Improved Bootstrapping by FFT on Encrypted Multi Operands Homomorphic Addition

Paulin Boale Bomolo, Simon Ntumba Badibanga, Eugene Mbuyi Mukendi

Abstract: ***Bootstrapping is a technique that was introduced by Gentry in 2009. It is based on reencryption which allows an encryption scheme to perform an unlimited number of processing on encrypted data. It is a bottleneck in the practicability of these schemes because of multiplication operations which are costly in complexity. This complexity was reduced in TFHE by processing bootstrapping on the result of a two-bit logic gate in thirteen milliseconds using the Fast Fourier Transform. Building on this advance, an implementation of the addition of ten (10) numbers of 32-bits was performed based on the 32-bit Carry Look ahead Adder and was executed in less than 35 seconds using the configured SPQLIOS Fast Fourier transform to manipulate AVX and FMA instructions. This connector improves performance to a higher level than FFTW3 and NAYUKI.***

Keywords: Fast Fourier Transform, libraries, homomorphic encryption, binary adder.

I. INTRODUCTION

Homomorphic encryption is an encryption which performs various processing on encrypted data. To do this, it is based on the bootstrapping technique introduced by Gentry which consists in refreshing the noise in the encrypted data by a re-encrypting operation [10,11]. In bootstrapping, the multiplication operation is a bottleneck for better performance. Since the advent of the encryption schemes of Ducas and Micciancio and Ilaria chillota and all [9, 28, 29,30],a glimmer of hope hasemerged for large-scale and industrial homomorphic encryption. They perform bootstrapping in less than a second and thirteen milliseconds respectively. Both schemes reduce bootstrapping processing time using implementations of the Discrete Fourier Transform called the Fast Fourier Transform that processes multiplication through logarithmic complexity [32,33]. Apart from the introduction and conclusion, this paper is divided into three sections which are the Fast Fourier Transform, the Torus Fully Homomorphic Encryption scheme and the implementation and interpretation of the results.

Manuscript received on October 02, 2021.
Revised Manuscript received on October 07, 2021.
Manuscript published on October 30, 2021.

*Correspondence Author

Paulin Boale B.*, Mathematics and Computer Sciences, University of Kinshasa, Kinshasa, Democratic Republic of Congo, Email: Paulin.boale@unikin.ac.cd

Simon Ntumba B., Mathematics and Computer Sciences, University of Kinshasa, Kinshasa, Democratic Republic of Congo, Email: eugene.mbuyi@unikin.ac.cd

Eugene Mbuyi M., Mathematics and Computer Sciences, University of Kinshasa, Kinshasa, Democratic Republic of Congo, Email: simon.ntumba@unikin.ac.cd

II. THE FAST FOURIER TRANSFORM [32,33]

The Fast Fourier Transform is the most widely used algorithm currently in the analysis and processing of digital data in several scientific fields including cryptography. It remarkably reduces the multiplication time from $(n \log n)$ to $O(n^2)$. It is based on the representation by values, the properties of the$n^{ièmes}$ roots of unity and the "divide and conqueror" strategy. It is an implementation of the Discrete Fourier Transform that adapts to the hardware for more performance.

A. Polynomials.

a. Definitions.

A polynomial of indeterminate x, defined on a commutative ring A, is the formal sum $p(x) = \sum_{i=0}^{n-1} a_i x_i = a_0 + a_1 x + \cdots \ldots \ldots \ldots + a_{n-1} x^{n-1}$ where $a_0, a_1, \ldots \ldots \ldots, a_{n-1}$ are called the coefficients of the polynomial that belongs to the ring A and often to the field of complexes C. $p(x)$ is said to be of degree k, if k is the largest integer such that $a_k$is not zero.

b. Operations on polynomials.

Let $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ let two polynomials of n terms.
Addition of polynomials: the sum of $A(x)$ and $B(x)$ is polynomial $c(x)$ of n terms also, such that :$C(x) = \sum_{i=0}^{n-1} c_i$ where $c_i = a_i + b_i$.
Multiplication of polynomials: the multiplication of $A(x)$ and $B(x)$is the product $C(x)\ of\ 2n-1$ terms such that: $C(x) = \sum_{i=0}^{2n-1} c_i$ where $c_i = \sum_{k=0}^{i} a_k b_{i-k}$. The vector of the resulting coefficients $c$is called the convolution of the input vectors a and b, and denoted $c = a \otimes b$.

c. Representations of polynomials.

A polynomial is represented in two equivalent ways that are by coefficients or values.

i. Representation by coefficients.

The coefficient representation of a polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$is the vector of the coefficients $a = (a_0, a_1, \ldots, a_{n-1})$. The evaluation of $A(x)$ at point x_0 performs this representation in $O(n)$ with the Horner method. As for the *multiplication* of two polynomials $A(x)$ and $B(x)$ takes a time of $O(n^2)$.

ii. Representation by values.

The value representation of a polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$ is a set of n points in the plane, represented by their coordinates $\{(x_0, y_0), (x_1, y_1), \ldots., (x_{n-1}, y_{n-1})\}$ such that the x_k are all distinct and $y_k = A(x_k)$, for $k = 0, 1, \ldots, n-1$. This evaluation takes $O(n^2)$by Horner's method.

iii. Interpolation.

The interpolation of a polynomial is the inverse process of evaluation. It consists in determining the coefficients of a polynomial from its representation by values.
Theorem: the uniqueness of an interpolation polynomial. A distinct set x_i of n coordinates $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ has one and only one polynomial $A(x)\ of\ n$ terms such that $y_k = A(x_k)$for $k = 0, 1, \dots, n-1$. The fastest algorithm for interpolation on n points is based on Lagrange's formula: $A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x-x_j)}{\prod_{j\neq k}(x_k-x_j)}$. It calculates the coefficients of $A\ in\ O(n^2)$.

B. Fast Fourier Transform (FFT) [33].

The integral Fourier transform $f(x)$ is the function $F(\omega)$ such that:

$$F(t) = \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt$$

And the inverse transform is given by the equation:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega)\, e^{-i\omega t} d\omega$$

Where $i^2 = -1$ and $e^{i\omega} = \sin\omega + \cos\omega$. $f(t)$ is considered to be a signal $F(t)$ and is called the signal spectrum.

a. Discrete Fourier Transform (DFT).

The Discrete Fourier Transform of a vector of n components $a = (a_0, a_1, \dots\dots\dots\dots\dots\dots\dots, a_n)$ with respect to a constant ω, denoted DFT(a), is a new vector $A = (A_0, A_1, \dots\dots\dots\dots\dots\dots\dots, A_n)$ such that: $A_j = \sum_{k=0}^{n-1} a_k \omega^{jk}, 0 \le j \le n-1$ where $\omega = e^{i\frac{2\pi}{n}}$, $n^{\text{ième}}$ called the primitive (or principal) root of unit, and the $\omega^k for\ k = 0, \dots, n-1$ are called the $n^{\text{ième}}$roots of unity.

b. Polynomial interpretation of DFT.

Given a vector a representing by coefficients of a polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$, the calculation of the DFT is equivalent to evaluating $p(x)$at the points $x_i = \omega^i \quad 0 \le k \le n-1$, more precisely the vector $A = DFT(a) = \{p(1), p(\omega), p(\omega^2), \dots, p(\omega^{n-1})\}$.

c. Reverse DFT.

Given n components of vector A such as $A = DFT(a)$ the original vector a can be retrieved by the reverse DFT. The inverse DFT of n components vector of A with respect to a constant ω, denoted DFT^{-1}, is a vector such that $a = (a_0, a_1, \dots, a_{n-1})$.
Thus

$$DFT^{-1}\big(DFT(a)\big) = a$$

Since ω is $n^{\text{ième}}$ root of unity, so ω^{-1} is it.
Convolution theorem 1. Let a and b two vectors of n components where n is a power of 2. The convolution of a and b is reverse DFT of the product $a \otimes b = DFT^{-1}{}_{2n}\big(DFT_{2n}(a).DFT_{2n}(b)\big)$ of their Fourier transform where $\otimes$ is the product term terms of two vectors while is their convolution.

C. Fast Fourier Transform.

It was in 1965 that JAMES COOLEY and JOHN TUKEY published this method. It was later discovered that the algorithm had already been invented by CARL FRIEDRICH GAUSS in 1805 and designed several times in different forms.
The Fast Fourier Transform is an algorithm that significantly reduces the number of operations to calculate the FFT. It is based on the divide-and-conqueror strategy and takes advantage of the particular properties of $n^{\text{ième}}$roots of unity.

a. nième roots of unity.

Definition: A constant ω is said to be the $n^{\text{ième}}$ root of the unity if $\omega^n = 1$. In addition, ω is said to be $n^{\text{ième}}$ root principal (or primitive) of the unity ω if the following additional conditions are met:
(1) $\omega^k \neq 1\ pour\ 0 < k < n$
(2) $\sum_{j=1}^{n-1} \omega^{jk} = 0\ pour\ 0 < k < n$

Theorem 2. Let be any commutative ring. Let $n > 1$ be a power integer of two and let ω be an element in this ring such that $\omega^{\frac{n}{2}} = -1$. Then

(1) $\omega$is a $n^{\text{ième}}$root principal of unity;
(2) ω^{-1} is the multiplicative inverse ω of in the ring;
(3) $\omega^{-1}$is also a $n^{\text{ième}}$root principal of unity;
(4) $1 = \omega^0, \omega^1, \dots, \omega^{n-1}$, called the $n^{\text{ième}}$root principal roots of unity are all distinct;
(5) If there is a multiplicative n^{-1} inverse of n in the ring, then ω^{-1} is a consequence of the fact that ω is a $n^{\text{ième}}$root principal of the unity.

Periodicity property. For all $n \geq 0$ et $k \geq 0$, $\omega^{n+k} = \omega^k$.
Symmetry property. For any even integer n, $\omega^{\frac{n}{2}} = -1$.
Bipartition property. For any n, the squares of n $n^{\text{ième}}$ roots of unity are $\frac{n}{2}$ $\left(\frac{n}{2}\right)^{\text{ième}}$ roots of unity.

b. FFT algorithm.

The best-known algorithm that requires a number of samples that is a power of 2 is that of Cooley-Tukey. It is based on the successive decomposition of size n input vector, into two other identical smaller size corresponding to the divide-and-conqueror strategy. It is combined with the properties of the nth roots for more efficiency.
Theorem 3: FFT where the input A has a size $N = 2^m$,for $m \geq 0$ be calculated in $O(N \log N)$ with the Cooley-Tukey recursive algorithm.

```
Algorithm 1: FFT(A)
Input: a vector $A$ of size $2^m$complexes where $m \geq 0$
Output: a vector of complexes which is the FFT of the input $A$.
$N = size\ of\ A$
If $n == 1$ then return $A$
Else
    $\omega_N = e^{2\pi i/N}$
$\omega = 1$
```

```
$A_{pair} = (A_0, A_2, A_4, \dots\dots\dots.., A_{N-2})$
$A_{impair} = (A_1, A_3, A_3, \dots\dots\dots.., A_{N-1})$
$Y_{pair} = FFT(A_{pair})$
$A_{impair} = FFT(A_{impair})$
$for\ j\ = 0\ to\ {}^{N}/_{2} - 1$
        $do$
$Y[j] = Y_{pair}[j] + \omega * Y_{impair}[j]$
$Y[j + {}^{N}/_{2}] = Y_{pair}[j] - \omega * Y_{impair}[j]$
        $\omega = \omega * \omega_N$
        $return\ Y$
```

The above algorithm splits the input into two parts of ${}^{N}/_{2}$. The splitting operation and updating the result takes in terms of complexity $O(N)$. From what preceded the following recurrence relation is derived $T(N) = 2T({}^{N}/_{2}) + O(N)$ showing that the total execution time is $O(N\ \log N)$.

c. Libraries of the Fast Fourier Transform: Preprocessors.

i. Single Instruction on Multiple Data [37].

Single Instruction on Multiple Data (SIMD) is one of the four categories of architecture defined by Flynn's taxonomy in 1966 and refers to a mode of operating of computers with parallelism processing. In this mode, the same statement is applied simultaneously to multiple data.
On January 8, 1997,Intel released the first microprocessor with MMX technology, the Pentium MMX at 166 MHz (P166MX) which is the first time that a SIMD is added to a Complex Instruction Set Computer (CISC) technology processor. Later in 1997, AMD also launched an Multiple Math eXtension (MMX)-compatible X86 processor (licensed to Intel) including an additional set instructions of SIMD, the Intel 3DNow would add a new set instructions of SIMD in 1999 with SSE technology, incompatible with 3DNow.

The hardware implementation of the SIMD paradigm can be done in various ways:

- through the use of SIMD instructions, usually in micro-code interpreted on CISC or wired on RISC;
- by vector processors;
- by stream processors;
- or through systems with multi-core or multiple processors.

In the first three cases, a single processor can perform same operation on multiple data. In the latter case, each processor will perform a single operation on a data. SIMD parallelism therefore comes from the use of several processors. All modern processors contain extensions to their instruction set, such as MMX, SSE, etc. These extensions have been added to modern processors to be able to improve processing speed on calculations. SIMD instructions are composed in particular of instruction sets: On x86 processor: MMX, 3DNow, SSE, SSE2, SSE3, SSSE3, SSE4,SSE4.1, SSE4.2, AVX,AVX2 and AVX512.
Streaming SIMD Extensions, usually abbreviated SSE, is a set of 70 additional instructions for x86 microprocessors, which appeared in 1999 on the Pentium III in response to AMD's 3DNow that appeared one year earlier. Instructions is SIMD type. Streaming SIMD Extension 2, usually abbreviated SSE2, is composed of 144 instructions and made its appearance with Intel's Pentium 4. It manages 128-bit registers for integers as well as single and double precision floats. SSE3, also known by its internal codename Prescott New Instructions (PNI), is the third generation of the SSE instruction set for the IA-32 architecture. Intel introduced SSE3 in early 2004 with the Prescott version of its Pentium 4 processor. In April 2005, AMD introduced a subset of SSE3 in revision E of their Athlon 64 processor (Venice and San Diego). Their SIMD instruction set for the x86 platform, from the oldest to the newest, are MMX, 3DNow, SSE, and SSE2.
Advanced Vector Extensions (AVX) is an instruction set of the x86 architecture from Intel and AMD, released by Intel in March 2008. It is supported by Intel Sandy Bridge processors and AMD Bulldozer processors in 2011. AVX offers new features, new instructions and a new "VEX" coding scheme. **AVX2** expands most 128-bit SSE and AVX commands to 256-bit. **AVX-512** expands the number of SIMD registers to 32 and expands them to 512 bits. It uses new coding using the EVEX *prefix* proposed by Intel in July 2013. The first processors supporting it were the Knights Landing.
Fuse Multiply Add (FMA) is an extension of the SSE Streaming SIMD Extensions instructions from 128 to 256 bits in the x86 microprocessor instruction set to perform FMA operations. These instructions perform addition and multiplication operations in a single instruction in a clock cycle. There are two variants which are respectively the FMA4 which is supported by the AMD processor with the Bulldozer architecture and the FMA3 which is supported in the AMD processors of the PileDriver architecture and Intel with Haswell and Broadwell processors since 2014.

ii. FFTW3[35].

Fftw3 is a library that was developed at MIT by Matteo Frigo and Steven G. Johnson. This GPL- open-source licensed library implements the Fast Fourier Transform at one or more dimensions, of arbitrary size and for real as well as complex. The latest free version for download on http://www.fftw.org is 3.3.10. Version 3.3 introduced AVX support, an implementation with MPI and an interface for Fortran 2003.

iii. Nayuki Project[36].

The default processor comes from Project Nayuki, which offers two implementations of the Fast Fourier Transform – one in portable C and the other using AVX assembly instructions. This component is licensed under the MIT license, and we have added the code of the reverse FFT (both in C and assembly). Original source : https://www.nayuki.io/page/fast-fourier-transform-in-x86-assembly.

iv. SPQLIOS [28].

The last processor named the SPQLIOS is provided by [28]. It which is written in AVX and FMA assembly in the style of the NAYUKI processor, and which is dedicated to the ring for a power of 2.

v. FFT processors.

To run the TFHE needs at least one of the processors listed in the table below:

Table 1: FFT processors

Name	**License**	**Language and portability**	**Performance**	**Website**
NAYUKI	MIT	C and AVX	1	www.nayuki.io
SPQLIOS	Apache 2	AVX and FMA	2 - 3	
FFTW3	GPL	C and FORTRAN	1	www.fftw.org

In terms of performance, the SPQLIOS processor performs better than the other two. It reduces their execution times by a factor of 2 or 3.

III. HOMOMORPHIC ENCRYPTION: TFHE.

The homomorphic encryption scheme is an encryption scheme that manipulates encrypted data to produce a corresponding result on plaintext messages. It is divided into three categories that are partial, little and complete. The partial homomorphic encryption scheme supports a single operation such as addition [38] or multiplication [39] on the encrypted data. They are not very useful in dealing with arbitrary operations on encrypted data. The somewhat homomorphic encryption scheme simultaneously supports addition and multiplication operations of the encrypted data but the number of operations is bounded once the threshold is reached and decryption fails [8,13] The fully homomorphic encryption scheme simultaneously supports unlimited addition and multiplication operations on the encrypted data. This property allows you to process any function on encrypted data [12,9,27].

The FHE uses a Learning With Errors problem (LWE) [18] where an error called noise is introduced to ensure safety (18). This error increases after each operation especially multiplication and causes invalid decryption after a number of operations. Therefore, this error must be reduced to a threshold to support an unlimited number of operations.

The refresh procedure is called bootstrapping [10,11]. It brings an additional number of operations that makes the algorithms inefficient [8]. Improvements have been made to this procedure to make it effective. Processing of a logical function on two encrypted messages corresponding to the free bits is performed in less than a second [9] and in 13 milliseconds [27] respectively.

A. TFHE [28,29,30].

Torus Fully Homomorphic Encryption (TFHE) is a homomorphic encryption scheme designed by Illaria Chillota and all [28,29,30] that work on the torus $T = {}^{\mathbb{R}}/_{\mathbb{Z}}$, a set of real numbers modulo 1, as space for plaintext and ciphertext. Ciphertexts are built under the LWE problem [19,22] and represented as the Torus LWE where an error or noise is added to each encrypted message.

For a given dimension $m \geq 1$, the secret key $\vec{s}$ is drawn in $\mathcal{B}^m \; with \; \mathcal{B} = \{0,1\}$, and the e error is drawn in a distribution χ, a sample is defined as $(\vec{a}, b)$ where $\vec{a} \in$ T such that $\vec{a}$ is a vector of coefficients drawn from size m and each element a_i is drawn in a uniform distribution under T and The $b = \vec{a} * \vec{s} + e$. The term e in the sample increases with the number of operations. Therefore, bootstrapping is introduced to decrypt and re-encrypt the encrypted message to remove unnecessary noise.

The space of plaintext in the TFHE is the bit represented by the set $\mathcal{B} = \{0,1\}$ and generates encrypted messages of the LWE type under the torus. Thus, processing on plaintext messages is comparable to processing on the bit. A binary vector represents a number in $\mathbb{N}$ or $\mathbb{R}$, the same is true of n sample LWE of n size represents a numerical number of n size. In TFHE, the Boolean gates of the addition circuit between two numbers correspond to the operations on LWE samples. The motivation for choosing the TFHE is summarized through the speed of its bootstrapping in 13 milliseconds, the acceptable size of the encrypted message and the multitude of Boolean operations supported.

The implementation of TFHE comes with the standard cryptography functions (key generation, encryption, decryption) and also all the logic gates for performing logical operations in $\mathcal{B}$. And its bootstrapping is improved by using of the three FFT processors which are the FFTW3, the NAYUKI project and the SPQLIOS.

B. *Homomorphic arithmetic addition with TFHE [34,31].*

The one-bit addition and multiplication operations are defined in using the XOR and AND logic gates respectively. These gates are the foundation for the implementation of increasingly complex circuits.

This section presents an implementation of arithmetic addition by composing the complete binary adder with the AND and XOR gates. The adder circuit made it possible to build the Carry Lookhead Adder (CLA)[31,34]. This addition arithmetic operation will be performed on integers with a size of 32 bits using one 32-bit block, two 16-bit blocks, four 8-bit blocks, and eight 4-bit blocks, respectively.

a. Adder.

The adder is a circuit that is made from two basic circuits which are the half-adder and the complete adder. These are used to make the CLAx adders where x is an integer representing the size of the block.

i. Half adder.

The half-adder is a circuit that allows the calculation of the sum s_i and the carry c_i when adding two bits a_i and b_i, the ith bit of a binary representation of a and b two integers of 32 bits.

$s_i = a_i \oplus b_i \; et \; c_i = a_i * b_i$ where $\oplus$, $*$ represents respectively the addition on a bit: XOR and the multiplication on a bit: AND.

ii. Full adder.

A full adder is a circuit that allows the calculation of the ith sum s_i and the (i+1) th carry c_{i+1} when adding two bits a_i , b_i and c_i an input carry.

It includes half-adders and full adders. The difference is that a half-adder does not accept a carry while the adder accepts it.
The implementation can vary as long as the logical expressions of different implementations are equivalent. In [25], for example, the expressions of sum and carry can be written as follows:

$$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i)$$
$$s_i = a_i \oplus b_i \oplus c_i$$

The expression of withholding can be reduced as follows:

$$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i) = (a_i \oplus c_i) . (b_i \oplus c_i) \oplus c_i$$

This optimized expression is found in [26]. It uses only for each bit an AND gate, and therefore a complete adder of one bit at a multiplicative depth that is equivalent to $1 (L = 1)$.

b. Carry Lookahead Adder.

In a ripple carry architecture, the addition depends on the propagation of carries through stages of the parallel adder [34]. To reduce the propagation time and speed up the addition process, it is possible to anticipate the output carry of each stage and to produce, from the inputs, the carry by generation or propagation. This technique is called "carry anticipation".
carry generation occurs when a carry is generated by the full adder. A carry can only occur when the two input bits are 1. The generated carry is noted g_i and is equal to $g_i = a_i * b_i$. A carry propagation is created when an input carry is passed on to output carry. In a full adder, the propagation of an input carry can occur when at least one of the bits is 1. The propagated carry noted p_i and is equivalent to $p_i = a_i \oplus b_i$. The output carry of full adder can be expressed as a propagated carry p_i or as a generated carry g_i. The noted output carry c_i is 1 if the generated output is 1 or if the propagated output is 1 and the input carry (c_{i-1}) is 1.
In other words, an output carry of 1 is generated by the full adder if $a_i = 1 \; et \; b_i = 1$ or by propagation of the additionor of the input carry $(a_i = 1 \; ou \; b_i = 1) \; et \; (c_{i-1} = 1)$. The expression below summarizes all the cases: $c_i = g_i + p_i c_{i-1}$.
Let's illustrate this concept by applying to a thirty-two (32) bit parallel adder with k-bits blocks. The stage i produces an output carry either by generating it or by propagating the internal carry to the output carry. For each stage i, it generates g_i and propagates p_i as follows:

- The column i produces an output carry if the inputs a_i and b_i are equal to 1 binary: $g_i = a_i * b_i$;
- The column i propagates the internal carry to the output carry if one of the inputs is equal to1: $p_i = a_i \oplus b_i$;
- The output carry of column i is given by the following expression:
  $$c_i = a_i * b_i \oplus (a_i \oplus b_i) * c_{i-1} = g_i \oplus p_i \, c_{i-1}.$$

The algorithm of the carry anticipation adder can be described in the steps below:

- Step 1: calculate g_i and p_i for all columns for i from 0 to 31;
- Step 2: calculate the g_k and p_k for each k −block of bits for k = 4, 8, 16;
- Step 3: The input carry c_0 propagates through the bit k −block by the functions of generating and propagating the carry.

Example for a block of 4 bits ($p_{3:0}$ and $g_{3:0}$):

$$g_{3:0} = g_3 + p_3(p_2 + p_2(g_1 + p_1 g_0)$$
$$p_{3:0} = p_3 p_2 p_1 p_0$$

In general,

$$g_{i:j} = g_i + p_i(g_{i-1} + p_{i-1}(g_{i-2} + p_{i-2} g_j))$$
$$p_{i:j} = p_i p_{i-1} p_{i-2} p_j$$
$$c_i = g_{i:j} + p_{i:j} \, c_{j-1}$$

The complexity of the algorithm of the added or with Carry Lookahead Adder respectively in time is $O(n \log n)$. The CLA of n is faster than the Ripple Carry Adder which has a complexity $O(n)$ in time and space of respectively.

IV. IMPLEMENTATION AND INTERPRETATION OF RESULTS

The implementations were tested on an **Intel® Core™ i7-5500 CPU @ 2.40 Ghz** laptop that has a 4096 kilobytes cache, 1100 Mhz frequency clock and **8 Gigabyte RAM.** It supports the following features: MMX, SSE, SSE2, FMA, SSE4_1, SSE4_2, AVX AND AVX2.

A. Default settings.

The default settings have been configured. They provide a security setting of at least 110 bits that are based on the difficult assumptions of the ideal lattices problem. The logical homomorphic AND performs on two bits with a secret key of 109 Megabytes and a bootstrapping key of the same size provided the results for each FFT type in Table 2. The FFT SPQLIOS configured with the AVX or FMA flag provides better performance than other FFTs.

Table 2: Execution time of a two-bit AND operation.

Duration(s)	FFTW3	NAYUKI-PORTABE	NAYUKI-AVX	SPQLIOS-AVX	SPQLIOS-FMA
AND	0.3	0.2	0.1	0.024	0.03

B. Performance and interpretation of results.

Table 3, the column represents the type of FFT used during the experiment and the row when it the CLAx where x represents the block size. The intersection between the row and the column represents the duration of the execution of an addition operation on ten numbers of 32-bits respectively.

Table 3: Performance of CLAx adders.

Duration(s)	FFTW3	NAYUKI-PORTABE	NAYUKI-AVX	SPQLIOS-AVX	SPQLIOS-FMA
CLA32	443	288.2	142.5	33.5	32.2
CLA16	440	284.6	141.6	32.1	31.8
CLA8	**440**	**284.3**	**141.2**	**31.9**	**31.6**
CLA4	438	282.6	141.0	31.6	31.3

Ten 32-bit numbers occupy a space of 320 bits on disk or about 40 Bytes. These numbers are encrypted in a file with a size of 795 Kilo-Bytes.

The expansion between the encrypted message and the plaintext message is very high by19 875. From the table 2, the FFT SPQLIOS reduces the execution time of the FFT NAYUKI-AVX by 343%. While the FFT FFTW3 provides poor performance even compared to the default FFT which is the FFT NAYUKI-PORTABLE, it increases its execution time by 35%. The size of the chosen block in the operation improves performance in a very small proportion. It is of the order on average of less than 1% for all FFTs used.

V. CONCLUSION

TFHE performs a homomorphic AND logical operation on two bits in thirteen milliseconds using bootstrapping by taking advantage of the implementation of the Discrete Fourier Transform called the Fast Fourier Transform.

The different implementations of the Fast Fourier Transform have made it possible to build a homomorphic addition circuit of 10 numbers of 32 bits with 9 carry lookahead adders that gives a result in 35 seconds with SPQLIOS configured with AVX or FMA instructions. In our configuration, the FFTW3 did not yield results to corroborate the assumptions against NAYUKI.

To further improve, other Fast Fourier Transform implementations such as IPP and OTFFT must be configured to evaluate their performance with TFHE with the same circuit or others. Another way to be explored is shared memory or distributed memory parallelism coupled with native MPI or OpenMP instructions from different Fast Fourier Transform implementations.

REFERENCES

1. [Bra12] Brakerski Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini R., Canetti R., (eds) CRYPTO 2012, LNCS, vol. 7417, pp. 868-886. Springer, Berlin (2012);
2. [BGV12] Brakerski Z., Gentry C., Vaikuntanathan V., (Leveled) fully Homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed) ITCS 2012, pp. 309–325. ACM, New York (2012);
3. [BV11] Brakerski Z., Vaikuntanathan V., Fully encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed) CRYPTO 2011, CNSL, vol. 6841, pp. 505–524. Springer, Berlin (2011);
4. [AoM15] Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: Proceedings of 2015 IEEE Conference on Communications and Network Security, pp. 628–632. IEEE, Piscataway, NJ (2015);
5. [CJWY15] Chen X., Jingwei C., Wenyuan W., Yong F.: Homomorphically Encrypted Arithmetic Operations over Integer Ring.;
6. [CCKLTY13] Cheon J.H., Coron J.S., Kim J., Lee M.S., Lepoint T., Tibouchi M., Yun A.: Batch fully Homomorphic encryption over integers. In: Johanson, T. Nguyen, P.Q. (eds) EUROCRYPT 2013, LNCS, vol 7881, pp. 315–335. Springer, Berlin 2013;
7. [CMNT11] Coron J.S., Mandal A., Naccache D, Tibouchi M., : Fully homomorphic encryption over the integers with shorter public keys. In: Rogaway, P. (ed) CRYPTO 2011, LNCS, vol 6841, pp. 487–504. Springer, Berlin 2011;
8. [DGHV10] Van Dijk M., Gentry C., Halevi S., Vaikuntanathan V.: Fully Homomorphic encryption over the integers. In: Gilbert, H. (ed) EUROCRYPT 2010, LNCS, vol 6110, pp. 24–43, Springer, Berlin (2010);
9. [DM15] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: EUROCRYPT 2015, Part I. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. CNSL. Springer, Heidelberg, Apr. 2015, pp. 617–640. doi: 10.1007/978-3-662-46800-5_24;
10. [Gen09a] Gentry C.,: A Fully Homomorphic Encryption Scheme. PhD, thesis, Stanford University, Stanford (2009);
11. [Gen09b] Gentry C.: fully Homomorphic encryption using ideal lattices. In: Mitzenmacher M. (ed) SOC 2009, pp. 169–178. ACM, New York (2009);
12. [GHS12] Gentry C., Halevi S., Smart N.P.: Fully Homomorphic encryption with polylog overhead. In: Pointcheval D., Johansson T. (eds) EUROCRYPT 2012, LNCS, vol. 7237, pp. 465482. Springer, Berlin (2012);
13. [GSW13] Genty C., Sahai A., Waters B.: Homomorphic encryption from learning with errors: Conceptually – simpler, asymptotically-faster, attribute-based. In: Canetti R., Garay J.A., (eds) CRYPT 2013, Part I, LNCS, vol 8042, pp. 75–92, Springer, Berlin (2013);
14. [GSH12] Gentry C., Halevi S., Smart N.P.: Fully Homomorphic Encryption with polylog overhead. In: pointcheval, D., Johanson, T. (eds) EUROCRYPT 2012, LNCS, vol. 7237, pp. 465–482. Springer, Berlin (2012);
15. [HS16] Halevi S., Shoup V., : Helib : An Implementation of Homomorphic encryption https://github.com/shaih/Helib, accessed in June 2016;
16. [KSS09] Kolesnikov, V., Sadeghi, A.R. Scheinder, T.: Improved garbled circuit building blocks and application to auctions and computing minima. In:
17. Garay, J.A., Miyaji, A, Otsuka, A. (eds) CANS 2009, CNSL, vol. 5888, pp. 1–20. Springer Berlin (2009);
18. [RSL10] Lyubashevsky, V., Peiker, C., Regev, O, : On ideal lattices and learning with errors over rings. In: Gilbert, H, (ed) EUROCRYPT 2010, LNCS, vol. 6110, pp1–23. Springer, Berlin (2010);
19. [Reg05] Regev O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N.., Fagin, R. (eds) STOC 2005, pp. 84–93. ACM, New York (2005);
20. [Shoup16] Shoup V., NTL: A library for doing number theory. http://shoup.net/ntl/, accessed in June, 2016;
21. [SV14] Smart N. P., Vercauteren F.,: Fully Homomorphic SIMD operations. Designs, Codes and Cryptography 71(1), 57-81 (2014);
22. [LP10] V. lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and Learning With errors over Rings. JACM, 60(6):43, 2013;
23. [AP14] J. Alperin-Sheriff and C. Peikert, "Faster Bootstrapping with polynomial erros" in Proceedings of the international Cryptology Conferences, pp. 297-314, Springer, Berlin, Germany, 2014;
24. [RAD78] Ron Rivest, Leonard Adleman, and Michael L. Detrouzos. On data banks and privacy homomorphisms. In foundations of secure computations, pp. 169-180, 1978;
25. [YG15] Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: Proceedings of 2015 IEEE Conference on Communications and Network Security, pp. 628–632. IEEE, Piscataway, NJ (2015);
26. [KSS09] Kolesnikov, V., Sadeghi, A.R. Scheinder, T.: Improved garbled circuit building blocks and application to auctions and computing minima. In: Garay, J.A., Miyaji, A, Otsuka, A. (eds) CANS 2009, CNSL, vol. 5888, pp. 1–20. Springer Berlin (2009);
27. [CGGI16a] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast Fully Homomorphic Encryption Library over the Torus. https://github. com /tfhe/tfhe. 2016;
28. [CGGI16b] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: ASIACRYPT 2016, Part I. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. CNSL. Springer, Heidelberg, Dec. 2016, pp. 3–33. doi: 10.1007/978-3-662-53887-6_1;
29. [CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE". In: ASIACRYPT 2017, Part I. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. CNSL. Springer, Heidelberg, Dec. 2017, pp. 377–408;
30. [Reg05] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: 37th ACM STOC. Ed. by Harold N. Gabow and Ronald Fagin. ACM Press, May 2005, pp. 84–93;
31. [ZIMMERMANN] Binary Adder Architectures for Cell-Based VLSI and their Synthesis. PhD Thesis Swiss FederaI lnstitute of Technology Zurich 1998.
32. [YKJH15] Young-so Park, Koo-Rack Park, Jin-Mook Kim, Hwa-Young Jeong Fast Fourier transform benchmark on X86 Xeon system for multimedia data processing, Springer Science+Business Media new York 2015.
33. [Kassem] Kassam Kalach, Implementation de la multiplication de grands nombres par FFT dans le context des algorithms cryptographiques, Université de Montréal, 2005.

34. [BNM21] Paulin Boale Bomolo,Simon Ntumba Badibanga,Eugene Mbuyi Mukendi, Performance of Adder Architectures on Encrypted Integers, JEAT, volume-10- Isuue-6, August 2021.
35. [MS] Matteo frigo, Steven G. Johnson, FFTW user's manual, www.fftw.org, forversion 3.3.10, December 10th, 2010.
36. Project NAYUKI, www.nayuki.io.
37. [Lomont] Chris Lomont, Introduction to Intel Advanced Vector Extensions, https://hpc.llnl.gov,March 2011.

AUTHORS PROFILE

Paulin BOALE B. is senior lecturer and PhD Student at university of Kinshasa in Mathematics and Computers sciences department. his field of research is cryptography, in particular homomorphic cryptography. he works to improve algorithms in everyday applications. he contributed to the publication of articles respectively in the journal IJCSI and IJSR such as « Study of Master-Slave Database replication in distributed database », IJCSI, 2011.

Simon NTUMBA B. is professor and head of Mathematic and computers sciences department of the University of Kinshasa. As publications, Author of many publications, such as: "Enhanced Parallel Skyline on multi-core architecture with lax Memory space Cost", IJCSI, volume 13, Issue 5, September 2016, Data mart approach for stock management model with a calendar under budgetary constraint, IJCSI, volume 15, Issue 5, September 2018, Poster et the 2nd International conference on Big Data Analysis and Data Mining, San Antonio, USA, 30 november- 01 December 2015 "; Data Mart Approach for Stock Management Model with a calendar Under Budgetary constraint, IJCSI, volume 15, Issue 5, September 2016,

Eugene MBUYI M. is professor at the Mathematic and Computers Sciences department of the University of Kinshasa. Director of informatics laboratory of the faculty of sciences at the university of Kinshasa. He is author of many articles in many scientific journals like in IJCSI. Poster et the 2nd International conference on Big Data Analysis and Data Mining, San Antonio, USA, 30 november- 01 December 2015.