FPGA Implementation of a Large-Number Multiplier for Fully Homomorphic Encryption

Wei Wang and Xinming Huang
Department of Electrial and Computer Engineering
Worcester Polytechnic Institute, Worcester, MA 01609, USA
{weiwang, xhuang}@ece.wpi.edu

***Abstract*—The first plausible scheme of fully homomorphic encryption (FHE), introduced by Gentry in 2009, was considered a major breakthrough in the field of information security. FHE allows the evaluation of arbitrary functions directly on encrypted data on untrusted servers. However, previous implementations of FHE on general-purpose processors had very long latency, which makes it impractical for cloud computing. The most computationally intensive components in the Gentry-Halevi FHE primitives are the large-number modular multiplications and additions. In this paper, we attempt to use customized circuits to speedup the large number multiplication. Strassen's algorithm is employed in the design of an efficient, high-speed large-number multiplier. In particular, we propose an architecture design of an 768K-bit multiplier. As a key compoment, an 64K-point finite-field fast Fourier transform (FFT) processor is designed and prototyped on the Stratix-V FPGA. At 100 MHz, the FPGA implementation is about twice as fast as the same FFT algorithm executed on the NVIDA C2050 GPU which has 448 cores running at 1.15 GHz but at much lower power consumption.**

***Index Terms*—Fully Homomorphic Encryption, FPGA, Large-number Modular Multiplication**

I. Introduction

In recent years, one of the most significant advances in cryptography was the introduction of the first fully homomorphic encryption scheme (FHE) by Gentry [1]. This advance not only resolved an open problem posed by Rivest decades ago, but also opened the door to many new applications. Indeed, FHE makes it possible for users to perform an arbitrary number of computations directly on the encrypted data without revealing its secret key to the server. Thus an untrusted party such as a remotely hosted server can be outsourced to perform computations on the encrypted data without compromising privacy. This property of FHE is precisely what makes it invaluable for the fast-growing cloud computing industry. For instance, it was recognized early on [1] that the privacy of sensitive data on cloud computing platforms are ideally suited to be protected using FHE.

Despite its promising prospectives, FHE is still far from practical implementations due to the computational complexity. The first attempt of implementing a FHE variant was proposed by Gentry and Halevi (GH) [2] and was developed in software targeted for a general-purpose processor. Despite of an impressive array of optimizations which reduces the size of the public key and also reduces the latencies of the primitives, encryption of one bit took 1.69 seconds on a high-end server with Intel Xeon processor, while the recryption primitive takes 27.68 seconds for the lowest security setting of dimension 2,048. It is worth noting that a recrypt operation is necessary after every few bit AND operations in order to reduce the noise in the ciphertext to a manageable level. It is clear that the software implementation of the FHE scheme on general purpose processor is too slow to be useful in practical applications.

Later we took the Gentry and Halevi's FHE algorithm and accelerated it on a GPU platform [3]. Targeted on a NVidia C2050 GPU with 448 core running at 1.15 GHz, the processing time for 1-bit encryption was reduced to 45 msec and the recyption is reduced to 1.8 seconds, which is about 37.6 and 15.4 times faster than the original implemenation on an Intel Xeon processor after we did further optimization on [3]. Although the encryption and decryption time is comparable to RSA in the length of 1,024 or length 2,048, the main issue is the frequent recryption in the FHE scheme. For a piece of data with a thousand bits, each recryption operation would take a few thousand seconds which is clearly not acceptable to the end users. Retrospectively, the RSA encryption with length 1,024 would take more than 10 minutes on an Intel 8086 processor when RSA was first introduced in 1978. As the algorithm improves and primarily as the computing technology evolves along Moore's law, nowadays RSA encryption process is completed in a few miliseconds on application-specific circuits that are often embedded inside a microprocessor.

Inspired by the history of RSA development, we propose to design a customized circuits for the Gentry and Halevi's FHE implementation. Since the most computationally intentive components in the FHE primitives are large-number modular multiplications and additions, our initial attempt is to tackle the design of a large-number multiplier. The rest of the paper is organized as follows: Section 2 gives a brief introduction of fully homomorphic encryption; Section 3 presents the Strassen's FFT-based multiplication algorithm; Section 4 shows the architecture of the VLSI implementation of the finite-field FFT engine; Section 5 gives the experimental results on FPGA; and Section 6 are the conclusion and future work.

II. Gentry's Fully Homomorphic Encryption

Informally a homomorphic encryption scheme refers to an encryption function that allows one to perform a binary operation on the plaintexts while only manipulating the

978-1-4673-5762-3/13/$31.00 ©2013 IEEE

ciphertexts without the knowledge of the encryption key: $E(x_1) \star E(x_2) = E(x_1 \otimes x_2)$. If the scheme supports the homomorphic computation of any efficiently computable function, it is called a fully homomorphic encryption scheme (FHE). With FHE, an honest but curious party can perform any computation directly with encrypted result without gaining access to the plaintext.

As mentioned before, the first FHE is proposed by Gentry in [1], [4]. However, this preliminary implementation is too inefficient to be used in any practical application. The Gentry-Halevi FHE variant including a number of optimizations as well as the results of a reference implementation were presented in [2]. Due to space constraints, we only present a high level overview of the primitives and the details can be referred to the original work [2].

Encrypt: To encrypt a bit $b \in \{0,1\}$ with a public key (d, r). Encrypt first generates a random $\{-1, 0, 1\}$ "noise vector" $u = \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with the probability 0.5 and as ± 1 with probability 0.25 each. Then the message bit b is encrypted by computing

$$c = [u(r)]_d = \left[b + 2 \sum_{i=1}^{n-1} u_i r^i \right]_d \quad (1)$$

where d and r are information from the public key. For the small setting with a lattice-dimension of 2,048, the d and r has the size of about 780,000 bits.

Decrypt: The encrypted bit b can be recovered by computing

$$m = [c \cdot w]_d \bmod 2 \quad (2)$$

where w is the private key. The size of the w is the same as the size of d and r.

Recrypt: Briefly, the recyption process is simply the homomorphic decryption of the ciphertext. The actual procedure of recyption is very complicated, so we choose not to explain it here. From the brief descriptions above, we can see that the fundamental operations for the FHE are the large-number addition and multiplication. The addition has far less computing complexity than the multiplication. So we focus on the hardware acceleration of the multiplication by using FPGA or ASIC.

III. Strassen Multiplication and Finite Field FFT

A. Strassen Multiplication

Large integer multiplication is by far the most time consuming operation in the FHE scheme. Therefore, it is selected as the first block for hardware acceleration. There are many multiplication algorithms in the literature. In [5], Strassen described a multiplication algorithm based on FFT, which offers an effective solution for parallel computation of the large-number multiplication. It breaks each large multiplicand into digits and each digit contains the same number of bits, i.e. 32-bit or 64-bit. Consequently, a large number is now represented as a signal series with each digit as a sample of the signal. The Strassen FFT algorithm can be summarized as follows:

1) Given a base b, compute the fast Fourier transform of the digits (with respect to the base) of A and B, treating each digit as a data sample.
2) Compute the dot-product of the FFT results: set $C[i] = FFT(A)[i] * FFT(B)[i]$.
3) Compute the inverse fast Fourier transform: set $C' = IFFT(C)$.
4) Resolve the carries: when $C'[i] \geq B$:set $C'[i+1] = C'[i+1] + (C'[i] \operatorname{div} b)$, and $C'[i] = C'[i] \bmod b$.

B. Finite Field FFT

In Strassen's multiplication algorithm, the FFT computation can be performed in complex field as widely used in signal processing with floating point operations. But the multiplication based on the complex-field FFT needs rounding operations, which makes it very difficult to get an accurate result. Also, the floating-point operations consume a large amount of hardware resources, compared with fixed-point operations. Instead, we choose to perform the computation in the finite field $\boldsymbol{Z}/p\boldsymbol{Z}$, with prime p. The computation of a finite-field FFT requires three operations: modulo addition, modulo subtraction and multiplication. By selecting a proper prime p, the modular multiplication in the finite field can be computed rapidly. In this paper, we choose the prime p =0xFFFFFFFF00000001 from a special family of prime numbers which are called Solinas Primes [6]. The prime p has special identities, such as ($2^{96} \bmod p$ = -1) and ($2^{64} \bmod p = 2^{32} - 1$), which makes it support highly efficient modulo multiplication z modulo p, where z is an arbitrary 128-bit number[7]. The 128-bit number z can be represent as $z = 2^{96}a + 2^{64}b + 2^{32}c + d$ (where a, b, c and d are each 32-bits). By using the properties of the prime p, the calculation can be simplified as follows [7]:

$$z = 2^{96}a + 2^{64}b + 2^{32}c + d = 2^{32}(b + c) - a - b + d \quad (3)$$

Consequently, the discrete Fourier transform in the finite field can be represent as (4) , where r is the k^{th} primitive root of unity. With the prime chosen, the number 8 is a 64^{th}root of unity and 4096 is a 16^{th} root of unity. This means that 64-point and 16-point FFTs can be done with shift operations rather than 64-bit by 64-bit multiplications. This is one of the most important techniques that reduce the complexity of the FFT engine significantly.

$$A(x) = \sum_{i=0}^{n-1} a_i r^i \; mod \, p \quad (4)$$

IV. VLSI Design for Large Number Multiplication

In Gentry-Halevi's implementation[2], even for the small setting with a lattice-dimension of 2,048, the determinant d has about 780,000 bits. It means we need to design a large-number multiplier. In our implementation, we choose the base b to be 24. Therefore, every sample has 24 bits and a total of 32K samples. As we know, the multiplication of two numbers is similar to the cyclic convolution result of two

signal each with 32K samples. Typically, cyclic convolution involves "zero padding" and the result contains approximately twice many samples as that of the input signal. For a 780K-bits multiplication, we need to apply 64K-point FFT. As described in Section 3.1, the FFT-based Strassen multiplication algorithm requires the FFT of each of the operands A and B and also it requires an IFFT operation on the dot-product of *FFT*(*A*) and *FFT*(*B*). The implementation of the 64K-point FFT consists of 2 stages, including a 4K-points FFT and a 16-points FFT. The detail design of each of the stages is described below.

A. Radix-64 Unit

In the selected finite field, the number 8 is a 64^{th}root of unity. The 64-point FFT can be computed without using multiplications as in (5).

$$A(x) = \sum_{i=0}^{63} a_i 8^i = \sum_{i=0}^{63} a_i 2^{3*i} \bmod p \tag{5}$$

Since $8^{64} \bmod p = 2^{192} \bmod p = 1$, so all the intermediate result in the 64-point FFT can not be exceed 192-bit. In this implementation, no multiplication is necessary and only shift and modulo addition are required. The 192-bit addition would generate a long carry chain in traditional carry-ripple adder, which would slow down the clock speed of the entire design. In order to solve the problem, carry-save adder is used instead. Given three n-bit numbers a, b and c, it produces a partial sum ps and a shift-carry sc as shown in Fig. 1, where $ps_i = a_i \oplus b_i \oplus c_i$ and $sc_i = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$. The carry-save adder performs simple boolean operations, which few resources. More importantly, a carry-save adder support high-performance calculations because we could do shifting at a high speed, and the carry save adder would do the additions at a high speed without waiting for the carries to propogate, and we could hold off the normalization back down to 64 bits until the end of the 64 points FFT instead of doing modulo addition after every step. The architecture for processing element is shown in Fig. 1. The processing element is pipelined. Eight samples can be fed into the processing element consecutively, presenting compact and regular data flow without data hazard and bubble cycle. The architecture for radix-64 unit is shown in Fig. 2. The radix-64 unit includes 64 shift registers to shift the samples and 64 processing element to accumulate the intermediate results. In each cycle, eight samples are fed to the radix-64 unit to be shifted and accumulated consecutively. For every eight cycles, one 64-point FFT result is computed with several cycles of pipeline delay.

B. Radix-16 Unit

For 16-point FFT, the number 4096 is a 16^{th}root of unity. The equation below can be used to calculate 16-point FFTs.

$$A(x) = \sum_{i=0}^{15} a_i 4096^i = \sum_{i=0}^{15} a_i 2^{12*i} \bmod p \tag{6}$$

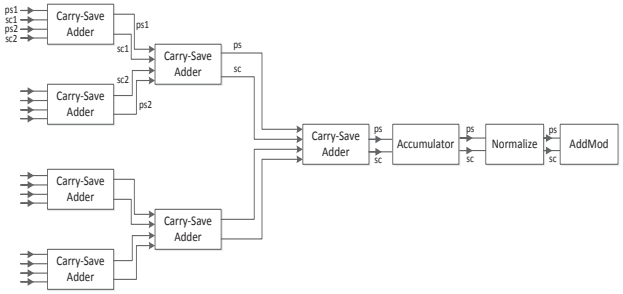

Figure 1. Architecture of processing element in radix-64 unit.

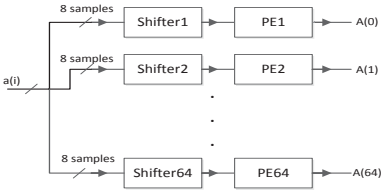

Figure 2. Architecture of the radix-64 unit.

The architecture of the processing element in the radix-16 unit is similar to the PEs in the radix-64 unit. The main difference is that more carry-save adders are used in the processing element so that 16 samples can be read and accumulated. Applying pipeline technique, the processing element can keep on reading 16 input samples and accumulating them continuously. For the radix-16 unit, it includes 16 shifters and 16 PEs so that the results of 16-point FFT can be completed and output every cycle.

C. 64K-point FFT

For the design of 64K-point FFT, the idea is to build it with the 64-point and 16-point FFTs with shifters and adders rather than multipliers. The equation belows describes the exploitation of the 64K point FFT as a radix-16 processor combined with a radix-4096 processor, which can be processed using a radix-64 unit. The DFT equation for a 64K-point FFT takes the form

$$X(k) = \sum_{n_1=0}^{16} W_{16}^{n_1 k_1} \Big(\sum_{n_2=0}^{4095} x[n] W_{4096}^{n_2 k_2} \Big) W_{16*4096}^{n_1 k_2}. \tag{7}$$

Furthermore, the 4K-point FFT can be calculated using the equation below.

$$X(k) = \sum_{n_1=0}^{63} W_{64}^{n_1 k_1} \Big(\sum_{n_2=0}^{63} x[n] W_{64}^{n_2 k_2} \Big) W_{4096}^{n_1 k_2}. \tag{8}$$

The radix-64 unit in Fig. 2 is used for the 4K-point FFT computation. The architecture of the 4K-point FFT processor is shown in Fig. 3. In this architecture, there are two memory banks and each of them has 64 subbanks in order to read/write data from/to the radix-64 unit. It is assumed that the data

are already loaded into *SRAM1s* on the left. During the first phase, the data are read from *SRAM1s* and fed to the Radix-64 Unit. Then the data calculated by Radix-64 Unit are written into *SRAM4Ks*. During the second phase, the data are read from *SRAM4Ks* and multiplied by the twiddle factor that was preloaded in a ROM. The multiplication results are fed to the Radix-64 Unit again. The data calculated by Radix-64 Unite are then written into *SRAM1s*.

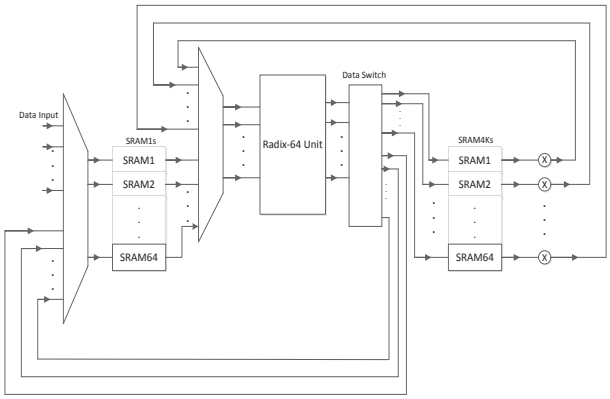

Figure 3. Architecture of the 4096-point FFT engine

The architecture for 64K-point FFT is depicted in Fig. 4. For the 64K-point FFT, we need to do 16 consecutive 4K-point FFT at the first stage. The first 4K data samples are read from SRAM1s and fed into the 4K-point FFT processor. Then the results of the 4K-point FFT are written back to SRAM1s. For the second 4K-point FFT, the data are read from *SRAM2s*, processed by 4K-point FFT processor, and written back to *SRAM2s*. This same procedure repeats until all 16 groups of 4K-point FFT is completed.

The next phase is to perform the 16-point FFT on the data. A set of 16 data can be read from 16 memory banks, each from one memory bank with 64 small *SRAMs*. Then the data are multiplied by the twiddle factors that are preloaded in ROMs. The multiplication results are fed into the radix-16 unit. The output of the radix-16 unit are the final results of the 64K-point FFT computation.

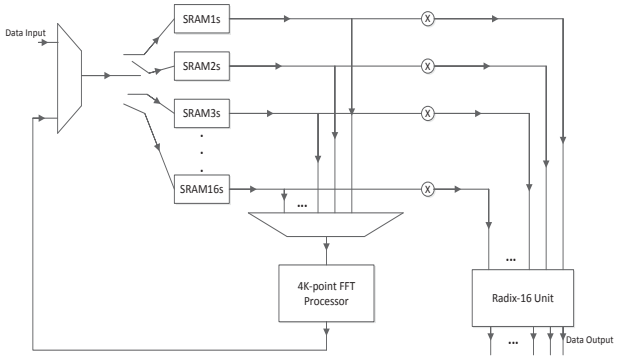

Figure 4. Architecture of the 64K-point FFT engine

V. Experimental Results

The proposed architecture are designed using SystemVerilog and prototyped on an FPGA. The design is targeted for Stratix-V 55GSMD8N3F45I4 FPGA using Altera synthesize tool. After place and route, the resource utilization of the 64K-FFT co-processor is shown in Table 1.

Table I
Device Utilization Summary

Logic Utilization	Used	Utilization
Combinational ALU	462983	88%
Dedicated Logic Register	336377	31%
DSP Blocks	720	37%

The same Strassen FFT multiplication algorithm is also implemented on NVIDA Tesla C2050 GPU, which has 448 cores running at 1.15 GHz. The execution time for 64K-point finite field FFT is about 0.26 ms. The FPGA implementation running at 100 MHz show the execution time of about 0.125 ms, which is about twice as fast as the powerful GPU. Further optimization of the critical path can lead to even higher clock speed, thus a greater speedup factor can be expected.

VI. Conclusion and Future Work

In this paper, the Strassen algorithm is used for the design of a large-number multiplier for fully homomorphic encryption. In the architecture of a 768K(786,432)-bits multiplier, a 64K-point finite field FFT is the key component. Several optimizing strategies are used to optimize the calculation of FFT in finite field by choosing a proper prime. The 4096-point FFT and 16-point FFT blocks are combined to perform the task. The FFT processor is implemented on Altera Stratix-V FPGA. When the FPGA running at 100 MHz, it is twice as fast as the comparable implementation on a C2050 GPU with 448 cores running at 1.15 GHz. As an on-going effort towards the design of the FHE processor for cloud computing, our initial exploration on the design of a large-number multiplier shows the hardware acceleration is promising. When compared with GPU acceleration, the FPGA-based design has much lower power consumption and better performance.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*. ACM, 2009, pp. 169–178.

[2] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *Advances in Cryptology–EUROCRYPT 2011*, pp. 129–148, 2011.

[3] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using gpu," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–5.

[4] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[5] A. Schönhage and V. Strassen, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, 1971.

[6] J. Solinas, "Generalized mersenne numbers," *Technical Reports*, 1999.

[7] N. Emmart and C. Weems, "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.