC, at each step of the algorithm each cluster needs to send a packet of $\sqrt[4]{N}$ coefficients. Thus, the communication is organized as follows: at iteration i of the outermost loop of Algorithm 4, cluster I sends $\frac{1}{4}\sqrt[4]{N}$ packets to cluster $I+i$ and receives $\frac{1}{4}\sqrt[4]{N}$ packets from cluster $I + (32 - i)$. Each packet takes 1 step to be loaded into the NoC, i steps inside the network and $\frac{1}{4}\sqrt[4]{N}$ steps to store in the correct locations. Notice that each packet can be loaded into the NoC in parallel as each pair of coefficients is loaded from a different tile. However, each packet is stored in a single tile.

Algorithm 3 Intra-cluster NTT_N Coefficient Reordering

Input: Reordering Matrix O
  for i in 0 to $\frac{1}{4}\sqrt[4]{N} - 1$ **do**
    for j in 0 to $\frac{1}{4}\sqrt[4]{N} - 1$ **do**
      $CH_{4j} \leftarrow Data\_Cache_{2j}[\frac{1}{4}\sqrt[4]{N}I + O_{ij}][0]$;
      $CH_{4j+1} \leftarrow Data\_Cache_{2j}[\frac{1}{4}\sqrt[4]{N}I + O_{ij}][1]$;
      $CH_{4j+2} \leftarrow Data\_Cache_{2j+1}[\frac{1}{4}\sqrt[4]{N}I + O_{ij}][0]$;
      $CH_{4j+3} \leftarrow Data\_Cache_{2j+1}[\frac{1}{4}\sqrt[4]{N}I + O_{ij}][1]$;
    end for
    for j in 0 to $\frac{1}{4}\sqrt[4]{N} - 1$ **do**
      $Data\_Cache_{2O_{ij}}[\frac{1}{4}\sqrt[4]{N}I + j] \leftarrow (CH_{4j+2}, CH_{4j})$;
      $Data\_Cache_{2O_{ij}+1}[\frac{1}{4}\sqrt[4]{N}I + j] \leftarrow (CH_{4j+3}, CH_{4j+1})$;
    end for
  end for

3.3 Time Complexity Analysis

Algorithm 1, NTT_N consists of three main parts, repeated twice:

Algorithm 4 Inter-cluster NTT_N Coefficient Reordering

for i in 0 to $2\sqrt[4]{N} - 1$ **do**
  for K in 0 to $\frac{1}{4}\sqrt[4]{N} - 1$ **do**
    for j in 0 to $\frac{1}{4}\sqrt[4]{N} - 1$ **do**
      $CH_{4j} \leftarrow Data\_Cache_{2j}[\frac{1}{4}\sqrt[4]{N}(I+i)+k][0]$;
      $CH_{4j+1} \leftarrow Data\_Cache_{2j}[\frac{1}{4}\sqrt[4]{N}(I+i)+k][1]$;
      $CH_{4j+2} \leftarrow Data\_Cache_{2j+1}[\frac{1}{4}\sqrt[4]{N}(I+i)+k][0]$;
      $CH_{4j+3} \leftarrow Data\_Cache_{2j+1}[\frac{1}{4}\sqrt[4]{N}(I+i)+k][1]$;
    end for
    for s in 1 to i **do**
      advance_ring;
    end for
    for j in 0 to $\frac{1}{4}\sqrt[4]{N} - 1$ **do**
      $Data\_Cache_{2k}[\frac{1}{4}\sqrt[4]{N}(I-i+32)+j] \leftarrow (CH_{4j+2}, CH_{4j})$;
      $Data\_Cache_{2k+1}[\frac{1}{4}\sqrt[4]{N}(I-i+32)+j] \leftarrow (CH_{4j+3}, CH_{4j+1})$;
    end for
  end for
end for

1. $\text{NTT}_{\sqrt{N}}$: has time complexity of $\mathcal{O}(\sqrt{N}\log\sqrt{N})$.
2. Intra-cluster Reordering: has time complexity of $\mathcal{O}(\sqrt[4]{N})$.
3. Inter-cluster Reordering: The number of steps in this part is $\sum_{i=0}^{\frac{1}{4}\sqrt[4]{N}} i + \frac{1}{4}\sqrt[4]{N} = \frac{\frac{1}{4}\sqrt[4]{N}-1}{2}(2(\frac{1}{4}\sqrt[4]{N}+1) + \frac{1}{4}\sqrt[4]{N} - 2)$. Thus, this part of the algorithm has time complexity of $\mathcal{O}(\sqrt{N})$.

Therefore, the overall time complexity of the proposed algorithm is $\mathcal{O}(\sqrt{N}\log\sqrt{N})$. The dominant part of the algorithm is the $\text{NTT}_{\sqrt{N}}$. When the initial load and final store operations are added, the complexity becomes $\mathcal{O}(N)$.

3.4 Valid values of N

The proposed algorithm works only for $N = 2^{4k+8}$ and needs modifications to work for any value of N. However, for $k = 2$, $N = 2^{16}$, which is suitable for the target applications of FHE.

4 Hardware Architecture

The proposed architecture, shown in Figure 1, targets evaluating FHE schemes based on Rings of dimension, i.e. the degree of polynomials involved in the NTT operation is $N = 2^{16} - 1$. In compliance with Algorithm 1, the architecture consists of 32 clusters, each cluster consists of 8 tiles. Each tile performs Algorithm 2. Each tile is also capable of performing polynomial addition/subtraction and coefficient-wise multiplication, in addition to the small CRT, large CRT and

Divide-and-Round operations described in [17]. The rest of this section describes the features of each of the building blocks of the proposed tile in addition to the NoC switch.

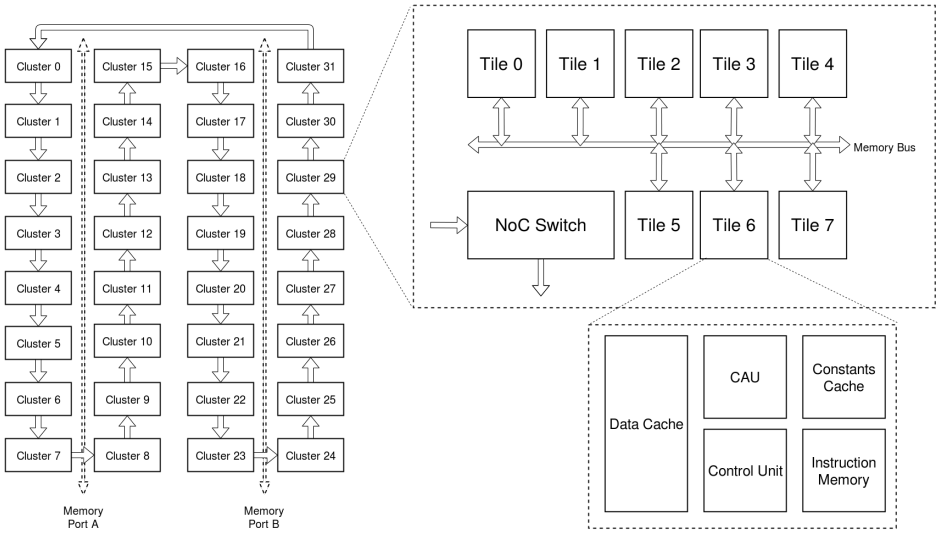

Fig. 1: The overall proposed architecture

4.1 Configurable Arithmetic Unit (CAU)

Each tile includes a configurable arithmetic unit (CAU) is responsible for performing the arithmetic operations used in the evaluated encryption schemes. These operations are:

1. Modular multiplication, based on Barrett reduction.
2. Barrett reduction.
3. Modular addition/subtraction.
4. The butterfly operation of the NTT algorithm.
5. Modular multiply-and-accumulate (MAC) operations.
6. Serial integer multiplication.

To perform these operations, the CAU unit includes three 32-bit integer multipliers, two 64-bit integer adders/subtractors and four 32-bit integer adders/subtractors, in addition to a group of registers. The CAU unit is organized as shown in Figure 2. A group of comparators and multiplexers are used to select the outputs, which are not shown in Figure 2. Additionally, the signal named *barret_out* refers to the output of the Barrett reducer.

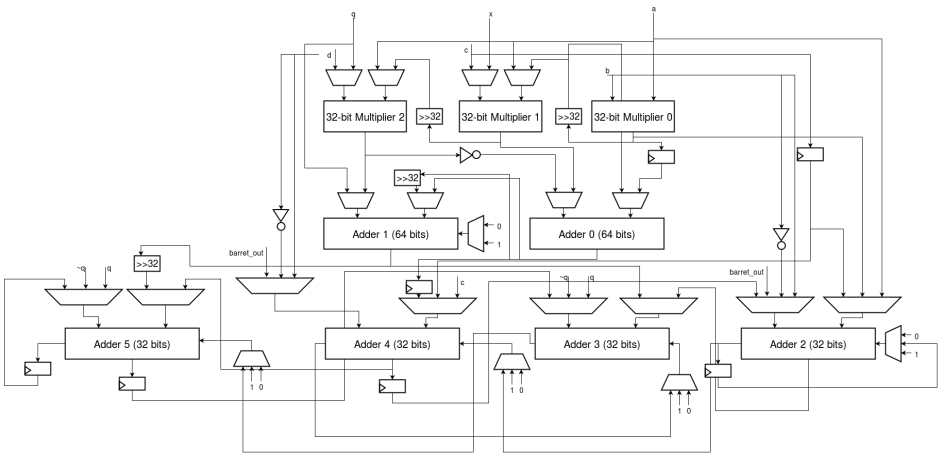

Fig. 2: Configurable Arithmetic Unit (CAU)

4.2 Data Cache

Each tile includes two dual port 64-bit 256 RAMs. The overall size of the available data cache in the architecture is enough to store four polynomials of degree 2^{16}. This makes it possible to perform more than one instruction with reduced memory overhead. For example, it is possible to perform one complete polynomial multiplication operation, which consists of 2 NTT_N, 1 point-wise multiplication and 1 INTT_N, with only 2 load and 1 store operations, minimizing the memory overhead.

4.3 Precomputed Constants

The architecture uses two types of constants:

1. Full width large constants: The constant of the Divide-and-Round operations $(\frac{2}{q})$ - The constants of the large CRT operations for both q (41 residues) and Q (84 residues).
2. 32-bit constants: The NTT operation twiddle factors - The CRT moduli q_i - The Barrett reduction constants x_i corresponding to each q_i - The INTT scaling factors $N^{-1} mod q_i$ - The small CRT constants: $41 * 84$ to convert from q to Q.

The full width constants are stored as 32 bit words in a 256 word ROM in each tile. On the other hand, the 32-bit constants cannot be stored using the same approach as their number is very large. For example, the small CRT need 3444 locations, while the NTT/INTT operation requires 86100 locations for each tile. Instead, each tile includes another 32-bit 1024 word RAM that is loaded with the 32-bit constants currently being used. A smart way to organize

operations to reduce memory overhead is to batch similar operations together. For example, do as many NTT operations as possible before doing the large CRT of Divide-and-Round operations.

4.4 Cluster NoC Switch

Each cluster includes a ring NoC switch (shown in Figure 3) and a switch controller. The switch consists of 16 channels, each of 32 bits. It is responsible of the following operations:

1. Storing the values inside the NoC registers into the local tiles.
2. Loading the NoC registers with values read from the local tiles.
3. Passing the values at the inputs of the switch to its outputs.

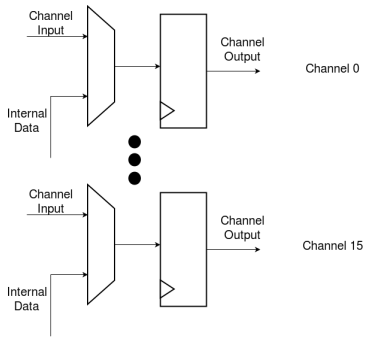

Fig. 3: The ring NoC switch

Switch controllers synchronize all the switches to do the same operations at the same time. Controllers are also responsible for executing Algorithms 3 and 4, where the matrix O_{4x4} is given by:

$$O_{4x4} = \begin{pmatrix} 0\,1\,2\,3 \\ 1\,0\,3\,2 \\ 2\,3\,0\,1 \\ 3\,2\,1\,0 \end{pmatrix}$$

4.5 Memory Interface

The external memory is assumed to be a dual-port RAM. Thus, the clusters are divided into two groups of 16 clusters each. The 128 tiles in each group share the same memory bus that is connected to one of the RAM ports. It is the responsibility of the main CPU to organize data inside the RAM into blocks to be read by the tiles. This simplifies the amount of logic required by the architecture in order to interface with the external memory.

5 Results and Comparison

The proposed design has been implemented using Xilinx ISE 14.2 Design Suite, with the target device Virtex-7 XC7VX1140T-2 which is the same device used in [17] and the largest device in the Virtex-7 FPGA family. The design achieves an overall utilization ratio of 60%. The amount of resources used is presented in Table 3. The regularity of the proposed architecture makes it possible to increase the utilization ratio in comparison to [17]. One sign of this regularity is percentage of LUT-FF paired registers out of all the registers used (61.6%), which indicates that most of the registers are placed as closely as possible to the combinational logic blocks, which simplifies placement and routing, which was the main problem preventing increasing the utilization ratio in [17]. On the other hand, a lot of low level optimizations have been performed to make it possible for the architecture to fit on a single FPGA. These optimizations include:

1. Since the implementation targets the evaluation of both the LTV and YASHE schemes for the equal dimensions, the NoC switch control unit has been implemented as a micro-coded ROM. This is due to the following observations:
   (a) The percentage of unused Block Rams is greater than the percentage of unused LUTs.
   (b) Block Rams are more compact and easier to place in the design than a logic control unit.
   (c) The 32 NoC switches required can be implemented using less than 2% of Block Rams.
   (d) The reordering algorithm is the only algorithm that requires on-chip communications between different tiles and it can easily be implemented as a fixed length parametrized microprogram (with the cluster id as a parameter).
2. Whenever values need to be moved within the external memory, the architecture relies on the main CPU to perform this task. Each additional mode of load and store operations in the external memory needs around 100 additional LUTs of control logic, which corresponds to around 25600 LUTs for the overall design (3.5% of the available LUTs).

	Clock Frequency	Registers	LUTs	Fully used LUT-FF pairs	BRAMs	DSP48E1
Value	209 MHz	324736	678368	200032	1600	3072
Percentage		22%	95%	61.6%[a]	85%	91%

Table 3: The implementation results on Xilinx Virtex-7

[a] Percentage of LUT-FF paired registers out of all the registers used.

Table 4 shows the time consumed by each of the primitive operations performed by the design. It is noticeable that all operations have been accelerated

due to two reasons; the increased clock frequency (e.g. 209 MHz vs. 143 MHz in [17]) and the increased number of processing cores (256 vs. 128 in [17]). However, the major gain in the proposed implementation is in the NTT/INTT operations, with a speed-up factor of 14x. This is due to the asymptotic performance gain of Algorithm 1. On the other hand, the proposed design achieves 12.25x and 11.5x speed-up over the design in [18] in both the NTT and large CRT operations respectively.

Operation	Clock Cycles	Time	Time[17]	Time[a]	Time[18][b]
ADD/SUB(n[c]=2^{15})	133	$0.63\mu s$	$15\mu s$[d]	$111\mu s$	
ADD/SUB(n=2^{16})	261	$1.3\mu s$	$29\mu s$	$221.5\mu s$	
Point-wise MULT	273	$1.31\mu s$	$29\mu s$	$223.5\mu s$	
NTT	4,964	$23.8\mu s$	$334\mu s$	$245\mu s$	$\sim 3ms$[e]
INTT	5,237	$25.1\mu s$	$363\mu s$	$258.47\mu s$	
Small-CRT	209,920	$1ms$	$0.8ms$	$1.35ms$	
Large-CRT($l = 41$)	1,178,368	$5.65ms$	N/A[f]	$7.75ms$	$89ms$
Large-CRT($l = 84$)	2,414,218	$11.5ms$	$19.248ms$	$15.05ms$	
Divide-and-Round	3,666,304	$17.6ms$	$19.678ms$	$23.18ms$	

Table 4: Timing Results

[a] With memory overhead

[b] The design in [18] does not include all the operations implemented in this paper. In addition, the results published in [18] include the timing performance of dominant operations.

[c] Polynomial degree.

[d] The authors of [17] did not clarify for which degree this result is. The degree $n = 2^{15}$ is used for operations such as YASHE/LTV.ADD, while the degree $n = 2^{16}$ is required for the polynomial multiplication operation. This number is estimated, because the point-wise multiplication (which consumes the same amount of time) is used only for $n = 2^{16}$.

[e] In [18], it is mentioned that 1 polynomial multiplication operation consumes $9.51ms$ and consists of 2 NTT transforms, 1 INTT transform and 1 point-wise multiplication.

[f] This operation is needed for the LTV scheme, which is not evaluated in [17].

Table 5 includes the evaluation results of the YASHE scheme and the time consumed to evaluate 1 block of the SIMON-64/128. The YASHE.MULT operation consists of the following steps:

1. The input polynomials are lifted from R_q to R_Q using the small-CRT operation.
2. Polynomial multiplication is computed, using Barrett reduction, which needs 4 NTT operations, 3 INTT operations, 3 point-wise multiplications and 1 polynomial subtraction operation.
3. Large-CRT operation for ($l = 84$) and divide-and-round operations are performed once.

4. Key Switching is performed:
   - 22 NTT operations, 22 point-wise multiplications, 21 polynomial additions and 1 INTT are performed, in addition to 2 NTT, 2 INTT and 2 point-wise multiplications and 1 polynomial subtraction for Barrett reduction.

The SIMON-64/128 block cipher consists of 44 Rounds, each includes 32 AND Gates and 96 XOR Gates. For comparison purposes, the results in Table 5 exclude the time consumed to load and store coefficients in the external memory. The proposed architecture is 1.5x faster than the estimates in [17] for evaluating one block of the SIMON-64/128 faster. Although it has been shown earlier that the NTT operation is 14x faster on this architecture, other dominant operations does not experience the same speed up. Additionally, in [17] the divide-and-round and large CRT operations are executed in parallel, which is not the case in the proposed architecture. It is also important to mention that it may be misleading to compare the performance with neglecting the memory overhead, which can be critical for some operations as shown in Table 4.

Operation	Time[a]	Time[17]
YASHE.ADD	$25.83\mu s$	$172\mu s$
YASHE.MULT	$74.51ms$	$112.025ms$
SIMON-64/128	$105s$	$157.731s$

Table 5: Evaluation results of the YASHE Scheme and homomorphic evaluation of the SIMON-64/128 block cipher

[a] Without memory overhead.

The other scheme that is used to evaluate this architecture is the LTV scheme, which used to evaluate 1 block of the AES block cipher. Table 6 includes the evaluation results of the LTV scheme. The LTV.ModSwitch operation consists of one large CRT operation, where $l = 41$, followed by a group of polynomial additions whose number is $l - 1$, depending on the level of the computation. For $l = 41$ the relinearization operation consists of one large CRT operation, 80 NTT operations, 41 INTT operations and 3280 point-wise addition and multiplication operations. These results indicate that the proposed architecture achieves a speed up of ∼11-14x, which is near the gain obtained by the new proposed NTT algorithm over the design proposed in [18].

The AES circuit evaluated in this paper includes the S-Box design proposed in [25] and consists of 18448 XOR gates and 5440 AND gates. The homomorphic evaluation of the AES circuit takes 4 minutes (3.7x faster than the estimates in [18]), at an amortized AES evaluation time of 117 ms/block. According to the estimation methodology in [18] and [26], the proposed architecture should have consumed ∼ 2.4 minutes, which agrees with the 7x speed up factor measured on

Operation	LTV.ADD	LTV.Relinearize	LTV.ModSwitch	LTV.MULT
Time	$4.55ms$	$36.34ms$	$7.86ms$	$72.3ms$
Time[18]		$526ms$ [a]	$89ms$	

Table 6: Evaluation results of the LTV Scheme

[a] The results for LTV operations are for the worst-case (Level 0 of the computation)

the level of the LTV scheme. However, it can be easily detected that the difference between the estimated and measured performance is due to the following factors:

1. The difference in the AES circuits: In [26] and [18], the estimations were done for a circuit that needed only 2880 relinearization operations, as opposed to 5440 in this paper. It is believed that the performance can be enhanced by implementing the appropriate circuit/optimizations.
2. The estimation methodology: In [18], the authors provide a rough estimate for the evaluation time of the AES circuit based on only the number of relinearizations (AND gates) that should be performed. Despite their high speed compared to AND gates, the number of XOR gates in the design affects the overall execution time significantly.
3. The estimation methodology also neglects the 41 polynomial multiplication operations performed for each AND gate before relinearization.

However, the proposed results show that hardware acceleration is crucial for accelerating FHE schemes. The overall results for evaluating AES and SIMON-64/128 on both YASHE and LTV scheme is presented in Table 7.

Algorithm	NTT	INTT	Add	Mult	lCRT (41)	lCRT(84)	sCRT	DIV	Execution Time
YASHE/ AND Gate	1320	375	986	1236	0	1	1	1	461.06 ms
YASHE/ XOR Gate	0	0	0	41	0	0	0	0	4.55 ms
YASHE/ SIMON-64/128									11.14 minutes
YASHE/ AES-128									43.2 minutes
LTV/ AND Gate (Level 0)	244	164	3280	3280	2	0	0	0	72.3 ms
LTV/ XOR Gate (Level 0)	0	0	0	41	0	0	0	0	4.55 ms
LTV/ SIMON-64/128									1 minutes
LTV/ AES-128									4 minutes

Table 7: The overall results for evaluating AES and SIMON-64/128 on both YASHE and LTV schemes using the proposed architecture

6 Conclusions and Future Work

In this paper, a new algorithm for executing the NTT operation in distributed-memory multi-core environments in time $\mathcal{O}(\sqrt{N}\log(\sqrt{N}))$ has been proposed. An architecture for homomorphic function evaluation using this algorithm has also been presented. The architecture has been implemented on Virtex 7 FPGA and has been studied using both the YASHE and LTV FHE schemes. This study has led to the following conclusions:

1. From a hardware perspective, the LTV scheme is $\sim$ 10x faster than the YASHE scheme. This is due to the complex rounding operation used in the YASHE. MULT operation, which requires polynomial multiplication in R instead of R_q and a full-width integer division operation for each AND gate evaluated. Additionally, the performance of the LTV scheme increases with the increase of circuit level, while the performance of the YASHE scheme is constant over the whole operation.
2. Although some papers have estimated the time required for different circuits to be evaluated, the actual practical numbers are sometimes much larger than these estimates. The reason for this is that the estimation methodologies sometimes neglect crucial time consuming operations, such as memory interfacing.

The future work includes evaluating the proposed architecture using ASIC. While the proposed results show that hardware acceleration can achieve a significant speed-up over software implementations, the performance is not yet within the practical bounds (the amortized time for evaluating 1 block of AES-128 takes 117 ms). Our target is to reach an amortized performance of 20 ms per block. Another direction is to design a multi-FPGA environment. The foreseen challenge in these direction is to evaluate their cost (in terms of money) as opposed to the performance gain. On the other hand, the study of performance of FHE schemes presented in section 5 has to be extended to other schemes, such as BGV and FV. Additionally, it has to be extended to other types of schemes that rely on Ring operations, such as [27], which will take the performance of the bootstrapping operation up to a few milliseconds.

References

1. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of secure computation **4** (1978) 169–180
2. Gentry, C., et al.: Fully homomorphic encryption using ideal lattices. In: STOC. Volume 9. (2009) 169–178
3. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Cryptography and Coding. Springer (2013) 45–64
4. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Advances in Cryptology–CRYPTO 2012. Springer (2012) 868–886

5. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive **2012** (2012) 144
6. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Advances in Cryptology–CRYPTO 2013. Springer (2013) 75–92
7. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the forty-fourth annual ACM symposium on Theory of computing, ACM (2012) 1219–1234
8. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on Computing **43** (2014) 831–871
9. Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: Public Key Cryptography–PKC 2012. Springer (2012) 1–16
10. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Advances in Cryptology–EUROCRYPT 2012. Springer (2012) 465–482
11. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ACM (2012) 309–325
12. Van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Advances in cryptology–EUROCRYPT 2010. Springer (2010) 24–43
13. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Designs, codes and cryptography **71** (2014) 57–81
14. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes fv and yashe. In: Progress in Cryptology–AFRICACRYPT 2014. Springer (2014) 318–335
15. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The simon and speck families of lightweight block ciphers. IACR Cryptology ePrint Archive **2013** (2013) 404
16. Costache, A., Smart, N.P.: (Which ring based somewhat homomorphic encryption scheme is best?)
17. Dimitrov, V., Verbauwhede, I.: Modular hardware architecture for somewhat homomorphic function evaluation. In: Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings. Volume 9293., Springer (2015) 164
18. Öztürk, E., Doröz, Y., Sunar, B., Savaş, E.: Accelerating somewhat homomorphic evaluation using fpgas. Technical report, Cryptology ePrint Archive, Report 2015/294 (2015)
19. Putnam, A., Macias, A.: Accelerating homomorphic evaluation on reconfigurable hardware. In: Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings. Volume 9293., Springer (2015) 143
20. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: (Compact ring-lwe based cryptoprocessor)
21. Pöppelmann, T., Güneysu, T.: Towards practical lattice-based public-key encryption on reconfigurable hardware. In: Selected Areas in Cryptography–SAC 2013. Springer (2013) 68–85
22. Aysu, A., Patterson, C., Schaumont, P.: Low-cost and area-efficient fpga implementations of lattice-based cryptography. In: Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on, IEEE (2013) 81–86

23. Baas, B.M.: A generalized cached-fft algorithm. In: Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on. Volume 5., IEEE (2005) v–89
24. Baas, B.M.: An approach to low-power, high-performance, fast Fourier transform processor design. PhD thesis, Citeseer (1999)
25. Boyar, J., Peralta, R.: A depth-16 circuit for the aes s-box. IACR Cryptology ePrint Archive **2011** (2011) 332
26. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic aes evaluation using ntru. IACR Cryptology ePrint Archive **2014** (2014) 39
27. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Advances in Cryptology–EUROCRYPT 2015. Springer (2015) 617–640