Accelerating Fully Homomorphic Encryption over the Integers with Super-size Hardware Multiplier and Modular Reduction

Xiaolin Cao, Ciara Moore, Maire O'Neill, Elizabeth O'Sullivan and Neil Hanley

CSIT, ECIT, Queen's University Belfast, Belfast, Northern Ireland, UK
xcao03@qub.ac.uk, cmoore50@qub.ac.uk, m.oneill@ecit.qub.ac.uk,
n.hanley@qub.ac.uk, e.osullivan@qub.ac.uk

Abstract. A fully homomorphic encryption (FHE) scheme is envisioned as being a key cryptographic tool in building a secure and reliable cloud computing environment, as it allows arbitrarily evaluation of a ciphertext without revealing the plaintext. However, existing FHE implementations remain impractical due to their very high time and resource costs. Of the proposed schemes that can perform FHE to date, a scheme known as FHE over the integers has the advantage of comparatively simpler theory, as well as the employment of a much shorter public key making its implementation somewhat more practical than other competing schemes.

To the author's knowledge, this paper presents the first hardware implementations of encryption primitives for FHE over the integers using FPGA technology. First of all, a super-size hardware multiplier architecture utilising the Integer-FFT multiplication algorithm is proposed, and a super-size hardware Barrett modular reduction module is designed incorporating the proposed multiplier. Next, two encryption primitives that are used in two schemes of FHE over the integers are designed employing the proposed super-size multiplier and modular reduction modules. Finally, the proposed designs are implemented and verified on the Xilinx Virtex-7 FPGA platform. Experimental results show that the speed improvement factors of up to 44.72 and 54.42 are available for the two FHE encryption schemes implemented in FPGA when compared to the corresponding software implementations. Meanwhile, the performance analysis shows that further improvement is speed of these FHE encryption primitives may still be possible.

Keywords: Barrett Modular Reduction, Fully Homomorphic Encryption, FPGA, Hardware, Integer-FFT Multiplication.

1 Introduction

Fully homomorphic encryption (FHE) is a significant breakthrough in cryptographic research in recent years [1]. A FHE scheme can be used to arbitrarily perform computations on a ciphertext, but without compromising the content of the corresponding plaintext. Therefore, a practical FHE scheme will open the door to numerous new security technologies and privacy related applications, such as the privacy-preserving search, cloud-based storage, computing outsourcing and identity preserving banking.

To date, many FHE schemes based on different computationally hard problems have been proposed [1 – 9], as well as their software implementations [8 – 13]. The first software implementation of the lattice-based FHE scheme was reported by Gentry and Halevi (GH) with a public key size from 17 Megabytes (MB) to 2.3 Gigabytes, and a ciphertext homomorphic evaluation time from 6 seconds to 30 minutes [10]. Then, Coron *et al.* [8] proposed a scheme of FHE over the integers with a reduced public key size from 0.95 MB to 802 MB, and an encryption time from 0.05 seconds to 3 minutes. Next, Coron *et al.* [9] further reduced the public key size to no more than 10.1 MB but with a longer encryption time of 0.05 seconds to 7 minutes. In more recent work, Gentry *et al.* described a homomorphic implementation of the block cipher Advanced Encryption Standard (AES), which requires 36 hours to evaluate a single AES encryption operation [11]. Lauter *et al.* presented a somewhat homomorphic encryption implementation that employs much shorter key, 29 Kilobytes, than its FHE counterpart, and requires a shorter encryption time of 0.024 seconds [12]. The most recent software implementation was done on a NVIDIA C2050 GPU [13], and it uses the integer-FFT multiplication algorithm [14] to compute the super-size multiplication and Barrett modular reduction [15] to implement Gentry's FHE scheme [2]. It gained almost 7 times speed improvement compared to the work in [10]. However, all these reported software implementations results show that current FHE schemes still face severe efficiency challenges, impractical public key sizes and a very large computational complexity. Hence, there is still a long way to go before a practical FHE scheme can be deployed in real-life applications. To date, the only previous hardware related FHE implementations are reported in [16, 17]. In these work, they look to obtain a scalable hardware implementation on a FPGA platform using the Matlab® HDL Coder tool, however they do not report any implementation or simulation results yet.

The objective of this paper is to accelerate the encryption primitives in FHE over the integers using FPGA technology. This algorithm was chosen as it uses smaller key size, and the FPGA platform is used as it provides a quick verification environment[1]. Specifically, we present the first full hardware implementation of the encryption primitives required for FHE over the integers. Our contributions are as follows: (i) a super-size hardware multiplier architecture using the Integer-FFT multiplication algorithm is proposed; (ii) a super-size hardware architecture of Barrett modular reduction is presented using the proposed multiplier as a sub-module; (iii) two encryption primitives of FHE over the integers are designed utilising the proposed super-size multiplier and modular reduction as sub-modules; (iv) our implementations are verified in Xilinx Virtex-7 FPGA, and the result shows our designs achieve a significant perfor-

[1] As low cost FPGAs decrease in price, physical size and power consumption, while at the same time increasing in density, they are also increasingly being used in released products. This has the added advantage in security scenarios of allowing in-situ upgrading of hardware as recommended protocols and algorithms are changed due to vulnerability concerns.

mance improvement of a factor of 44.72 and 54.42 over their prior corresponding software counterparts.

The rest of the paper is organised as follows. In Section 2, the previously related works are reviewed. In Section 3 the proposed hardware architecture of the super-size multiplier is described. Next, Section 4 details the hardware architectures of the proposed super-size Barrett reduction and two FHE encryption primitives. The implementation and performance comparison results are given in Section 5. Finally, Section 6 concludes the paper.

2 Review of Related Work

2.1 Encryption Primitives in FHE over the Integers

Currently, there are three different schemes of FHE over the integers. The first scheme was proposed by van Dijk *et al.* [3]. Then this scheme was improved by Coron *et al.* [8] by reducing the public key size, this scheme is referred to as CMNT in this paper. The last scheme with the smallest public key size was presented in 2012 also by Coron *et al.* [9], and is denoted as CNT here. As the encryption primitives of the two shorter key size schemes, CMNT and CNT, are implemented in this paper, their mathematical definitions are listed in Equation (1) and Equation (2).

$$C = \left(M + 2R + 2\sum\nolimits_{1\le i,j\le\theta} B_{i,j} \times A_{i,0} \times A_{j,1}\right) mod\ A_0 \quad (1)$$

$$C = \left(M + 2R + 2\sum\nolimits_{i=1}^{\theta} B_i \times A_i\right) mod\ A_0 \quad (2)$$

In both equations, C denotes the ciphertext; $M \in \{0,1\}$ is a 1-bit plaintext; R is a random signed integer in $(-2^{\rho}, 2^{\rho})$; $A_0 \in [0, 2^{\varphi})$, is a part of the public key. In Equation (1), $\{B_{i,j}\}$ with $1 \le i,j \le \theta$ is a random integer sequence, and each $B_{i,j}$ is a δ-bit integer. $\{A_{i,0}\}$ and $\{A_{i,1}\}$ with $1 \le i \le \theta$ are two public key sequences, and each entry is a φ-bit integer. In Equation (2), $\{B_i\}$ with $1 \le i \le \theta$ is a random integer sequence, and each B_i is a δ-bit integer. $\{A_i\}$ with $1 \le i \le \theta$ is again the public key sequence, and each A_i is a φ-bit integer. The parameter bit-length of four test groups for both equations, which will be used in the following performance comparison in Section 5, are individually listed in Table 1 and Table 2.

Table 1. The four groups of parameters for Equation (1) in CMNT [8]

Group	δ	$\rho = 4\delta$	$\varphi \times 10^{-6}$	θ
Toy	42	168	0.16	12
Small	52	208	0.86	23
Medium	62	248	4.20	44
Large	72	288	19.00	88

To implement the above two equations, the first challenge is the super-size multiplication. A typical multiplication algorithm for very large bit-length operands is Integer-FFT [14, 18, 19]. It conquers large bit-length multiplication by first dividing it into small bit-length multiplication then accumulating. For example, the widely used open-source GMP library uses the Schönhage-Strassen Integer-FFT algorithm [14] for multiplication when the bit-length of operands is greater than 2^{15} bits [20]. There are many different Integer-FFT variants that use different methods to improve the small bit-length multiplication speed, as it is the performance bottleneck of the Integer-FFT algorithm. However, a Xilinx Virtex-7 FPGA device can help to solve this problem by using its embedded multipliers, which are specially optimised for high-speed performance of up to 750MHz [21]. Thus, the basic Integer-FFT algorithm [19] combined with these embedded FPGA multipliers is used in our work.

Table 2. The four test groups of parameters for Equation (2) in CNT [9]

Group	$\boldsymbol{\delta}$	$\boldsymbol{\rho = \delta}$	$\boldsymbol{\varphi \times 10^{-6}}$	$\boldsymbol{\theta}$
Toy	936	936	0.15	158
Small	1476	1476	0.83	572
Medium	2016	2016	4.20	2110
Large	2556	2556	19.35	7695

The super-size modular reduction is also a considerable challenge. Generally, the modular reduction algorithms used in traditional long bit-length cryptography implementations are Montgomery [22] and Barrett reduction [15]. However, Montgomery reduction algorithm is only suitable for scenarios where successive modular operations with the same operands are required, such as exponentiation for example. Otherwise, a heavy pre-computation and post-processing penalty is incurred. On the other hand, Barrett reduction only requires a one-time pre-computation, and is typically used after the multiplication is completed. Therefore, Barrett reduction is adopted for the modular reduction in the proposed hardware implementations.

The objective of this paper is to accelerate the speed of Equations (1) and (2), rather than dealing with the storage bottlenecks. Therefore, it is assumed that there is sufficient off-chip memory available for the designed FPGA accelerator to store its intermediate variables and final results. This is a reasonable assumption as the accelerator could be viewed as a powerful coprocessor device, sharing memory with the main workstation (be it a server or PC) over a high speed PCI bus. However, it is acknowledged that off-chip memory I/O can become a bottleneck and that the latency of the bus becomes an issue. Investigations into such issues will be the subject of future work.

2.2 The Integer-FFT Multiplication Algorithm

The Integer-FFT multiplication treats each multiplication operand as a sequence of smaller, computationally efficient numbers instead of a single super-size integer. The input parameters to the Integer-FFT multiplication are:

- p, a m-bit prime number, used as the modulus in the Integer-FFT modular reduction.
- k, the FFT point number.
- ω, the twiddle factor of the FFT.
- b, the base unit bit-length when transforming the input super-size operand into a b-bit digit sequence.

To ensure the Integer-FFT algorithm works correctly, it is required that the FFT point number k divides $q-1$ for every prime factor q of p (in this paper, because p is a prime, q is equal to p), the twiddle factor ω is a primitive k-th root of unity (meaning that $\omega^k = 1\ (mod\ p)$ and $\omega^{k/q} - 1 \neq 0\ (mod\ p)$ for any prime divisor q of p [14]), and all operations used in the FFT should be modular with respect to the prime modulus, p.

Table 3. The four groups of Integer-FFT parameters in our experiments.

Group	p	m	k	ω	b
Special modulus form [23]	$2^{32}+1$	33	64	2	8
	$2^{64}+1$	65	128	2	24
Solinas modulus form [24]	$2^{64}-2^{32}+1$	64	128	7	28
General modulus form [19]	$2^{32}-2^{20}+1$	32	64	17	12

The Integer-FFT parameters used in our experiments are listed in Table 3. As the selection of a reasonable modulus, p, heavily influences the modular multiplication performance in Equations (4 – 6), four different moduli are implemented and compared in this paper. Their different characteristics are detailed in Section 3.3. In the following Algorithm 1, we take an example, $z = x \times y$, to explain the Integer-FFT algorithm [14] used in this paper as illustrated in Fig. 1(i).

Algorithm 1: Integer-FFT Multiplication Algorithm

Input: x, y, p, k, ω, b

Output: $z = x \times y$

- Step-1: x is processed as a b-bit digit sequence, $\{x_t\}$ with $0 \leq t < k$. The sequence $\{x_t\}$ should be treated as: x_0 to $x_{k/2-1}$ are filled by the real data of x from the least significant bit (LSB) to the most significant bit (MSB), while $x_{k/2}$ to x_{k-1} are filled with 0. Performing the same operations to y to obtain $\{y_t\}$. Their relationship is expressed in Equation (3):

$$x = \sum_{t=0}^{k} x_t\ (mod\ p),\ y = \sum_{t=0}^{k} y_t\ (mod\ p) \tag{3}$$

- Step-2: Perform a k-point FFT over the finite field Z/pZ with the sequence $\{x_t\}$ as inputs to obtain a k-point sequence, $\{X_T\}$ with $0 \leq T < k$. The same operations are applied to $\{y_t\}$ to obtain the sequence $\{Y_T\}$. Equation (4) is used to describe this relationship.

$$X_T = \sum_{t=0}^{k-1} x_t\, \omega^{Tt}\ (mod\ p),\ Y_T = \sum_{t=0}^{k-1} y_t\, \omega^{Tt}\ (mod\ p) \tag{4}$$

- Step-3: Perform a point-wise multiplication over the finite field Z/pZ, as in Equation (5), to get a k-point sequence $\{Z_T\}$ with $0 \leq T < k$:

$$Z_T = X_T \times Y_T\ (mod\ p) \tag{5}$$

- Step-4: Using the sequence $\{Z_T\}$ to perform a k-point IFFT, as in Equation (6), in the finite field Z/pZ to get a k-point sequence $\{z_t\}$:

$$z_t = k^{-1} \sum_{T=0}^{k-1} Z_T \, \omega^{-tT} \ (mod\ p) \tag{6}$$

- Step-5: Resolve the long carry chain to obtain the product $z = x \times y$, as described in Equation (7).

$$z = \sum_{t=0}^{k-1} \big(z_t \ll (t \times b)\big) \tag{7}$$

2.3 The Barrett Modular Reduction Algorithm

Algorithm 2: Barrett Reduction Algorithm
Input: x ($2m$ bits), p (m bits)
Output: $y = x \ (mod\ p)$
- Step-1: Pre-computing a constant number, $p_1 = 2^{m+\alpha}/p$;
- Step-2: Computing $\hat{\sigma}$ according to Equation (8);
- Step-3: Computing $p_2 = \hat{\sigma} \times p$;
- Step-4: Computing $y_1 = x - p_2$ and $y_2 = y_1 - p$;
- Step-5: If $y_2 < 0$, $y = y_1$, otherwise $y = y_2$.

In this paper, two versions of Barrett modular reduction are designed. The first one is for the small size reduction used in the Integer-FFT algorithm, and the second is the proposed super-size Barrett reduction. Both of them adopt the Barrett reduction algorithm introduced in [25].

$$\hat{\sigma} = \frac{\frac{x}{2^{m+\beta}} \times \frac{2^{m+\alpha}}{p}}{2^{\alpha-\beta}} \tag{8}$$

The essence of the Barrett reduction is that $\hat{\sigma}$ as given in Equation (8) is used to estimate x/p, then $x - \hat{\sigma}p$ is used to approximate $x \ (mod\ p)$. The advantage of this algorithm is that it has been proved that if $\beta < -2$ and $\alpha > m$, at most only one subtraction is required in the final reduction [25]. Algorithm 2 outlines the Barrett reduction algorithm.

3 The Proposed Super-Size Multiplier Architecture

3.1 The Architecture Overview

An overview diagram of the proposed super-size hardware multiplier architecture is illustrated in Fig. 1(ii). It consists of a shared RAM, a finite state machine (FSM) controller, and an Integer-FFT unit. The shared RAMs are assumed to be off-chip, and are used to store the input operands, the intermediate and final results. The FSM controller is responsible for distributing the signals to schedule to algorithm. The proposed FSM scheduling mechanism can be viewed as a combination of school-book multiplication [20] and the Integer-FFT multiplication [14]. The core element of the design is an Integer-FFT module that executes a block multiplication for calculating partial products of the entire super-size multiplication, while the FSM controller schedules an iterative school-book multiplication to accumulate the block products. The proposed architecture is also a fully pipeline architecture, as the RAM reading, RAM writing and Integer-FFT operations are executed in parallel.

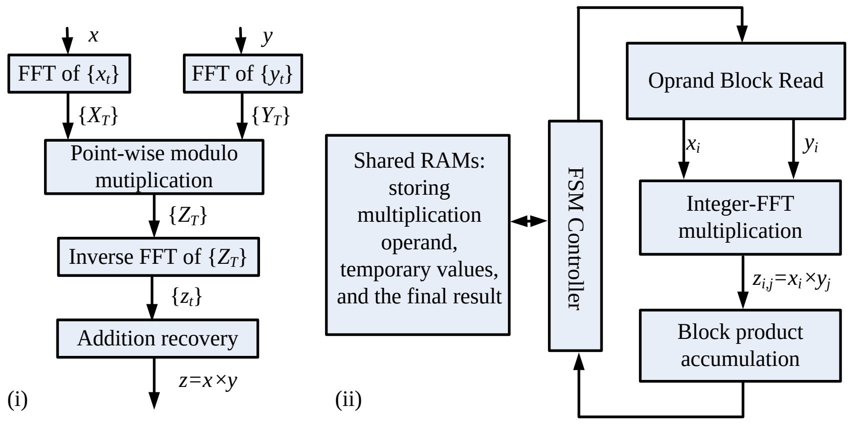

Fig. 1. (i) Diagram of the Integer-FFT multiplication algorithm; (ii) Overview of the proposed super-size hardware multiplier architecture

In the following of this subsection, we take an example, $z = x \times y$, to explain the operation of the proposed architecture. It is assumed that the super-size operands, x and y, are already stored in the shared RAMs before the multiplication starts. The input parameters to the proposed multiplier are divided into two groups. The first group relates to multiplication operands, x (n_x-bit) and y (n_y-bit). The second group is the Integer-FFT related parameters as described in Section 2.2. The steps of computing $z = x \times y$ are as follows:

- Step-1: In this step the operands are read from the shared RAMs. The operand x is processed as a sequence of $\frac{kb}{2}$-bit data blocks, $\{x_i\}$ with $0 \le i < \frac{2n_x}{kb}$, from LSB to MSB. Similarly, the sequence $\{y_j\}$ with $0 \le j < \frac{2n_y}{kb}$ is obtained. Each iteration, $\frac{kb}{2}$-bit x_i and $\frac{kb}{2}$-bit y_j are read into the proposed Integer-FFT multiplier. Therefore, the total count of RAM read access is $\frac{2n_x}{kb} \times \frac{2n_y}{kb}$ due to the use of school-book multiplication method.
- Step-2: The Integer-FFT multiplication is performed to calculate the block product, $z_{i,j} = x_i \times y_j$, as described in Algorithm 1.
- Step-3: Following the school-book multiplication method, this step accumulates the block products to obtain the final product, z. This step also determines how to write/read the partial products to/from the shared RAMs. The final product, z, is written to the share RAM as $\frac{kb}{2}$-bit data blocks, $\{z_k\}$ with $0 \le k < \frac{2(n_x+n_x)}{kb}$, from LSB to MSB as follows:
  - Step-3.1: Read the $\frac{kb}{2}$-bit partial product, z_{i+j}, from the shared RAMs into the proposed multiplier. The step is only done when both $i > 0$ and $j > 0$.

- Step-3.2: Write the partial/intermediate block product into the shared RAMs. In this step, the block product, $z_{i,j}$, is processed as two $\frac{kb}{2}$-bit parts from LSB to MSB, ${z_{i,j}}^{MSB\_half}$ and ${z_{i,j}}^{LSB\_half}$. If $i == j == 0$, ${z_{i,j}}^{LSB\_half}$ is directly written into memory, and ${z_{i,j}}^{MSB\_half}$ remains the same; else if $i == 0$ and $j > 0$, ${z_{i,j}}^{LSB\_half} + {z_{i,j-1}}^{MSB\_half}$ is written to memory; else if $i > 0$ and $j == 0$, the addition, ${z_{i,j}}^{LSB\_half} + z_{i+j}$, is performed prior to writing memory; else if $i > 0$ and $j > 0$, ${z_{i,j}}^{LSB\_half} + {z_{i,j-1}}^{MSB\_half} + z_{i+j}$ is written to memory. When $i + j > 0$, ${z_{i,j-1}}^{MSB\_half}$ is kept in an on-chip register array for faster accumulation.
- Step-3.3: Determine the index of the operand data block to be read and determine the conditions for iterating the block multiplication as follows: If $j < \frac{2n_y}{kb} - 1$, increment j, then go to Step-1; Else if $j == \frac{2n_y}{kb} - 1$ and $i < \frac{2n_x}{kb} - 1$, reset $j = 0$, increment i, then go to Step-1; Else when $j == \frac{2n_y}{kb} - 1$ and $i == \frac{2n_x}{kb} - 1$, the whole multiplication is completed.

In the following sections, the key components in the proposed multiplier architecture are described. In all the following diagrams the outputs of multiplication and submodules are registered.

3.2 The FFT/IFFT Module and Its Butterfly Unit

There are various different FFT algorithms and architectures that can be used to implement Equations (4) and (6) for different tradeoff purposes [18, 19]. In this paper, the radix-2 fully parallel architecture is adopted for the FFT and IFFT in order to obtain the highest multiplication throughput. The architecture is illustrated in Fig. 2(i). For example, there are $\log_2 k$ butterfly stages for a k-point FFT, and each butterfly stage is composed of $k/2$ parallel butterfly units, which is plotted in Fig. 2(ii).

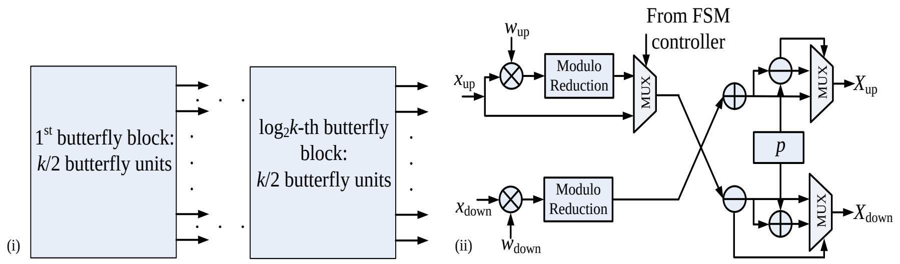

Fig. 2. (i) The radix-2 parallel FFT diagram; (ii) The proposed FFT/IFFT butterfly unit

The IFFT in Equation (6) needs to multiply $k^{-1}\ (mod\ p)$, which is not needed in the FFT. If an identical architecture is used to implement both of FFT and IFFT, a point-wise module multiplication stage is additionally required for the IFFT, and the cycle latency of the IFFT is increased compared to the FFT. The problem is solved by pre-computing $\widehat{\omega}^{-1} = k^{-1} \times \omega^{-1}\ (mod\ p)$ to incorporate $k^{-1}\ (mod\ p)$ into the IFFT twiddle factors, then $\widehat{\omega}^{-1}$ is used in the final IFFT butterfly stage, while the other stages still use ω^{-1}. In order to meet the butterfly requirement of both of FFT and IFFT, a unified butterfly unit is proposed in Fig. 2(ii). The multiplication operation at the bottom left-hand side in Fig. 2(ii), $x_{\text{down}} \times \omega_{\text{down}}$, is the same for all FFT/IFFT butterfly stages, as well as the operation $X_{\text{up}}/X_{\text{down}}$ as shown on the right-hand side of Fig. 2(ii). However, the operation of $x_{\text{up}} \times \omega_{\text{up}}$ illustrated on the upper left-hand side of Fig. 2(ii) is only required at the final stage of the IFFT.

In our designs, if the special modulus form $p = 2^{m-1} + 1$ as listed in Table 3 is used, each (m-bit $\times$ m-bit) multiplier in a butterfly is implemented as bit-shifting, as the k-th primitive root of unit ω is equal to 2 in this situation. Otherwise, each butterfly multiplier is designed using a multi-stage pipelined multiplier, which is implemented using the FPGA embedded multipliers through the use of the Xilinx Core Generator [21] tools. This prevents the multipliers becoming the timing performance bottleneck in our design.

3.3 The Modular Reduction Module

The addition/subtraction modular reduction is very simple and is illustrated in the right-hand part in Fig. 2(ii). Therefore, this subsection introduces the modular reduction unit used after the butterfly and point-wise multiplication. Three reduction methods are designed and tested in our work: the first is the Barrett modular reduction that can be used for any modulus ($2^{32} - 2^{20} + 1$ here), the second is the simplest reduction method of the three that is only suitable for a modulus with the special form $p = 2^{m-1} + 1$, and the third is suitable for the Solinas modulus $2^{64} - 2^{32} + 1$ [24].

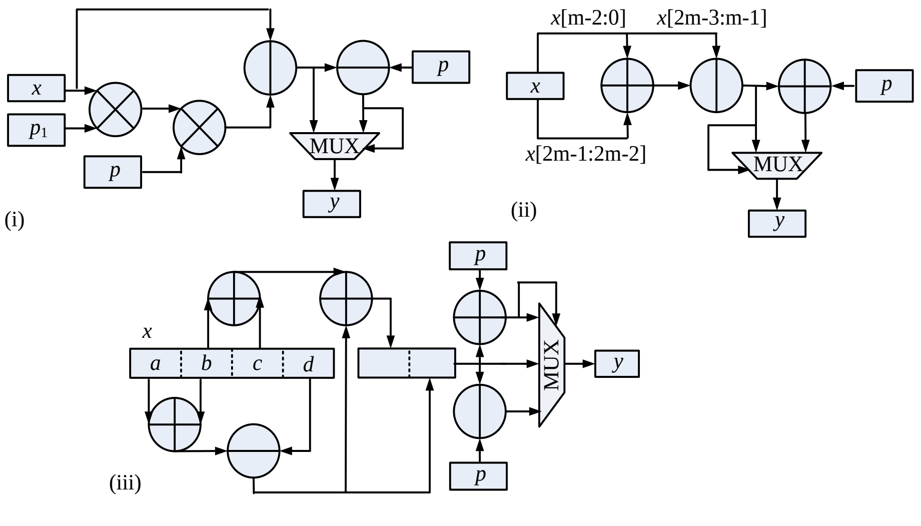

Fig. 3. The proposed modular reductions used in FFT butterfly and point-wise multiplication: (i) Barrett reduction suitable for all modulus; (ii) The simplest reduction only suitable for a special form modulus, $p = 2^{m-1} + 1$; (iii) A simpler reduction only suitable for the Solinas form modulus

The Barrett reduction architecture is shown in Fig. 3(i). Following the Barrett reduction algorithm outline in Section 2, in our design we set $\beta = -4$ and $\alpha = m + 4$, thus, $p_1 = 2^{2m+4}/p$, and the pre-computed constant number in Fig. 3(i) is $m + 4$ bits.

The design of the special form modulus reduction algorithm [14] is shown in Fig. 3(ii). The input parameters are x ($2m$-bit) and $p = 2^{m-1} + 1$, the reduction $y = x\ (mod\ p)$ is easily to be obtained using the logic in Fig. 3(ii) as follows: let $y_1 = x[m-2:0] + x[2m-1:2m-2] - x[2m-3:m-1]$ and $y_2 = y_1 + p$; If $y_1 < 0, y = y_2$; else $y = y_1$. As no multiplication is required here, this circuit obviously consumes less hardware resource than Barrett reduction, and its speed performance is better.

The design of the Solinas modulus reduction is shown in Fig. 3(iii). If the Solinas modulus $p = 2^{64} - 2^{32} + 1$ is used, the 128-bit multiplication product can be expressed as $x = 2^{96}a + 2^{64}b + 2^{32}c + d$, where a, b, c and d are 32-bit numbers. As $2^{96}\ (mod\ p) = -1$ and $2^{64}\ (mod\ p) = 2^{32} - 1$, the reduction can be quickly computed as $x\ (mod\ p) = 2^{32}(b + c) - a - b + d\ (mod\ p)$. Thus, the upper-half 64-bit ($2^{32}(b + c)$) and the lower-half 64-bit $(-a - b + d)$ results can be computed independently. As the result of $2^{32}(b + c) - a - b + d$ is within the range of $(p, 2p)$, an addition, a subtraction and a 3:3 multiplexer are needed for the final reduction. Although it is a little more complex than the special modulus form, it is much simpler than the Barrett reduction as no multiplication is required. However, according to the condition described in Section 2.2, not every Solinas modulus is suitable for the Integer-FFT algorithm [14, 19].

3.4 The Addition Recovery and Product Accumulation Module

The addition recovery module is responsible for converting the IFFT outputs back to an integer by resolving a very long carry chain, as is shown in Equation (7). The product accumulation module is used to generate intermediate product results that can be written to memory. As these two modules are tightly coupled together in our proposed design, they are described in the same section.

The addition recovery architecture is composed of two parts, respectively depicted in Fig. 4(i) and 4(ii). The part in Fig. 4(i) is a parallel two-by-two adder tree, which means at each addition level, the adjacent two entries are added from the least significant entry to the most significant entry. Let d represent the data bus bit-width between our super-size multiplier and the shared RAMs. There are in total $\frac{d}{b}$ levels two-by-two adder trees (provided suitable values of d and b are chosen). Let the sequence $\{z'_i\}$ with $0 \le i < \frac{kb}{d}$ be the result after $\frac{d}{b}$ levels two-by-two addition. The addition results in each level are registered. Otherwise, a very long carry chain will become the time performance bottleneck of the design. It can be shown that the first useful d-bit output value is included in z'_0, the second available d-bit product can be obtained by $z'_1 + (z'_0 \gg d)$, and the third d-bit product can be obtained by $z'_2 + \left(z'_1 + (z'_0 \gg d)\right) \gg d$. Therefore, this can be achieved by a registered carry chain addition, which is illustrated in Fig. 4(ii).

The advantage of the above architecture is that each clock we have a d-bit result to write into memory, thus, there is no bus bit-width wastage. However, as the carry chain of length d/b is needed to generate the d-bit result, the parameter d/b should be carefully chosen to avoid this carry chain becoming the performance bottleneck. This parameter is determined by iterative experiments in our implementation (here the values $\frac{d}{b} = \{4,4,8,8\}$ are used for the four test groups in Table 4 respectively).

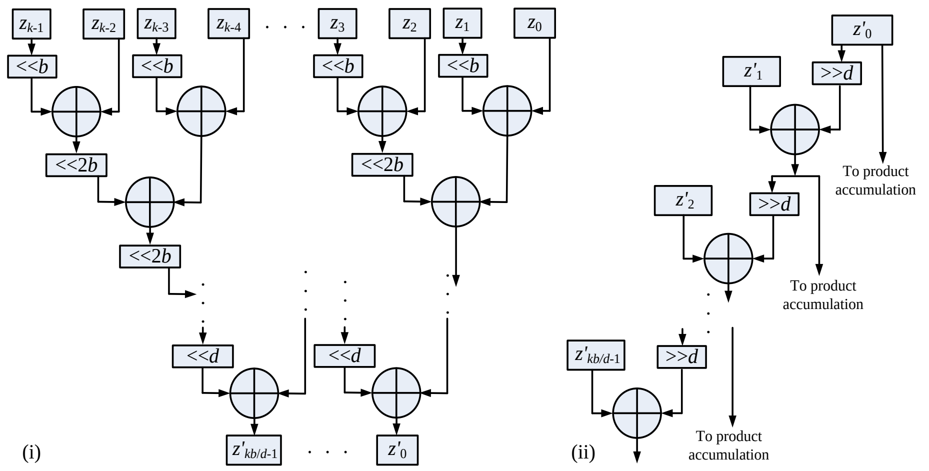

Fig. 4. The proposed addition recovery architecture: (i) a parallel two-by-two adder tree used as the 1st part; (ii) a serial and registered carry chain used as the 2nd part.

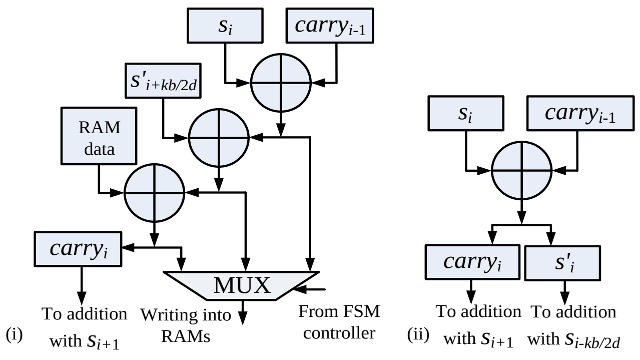

Fig. 5. The proposed serial and registered multiplication product accumulation architecture: (i) when $0 \leq i < \frac{kb}{2d}$; (ii) when $\frac{kb}{2d} \leq i < \frac{kb}{d}$

The proposed product accumulation architecture is illustrated in Fig. 4(ii). It is assumed that the sequence of $\{s_i\}$ with $0 \leq i < (kb)/d$ is the input. This architecture is a detailed diagram illustrating how to execute the control logic explained in Step-3.2 and Step-3.3 in Section 3. Since each iteration a $(kb)/d$-bit partial/intermediate product is written into memory, the LSB product (i.e. when $i < (kb)/(2d)$) has a different processing procedure from the MSB product (i.e. when $i \geq (kb)/(2d)$), which is shown in Fig. 5(i) and Fig. 5(ii) respectively.

From the carry chain logic perspective, the logic in Fig. 4(ii) should be executed after that in Fig. 4(i). Actually in our design, they are concurrently and pipelined executed. We take the example in Step-3.2 to explain in Section 3. Each block product, $z_{i,j}$, is processed as two $(kb)/2$-bit parts, ${z_{i,j}}^{MSB\_half}$ and ${z_{i,j}}^{LSB\_half}$. In the first block round (i.e. $i == j == 0$), ${z_{i,j}}^{LSB\_half}$ is first generated and written into memory. At the same time, ${z_{i,j}}^{MSB\_half}$ cannot be obtained due to the lacking of carry bits. Then from the next block iteration (i.e. $i \neq 0$ or $j \neq 0$), ${z_{i,j-1}}^{MSB\_half}$ and ${z_{i,j}}^{LSB\_half}$ can be calculated simultaneously, as all the parameters in Fig. 4(i) and Fig. 4(ii) are already ready to use from this moment.

4 The Proposed FHE Encryption and Super-Size Modular Reduction Architecture

The proposed hardware architecture for the two encryption primitives in CMNT and CNT are plotted in Fig. 6(i). These two primitives share the same architecture but with different FSM controller logic (i.e., different accumulation and multiplication schedule procedure). Due to that only one instance of the proposed multiplier is implemented in our architecture, the FHE encryption architecture is tightly coupled with the super-size Barrett modular reduction using the FSM controller. Thus, the overview diagram in Fig. 6(i) can also represent the proposed architecture of the super-size Barrett reduction, which means that all the multiplications, in Equations (1) and (2) and the super-size Barrett reduction, are completed by the single Integer-FFT instance.

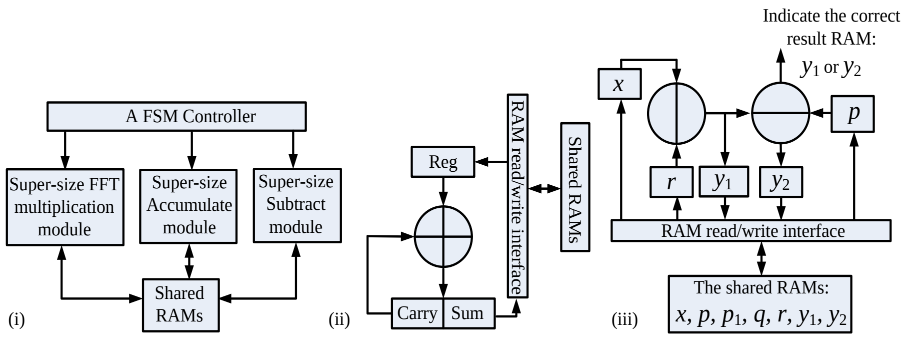

Fig. 6. The proposed FHE encryption and super-size Barrett modular reduction architecture: (i) an overview diagram; (ii) a super-size accumulation module for FHE encryption; (iii) the subtraction module for the super-size Barrett reduction.

The proposed super-size accumulation module is illustrated in Fig. 6(ii) to complete the required product accumulation operation in Equations (1) and (2). It is basically a general d-bit width accumulation operation, which is interleaved with the memory reading/writing operations. As the bit-length of the modulus A_0 in Equations (1) and (2) is φ, the bit-length of the accumulation result in Equations (1) and (2) should be defined as $\varphi + \gamma$ according to the adopted Barrett reduction algorithm [25]. As the bit-length of B_i or $B_{i,j}$ is much less than φ and the bit-length of the accumulation counter θ is much less than φ, it is clear that γ is also much less than φ. This is beneficial to the super-size Barrett reduction hardware design, as it allows more flexibility to choose the parameters. In this super-size situation, Equation (8) becomes (9):

$$\hat{\sigma} = \frac{\frac{x}{2^{\varphi+\beta}} \times \frac{2^{\varphi+\alpha}}{A_0}}{2^{\alpha-\beta}} \tag{9}$$

Here, we still use the symbols x and $\hat{\sigma}$ in our explanation. As x, $\frac{x}{2^{\varphi+\beta}} \times \frac{2^{\varphi+\alpha}}{A_0}$ and $\hat{\sigma}$ are super-size parameters, it is impossible to directly load all the required bits of $\frac{x}{2^{\varphi+\beta}}$ and $\hat{\sigma}$ into the multiplier directly. Therefore, we need to iteratively access the memory to obtain the full values. An implementation issue of the super-size Barrett reduction is how to choose the suitable value of β and α in Equation (9) so that the correct values of $\frac{x}{2^{\varphi+\beta}}$ and $\hat{\sigma}$ can be read out from the right address of the shared RAMs when computing $\frac{x}{2^{\varphi+\beta}} \times \frac{2^{\varphi+\alpha}}{A_0}$ and $\hat{\sigma} \times p$. In our work, this issue is solved by relating the value of β and α to the data bus bit width, d, as given in Equations (10) and (11),

$$\beta = \begin{cases} \frac{\varphi}{d} \times d - \varphi, if\ \left(\frac{\varphi}{d} \times d - \varphi\right) < -2 \\ \frac{\varphi}{d} \times d - \varphi - d, otherwise \end{cases} \tag{10}$$

$$\alpha = \left(\frac{\varphi-\beta}{d} + 1\right) \times d + \beta \tag{11}$$

In Equation (10) it is assumed that $d > 2$ in super-size FHE implementation. Through setting the initial RAM storage address of x and $\frac{x}{2^{\varphi+\beta}} \times \frac{2^{\varphi+\alpha}}{p}$ in the shared RAMs to 0, the above parameter setting makes sure the initial read address of x, $\frac{\varphi+\beta}{d}$, is a multiple of d, and the starting read address of $\frac{x}{2^{\varphi+\beta}} \times \frac{2^{\varphi+\alpha}}{p}$, $\frac{\alpha-\beta}{d}$, is also a multiple of d. At the same time, Equation (10) obviously meets the requirement of $\beta < -2$, and Equation (11) also has $\alpha > \gamma$ as it is mentioned that both γ and d are much smaller than φ. Thus, the procedure of the proposed super-size modular reduction is as follows:

- Step-1: The constant number $A_{0,1} = 2^{\varphi+\alpha}/A_0$ is pre-computed and stored in the shared RAM before the reduction starts.
- Step-2: Read x from its RAM address $\frac{\varphi+\beta}{d}$, and read $A_{0,1}$ from its RAM address 0, then use the super-size FFT multiplication module to calculate $q = x[\varphi + \gamma - 1 : \varphi + \beta] \times A_{0,1}$.
- Step-3: Read q from its RAM address $\frac{\alpha-\beta}{d}$, and read A_0 from its RAM address 0, then use the super-size FFT multiplication to compute $r = q[\alpha + \gamma - \beta : \alpha - \beta] \times A_0$.

- Step-4: Read x, r and A_0 from their RAM address 0, and use the super-size subtract module to calculate $y_1 = x[\varphi : 0] - r[\varphi : 0]$, $y_2 = y_1 - p$, and output a signal that indicates which of y_1 or y_2 is the correct reduction result.

As the subtraction also a super-size operation, a super-size subtraction module is designed in Fig. 6(iii) to complete the required subtraction operation in the super-size Barrett reduction. It is essentially a successively applied d-bit subtraction operation, but a memory read/write interface is added to extend its bit width

5 Implementation, Performance and Comparison

All the proposed architectures were designed and implemented using Xilinx FPGA technology. Modelsim 6.5a was used as the functional and post-synthesis timing simulation tool. The synthesis tool used was Xilinx ISE Design Suite 14.4. The synthesis strategy is set to balance between speed and area. The optimisation objective is set to speed. The target device is Virtex-7 XC7VX980T. The test vectors are generated as random numbers using C++ according to the parameter requirements in Table 1 and Table 2.

The proposed super-size multiplier architecture has been implemented as a fully pipelined and parallel circuit. At the outer interface, three RAM read buses and one RAM write bus are implemented, so that the two super-size multiplication operands can be simultaneously read into the multiplier, and at the same time the final block product accumulation can also be executed simultaneously. At the inner layer, the Integer-FFT is also pipelined.

From the Table 3, it can be found that the base unit bit-length is chosen as the maximum even number of the suitable numbers according to the parameter requirements described in Section 2. The reason is that the basic computation bit-length in Integer-FFT is determined by the modulus bit-length rather than the base unit bit-length, so increasing the base unit bit-length will not decrease the speed performance of Integer-FFT's multiplication and addition. However, the base unit bit-length is related to the multiplication product throughput, that is, the larger the base unit bit-length, the less the consumed clock cycle count.

In our implementation, the ratio of multiplication operand's block bit-length and the data bus bit-width (for both RAM read and write), $\frac{kb}{2d}$, to be equal to 8. Thus, the data bus bit-width is respectively equal to 32, 48, 192 and 224 corresponding to the four group parameters in Table 3. It means that the clock count is equal to 8 for each $(kb)/2$-bit input operand, and the clock cycle count is also equal to 8 for each output block product except the first block product, according to our described hardware architecture. Meanwhile, the pipeline stage count in each FFT butterfly stage is designed to be also equal to 8, which accounts for all the m-bit multiplication, addition/subtraction and modular reduction.

The latency of the proposed architecture is analysed as follows. The total latency for the designed pipelined multiplication is composed of two parts. The first part is the time cost required by the preparation work before outputting the first block product, and the second part is determined by the count of block products multiplying a fixed parameter for a pipelined architecture. It is equal to 8 in our design.

Let N_F, N_{PW} and N_{AR} respectively be the count of pipeline stage in a FFT/IFFT butterfly, in a point-wise multiplication, and in the addition recovery. Then, the time latency of the preparation work is equal to $(kb)/(2d) + 2N_F \log_2 k + N_{PW} + N_{AR}$, where $(kb)/(2d)$ denotes the latency for reading the operand, and $2N_F \log_2 k + N_{PW} + N_{AR}$ denotes the block product computation latency. Next, the common time latency of outputting each block products is analysed according to the logic described in Step-3.2 and Step-3.3 in Section 3. If $j < (2n_y)/(kb) - 1$, $(kb)/2$-bit intermediate product is written into memory, and the latency is equal to $(kb)/(2d)$; if $j == (2n_y)/(kb) - 1$, kb-bit intermediate product is written into memory, and the latency is $(kb)/(2d)$. As there are totally $(4n_x n_y)/((kb)^2)$ block products, in which there are $(2n_x)/(kb)$ blocks with the condition $j == (2n_y)/(kb) - 1$. As in our design we have $N_F = N_{PW} = N_{AR} = 8$, therefore, combining the two parts, the whole time latency can be expressed as:

$$T_l = 16(log_2 k + 1) + \frac{2}{d}\left(\frac{n_x n_y}{kb} - n_x + \frac{kb}{2}\right) \tag{12}$$

We have implemented four different designs using the four groups of parameters in Table 3. For each design, the FFT point num k is fixed, unrelated to the multiplication operand bit-length. With the input multiplication operand bit-length increasing, $16(log_2 k + 1)$ quickly becomes much smaller than $2\big((n_x n_y)/(kb) - n_x + (kb)/2\big)/d$, as the data bus bit-width d is fixed. Therefore, the time latency of the proposed implementations with the four forms of modulus is almost the same, which can be estimated by multiplying the total count of block products, $(n_x n_y)/(kb)$, and the output latency of each block product, $(kb)/(2d) == 8$.

For the small size 33-bit×33-bit, 32-bit×32-bit, 65-bit×65-bit and 64-bit×64-bit multiplication used in FFT butterfly and point-wise multiplication, Xilinx Core Generator is employed to generate a 4 stage pipelined multiplier using the embedded multipliers on the Virtex-7 FPGA device. Meanwhile, in order to meet the demand of the addition recovery latency, that is $N_{AR} == 8$, the adder tree length is equal to d/b, 4, 4, 8, and 8 in our implementations respectively.

Table 4. The synthesis results of the proposed super-size multiplier.

Group	Frequency (MHz)	DSP48E1 Utilisation (total: 3600)	Slice register Utilisation (total: 1224000)	Slice LUT Utilisation (total: 612000)
Multiplier with $2^{32}+1$	304.990	256	191151	237034
Multiplier with $2^{32}-2^{20}+1$	254.065	6292	213403	170919
Multiplier with $2^{64}+1$	179.346	2048	955974	1214269
Multiplier with $2^{64}-2^{32}+1$	166.450	18496	1122826	954737

The synthesis results of the proposed super-size multiplier are displayed in Table 4. The hardware resource utilisation and frequency of the two designs with the special modulus ($2^{32}+1$ and $2^{64}+1$) are obviously better than the multipliers with the general modulus ($2^{32}-2^{20}+1$) and the Solinas modulus ($2^{64}-2^{32}+1$). The reason the special modulus multiplier requires the smallest hardware resource is that it does not need multiplication in the FFT butterfly unit and no multiplication is needed for reduction with the special modulus. The reason the Solinas modulus multiplier requires more hardware than the general modulus multiplier is that it does not need multiplication in the modular reduction units. It can also be seen that the smallest bit-length modulus multiplier has the highest frequency, as the four multipliers needed for the point-wise multiplication in the Integer-FFT algorithm contain the critical path, and are implemented as 4 stage pipelined multipliers. However, this does not mean that the super-size multiplier with the special modulus will definitely outperform the other two, as the other two multipliers allow a larger base bit-length that implies that in one clock cycle the Solinas modulus multiplier can produce the longest product.

Meanwhile, we can find that only the multiplier with the smallest modulus $2^{32}+1$ is within the hardware resource budget, and there are not enough hardware resources for the other architectures. Thus, the conclusion is that the parallel FFT architecture is not suitable for a practical or implementable design due to the excessive hardware cost, although this FFT architecture has the highest product throughput.

Table 5. The synthesis results of the FHE encryption primitives.

Group	**Frequency (MHz)**	**DSP48E1 Utilisation (total: 3600)**	**Slice register Utilisation (total: 1224000)**	**Slice LUT Utilisation (total: 612000)**
CMNT with $2^{32}+1$	292.410	256	191176	237031
CMNT with $2^{32}-2^{20}+1$	254.054	6292	213788	171450
CMNT with $2^{64}+1$	179.346	2048	956974	1215166
CMNT with $2^{64}-2^{32}+1$	166.450	18496	1123001	954955
CNT with $2^{32}+1$	292.410	256	191167	236979
CNT with $2^{32}-2^{20}+1$	254.054	6292	213777	171494
CNT with $2^{64}+1$	179.346	2048	956973	1215150
CNT with $2^{64}-2^{32}+1$	166.450	18496	1122933	954947

The proposed super-size Barrett reduction has been implemented by serially calling the super-size multiplier to finish Step-2 and Step-3 in Section 4. For the super-size subtraction module, additional RAM read and write buses are implemented in parallel to simultaneously execute with the Integer-FFT, so the cycle latency of the super-size

Barrett reduction can be estimated at twice the time cost that is required by a super-size multiplication with the input operand's bit-length.

The two encryption primitives in Equation (1) and (2) have been implemented by adding a super-size accumulation module following the super-size Barrett reduction. It can be seen as a serial procedure of two stages. In the first stage, the super-size accumulation module and the super-size multiplier are parallel executed; in the second stage, the super-size Barrett reduction is executed. The reason is that there is only one super-size multiplier is implemented. The synthesis results of the implementations of FHE encryption primitives are listed in Table 5. It can be seen that the frequency and hardware resource utilization are almost the same as that in Table 4. The synthesis performance of CMNT and CNT is also nearly identical. The reason is that the controller part only occupies a very small percentage of hardware resource compared to the super-size multiplier, and the only difference between the CMNT's and CNT's controller parts is the different multiplier scheduling procedures.

Table 6. The average running time of the proposed FHE encryption designs.

Group	Toy	Small	Medium	Large
CMNT with $2^{32}+1$	0.003 s	0.050 s	0.872 s	15.735 s
CMNT with $2^{32}-2^{20}+1$	0.003 s	0.057s	1.003 s	18.110 s
CMNT with $2^{64}+1$	0.000854 s	0.0139 s	0.239 s	4.284 s
CMNT with $2^{64}-2^{32}+1$	0.000815 s	0.0130 s	0.221 s	3.958 s
CMNT [8]	0.05 s	0.79 s	10 s	2 min 57 s
CNT with $2^{32}+1$	0.011 s	0.306 s	7.586 s	159.173 s
CNT with $2^{32}-2^{20}+1$	0.012 s	0.352 s	8.731 s	183.204 s
CNT with $2^{64}+1$	0.000786 s	0.0142 s	0.5165 s	8.655 s
CNT with $2^{64}-2^{32}+1$	0.000739 s	0.0132 s	0.4772 s	7.994 s
CNT [9]	0.05 s	1.0 s	21 s	7 min 15 s
[13]		1.69 s		

The running time of the proposed hardware implementations of FHE encryption primitives using the 4 group parameters listed in Table 1 and Table 2 is provided in Table 6, and the previously reported corresponding software results are also given here. The running time is obtained by averaging the simulated latency of the test vectors and multiplying by the synthesis frequency. It can be seen that although the special modulus multiplier has a higher frequency, it needs more running time than the Solinas modulus multiplier. The reason is that for the proposed multiplier, the multiplication product throughput is mainly determined by the product of the data bus bit-width, the base unit bit-length and the frequency, rather than only the frequency, which can be deduced from the time latency expression in Equation (12). Comparing the four group parameters in Table 3, it is also easy to see that although the special modulus can use twiddle factor of 2, its obvious disadvantage is that its base unit bit-length is much smaller than the other two kinds of modulus when almost the same bit-length modulus is employed. Therefore, for the proposed FHE encryption architecture, the Solinas modulus is the best choice in our proposed designs, which enables a comparable simpler modular reduction and a larger base unit bit-length.

It can be seen that when the Solinas modulus multiplier is used, the proposed implementations are 61.34 and 67.65 times faster than the software implementations of CMNT and CNT respectively for the toy parameter group. And the proposed designs also respectively achieve a time improvement of 44.72 and 54.42 for the large group. It must be noted that only the experimental results using small FFT parameters (i.e., $k = 128$ and $m \leq 65$) are reported in this paper. As the product of the data bus bit-width and the frequency determines the proposed super-size multiplier product throughput performance, we have enough reason to believe that there is still a great potential to improve the time performance for encryption primitives in FHE over the integers if larger FFT parameters are used. For example, the recent batch FHE over the integers proposed in [26] shows that the parallel super-size multiplication can be applied to improve the FHE encryption speed.

6 Conclusion

In this paper, the first complete hardware implementations of two encryption primitives employed in schemes of FHE over the integers proposed by Coron *et al.* are presented. For this purpose, an Integer-FFT based super-size hardware multiplier module and a super-size Barrett modular reduction module are proposed. These proposed hardware architectures are designed and verified on a Xilinx Virtex-7 device using four groups of Integer-FFT parameters. When the super-size multiplier is implemented with a Solinas Integer-FFT modulus, the synthesis results show that our hardware implementations achieve speed improvement factors of 44.72 and 54.42 compared to the corresponding software implementations for the large scale test data used in FHE over the integers. Meanwhile, as our implementations only use 128 point FFT and small base unit bit-length, 24 and 28, for the super-size hardware multiplier, there is still great potential to further improve the encryption speed in FHE over the integers. Recently research improvements in this area such as the batch FHE over the integers [26] confirm that this potential indeed exists for both hardware architecture and implementation developments.

References

1. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University, (2009). Http://crypto.stanford.edu/craig.
2. Gentry, C.: Fully homomorphic encryption using ideal lattices. The 41st annual ACM symposium on Theory of computing. pp. 169–178. ACM (2009).
3. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. EUROCRYPT 2010, LNCS, vol. 6110, pp. 24–43. Springer (2010).
4. Smart, N.P., Vercauteren, F: Fully homomorphic encryption with relatively small key and ciphertext sizes. PKC 2010, LNCS, vol. 6056, pp. 420–443. Springer (2010).
5. Brakerski, Z., Vaikuntanathan, V.: Efficient Fully Homomorphic Encryption from (Standard) LWE. FOCS (2011).
6. Brakerski, Z., Vaikuntanathan, V.: Fully Homomorphic Encryption for Ring-LWE and Security for Key Dependent Messages. CRYPTO 2011, LNCS, vol. 6841, pp. 505–524. Springer (2011).

7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Report 2011/277 (2011).
8. Coron, J.S., Mandal, A., Naccache, D., Tibouchi, M: Fully Homomorphic Encryption over the Integers with Shorter Public Keys. CRYPTO 2011, LNCS, vol. 6841, pp. 487–504. Springer (2011).
9. Coron, J.S., Naccache, D., Tibouchi, M: Public key compression and modulus switching for fully homomorphic encryption over the integers. EUROCRYPT'12. pp. 446–464. Springer (2012).
10. Gentry, C., Halevi, S.: Implementing Gentry's fully homomorphic encryption scheme. EUROCRYPT 2011, LNCS, vol. 6632, pp. 129–148. Springer (2011).
11. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic Evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099 (2012).
12. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can Homomorphic Encryption be Practical? Cryptology ePrint Archive, Report 2011/405 (2011).
13. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. High Performance Extreme Computing (HPEC), pp. 1-5. IEEE (2012).
14. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing, vol. 7, no. 3, pp. 281–292. (1971).
15. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Advances in CRYPTO'86. pp. 311–323. Springer, (1987).
16. Cousins, D.B., Rohloff, K., Peikert, C., Schantz, R.: SIPHER: Scalable implementation of primitives for homomorphic encryption–FPGA implementation using Simulink. High Performance Extreme Computing Conference (2011).
17. Cousins, D.B., Rohloff, K., Peikert, C., Schantz, R.: SIPHER: An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) — FPGA implementation using Simulink. High Performance Extreme Computing Conference (2012).
18. Craven, S., Patterson, C., Athanas, P.: Super-sized multiplies: how do FPGAs fare in extended digit multipliers. 7th International Conference on Military and Aerospace Programmable Logic Devices. (2004).
19. Emmart, N., Weems, C.: High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes. Parallel Processing Letters, vol. 21, no. 3, p. 359. (2011).
20. http://gmplib.org/manual/Multiplication-Algorithms.html#Multiplication-Algorithms.
21. Xilinx Product Specification: LogiCORE IP Multiplier v11.2. http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf
22. Montgomery, P.: Modular multiplication without trial division. Mathematics of computation, vol. 44, no. 170, pp. 519–521. (1985).
23. Kalach, K., David, J.P.: Hardware implementation of large number multiplication by FFT with modular arithmetic, the 3rd International IEEE-NEWCAS Conference, pp.267 – 270. (2005).
24. Solinas, J. A.: Generalized Mersenne Numbers, Issue 39 of Research report (University of Waterloo. Faculty of Mathematics). (1999).
25. Dhem, J. F.: Design of an efficient public-key cryptographic library for risc-based smart cards. PhD thesis. (1998).
26. Cheon, J. H., Coron, J.-S., Kim, J., Lee, M. S., Lepoint, T., Tibouchi, M., and Yun, A.: Batch Fully Homomorphic Encryption over the Integers, Advances in Cryptology – EUROCRYPT 2013, Springer, LNCS, Volume 7881, pp. 315-335. (2013).