

FPGA Acceleration of Homomorphic Encryption

Zhaorui Ni
dept. of computer science
University of Southern California
Los Angeles, United States of America
zhaoruin@usc.edu

Zhengrong Li
Ming Hsieh Department of Electrical
and Computer Engineering
University of Southern California
Los Angeles, USA
zhengron@usc.edu

Abstract: Fully Homomorphic Encryption (FHE) can truly fundamentally solve the data security problem when entrusting data and its operations to a third party in cloud computing. For the Homomorphic Encryption, it has two stages: automorphism and number theoretic transformation (NTT). This paper use the FPGA HLS pragma implement two algorithm separately and then use different clock cycle and different input data points to generate the results about the hardware resource consumption as well as the latency. The result shows that the NTT and automorphism can be fully pipelined.

Keywords—NTT, FPGA, automorphism, HLS pragma

I. INTRODUCTION

Cloud services is more and more important in our daily life. Cloud service happens when we update our social media status, check our bank balance or upload our daily photo into the internet, this service helps us storing and processing the information, analyzing the big-data and so on. Recently Deep neural network (DNN) has rapidly developed and is widely used in many applications. Recently, deep learning as a service (DLaaS) has emerged as a promising way for further application of DNNs [1]. However, cloud computing will cost a significant risks that have been analyzed over the last decade [2] – [4]. In the process of DLaaS, clients need to send their private data to the cloud servers, which may cause privacy leakage [5]. By employing the Homomorphic Encryption (HE) schemes, this problem can be efficiently handled. HE can provide protection to clients when they are sending data to the cloud while keeping some of the useful properties of cloud services [6]. This is because HE allows computations over encrypted data without access to the secret key [7]. The Homomorphic Encryption happens on the server end that shows in Figure 1. However, the high computation complexity and latency of the HE schemes still form the bottleneck for applications. As a phase of HE, homomorphic rotation has a significant portion of the computation [8] - [10].

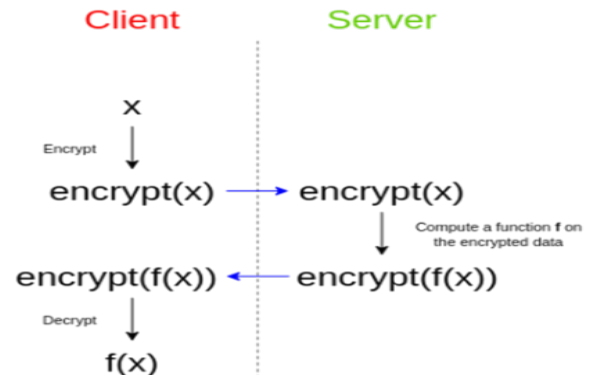


Fig. 1. Homomorphic Encryption

FPGA is a parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor. FPGA is become more and more popular due to their high flexibility and energy efficiency; it had been widely used to be attached to the data center nodes as a customized and specified accelerators [11], also, the high-bandwidth and low-latency on chip STAM and external DRAMS that the FPGA provided gives the user a fine-grained access to the memory system which makes FPGA suitable to accelerate the intensive compute task. With the help of High-Level Synthesis (HLS), FPGA programmer can work at a higher-level of abstraction by using a software algorithm to specify the hardware functionality [12].

The goal of this project is to develop a parallel architecture for HE rotation on FPGA using HLS. Since the most proportion of the computations occur on this stage, we assume that it can improve the performance of the homomorphic encryption by accelerating the homomorphic rotation.

The HE rotation has two core algorithms, which are automorphism and Number Theoretic Transform (NTT). These two algorithms occupy the largest number of computations. This project will implement optimization and parallelization into these two algorithms to achieve the high performance of the HE rotation.

However, the parallelization for these two algorithms is not straightforward, which leads two main challenges in this project: (1) the automorphism algorithm is a rearrangement for the ciphertext which needs complex data communication between the computation stages [13]. (2) the modular multiplication in NTT is usually resource-consuming and slow, the high resource requirements of modular arithmetic are needed for designing the low latency NTT cores. To

address these challenges and make HE rotation hardware friendly, we implement optimization on these two algorithms. For automorphism, this project will use loop unroll to reduce the computation time and use on-chip SRAM to reduce the off-chip memory access. For NTT, we implement and parallelize the butterfly unit which can increase the parallelism and the performance of the operation. The main contributions of this project are listed here:

- We implement the two algorithms (automorphism and NTT) in HLS, and we experiment the different pragmas in HLS to see how different pragmas may affect the latency and hardware-cost.
- We make the optimization and parallelization based on the serial version of these two algorithms. In Vitis HLS, the result of the performance and resource estimates can give us the detailed latency and the resources of the parallelization.
- We conduct a comprehensive analysis of the number of resources that are needed to parallelize these two algorithms. Based on this analysis, we can get the relationship between the resources and performance, and make some optimization based on this relationship.
- This project evaluates the performance and resources of the two algorithms (automorphism and NTT) under different range of ciphertext size N . The evaluation result enables us to project the performance and resource of the whole HE rotation based on the implementation.

The rest of the paper is as follow. Section II gives the background on Homomorphic Rotation based on the data type, Automorphism, NTT and the Homomorphic operation. Section III introduce the related work about this project and give a conclusion of the difference of our project and prior word. Section IV discuss the HLS-based design method and how we parallelize the automorphism and NTT. Section V explain the environment setup and analyze the result based on this setup. Section VI conclude the paper.

II. BACKGROUND

In Homomorphic Rotation, we need the ciphertext as the input and output the rotated ciphertext, so that the server can do some operation based on the rotated ciphertext. We will discuss more detail about how and why this homomorphic rotation works, based on the following sections. By introducing the background of the Homomorphic Rotation, it led us how we can parallelize this algorithm.

A. Data type

Even though the Homomorphic Rotation is taken the ciphertext as the input, and encryption and decryption are performed by the client, it is useful to understand how the encryption works and how the plaintext represents. Each plaintext vector can be encoded as a polynomial with N coefficients mod t . We represent plaintext space as R_t , so

$$a = a_0 + a_1x + \dots + a_{N-1}x^{N-1} \in R_t$$

is plaintext. Plaintext is encrypted into a ciphertext consisting of two polynomials of N integer coefficients modulo some Q

$\gg t$. In here, we take each ciphertext polynomial as a member of R_Q .

To encrypt a plaintext $m \in R_t$, we need use a secret key that represented as a polynomial $s \in R_Q$, one samples as uniformly random $a \in R_Q$, an error $e \in R_Q$ with small entries, and computes the ciphertext ct as

$$ct = (a, b = as + te + m)$$

Here, ciphertext $ct = (a, b)$ is the input of the Homomorphic Rotation. This ciphertexts should not reveal the information of the plaintext or the secret key. In short, the input of our algorithm would two 1D vectors that represent the coefficients of the ciphertext polynomials. And these two vectors a and b will do the following operation to get our final result.

B. Automorphism

Automorphism is used to rearrange the ciphertext based on the rotation step k , and its results is the special permutation of the coefficients of the ciphertext polynomial. To specify what automorphism do, there are N size of 1D, and the result of the automorphism denoted $\sigma_k(a)$ or $\sigma_{-k}(a)$ for the rotation step $k < N$. When k is positive, the vector will be rotated to the right side, and when k is negative, the vector will be rotated to the left side. Specifically,

$$\sigma_k(a): a_i \rightarrow (-1)^s a_{ik \bmod N} \text{ for } i = 0, \dots, N-1$$

For example, $\sigma_4(a)$ permutes a 's coefficients so that a_0 stays at position 0, a_1 goes from position 1 to position 4, and so on, e.g., with $N = 1024$, the element in index 257 of the original vector a_{257} goes to position 4, since $257 \cdot 4 \bmod 1024 = 4$. The picture below shows another example when $N = 16$ and $k = 3$,



Fig. 2. Example of Automorphism

To perform Homomorphic Rotation, we first compute an automorphism on the ciphertext polynomial, it can be represented as: $ct_\sigma = (\sigma_k(a), \sigma_k(b))$. The challenge of implementing the automorphism is that each coefficient of the output may depend on any of the input coefficient, and the coefficient of the input polynomial may distribute across the whole vector, so creating a pipeline data path that produces these outputs is non-trivial. This is a crucial reason why pure SIMD or GPU architectures do not work well on HE.

C. NTT

Number Theoretic Transform (NTT) is used to multiply two polynomials, naively, the multiplication of two polynomials requires convolving their coefficients, which is an expensive operation that cost $O(N^2)$. However, just like convolutions can be made fast using Fast Fourier Transform (FFT), polynomial multiplication can be made faster with the Number Theoretic Transform [14]. NTT is a variant of the

Discrete Fourier Transform (DFT) for modular arithmetic. The NTT takes an N -coefficient polynomial as input and returns an N -element vector representing the input in the NTT domain.

The NTT is a multiplication of two polynomials with the ring of polynomials $Z_q[x]/\phi(x)$ (here, $Z_q[x]$ denote the polynomials with coefficients in Z_q , and $\phi(x)$ denote the reduction polynomial) which is the most time-consuming but fundamental operation utilized in Homomorphic Encryption. The multiplication operation in the ring basically takes the coefficients a_i and b_i that are in Z_q which denotes the numbers $\{0, 1, \dots, q-1\}$ as input, these input coefficients can be represented as the polynomials $A(x)$ and $B(x)$, and it returns the output polynomial $C(x)$.

It is hard for the classical polynomial multiplication technique to provide a fast polynomial multiplication operation in the HE schemes, since the classical implementation has a high time complexity of $O(N^2)$. Also, the polynomial output of the classic polynomial multiplication needs to be reduced with the reduction polynomial $\phi(x)$ and it will cost some cycles to this reduce operation. In this way, the NTT is implemented in the polynomial multiplication technique in the efficient HE schemes, which can reduce the complexity to $O(N \log N)$.

Because of the cheap operation of NTT (compared to the naive polynomial multiplication), the modular multiplication in the Homomorphic Rotation can be performed in $O(N \log N)$ time complexity by using two forward NTTs, one as the element-wise multiplication of two vectors, and another as an INTT (Inverse Number Theoretic Transform). And because the NTT is a linear transformation, so automorphisms and the homomorphic operations can also be performed in the NTT domain:

$$NTT(\sigma_k(a)) = \sigma_k(NTT(a))$$

$$NTT(a + b) = NTT(a) + NTT(b)$$

To make the NTT more efficient and hardware-friendly, we need implement the butterfly unit (BU) into the NTT algorithm, the main process of NTT is calling the BU operation repeatedly. In this project we use Gentleman-Sande butterfly unit, in this unit the input of each unit is in the normal order and the output of each unit are in the bit-reversed order. The parallelization of the NTT operation is performing the butterfly operations concurrently. However, the input of a stage in NTT in corresponding to the output of the pervious stage of NTT operation, which will cause the limited parallelism. Also, with the concurrent access to the elements in multiple BU, the memory access will become more complicated.

D. Homomorphic operations

The Homomorphic operation in Homomorphic Rotation is compose two operations: Homomorphic addition and Homomorphic multiplication. Like the FFT which reduce the

time complexity of convolution from $O(N^2)$ to $O(N \log N)$, which also requires element-wise multiplication and addition.

Homomorphic addition of the ciphertext is the operation that add the corresponding polynomials (represent as vectors) of two ciphertext together, it can be represented as follow:

$$ct = ct_0 + ct_1 = (a_0 + b_0, a_1 + b_1, \dots)$$

Homomorphic multiplication of the ciphertext is the operation that multiple the corresponding polynomials (represent as vectors) of two ciphertext together, it can be represented as follow:

$$ct_0 \times ct_1 = (a_0 \times b_0, a_1 \times b_1, \dots)$$

In the project, after we do the automorphism and NTT of the ciphertext, we will do the element-wise multiplication of the output NTT with the rotation key vector and do the addition to the original ciphertext vector to make it as the same result as the naive polynomial multiplication and save the time from $O(N^2)$ to $O(N \log N)$.

III. RELATED WORK

Gentry in 2009 first camp up the first construction of the HE that conceptualized by Rivest, Adleman and Dertouzos in 1977 [15][16]. However, the performance of the first generation of the HE schemes were extremely slow, it did not provide some efficient and practical solutions. Over the last decade, prior work has proposed multiple popular Homomorphic Encryption schemes such as BGV, B/FV, GSW, CKKS and Gazelle library. Also, the implementation in a high-end GPU reduces the computation time by several factors [17]. Hardware accelerators like FPGA or GPU make parallel processing capabilities in HE become possible and achieve the fast computation time.

The permutation operation in the Gazelle library is similar to the Homomorphic Rotation. The Gazelle library is an open-source library based on the latticed-based packed additive homomorphic encryption (PAHE) scheme [18]. In Gazelle library the permutation operation contains two stages the *TransDecomp* and *Automorph*. The *TransDecomp* stage is used to transform and decompose the input vector preparing of the automorph operation, and the automorph stage is to rearrange the ciphertexts according to the different values of rotation step which is composed of simple mapping operation, multiplications, and additions. The Gazelle library has a huge potential for real-time application since the Gazelle library takes the optimized parameter set of the HE and operate the efficient homomorphic operation [19] [20]. The difference between the Gazelle library and our project is that: we make simplify this Homomorphic Rotation into two main algorithm and some homomorphic operation and make them hardware friendly.

For Automorphism, Axel, Nikola and etc. in their F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption (Extended Version) had mentioned a Automorphism unit that will compute N Automorphism. In their approach, their break down the vector into multiple chunks which can be represented as the 2D matrix and do the row and column permutation into these chunks, for every permutation, they transpose the vector, in this way, they finish the vectorizing automorphism. In the Automorphism in the

F1, they fully pipelined the automorphism by implementing the method above. However, in the Homomorphic Rotation, we just need to take one Automorphism in every vector (two vector total), so the method of the F1 is not applied in this project.

Fully Homomorphic Encryption (FHE) can perform any number of homomorphic addition and homomorphic multiplication operations. Many cryptographers put the idea of fully homomorphic encryption in an important position on par with the idea of public-key encryption. Since Gentry [21] proposed the first FHE scheme in 2009, fully homomorphic encryption has made considerable progress, but there are still many shortcomings, especially in terms of computing efficiency, security and the application of fully homomorphic encryption. For a lattice base FHE with a dimension of only 2048, a multiplication of 785006 bits (approximately) is required. By analyzing the analysis result of each function in the cryptographic library of the FHE algorithm, most of the execution time of the calculation is large integer multiplication (including square operation), which consumes 53% to 62% of the total execution time of FHE [22]. Using dedicated hardware to accelerate large integer multiplication can greatly improve the calculation efficiency of FHE and promote the marketization of FHE. In addition to the traditional long multiplication of large integers, there are Karatsuba algorithm, Toom-Cook algorithm and Schönhage-Strassen algorithm[23]. The complexity of the above algorithms decreases sequentially, and the complexity of implementation increases sequentially [24, 25]. For digital multiplications greater than 10^{40000} (approximately), the Schönhage-Strassen algorithm is beginning to outperform Karatsuba and Toom-Cook multiplication [26]. This feature makes the Schönhage-Strassen algorithm suitable for multiplication in FHE. Using Number Theoretical Transform (NTT) instead of Discrete Fourier Transform (DFT), you can avoid rounding errors by using integer arithmetic instead of floating-point arithmetic. In order to accelerate the operation of FHE, Wang et al. [27] proposed a large integer multiplication architecture and implementation for fully homomorphic encryption. Feng et al. [28] proposed a dual-modulus number theory transformation structure using the Chinese remainder theorem, which improved the efficiency of number theory transformation.

IV. HYPOTHESIS & KEY IDEA, ARCHITECTURE, ALGORITHM

A. hypothesis

In this project, we implement and parallelize the two algorithms: Automorphism and NTT and based on the result of these two algorithms, we project the whole performance of the Homomorphic Rotation. The hypothesis for these two algorithms and the performance of the Homomorphic Rotations is list below:

- For the Automorphism algorithm we assume that using loop unroll can reduce the computation time and using on-chip SRAM can reduce the off-chip memory access
- For NTT part, we assume that implementing and parallelizing the butterfly unit to increases the parallelism and the performance of the operation.
- For the whole Homomorphic Rotation, two stages of algorithm should be run in sequence since the input of NTT depends on the output of Automorphism.

When we get the result of these two algorithms, we can easily project to the whole Homomorphic Rotation.

The expected result of the two algorithms is that: for the Automorphism, elementwise multifaction and addition, the input of algorithm and the output has no data dependency, we use the unroll factor f , and the size N as the input parameter and the result will be N/f cycles add some setup time T_s ; For the NTT, since we do the BU concurrently, the input parameter will be size N and the number of PEs that represents as B , the time complexity of the serial version NTT is $O(N\log N)$, so the total latency of the NTT will be $N\log N / B$. and since the input of these two algorithms is two vectors, so the total expected latency will be:

$$T_{HR} = 2 \times (T_{\text{Automorphism}} + T_{NTT} + T_{\text{homomorphic operation}} + T_s)$$

$$T_{HR} = 2 \times (N/f + N\log N / B + T_s)$$

B. Homomorphic Rotation framework

The framework of Homomorphic Rotation is shown below, the input ciphertext $ct[0]$, $ct[1]$ and the rotation key $rk[0]$, $rk[1]$ and the output of the automorphism $autoct[0]$, $autoct[1]$, the output of the NTT $ct'[0]$ and $ct'[1]$ are 1D vector of size N . N is one of parameters that we evaluate the algorithm that will discuss in the next section. And the '*' in the Algorithm 1 denotes the element-wise multiplication of two vectors, the '+' denotes the element-wise addition of two vectors.

Algorithm 1: The framework of Homomorphic Rotation

Input: ciphertext(ct), represented as ct[0] and ct[1]
Input: rotationkey(rk), represented as rk[0] and rk[1]
Output: ciphertext(ct'), represented as ct'[0] and ct'[1]
/* Automorphism is used */
1 autoct[0] = automorphism(ct[0])
2 autoct[1] = automorphism(ct[1])
/* NTT / INTT is used */
3 ct'[0] = intt(ntt(autoct[0]) * rk[0]) + ct[0]
4 ct'[1] = intt(ntt(autoct[1]) * rk[1]) + ct[1]

In this Homomorphic Rotation, the input is the cipher text 1D vector, it represents the coefficients of the cipher text polynomials, at first, we do the automorphism or form a permutation of the cipher text and then it do the NTT to the output of automorphism, after that it do some element-wise multiplication and addition to get the final output. In this project, all the input and output are a 1D vector of size N . Figure 3 below show the structure of the Homomorphic Rotation.

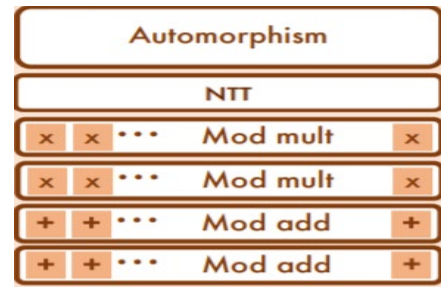


Fig.3. Structure of the Homomorphic Rotation

C. Automorphism

Automorphism σ_k maps the element at index i to index $ki \bmod N$. It is used to rearrange the cipher text based on the rotation step k , and its result is the special permutations of the coefficients of the cipher text polynomial. For every iteration in the Automorphism, it will read the input element one by one, and based on the rotation step, it calculates the corresponding index of the output array, and then write to the element into this index. Since it is single input and output in each iteration, we can do the loop unroll to achieve the parallelization.

This project aim synthesizing our two algorithms using the Xilinx HLS tool which takes C/C++ codes as input and generates synthesizable Verilog or VHDL code. It provides specific directives(pragmas) for users to optimize the generated RTL design. The *loop unrolling* directive (UNROLL) in HLS will transfers a single loop operation into multiple independent copies of the operations performed in the loop body and runs this operation concurrently. By applying this directive, it will increase the parallelism and the performance of the operation. This directive will apply to the iteration of the Automorphism operation. In each iteration, the algorithm will read one element from the vector and take it into another index of the vector.

Another challenge of the Automorphism is the memory control as it need lots of read and writer operation in an irregular pattern. By changing the index of the element in the array, it enforces this architecture need multiple small memory block or a large memory with multiple read and write ports to support this operation. In this project we use the array partitioning (ARRAY PARTITION) directive in HLS to automate the partition of an array into smaller memories or individual registers, which can increase the latency of the read or write operation from the memory at the expense of more hardware resources.

In our design, we partitioned each array ct that represent the coefficients of the ciphertext polynomial into register fire using the *complete*, the option of the ARRAY_PARTITION directive provided by the HLS, in this way, the program can read or write the data at any time. This design can reduce the latency, but it will cause the high hardware usage, since it need large multiplexers to read or write multiple data.

Algorithm 1 Automorphism

Input: Ct(1D array of size N), rotationStep, modulus
Output: autoCt(1D array)
 $galoisElement \leftarrow getGalois(rotationStep)$
for $element \in Ct$ **do**
 $indexRaw \leftarrow indexRaw + galoisElement$
 $index \leftarrow getIndex(indexRaw, N - 1)$
 $result \leftarrow element$
 if $(index_{raw} \gg \log_2(N))$ **then**
 $nonZero \leftarrow (result \neq 0)$
 $result \leftarrow (modulus - result) \bmod (nonZero)$
 end if
 $autoCt[index] \leftarrow result$
end for

Automorphism algorithm shown above. To parallelize this algorithm, we use ARRAY_PARTITION to partition the input vector and output vector, the *getIndex()* and *getGalois()* is used to operate the *rotationStep*, so that it can calculate the index easily, these two function takes 1 cycles in general to get the correct result. After partitioned the input and output

vector, we can do the loop unroll to the main for loop function, to achieve calculate concurrently. The result of the automorphism and the range of parameters will be discussed in next section.

D. key idea and hypothesis of Number theoretic theorem(NTT)

NTT is a generalization of traditional DFT on a finite field. The number theory transformation uses the rotation factor to be equivalent to in the DFT operation, where is the primitive root of the modulus prime number. Since is a prime number, according to Dirichlet's theorem, its primitive root must exist. The calculation formula of point NTT and its inverse process INTT is

$$X_k = \sum_{n=0}^{N-1} x_n (W_N)^{nk} \pmod{p} \quad (1)$$

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k (W_N)^{-nk} \pmod{p} \quad (2)$$

The W_n here is the n th root of the unity. So the algorithm is [9]:

Algorithm 3 HLS-friendly algorithm

Input a polynomial $a(x) \in \mathbb{Z}_q[X]/\langle X^n + 1 \rangle$
Input $n = 2^l$, number of Parallel PEs B
Input $\omega[N]$, where $\omega[i] = \omega^i$
Output $\hat{a} = \text{NTT}(a)$
1: $\hat{a} \leftarrow \text{bit-reverse}(a)$
2: **STAGE LOOP:**
3: **for** $(i = 0 : i < l : i++)$ **do**
4: **BUTTERFLY LOOP:**
5: **for** $(s = 0 : s < N/2 : s = s + b)$ **do**
6: **IDX_LOOP:**
7: **for** $(b = 0 : b < B : b++)$ **do**
8: $j[b] \leftarrow (s + b) \gg (l - 1 - i)$
9: $k[b] \leftarrow (s + b) \& ((u \gg i) - 1)$
10: $i_e[b] \leftarrow j[b] \cdot (1 \ll (l - u)) + k[b]$
11: $i_o[b] \leftarrow i_e[b] + (1 \ll (l - i - 1))$
12: $i_w[b] \leftarrow (1 \ll i) + ((s + b) \gg (l - i - 1))$
13: **end for**
14: **MEM_READ_LOOP:**
15: **for** $(b = 0 : b < B : b++)$ **do**
16: $U[b] \leftarrow \hat{a}[i_e[b]]$
17: $V[b] \leftarrow \hat{a}[i_o[b]]$
18: $W[b] \leftarrow \omega[i_w[b]]$
19: **end for**
20: **OP_LOOP:**
21: **for** $(b = 0 : b < B : b++)$ **do**
22: $t \leftarrow W[b] \cdot V[b] \bmod q$
23: $E[b] \leftarrow U[b] - t$
24: $O[b] \leftarrow U[b] + t$
25: **end for**
26: **MEM_WRITE_LOOP:**
27: **for** $(b = 0 : b < B : b++)$ **do**
28: $\hat{a}[i_e[b]] \leftarrow E[b]$
29: $\hat{a}[i_o[b]] \leftarrow O[b]$
30: **end for**
31: **end for**
32: **end for**
33: **return** $\hat{a}(x)$

Similar to the relationship between DFT and IDFT, NTT and INTT are very similar, and their optimization methods and hardware structure can be discussed together. Because NTT is

a generalization of DFT in a finite field, for the number-theoretic transformation of large numbers of points, you can also refer to the idea of the Curry-Tukey fast Fourier transform algorithm in the fast Fourier transform algorithm. For current FHE applications, the use of 12 Mbit multiplicative encryption is sufficient. Under this length multiplication, a number theory transformation of 1,048,576 points is needed. According to different application requirements and structures, large-point number theory transformations usually decompose from base 4 to base 64. Butterfly operation is realized by different basic operation units. The larger the base, the fewer stages that need to be cascaded for number-theoretic transformation and multiplication, and the smaller the delay, but the amount of calculation for each stage will increase. For the number-theoretic transformation of points, it can be decomposed according to the 5-level base 16 or the 4-level base 32. This paper optimizes the base 32 arithmetic units so that the computational capacity and efficiency of the base 32 arithmetic unit are suitable for the realization of the hardware system. Rewriting the index of the sum in formula (1) and applying the characteristics of the unit root, we can get

$$\begin{aligned}
& X(32768k_1 + 1024k_2 + 32k_3 + k_4) \\
&= \sum_{n_1=0}^{31} W_{32}^{n_1 k_1} \left(\sum_{n_2=0}^{31} W_{32}^{n_2 k_2} \left(\sum_{n_3=0}^{31} W_{32}^{n_3 k_3} \right. \right. \\
&\quad \cdot \left. \left. \left(\sum_{n_4=0}^{31} x(n) W_{32}^{n_4 k_4} \right) W_{1024}^{n_3 k_4} \right) W_{32768}^{n_2 k_4} \right. \\
&\quad \cdot \left. W_{1048576}^{n_1 k_4} \right) \pmod{p}
\end{aligned} \tag{3}$$

Among them, $n_1, n_2, n_3, n_4, k_1, k_2, k_3, k_4$ is an integer of $[0, 32)$.

It can be seen from the decomposition of formula (3) that a number theory transformation of 1048576 points can be decomposed into a 4-level base 32 arithmetic unit and a multiplication of twiddle factors. The rotation factor can be calculated in advance and stored in ROM, and it can be read directly when needed. The calculation amount of base 32 butterfly operation accounts for more than 90% of the number theory transformation multiplication, and its optimization of the logarithmic transformation efficiency is very important.

In the realization of number theory transformation, it is often necessary to perform a series of modular operations, such as addition, subtraction, and multiplication by a power of 2. If it is implemented as a 64-bit wide operation, the intermediate or final result needs to be modulo. Although the process of directly calculating modulo is very fast, it still requires a lot of hardware. However, if you take advantage of the characteristics of Solinas prime numbers, expand each 64-bit value to 192 bits, and operate on the pipeline with a 192-bit wide value, you can avoid performing modulo operations after each operation to reduce the area. In the choice of prime numbers, Solinas prime numbers are usually chosen as $2^{64} - 2^{32} + 1$.

As we can see that there are three main for loop for the NTT algorithm. For the innermost loop, there are 4 stages: index-calculation loop, memory read loop, calculation loop and write-back loop. And for the first loop is about inverting the array and the second and third loop is for calculation of the array for the next stage and then writing back to the array, so the loop can be pipelined because that when calculating the

even and the odd part, it uses the same array element but output different number which is the odd and even part. And when it writes back, the two array element is also independent of each other. So this article proposes to unroll the innermost loop and pipeline the second loop.

So the initial hypothesis for that is when the clock rate increases, the more resources it will consume because the NTT algorithm needs more elements to fully pipeline its parallel parts. And another assumption is that when the input array size becomes large, it also should consume more resources to fully unroll the loop.

V. EVALUATIONS (EXPEREMENTS)

A. Experimental Setup

For Automorphism we use C/C++ to implement this algorithm and parallelize it in the FPGA HLS using the *pragma* directive. For the target device in FPGA, we use *xcu19p-fsva3824-2-e*, and the Vivado IP Flow Target to observe the performance and resources. As the parameters, there are two parameters related to the parallelization: N and f , N is the size of the input vector ct , and f represent as how many unroll factor to unroll the automorphism unit. The N is range from 1024 to 8192 and f range from 1 to 8.

For the NTT part, the language is C and the platform is Xilinx FPGA on USC carc {xcvu11p-flga2577-1-e}, and I choose change the clock rate from 2ns to 20ns and the input size from 1k to 80k, and the unroll factor chose as 2.

B. Results and Analysis

Table I shows the result of the Automorphism, we use 16 sets of data to test this algorithm, there are four options of input size N which represent the count of the coefficient element: 1024, 2048, 4096 and 8192, and the f will be 1, 2, 4, 8. As we can see in the table, when the N increase it will increase the latency and the resources, the result of latency is related with the input size. And when the f can help us reduce the latency, take (1024, 8) as an example, the speed up is 7.8X. However, high usage of hardware also comes with when the factor f increase, in this program, we take FF and LUT in FPGA as the resources result, we can observer that when f doubles, the LUT also doubles, and the FF will increase a little bit when f is 1 to 4, when f become 8 the ff will exponentially increase. And when we do the experiment, we cannot set f to 16, which will lead resource limitation in FPGA, this is because the unroll in HLS will create the hardware to consume the loop operation. In the future work, we will try to figure a better solution to solve the high hardware usage of the Automorphism.

TABLE I. RESULT OF AUTOMORPHISM

(N, f)	Performance and Resources		
	Latency(cycles)	FF	LUT
(1024, 1)	1026	33963	38161
(1024, 2)	514	35142	68055
(1024, 4)	259	37675	140143
(1024, 8)	131	107963	284336
(2048, 1)	2050	66733	38167
(2048, 2)	1026	67914	48067
(2048, 4)	515	70455	140167
(2048, 8)	259	206286	284384
(4096, 1)	4098	66736	38172
(4096, 2)	2050	67919	68077
(4096, 4)	1027	70462	140187
(4096, 8)	515	206299	284424
(8192, 1)	8194	66739	38179
(8192, 2)	4098	67924	68091
(8192, 4)	2051	70475	140215
(8192, 8)	1027	206321	284480

For NTT part, the result is:

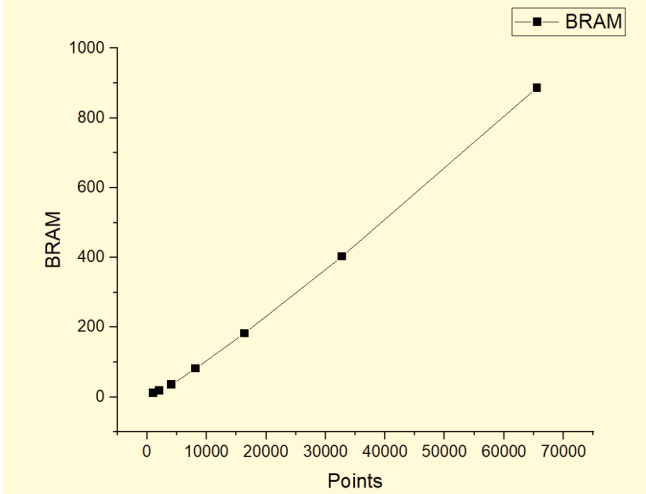


Fig. 4. The relationship between number of BRAM and points

For the BRAM, as we can see, when the points increases, the number of the BRAM usage is increases linearly.

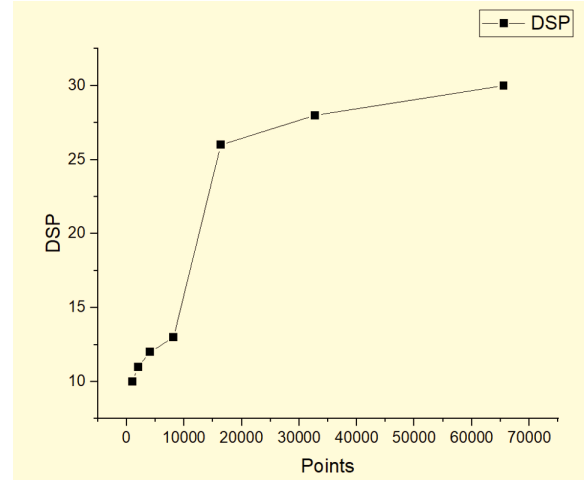


Fig. 5. The relationship between number of DSP and number of points

And for the DSP resources consumption, as we can see that the DSP firstly increases linearly in the range of 1024-8192 and when input points from 8192 to 16384, the DSP resources consumption increases several times compared the input size under 10000. And when input size larger than 20,000, the DSP consumption become increase slowly.

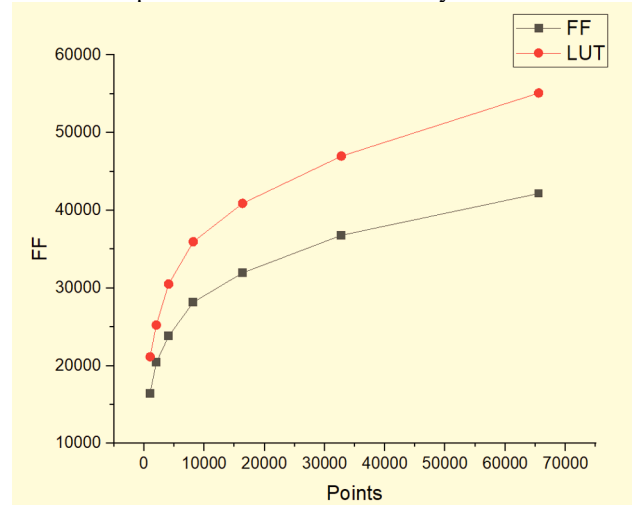


Fig. 6. The relationship between number of FF and number of points

For the FF and LUT, they have the same increasing rate for the input size and as the line shows, the consumption of the two parts increases soon when input size less than 10,000, but the trend become slow when the input size larger than 30,000. The reason is because when input array size become large enough, the FF and LUT can be reused for one cycle.

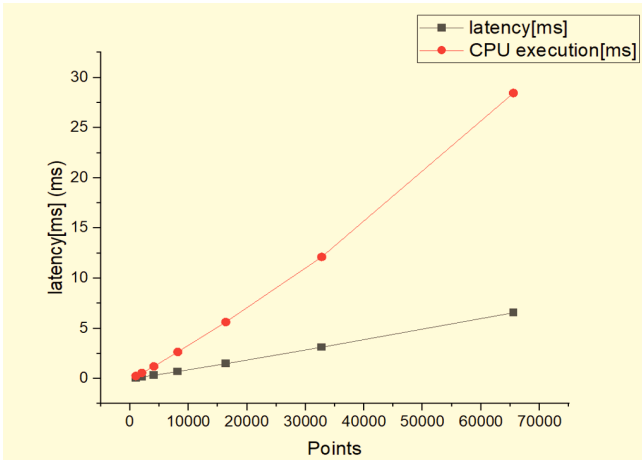


Fig. 7. The relationship between number of latency and number of points

For the latency and the CPU execution time, it can be seen it increases both linearly due to the fact the array input size become doubled, but the latency increase rate is lower than the CPU execution time, this is mainly because CPU needs to manage the input data one by one.

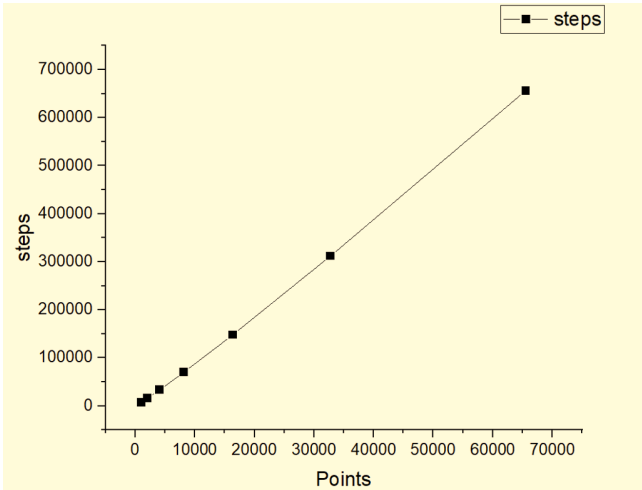


Fig. 8. The relationship between the number of steps and number of points

For the steps, it can also be seen that this is increase linearly as the input size doubled.

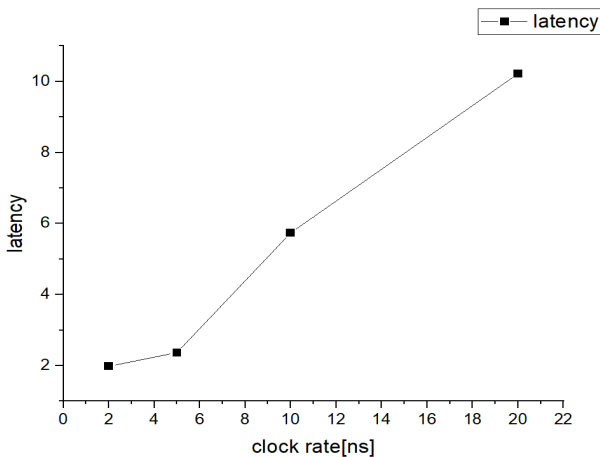


Fig. 9. The relationship between number of latency and clock rate

And for the latency and the clock rate, when we increase the clock rate for the FPGA, it also shows increases linearly for the latency.

So when combine the automorphism and NTT part, the result is showing below:

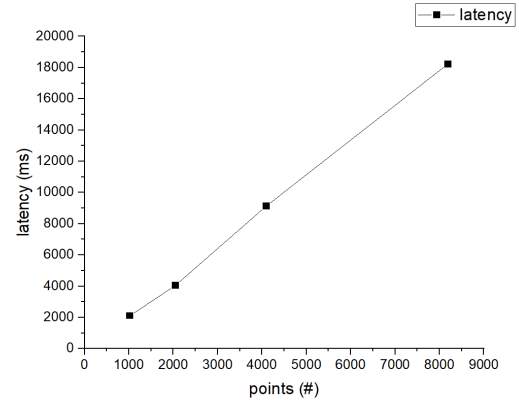


Fig. 10. The relationship between overall latency and number of points

As we can see, the latency is increasing as the # of inputs become larger.

VI. CONCLUSION

This article analyze the Fully Homomorphic Encryption algorithm based on two basic algorithm automorphism and number theoretic transformation. It utilize the Xilinx HLS pragma as a tool to analyze the algorithm and pipeline for the algorithm and verify the hypothesis based on the algorithm analysis. The experiment result shows it can be fully pipelined as interval equals 2. And when the input size become large, the electronic parts and units will also increase linearly. But when the input size is vary large, the electronic units will not increase so much when input size large than 60,000. It did some experiments on the FPGA units, fully work should also consider the array partition optimization as well as unroll factor and generate more data for future analysis. Experimental results show that the optimized algorithm improves the computational efficiency of the butterfly operation of number theory transformation multiplication.

ACKNOWLEDGMENT

The completion of this project is inseparable from Yang's help. He gives us lot of guidance and help when we get stuck. This is a challenging project; we do lots of research and struggle for the implementation. The partition work for this project is list as follow:

- Zhengrong Li:
 1. Write the most part of the proposal
 2. Finish the Yellow part of the PowerPoint to the presentation.
 3. Complete the implementation and parallelization of the Automorphism, experiment this algorithm and analysis the result.
 4. Finish the Section I; II; first three paragraph of III; part A, B, C of the IV and the automorphism part of the V
- Zhaorui Ni:

1. Discuss the baseline of the proposal with my partner and write the parallel programming part of the proposal
2. Write the baseline of the presentation before my partner makes presentation slides and discuss the idea with him, and give opinion and modification after my partner made slides, and finish the number of theory transformation parts of the slides.
3. Write the sequential code of number theory transformation and make sure the result is correct before write the parallel version.
4. Write the HLS pragma version of number theory transformation and did the experiments of the number theory transformation on USC carc.
5. Finish the abstract, the NTT and HE related work of part III, and hypothesis of number theory transformation(part IV) as well as the result analysis (part V)of number theory part and combine two parts results and made analysis of overall results(part V) and the conclusion of this paper(part VI).

REFERENCES

- [1] B. Bhattacharjee, S. Boag, C. Doshi, P. Dube, B. Herta, V. Ishakian, K. Jayaram, R. Khalaf, A. Krishna, Y. B. Li et al., "Ibm deep learning service," IBM Journal of Research and Development, vol. 61, no. 4/5, pp. 10–1, 2017.
- [2] Tharam Dillon, Chen Wu, and Elizabeth Chang. 2010. Cloud computing: issues and challenges. In 2010 24th IEEE international conference on advanced information networking and applications. Ieee, 27–33.
- [3] Jay Heiser and Mark Nicolett. 2008. Assessing the security risks of cloud computing. Gartner report 27 (2008), 29–52.
- [4] SEAL. 2019. Microsoft SEAL (release 3.3). <https://github.com/Microsoft/SEAL>. Microsoft Research, Redmond, WA.
- [5] X. Hu, J. Tian and Z. Wang, "Fast Permutation Architecture on Encrypted Data for Secure Neural Network Inference," 2020 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), 2020, pp. 141–144, doi: 10.1109/APCCAS50809.2020.9301698.
- [6] Ishimaki, Yu, and Hayato Yamana. "Faster Homomorphic Trace-Type Function Evaluation." IEEE Access, vol. 9, 2021, pp. 53061–53077., <https://doi.org/10.1109/access.2021.3071264>.
- [7] Will, Mark A., and Ryan K.L. Ko. "A Guide to Homomorphic Encryption." The Cloud Security Ecosystem, 2015, pp. 101–127., <https://doi.org/10.1016/b978-0-12-801595-7.00005-7>.
- [8] Reagen, Brandon, et al. "Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference." 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021, <https://doi.org/10.1109/hpca51647.2021.00013>.
- [9] Ye, Tian, et al. "FPGA Acceleration of Number Theoretic Transform." Lecture Notes in Computer Science, 2021, pp. 98–117., https://doi.org/10.1007/978-3-030-78713-4_6.
- [10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," ACM Transactions on Computation Theory (TOCT), vol. 6, no. 3, p. 13, 2014.
- [11] A Reconfigurable Fabric for Accelerating Large-Scale ... https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Catapult_ISCA_2014.pdf.
- [12] Nane, Razvan, et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, 2016, pp. 1591–1604., <https://doi.org/10.1109/tcad.2015.2513673>.
- [13] "An Algorithm for the Machine Calculation of Complex Fourier Series." Papers on Digital Signal Processing, 1969, <https://doi.org/10.7551/mitpress/5222.003.0014>.
- [14] R. T. Moenck, "Practical fast polynomial multiplication," in Proceedings of the third ACM symposium on Symbolic and algebraic computation, 1976.
- [15] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," Foundations of secure computation, 1978.
- [16] C. Gentry, "Fully homomorphic encryption using ideal lattices," in Proceedings of the 41st ACM Symposium on Theory of Computing (STOC 2009), pp. 169–178, 2009
- [17] A. Badawi, B. Veeravalli, C. Mun, and K. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018.
- [18] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," IACR Cryptology ePrint Archive, p. 2012:144, 2012.
- [19] P. Xie, B. Wu, and G. Sun, "Bayhenn: combining bayesian deep learning and homomorphic encryption for secure dnn inference," arXiv preprint arXiv:1906.00639, 2019.
- [20] H. Chabanne, R. Lescuyer, J. Milgram, C. Morel, and E. Prouff, "Recognition over encrypted faces," International Conference on Mobile, Secure, and Programmable Networking, pp. 174–191, 2018.
- [21] GENTRY C. Fully homomorphic encryption using ideal lattices[C]. The 41st Annual ACM Symposium on Theory of Computing, Bethesda, USA, 2009: 169–178. doi: 10.1145/1536414.1536440.J..
- [22] ABOZAIID G, EL-MAHDY A, and WADA Y. A Scalable multiplier for arbitrary large numbers supporting homomorphic encryption[C]. 2013 Euromicro Conference on Digital System Design, Los Alamitos, USA, 2013: 969–975. doi: 10.1109/DSD.2013.110.
- [23] RAFFERTY C, O'NEILL M, and HANLEY N. Evaluation of large integer multiplication methods on hardware[J]. IEEE Transactions on Computers, 2017, 66(8): 1369–1382. doi: 10.1109/TC.2017.2677426.
- [24] VAN METER R and ITOH K M. Fast quantum modular exponentiation[J]. Physical Review A, 2005, 71(5): 052320. doi: 10.1103/PhysRevA.71.052320.
- [25] GARCÍA L. Can Schönhage multiplication speed up the RSA encryption or decryption?[EB/OL]..
- [26] FÜRER M. Faster integer multiplication[C]. The 39th Annual ACM Symposium on Theory of Computing, San Diego, USA, 2007: 57–66. doi: 10.1145/1250790.1250800.
- [27] WANG Wei, HUANG Xinming, EMMART N, et al. VLSI design of a large-number multiplier for fully homomorphic encryption[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2014, 22(9): 1879–1887. doi:10.1109/TVLSI.2013.2281786.
- [28] FENG Xiang and LI Shuguo. Design of an area-efficient million-bit integer multiplier using double modulus NTT[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2017, 25(9): 2658–2662. doi: 10.1109/TVLSI.2017.2691727.
- [29] Mert, Ahmet & Karabulut, Emre & Ozturk, Erdinc & Savas, Erkay & Becchi, Michela & Aysu, Aydin. (2020). A Flexible and Scalable NTT Hardware : Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography. 346-351. 10.23919/DAT48585.2020.9116470.