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. In this paper, a new architecture for accelerating homomorphic function evaluation on FPGA is proposed. A parallel cached NTT algorithm with an overall time complexity $\mathcal{O}(\sqrt{N}\log\sqrt{N})$ is presented. The architecture has been implemented on Xilinx Virtex 7 XC7V1140T FPGA. achieving a 60% utilization ratio. The implementation performs 32-bit 2^{16}-point NTT algorithm in $23.8\mu s$, achieving speed-up of 2x over the state of the art architectures. 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

Since Gentry [1] proposed the first FHE scheme based in 2009, a lot of efforts have been directed towards constructing practical FHE schemes. The recently proposed FHE schemes are either based on the Learning with Errors (LWE)[2][3][4][5], Ring-LWE[6][7], N-th degree Truncated Polynomial Ring (NTRU) [8][9] or the Approximate Greatest Common Divisor (AGCD)[10] problems. Ring-based FHE schemes, such as BGV[6], FV[7], YASHE[8] and LTV[9], achieve the best performance results, due to the possibility to use the Single-Instruction-Multiple-Data (SIMD) technique[11], 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 [12], a comparison between the YASHE scheme and the FV scheme has been presented. A more recent study[13] has expanded the comparison to all the 4 ring-based schemes: BGV, LTV, YASHE and FV. These studies has shown that NTRU schemes have better performance compared to ring LWE schemes.

In 2015, three implementations, [14], [15] and [16], have been published targeting the acceleration of the whole operation of homomorphic function evaluation using FPGA. The design proposed in [16] uses an efficient double-buffered memory access scheme and a polynomial multiplier based on the Number Theoretic Transform (NTT). The implementation if presented for the parameter set ($n = 16384, \lceil log_2(q) \rceil = 512$), which is only capable of evaluating 9 levels of multiplication before bootstrapping is required. 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 [15], a hardware/software implementation is proposed, including a large NTT based multiplier capable of multiplying very large degree polynomials. The design is highly optimized using numerous techniques to speedup the NTT computations, and to reduce the burden on the PC/FPGA interface. On the other hand, a modular implementation for all building blocks required in polynomial ring based fully homomorphic schemes is presented in [14] and used to instantiate the YASHE scheme. 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. However, 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 Homomorphic Encryption Schemes (LTV/YASHE)

In this section, the two encryption schemes used to evaluate the architecture proposed in this paper are briefly discussed. The LTV scheme (Lopez-Tromer-Vaikuntanathan Leveled Fully Homomorphic Encryption Scheme) [9] was introduced in 2012 as the first fully-fledged FHE 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). On the other hand, the YASHE scheme (Yet Another Somewhat Homomorphic Encryption Scheme) [8] 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). Table 1 includes the routines related to homomorphic function evaluation of each of those two schemes and the parameters used to implement them.

	LTV	YASHE
ADD	$c_{add}^{(i)} = c_1^{(i)} + c_2^{(i)}$	$c_1 + c_2$
MULT	$c_{mult}^{(i-1)} = ModSwitch\ (Relinearize\ (c_1^{(i)} * c_2^{(i)}))$ is computed. To relinearize 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.	$c = KeySwitch(c_0, evk)$ 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 $= \sum_{i=0}^{u} a_i w^i$.
Parameters	The parameter set used in this paper to evaluate the LTV scheme is the same parameter set used in [15]. These parameters are $d = 65536, n = \Phi(d) = 32768, log_2(q) = 1271, r = 16^a$ and $l = 41^b$.	In [12] 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 [14], which is parameter set III in [12]. These parameters are $d = 65536, n = \Phi(d) = 32768, log_2(q_=1225$ and $log_2(w) = 205$.

Table 1: Homomorphic Evaluation Operations of the LTV and YASHE schemes

[a] The relinearization window[15]
[b] Number of levels

2.2 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. Implementing a multiplier that can multiply coefficients of more than 1000 bits will lead to low performance an is impractical. 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 [14], [15], [17], [18] and [19].

In [14], the authors showed how to 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.

2.3 Cached Number Theoretic Transform

The bottleneck of NTT hardware design is 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, which leads to routing congestion due to the increasing gap between addresses accessed simultaneously by the NTT algorithm. In order to overcome this problem, the cached-FFT algorithm presented in [20] is used. It was proposed to enhance FFT performance on devices with hierarchical memory systems. In this paper, the 2-epoch implementation of the cached-FFT algorithm [20] is chosen, where the N-point NTT is computed using a series of $\sqrt{N}$-point NTTs.

3 Parallel Cached NTT$_N$

In [15] and [14], the authors tried to adapt the iterative NTT algorithm to a multi-core environment. However, the performance gain was not as expected due to bottlenecks in memory access. In this paper, a parallel cached-NTT algorithm is proposed in order to compute a complete NTT operation with only one memory access. The algorithm runs on an environment that consists of $\sqrt{N}$ processing tiles. Each tile includes:

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 [16], [20] and [21]). 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 $\mathcal{O}(\sqrt{N})$.

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 $A(x) \in Z_q[x] = \mathrm{NTT}(a)$
  $A(x) \leftarrow bit\_reverse(a(x))$
  for i = 0 to 1 do
      for j = 0 to $\sqrt{N}-1$ do
          $\mathrm{NTT}_{\sqrt{N}}(a[(j+1)*\sqrt{N}-1 : j*\sqrt{N}], w[i*\sqrt{N}+(j+1)*\sqrt{N}/2-1 : i*\sqrt{N}+j*\sqrt{N}/2]);$
      end for
      Reorder;
  end for
```

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

Algorithm 2 is a slightly modified version of the compact memory efficient NTT algorithm proposed in [17]. The main contribution in [17] 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.

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. The first step is reordering between tiles of the same cluster, while the second step is reordering between different clusters. The algorithm responsible for reordering coefficients between tiles of the same cluster should read the pair of coefficients stored $\mathrm{Data\_Cache}_T[\frac{1}{2}\sqrt[4]{I}+j]$, 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 $\mathrm{Data\_Cache}_{(2j)}[\frac{1}{2}\sqrt[4]{I}+T]$ and $\mathrm{Data\_Cache}_{(2j+1)}[\frac{1}{2}\sqrt[4]{I}+T]$, 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 pre-computed. 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 $o_{ij} = k$.

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 $A(x) \in Z_q[x] = \mathrm{NTT}(a)$
  $A \leftarrow bit\_reverse(a(x))$ 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 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:

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.

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
```

4 Hardware Architecture

The proposed architecture 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 [14]. 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. Each tile consists of the following components:

1. **Configurable Arithmetic Unit (CAU)**: responsible for performing the arithmetic operations used in the evaluated encryption schemes. These operations are:
   (a) Modular multiplication, based on Barrett reduction.
   (b) Barrett reduction.
   (c) Modular addition/subtraction.
   (d) The butterfly operation of the NTT algorithm.
   (e) Modular multiply-and-accumulate (MAC) operations.
   (f) Serial integer multiplication.

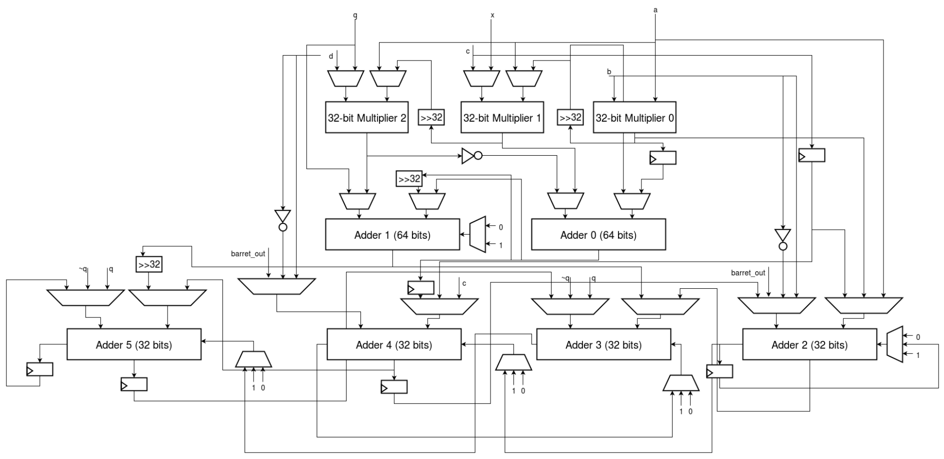

Fig. 1: Configurable Arithmetic Unit (CAU)

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 1. A group of comparators and multiplexers are used to select the outputs, which are not shown in Figure 1. Additionally, the signal named *barret_out* refers to the output of the Barrett reducer.

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.
3. **Precomputed Constants**: The architecture uses two types of constants:
   (a) 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). They are stored as 32 bit words in a 256 word ROM in each tile.
   (b) 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. Each tile includes a 32-bit 1024 word RAM that is loaded with the constants currently being used.

Cluster NoC Switch each cluster includes a ring NoC switch (shown in Figure 2) and a switch controller. The switch consists of 16 channels, each of 32 bits. It is responsible accessing the data caches of its tiles and passing coefficients to the adjacent switches.

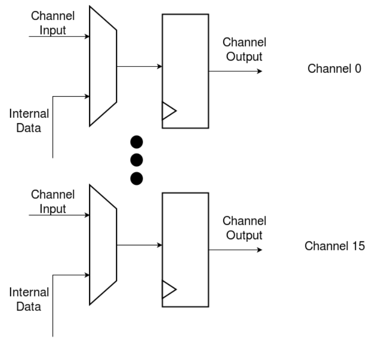

Fig. 2: The ring NoC switch

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 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 2. Due to the regularity of the proposed architecture, 61.6% of the registers used are in the form of fully used LUT-FF pairs, which simplifies placement and routing, which was the main problem preventing increasing the utilization ratio in [14].

	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 2: The implementation results on Xilinx Virtex-7

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

Table 3 shows the time consumed by each of the primitive operations performed by the design. All operations have been accelerated in comparison to [14] due to two reasons; the increased clock frequency (e.g. 209 MHz vs. 143 MHz) and the increased number of processing cores (256 vs. 128). While the design in [15] consists of 256 cores, the architecture proposed in this paper achieves a 2x speed up. Table 4 includes the evaluation results of the YASHE scheme and the time consumed to evaluate 1 block of the SIMON-64/128. In order to multiply two cipher-texts using YASHE.MULT, first the input polynomials are lifted from R_q to R_Q using the small-CRT operation. Second, polynomial multiplication is computed, using Barrett reduction, which needs 4 NTT operations, 3 INTT operations, 3 point-wise multiplications and 1 polynomial subtraction operation.

Finally, large-CRT operation for ($l = 84$) and divide-and-round operations are performed, followed by key switching. Key switching consists of 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.

Operation	Clock Cycles	Time	Time[14]	Time[15]
ADD/SUB(n[a]=2^{15})	133	$0.63\mu s$	$15\mu s$	
ADD/SUB(n=2^{16})	261	$1.3\mu s$	$29\mu s$	
Point-wise MULT	273	$1.31\mu s$	$29\mu s$	
NTT	4,964	$23.8\mu s$	$334\mu s$	$50\mu s$
INTT	5,237	$25.1\mu s$	$363\mu s$	
Small-CRT	209,920	$1ms$	$0.8ms$	
Large-CRT($l = 41$)	1,178,368	$5.65ms$	N/A[b]	$89ms$
Large-CRT($l = 84$)	2,414,218	$11.5ms$	$19.248ms$	
Divide-and-Round	3,666,304	$17.6ms$	$19.678ms$	

Table 3: Timing Results

[a] Polynomial degree.

[b] This operation is needed for the LTV scheme, which is not evaluated in [14].

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 4 exclude the time consumed to load and store coefficients in the external memory. The proposed architecture is 1.5x faster than the estimates in [14] for evaluating one block of the SIMON-64/128 faster.

Operation	Time	Time[14]
YASHE.ADD	$25.83\mu s$	$172\mu s$
YASHE.MULT	$74.51ms$	$112.025ms$
SIMON-64/128	$105s$	$157.731s$

Table 4: Evaluation results of the YASHE Scheme and homomorphic evaluation of the SIMON-64/128 block cipher

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 5 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.

Operation	LTV.ADD	LTV.Relinearize	LTV.ModSwitch	LTV.MULT
Time	4.55ms	36.34ms	7.86ms	72.3ms
Time[15]		526ms	89ms	

Table 5: Evaluation results of the LTV Scheme

The AES circuit evaluated in this paper includes the S-Box design proposed in [22] 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 [15]), at an amortized AES evaluation time of 117 ms/block. The overall results for evaluating AES and SIMON-64/128 on both YASHE and LTV scheme is presented in Table 6.

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	41	0	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	41	0	0	0	0	0	4.55 ms
LTV/ SIMON-64/128									1 minutes
LTV/ AES-128									4 minutes

Table 6: The overall results for evaluating AES and SIMON-64/128 on both YASHE and LTV schemes using the proposed architecture

6 Conclusion

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.

The future work includes evaluating the proposed architecture using ASIC and/or a multi-FPGA environment. 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.

References

1. Gentry, C., et al.: Fully homomorphic encryption using ideal lattices. In: STOC. Volume 9. (2009) 169–178
2. 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
3. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on Computing **43** (2014) 831–871
4. Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: Public Key Cryptography–PKC 2012. Springer (2012) 1–16
5. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Advances in Cryptology–EUROCRYPT 2012. Springer (2012) 465–482
6. 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
7. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive **2012** (2012) 144
8. 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
9. 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
10. 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
11. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Designs, codes and cryptography **71** (2014) 57–81
12. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes fv and yashe. In: Progress in Cryptology–AFRICACRYPT 2014. Springer (2014) 318–335
13. Costache, A., Smart, N.P.: (Which ring based somewhat homomorphic encryption scheme is best?)
14. 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
15. Öztürk, E., Doröz, Y., Sunar, B., Savaş, E.: Accelerating somewhat homomorphic evaluation using fpgas. Technical report, Cryptology ePrint Archive, Report 2015/294 (2015)
16. 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
17. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: (Compact ring-lwe based cryptoprocessor)

18. 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
19. 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
20. 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
21. Baas, B.M.: An approach to low-power, high-performance, fast Fourier transform processor design. PhD thesis, Citeseer (1999)
22. Boyar, J., Peralta, R.: A depth-16 circuit for the aes s-box. IACR Cryptology ePrint Archive **2011** (2011) 332