VLSI Design of a Large-Number Multiplier for Fully Homomorphic Encryption

Wei Wang, Xinming Huang, *Senior Member, IEEE*, Niall Emmart, and Charles Weems

***Abstract*— This paper presents the design of a power- and area-efficient high-speed 768 000-bit multiplier, based on fast Fourier transform multiplication for fully homomorphic encryption operations. A memory-based in-place architecture is presented for the FFT processor that performs 64 000-point finite-field FFT operations using a radix-16 computing unit and 16 dual-port SRAMs. By adopting a special prime as the base of the finite field, the radix-16 calculations are simplified to requiring only additions and shift operations. A two-stage carry-look-ahead scheme is employed to resolve carries and obtain the multiplication result. The multiplier design is validated by comparing its results with the GNU Multiple Precision (GMP) arithmetic library. The proposed design has been synthesized using 90-nm process technology with an estimated die area of 45.3 mm^2. At 200 MHz, the large-number multiplier offers roughly twice the performance of a previous implementation on an NVIDIA C2050 graphics processor unit and is 29 times faster than the Xeon X5650 CPU, while at the same time consuming a modest 0.97 W.**

***Index Terms*— Fully homomorphic encryption (FHE), large-number multiplication, VLSI design.**

I. Introduction

THE growth of cloud computing is deepening concerns over data privacy, especially when outsourcing computations to an untrusted service, which typically involves allowing the service to work with client data in decrypted form. Fully homomorphic encryption (FHE) is a technique enabling computation to be performed directly on encrypted data, thereby preserving privacy. The Gentry–Halevi scheme was the first software implementation of FHE but its computing latency is prohibitive for practical applications due to its intensive use of large-number (hundreds of thousands of bits) multiplications. Subsequent research has shown that performance can be improved through the use of parallelism on a general purpose graphics processor unit (GPU). However, the 200–400-W power consumption of a typical GPU makes it impractical to employ such an approach at data center scales. Because multiplication is the dominating component of FHE operations, it will be a significant step toward practical application of FHE if a high-performance, low-power, area efficient, and high precision integer multiplier architecture can be developed.

Manuscript received February 11, 2013; revised July 14, 2013 and August 10, 2013; accepted August 26, 2013. Date of publication November 1, 2013; date of current version August 21, 2014. This work was supported by the National Science Foundation under Award CCF-1217590.

W. Wang and X. Huang are with the Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA 01609 USA (e-mail: weiwang@wpi.edu; xhuang@wpi.edu).

N. Emmart and C. Weems are with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003 USA (e-mail: nemmart@yrrid.com; weems@cs.umass.edu).

Digital Object Identifier 10.1109/TVLSI.2013.2281786

Enabling general purpose computation on encrypted data was a problem introduced in [1] about three decades ago. In a major breakthrough, Gentry [2] introduced the first plausible FHE scheme in 2009. Using FHE, one may perform arbitrary computations directly on encrypted data without having access to the secret key. Thus an untrusted party, such as a cloud server, may perform computations on behalf of a client without compromising privacy. This property of FHE makes it potentially very valuable for the fast-growing cloud computing industry.

Although Gentry's FHE scheme is theoretically promising, it has been impractical for actual deployment. For instance, Gentry and Halevi [3] presented the first implementation of an FHE variant using software. They used sophisticated optimizations to reduce the size of the public key and the computing time of the large-number primitives based on the GMP library [4]. For the lowest security setting of dimension 2048, every source bit becomes encrypted as about 760 000 bits. The encryption of one bit took 1.69 s on a high-end Xeon processor, while the recryption primitive took 27.68 s. After every few bit-AND operations, a recrypt operation must be applied to reduce the noise in the ciphertext to a manageable level, thus inducing significant overhead.

Subsequently, we took Gentry and Halevi's FHE algorithm and accelerated it on a GPU platform [5]. Targeted to an NVIDIA C2050 GPU with 448 cores running at 1.15 GHz, the processing time for 1-bit encryption was reduced to 45 ms and the recyption was reduced to 1.8 s, which are about 37.6 and 15.4 times faster than the original implementation on the CPU. Although the GPU trial provided significant acceleration, the major problem remains that the power consumption of a high-end GPU today is about 200–400 W. Using GPUs to scale FHE up to data center levels is thus infeasible. The solution is to build low-power customized circuits that can provide comparable or superior performance to the fastest GPU while reducing power consumption by orders of magnitude.

Previously, the general-purpose GPU had also been used for acceleration of security algorithms such as elliptic curve cryptography [6]. But the GPU architecture was originally geared for graphics operations and later extended for general-purpose computations. It is not the most power-efficient architecture for a specific algorithm or applications. One approach is to attach an application-specific integrated circuit (ASIC) to the CPU which is dedicated to encryption/decryption operations. At the microarchitectural level, it can be implemented as an extension of the instruction set. Previously, customized ASIC or IP blocks have been designed to accelerate the well-known encryption schemes such as the Advanced

Encryption Standard (AES) and Rivest-Shamir-Adleman (RSA) [7], [8]. Today, many embedded processors have AES or RSA cores included. This paper is aimed at taking a similar approach and designing a specific hardware or IP blocks for accelerating the core computations in FHE.

Since the most computationally intensive operations in the FHE primitives are large-number modular multiplications, our initial attempt is to tackle the design of a large-number multiplier that can handle 768 000 bits, in support of the 2048-dimension FHE scheme demonstrated by Gentry and Halevi. In addition to FHE, large-number arithmetic also has other important applications in science, engineering, and mathematics. Specifically, when we need exact results or the results that exceed the range of floating point standards, we usually turn to multiprecision arithmetic [9]. An example application is in robust geometric algorithms [10]–[12]. Replacing exact arithmetic with fixed-precision arithmetic introduces numerical errors that lead to nonrobust geometric computations. High-precision arithmetic is a primary means of addressing the nonrobustness problem in such geometric algorithms [10].

The rest of this paper is organized as follows. Section II gives a brief introduction to FHE. Section III presents Strassen's fast Fourier transform (FFT) based multiplication algorithm. Section IV shows the architecture of the VLSI design of the finite-field FFT engine and the multiplier. Section V gives results based on VLSI synthesis and simulation. Conclusions follow in Section VI.

II. Gentry's FHE

One of the holy grails of modern cryptography is FHE, which allows arbitrary computation on encrypted data. Given a need to perform a binary operation on the plaintext, FHE enables that to be accomplished via manipulation of the ciphertext without the knowledge of the encryption key. For example, $E(x_1)+E(x_2) = E(x_1+x_2)$ and $E(x_1)\star E(x_2) = E(x_1 \otimes x_2)$.

The first FHE was proposed by Gentry in [2] and [13] and was seen as a major breakthrough in cryptography. However, its preliminary implementation is too inefficient to be used in any practical applications. A number of optimizations were used in the Gentry–Halevi FHE variant, and the results of a reference implementation were presented in [3]. Due to limited space, here we only provide a high-level overview of the primitives.

Encryption: To encrypt a bit $b \in \{0, 1\}$ with a public key (d, r), encryption first generates a random "noise vector" $u = \langle u_0, u_1, \ldots, u_{n-1}\rangle$, with each entry chosen as 0 with the probability p and as ± 1 with probability $(1-p)/2$ each. Gentry [3] showed that u can contain a large number of zeros without impacting the security level, i.e., p could be very large. A message bit b is then encrypted by computing

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

where d and r are parts of the public key. For the small setting with a lattice dimension of 2048, d and r have a size of about 785 000 bits [3].

When encrypted, arithmetic operations can be performed directly on the ciphertext with the corresponding modular operations. Suppose $c_1 = \text{Encrypt}(m_1)$ and $c_2 = \text{Encrypt}(m_2)$; then we have

$$\text{Encrypt}(b_1 + b_2) = (c_1 + c_2) \bmod d \tag{2}$$

$$\text{Encrypt}(\text{b}_1 * \text{b}_2) = (c_1 * c_2) \bmod d. \tag{3}$$

Decryption: The source bit b can be recovered by computing

$$b = [c \cdot w]_d \bmod 2 \tag{4}$$

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

Recryption: 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. But from the brief description above, we can see that the fundamental operations for FHE are large-number addition and multiplication. Addition has far less computing complexity than multiplication, so we focus on the hardware architecture of the multiplication using VLSI design.

III. Large-Integer Modular Multiplication

A. Multiplication Algorithms

Large-integer multiplication is by far the most time-consuming operation in the FHE scheme. Therefore, we have selected it as the first block for hardware acceleration. A review of the literature shows that there is a hierarchy of multiplication algorithms. The simplest algorithm is the naive $O(N^2)$ algorithm (often called the grade school algorithm).

The first improvement to the grade school algorithm was due to Karatsuba [14] in 1962. It is a recursive divide-and-conquer algorithm, solving an N bit multiplication with three $N/2$ bit multiplications, giving rise to an asymptotic complexity of $O(N^{\log_2 3})$. Toom and Cook generalized Karatsuba's approach, using polynomials to break each N bit number into three or more pieces. Once the subproblems have been solved, the Toom–Cook method uses polynomial interpolation to construct the desired result of the N bit multiplication [15]. The asymptotic complexity of the Toom–Cook algorithm depends on k (the number of pieces) and is $O(N^{\log(2k-1)/\log(k)})$.

The next set of algorithms in the hierarchy are based on using FFTs to compute convolutions. According to Knuth [15], Strassen came up with the idea of using FFTs for multiplcation in 1968, and worked with Schönhage to generalize the approach, resulting in the famous Schönhage–Strassen algorithm [16], with an asymptotic complexity of $O(N \cdot \log N \cdot \log\log N)$.

All the operations in FHE are modular operations. Usually, two different approaches are used to address the modular multiplication. The first is to do multiplication first, followed by modular reduction. The other approach, proposed in [17], interleaves the multiplication with modular reduction. This is an efficient grade-school approach, performing the equivalent of two $O(N^2)$ multiplications. The interleaved Montgomery approach is quite commonly used for modular multiplication in the RSA algorithm, see [8] and [18].

TABLE I
OPERATION COUNTS FOR A 786 432-bit MODULAR MULTIPLICATION

operation	Interleaved Montgomery	Karatsuba	FFT
dmultu	302,002,176	26,138,787	2,083,530
mflo/mfhi	603,992,064	52,277,574	4,167,060
daddu/dsubu	1,207,898,112	249,505,992	49,345,563
ddrl/dsll	0	0	14,112,477
and/or	0	0	4,639,830
sltu	603,942,912	155,445,660	25,947,906
movz/movn	0	0	25,947,906
load immediate	0	0	1,886,934
TOTAL	2718.0 M	483.4 M	128.1 M

To understand the arithmetic cost of different multiplication algorithms, we implement three different modular multiplication algorithms in carefully tuned MIPS 64 assembly and count the number of ALU operations for each. The first algorithm uses the interleaved version of Montgomery multiplication proposed in [17]. This is an efficient grade-school approach, performing the equivalent of two $O(N^2)$ multiplications. The second algorithm uses the noninterleaved three-multiplication Montgomery reduction implemented with Karatsuba multiplication (it uses the Karatsuba method if the arguments are larger than three words, and switches to grade-school multiplication to handle the base case when the arguments are small). The third algorithm adopted in this paper is based on FFT multiplication and is described in detail in the next section. This algorithm also uses a traditional three-multiplication Montgomery reduction. The operation counts of the three algorithms are presented in Table I.

Comparing the Karatsuba and FFT multipliers, both of which compute the product and then reduce the result modulo N, we can see that FFT multiplication is faster, requiring only one-third of the number of instructions as the Karatsuba multiplier. Comparing the FFT multiplier with interleaved Montgomery approach which is widely used in RSA for modular multiplication, we see that the FFT multiplier uses only 1/20th of the number of instructions. The interleaved version of Montgomery multiplication is popular and efficient in RSA, but it is no longer efficient for the modular multiplication in FHE. In all, the approach we adopt for modular multiplication is the most efficient algorithm. From above, we can see that large-number multiplication is the most crucial part for the modular multiplication. Therefore, we take the first step to design a fast multiplier for hardware implementation.

For further reading, there are a number of papers that cover hardware implementation of large-number multiplication. Yazaki and Abe [19] implement a 1024-bit Karatsuba multiplier, and in [20] they investigate a hardware implementation of FFT multiplication. Kalach [21] investigates a hardware implementation of finite field FFT multiplication. However, that paper does not present any information about the hardware resources and performance.

B. FFT Multiplication

FFT multiplication is based on convolutions. For example, to compute the product A times B, we express the numbers A and B as sequences of digits (in some base b) and then compute the convolution of the two sequences using FFTs.

1: Decompoisition: Break the large numbers A and B into a series of words $a(n)$ and $b(n)$.

2: FFT: Compute finite−field FFT of $a(n)$ and $b(n)$, treating each word in the series as FFT samples in the time domain

3: Component-wise Product: Compute the component−wise product of the FFT results:
$C[i]$ = FFT$(A)[i]*$FFT$(B)[i]$

4: IFFT: Compute the inverse FFT of $C[i]$:
$c(n)$=IFFT(C)

5: Re-assembly: Resolve the carries, when $c[i] \geq b$:
set $c[i+1] = c[i+1] + (c[i]$ div $b)$
set $c[i] = c[i]$ mod b

Fig. 1. FFT multiplication.

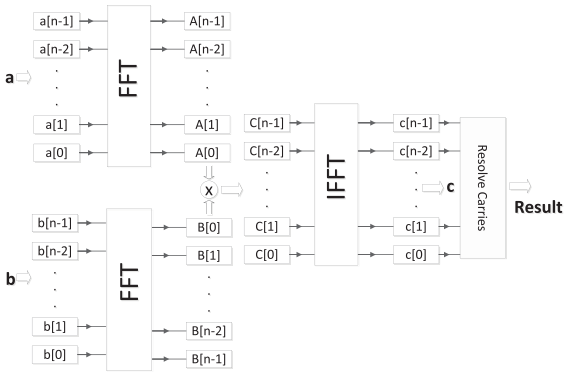

Fig. 2. FFT-based multiplication algorithm.

Once we have the convolution of the digits, the product A times B can be found by resolving the carries between digits. The FFT multiplication algorithm is presented in Fig. 1 and as a diagram in Fig. 2.

The FFT computations can done either in the domain of complex numbers or in a finite field or ring. In the complex number domain, it is trivial to construct the roots of unity required for the FFT, but the computations must be done with floating point arithmetic and the round-off error analysis is quite involved. In the finite field/ring case, all the computations are done with integer arithmetic and are exact. However, the existence and the calculation of the required root of unity will depend heavily on the structure of the chosen finite field/ring.

For our FFT multiplier, we follow the steps of our previous work [9] and implement the FFT in the finite field $\mathbf{Z}/p\mathbf{Z}$, where p is the prime 2^{64}–$2^{32}+1$. This prime is from a special class of numbers called Solinas primes (see [22]). As we shall see, this choice of p has three compelling advantages for FFTs.

1) We can do very large FFTs in $\mathbf{Z}/p\mathbf{Z}$. Since 2^{32} divides $p-1$, we can do any power-of-2-sized FFT up to 2^{32}.
2) There exists a very fast procedure for computing x modulo p for any x.
3) For small FFTs (up to size 64), the roots of unity are all powers of 2. This means that small FFTs can be done entirely with shifting and addition, rather than requiring expensive 64-bit multiplications.

C. FFTs in the Finite Field $\mathbf{Z}/p\mathbf{Z}$

To perform FFTs in a finite field, we need three operators: addition, subtraction, and multiplication, all modulo p, where $p = 2^{64}–2^{32}+1$. Addition and subtraction are straightforward (if the result is larger than p then subtract p, and if the result is negative, then add p). For multiplication, if X and Y are in $\mathbf{Z}/p\mathbf{Z}$, then $X * Y$ will be a 128-bit number, which we can represent as $X * Y = 2^{96}a + 2^{64}b + 2^{32}c + d$ (where a, b, c, and d are each 32-bit values). Next, using two identities of p, namely, $2^{96} \bmod p = -1$ and $2^{64} \bmod p = 2^{32} - 1$, we can rewrite the product of $X * Y$ as

$$\begin{aligned} X * Y &\equiv 2^{96}a + 2^{64}b + 2^{32}c + d \pmod{p} \\ &\equiv -1(a) + (2^{32} - 1)b + (2^{32})c + d \\ &\equiv (2^{32})(b + c) - a - b + d. \end{aligned}$$

This means that a 128-bit number can be reduced modulo p to just a few 32-bit additions and subtractions.

Further, note that $2^{192} \bmod p = 1$, $2^{96} \bmod p = -1$, $2^{384} \bmod p = 1$, etc. This leads to a fast method to reduce any sized value modulo p. Break the value up into 96-bit chunks and compute the alternating sum of the chunks. Then reduce the result as above.

In addition to the arithmetic operator, there are three other criteria in order to perform multiplication with finite field FFTs. First, to compute an FFT of size k, a primitive root of unity r_k must exist such that $r_k \hat{} k \bmod p = 1$ and $r_k \hat{} i \bmod p \neq 1$ for all i between 1 and $k-1$. Second, the value k^{-1} must exist in the field. Third, we must ensure that the convolution does not overflow, i.e., $k/2(b-1)^2 < p$, where k is the FFT size and b is the base used in the sampling. Finally, we must ensure that the numbers we are multiplying are less than $b^{k/2}$.

In a finite field, the process for doing an FFT is analogous to FFTs in the complex domain; thus

$$X_i = \sum_{j=0}^{k-1} x_j (r_k)^{ij} \pmod{p}. \tag{5}$$

The inverse FFT (IFFT) is just

$$x_i = k^{-1} \sum_{j=0}^{k-1} X_j (r_k)^{-ij} \pmod{p} \tag{6}$$

for all the usual methods for decomposing FFTs, such as Cooley–Tukey [23], except $(r_k)^j$ takes the place of $e^{j2\pi i/k}$.

With large FFTs, the primitive roots almost always look like random 64-bit numbers; e.g., the $r_{65\,536}$ that we use is 0xE9653C8DEFA860A9. However, for FFTs of size 64 or less, the roots of unity will always be powers of 2. As noted above, $2^{192} \bmod p = 1$, which means $(2^3)^{64} \bmod p = 1$ and therefore $r_{64} = 2^3 = 0 \times 08$. Likewise, $r_{16} = 2^{12}$.

For our hardware implementation, we will choose $k = 65\,536$ and $b = 2^{24}$. These values meet the criteria above and allow us to multiply two numbers up to $b^{k/2} = 2^{786\,432}$, i.e., 786 432 bit in length, which is sufficient to support Gentry–Halevi's FHE scheme for the small setting with a lattice dimension of 2048.

D. 192-bit Wide Pipelines

It is often the case in our hardware FFT implementation that we need to perform a sequence of modular operations (additions, subtractions, and multiplications by powers of 2).

If we were to implement this as 64-bitwide operations, we would need to reduce the result modulo p between each stage of the pipe. Although the process to reduce a value modulo p is quite fast, it still requires a lot of hardware. It turns out that, if we extend each 64-bit value to 192 bits (by padding with zeros on the left) and run the pipeline with 192-bitwide values, then we can avoid the modulo p operations after each pipeline stage by taking advantage of the fact that $2^{192} \bmod p$ is 1. We do this as follows:

1) Addition: Suppose we wish to compute $x + y$. There are two cases: If we get a carry out from the 192nd bit, then we have $\text{trunc}(x+y) + 2^{192}$, which is the same as $\text{trunc}(x+y)+1$ modulo p (where trunc(z) returns the least significant 192 bits of z). If it did not carry out, then the result is just $x + y$. We can implement this efficiently in hardware using circular shifting operations.

2) Multiplication by a Power of 2: First, let us consider multiplication by 2. Suppose we have a 192-bit value x and we wish to compute $2x$. There two cases. If the most significant bit of x is zero, then we simply shift all 1-bits to the left. If the top bit is set, then we need to compute $\text{trunc}(2x) + 2^{192}$, which is the same as $\text{trunc}(2x) + 1$ modulo p. In both case, it is just a left circular shift by 1 bit. Thus to compute $2^j * x$, we simply do a left circular shift by j bits.

3) Subtraction: Since $2^{96} \bmod p = -1$, we can simply rewrite $x - y$ as $x + 2^{96}y$. The 2^{96} is a constant shift.

For the final reduction from 192 bits back down to 64 bits, as above, we can represent a 192-bit number z as $z = 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f$, where a, b, c, d, e, and f are each 32 bits

$$\begin{aligned} z &\equiv 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f \\ &\equiv -(2^{32} - 1)a - 2^{32}b - c + (2^{32} - 1)d + 2^{32}e + f \\ &\equiv (2^{32}e+f)+(2^{32}d+a)-(2^{32}b+c)-(2^{32}a+d). \end{aligned} \tag{7}$$

IV. VLSI Design of the Large-Number Multiplier

For high-throughput applications, a pipelined FFT architecture is often used [24]. However, the pipelined design requires a memory buffer at every stage [24], which becomes problematic in the context of large-integer operations. For a 64 000 FFT and 64 bits per data sample, we would need 4 Mbits of memory after each stage. Generally, a large FFT involves numerous stages, which makes the total area for memory too large to be considered for hardware implementation.

In contrast to the pipelined FFT design, a memory-based FFT architecture adopts an in-place strategy, which allows us to store the intermediate results into the same memory as the input data. Doing so effectively minimizes the memory requirement for the FFT computation [25]. To improve throughput, multiple memory banks can be used for parallel access. In our 64 000 FFT architecture, a total of 16 dual-port memory banks are used, and each memory bank is 256 000 bits in size. Fundamentally, the 64 000 FFT is implemented using

four stages of 16-point FFTs. The basic concept of a stage is to perform 4096 16-point FFTs, followed by application of twiddle factors and then transposition. If we repeat that process four times ($16^4 = 64\,000$), then the result is a 64 000 FFT. Using an in-place memory-based design, these four stages are computed sequentially using the same hardware unit and memory.

A. Radix-16 FFT Unit

One of the key elements of our design is a high-throughput 16-point FFT engine. As discussed in Section III-C, for small ($k \leq 64$) FFTs, the root of unity will always be a power of 2.

In a finite field based on the Solinas prime p, a 16-point FFT can be performed using just shift and modulo addition operations. A 16-point FFT can be expressed as (8), noting $4096^{16} \bmod p = 2^{192} \bmod p = 1$. As discussed above, for 192-bit operations, any carry-out bit can be simply routed back as a carry-in bit, which is particularly suitable for hardware design

$$X(k) = \sum_{n=0}^{15} x(n) 2^{12 \cdot nk \% 192} \bmod p \tag{8}$$

$$x(n) = \frac{1}{16} \sum_{k=0}^{15} X(k) 2^{(192-12nk) \% 192} \bmod p. \tag{9}$$

For a 192-bit addition, a traditional ripple-carry adder would generate a long carry chain and slow the clock speed considerably. Thus we employ carry-save adders as the basis for our high-speed design. Given three n-bit numbers a, b, and c, the carry-save approach produces a partial sum ps and a shift-carry sc, where $\text{ps}_i = a_i \oplus b_i \oplus c_i$ and $\text{sc}_i = \text{BarrelLeftShifter}((a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i), 1)$. We can cascade 2 three-input carry-save adders to form a four-input adder. A diagram of the sum-16 unit is shown in Fig. 3. The summation unit is a pipeline architecture that takes 16 inputs every clock cycle. A normalization unit at the end performs a modulus p operation shown in (7) and converts the 192-bit result back to 64 bits.

The architecture for a radix-16 finite field FFT unit is shown in Fig. 4. It consists of 16 shifters and 16 summation units. At each clock cycle, the radix-16 unit takes 16 data inputs and outputs the 16-point FFT results after a few cycles of pipeline delay.

B. 64 000-Point FFT Processor

The 64 000-point FFT can be decomposed into four stages of 16-point FFTs. At each stage, a total of 64 000 samples are processed through the radix-16 FFT unit. At 16 samples per cycle, that gives a total of 4096 cycles per stage. This architecture reads 16 input values from memory and writes 16 output values to the memory every clock cycle. Therefore, the memory needs to be partitioned into 16 banks. An in-place memory addressing scheme is applied to ensure there is no memory access conflict. With reference to the derivation in [25] and [26], a conflict-free in-place scheme for radix-16

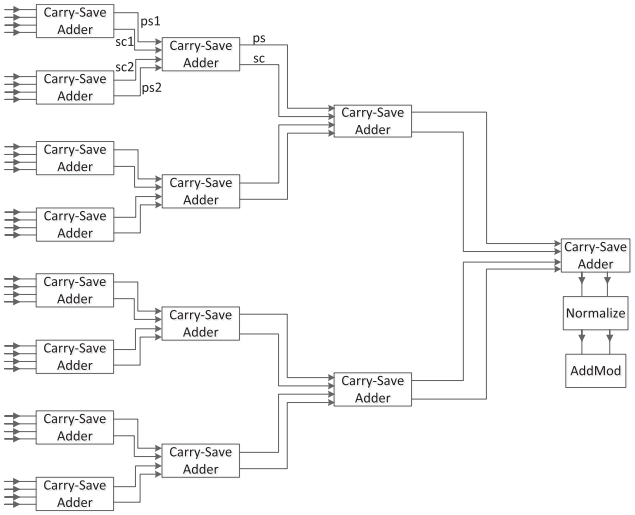

Fig. 3. Diagram of sum-16 unit.

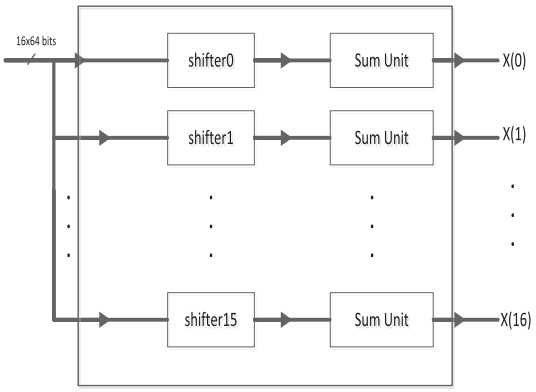

Fig. 4. Architecture of the radix-16 FFT unit.

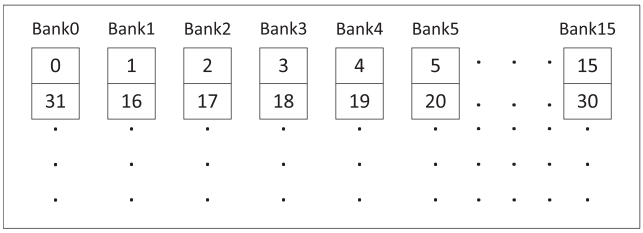

Fig. 5. Data storage pattern in the memory banks.

FFT can be described as follows:

$$\text{DataCount} = [d_{15}, d_{14}, \ldots, d_0] \tag{10}$$

$$\text{BankNum} = ([d_{15}, d_{14}, d_{13}, d_{12}] + [d_{11}, d_{10}, d_9, d_8] + [d_7, d_6, d_5, d_4] + [d_3, d_2, d_1, d_0]) \bmod 16 \tag{11}$$

$$\text{Address} = [d_{15}, d_{14}, \ldots, d_4]. \tag{12}$$

DataCount denotes the original address of the input data sample. BankNum is the corresponding bank assignment after partitioning. Address is the new address in the assigned bank. For 64 000 samples, the memory is partitioned into 16 banks, and each bank has 4096 samples. The data storage pattern in the memory banks is shown in Fig. 5.

The overall architecture of the FFT processor is shown in Fig. 6. Before entering the processor, the data has been

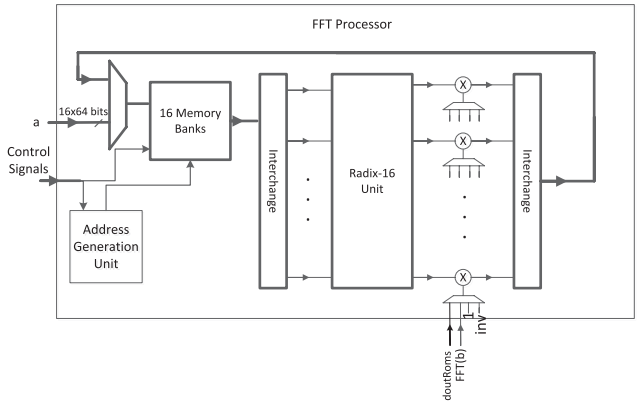

Fig. 6. Architecture of the 64 000-point FFT processor.

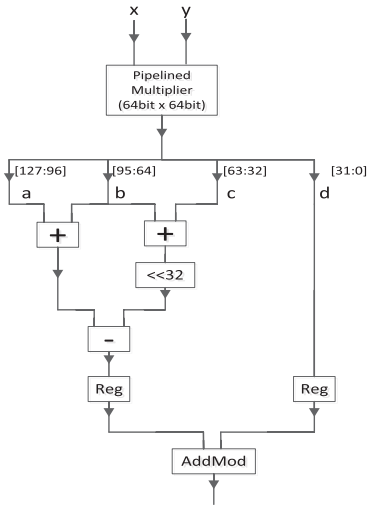

Fig. 7. Architecture of modular multiplication unit.

reshuffled according to (11) and (12). The Address generation unit generates the corresponding bank number and address for each data sample. After all 64 000 samples have been received and stored in the memory banks, the FFT processor begins the computation. At each clock cycle, it reads 16 samples from the memory banks according to the BankNo and Address generated by the address generation uinit. These 16 values are then permuted into a proper order by the interchange unit and fed to the Radix-16 unit. Subsequently, the radix-16 FFT results are modular-multiplied with twiddle factors supplied from ROMs. The final results of each stage are permuted to the desired order before being stored back into the memory banks.

The modular multiplier is designed as shown in Fig. 7. The 64-bit multiplier has four pipeline stages. The 128-bit multiplication result is then split into four 32-bit components a, b, c, and d. After going through the addition, shifting, and subtraction as in Fig. 7, a 64-bit modular multiplication result is obtained.

C. Large-Number Multiplier

The high-level architecture of the large-number multiplier is shown in Fig. 8. It consists of two FFT processors for computing the FFTs of the two inputs a and b. Then a

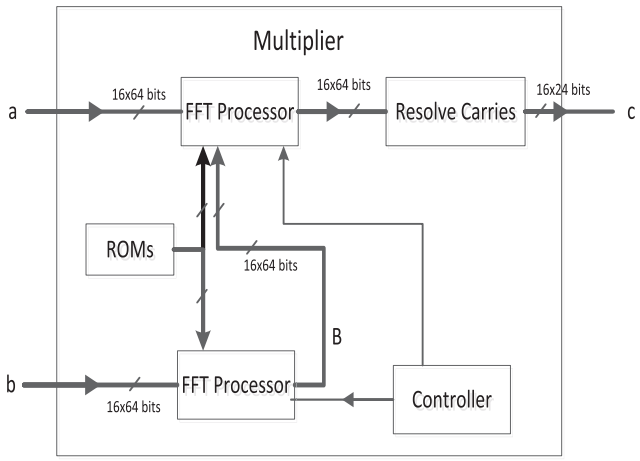

Fig. 8. Architecture of the large-number multiplier.

component-wise product is performed on the two FFT results. Subsequently, we reuse one of the FFT processors to perform the IFFT operation. The operations in each step are described as follows.

1) *Data Input:* The input data samples from a and b are reshuffled and stored in the corresponding addresses in the memory banks.
2) *FFT:* Two 64 000-point FFT processors are used in the architecture. To reduce the hardware needed, both FFT processors share the twiddle factor ROMs. They also share the control signals generated by the Controller.
3) *Component-Wise Product:* For the pointwise product, we reuse the modular multipliers in the FFT processor. Specifically, at the fourth stage of FFT(a), instead of multiplying by constant 1, the result of FFT(b) is fed to the modular multipliers. Effectively this computes the component-by-component pointwise product of FFT(a) and FFT(b). We thus avoid adding another set of multipliers into the design and thereby save chip area.
4) *IFFT*: One of the FFT processors is reused for the IFFT computation. This reuse effectively saves about one-third of the chip area.
5) *Resolve Carries:* A customized Resolve Carries unit produces the final result of large-number multiplication.

D. Resolve Carries

To further explain the process of resolving carries, we take the 768 000-bit Strassen's multiplier as an example. But note that the design approach is general. We first decompose each 768 000-bit multiplicand into 32 000 groups of 24-bit numbers. Each 24-bit number is then extended to a 64-bit data sample. Owing to the convolution property of multiplication, the multiplication results are expected to be 64 000 groups of 24-bit numbers, or up to 1 536 000 bits, which leads to the 64 000 FFT in the design. Following Strassen's algorithm, the IFFT output is 64 000 samples of 64-bit data. The Resolve Carries unit must then obtain the actual 1 536 000 bits results from the IFFT output data.

Since each group of data is supposed to be 24-bits, each 64-bit value in the IFFT output is actually overlapped by

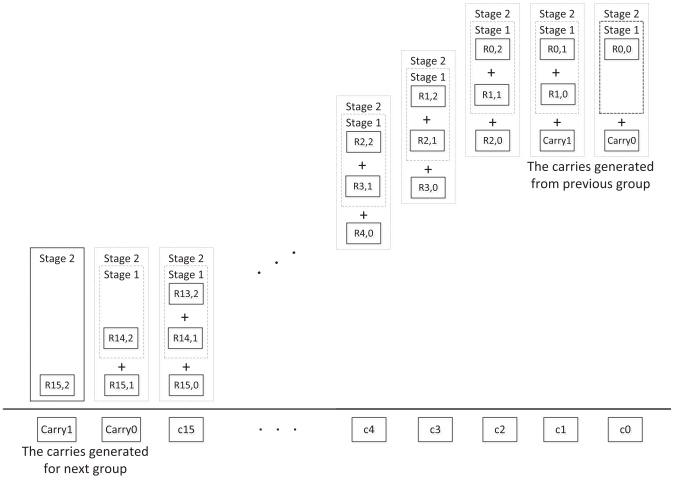

Fig. 9. Two-stage pipeline carry resolving unit.

40 bits with the next value. For our design, we extend the 64-bit numbers into a 72-bit format having three blocks of 24-bit numbers. The alignment among the words is illustrated in Fig. 9.

Recall that the IFFT module outputs 16 data samples per clock cycle. A total of 64 000 data values are output in 4096 consecutive cycles. Therefore, we must resolve the carries quickly to match the pipeline throughput. A traditional ripple-carry adder is again too slow to add 16 numbers in a row. Thus, a hierarchical carry-look-ahead scheme is employed as in Fig. 9. The algorithm has two steps. It first adds the words in parallel, followed by resolving the carry chain in one cycle [27]. The carry look-ahead function is shown in (13)

$$\text{carry} = ((c << 1) + \text{carryin} + \text{critical}) \text{ XOR critical} \quad (13)$$

where critical[i] and $c[i]$ are two Boolean arrays and carryin is a single carry bit from the previous word. If z_i is critical (z_i = MAX_INT), the ith bit of critical[i] is set, while the ith bit of $c[i]$ is set if z_i always generates a carry ($z_i >$ MAX_INT). For a 24-bit word, MAX_INT = 0 xFFFFFF. For a best performance, we use a two-stage pipeline design for the Resolve Carries unit as shown in Fig. 9. The carry-look-ahead scheme and two-stage pipeline enable the Resolve Carries unit to match the throughput of the FFT/IFFT processor output data at a high clock speed.

V. Experimental Results

The design of the large-number multiplier was implemented using System Verilog. The multiplier ASIC was synthesized for 90-nm technology, using the Synopsys Design Compiler, the DesignWare building block libraries, and IBM 90-nm CMOS 9 FLP standard-cell library. Table II lists the synthesis results for the radix-16 unit, the 64 000 FFT processor, and the multiplier. The number of logic equivalent gates (two-input NAND) of the chip is 20.6 M gates. A large portion of the chip area is occupied by the memories. For the large-number multiplier, we have two FFT processors, each of which has

TABLE II
Synthesis Results Using 90-nm CMOS Technology
(IBM 90-nm 9 FLP Process)

	Radix-16 unit	FFT processor	Multiplier
Core Area	2.2 mm^2	20.7 mm^2	45.3 mm^2
Dynamic Power	313.8 mW	562.2 mW	968.7 mW
Leakage Power	25.8 uW	202.68 uW	433.1 uW
Total Power	313.83 mW	562.4 mW	969.2 mW
Clock Frequency	200 MHz	200 MHz	200MHz
Core Voltage	1.32 V	1.32 V	1.32 V

TABLE III
Synthesis Results on Altera's Stratix-V FPGA

Logic Utilization	Device Utilization Summary		
	Used	Available	Utilization
Combination ALUTs	243,402	718,400	34%
Dedicated logic registers	245,257	1,436,800	17%
Total block memory bits	8.912,896	54,067,200	16%
Total DSP blocks	288	352	82 %
Maximum Frequency	229.4 MHz		

TABLE IV
Performance Comparison Between the
Proposed Design, CPU, and GPU

	Computing Time	Speedup factor
Intel Xeon X5650 processor	6 ms	1
NVIDA Tesla C2050 GPU	0.42 ms	14.5
The proposed Multiplier	0.206 ms	29

16 dual port SRAM banks of size 4096 × 64 bits. The estimated area of each SRAM is about 1.07 mm^2, so the total area for the SRAMs is about 34.24 mm^2. In addition, the FFT/IFFT processors also require a set of 30 ROMs to store the twiddle factors. Each ROM is 4096 × 64 bits with an estimated chip area of 0.154 mm^2. So the total area for the ROMs is about 4.63 mm^2. If combined, the total area for the RAMs and ROMs is about 38.87 mm^2, which occupies 85.8% of the chip. Thus, the architecture of the large-number multiplier is memory-constrained. In fact, the optimized radix-16 units occupy just 5% of the entire multiplier area. The proposed multiplier was also synthesized using Altera Quartus-II synthesize tool. After place and route, the design is implemented on Altera's Stratix-V 5SGXMABN1F45I2 field-programmable gate array (FPGA). The resources utilized by the multiplier are listed in Table III.

We validated the simulation results for the hardware multiplier against a software implementation using the GMP library [4]. Random numbers generated by C code were used as test vectors. The results match perfectly, thus showing that the architecture as well as the synthesized design of the large-number multiplier operates correctly.

For performance evaluation, we compare the throughput of our multiplier with the software implementations on CPU and GPU. The 768 000-bit multiplication was evaluated on a high-end server with an Intel Xeon X5650 processor running at 2.67 GHz with 24 GB RAM using the GMP library, which supports arbitrary precision arithmetic, and is carefully

designed using fast algorithms and highly optimized assembly code, as necessary [4]. The execution time on the CPU is about 6 ms. The same Strassen's multiplication algorithm was also implemented on an NVIDA Tesla C2050 GPU, which has 448 cores running at 1.15 GHz as in [5]. It takes 0.0657 ms to transfer a 786 432-bit number from Xeon processor to the GPU or transfer a 786 432-bit number from the GPU back to the Xeon processor. When the data has been transferred to the GPU, we measure the runtime of the GPU kernel, and then transfer the results back to the GPU. The GPU kernel execution time is 0.42 ms, excluding the data transfer time between CPU and GPU. For our hardware implementation, it takes 4096 cycles to load the samples into SRAMs, eight stages of FFT/IFFT with 4119 cycles per stage, and 4098 cycles to read the multiplication results out of the memory. At 200 MHz, the execution time of the VLSI implementation is 0.206 ms, which is twice as fast as the GPU and 29 times faster than the CPU as listed in Table IV. More importantly, the proposed VLSI implementation uses approximately 0.97 W, which is significantly less than either the GPU or CPU, making it more suitable for scaling up.

For comparison, Yazaki and Abe [20] implemented a 32 768-bit FFT-based multiplier in hardware in an area of 9.05 mm^2 using a 0.18-μm process. They achieved a runtime 1.02 ms for a 32 768-bit multiplication. Our multiplier handles numbers 24 times larger and at 5 times the speed.

VI. Conclusion

In this paper, an efficient VLSI implementation of a large-number multiplier was presented using Strassen's FFT-based multiplication algorithm. To the best of our knowledge, this is the largest multiplier that has been implemented using VLSI design. Because of memory constraints, a memory-based in-place FFT architecture was used for the FFT processor. A set of design optimization strategies were applied to improve the performance and reduce the area of both the Radix-16 unit and the Resolve Carries unit. The multiplier was synthesized for 90-nm technology with an estimated core area 45.3 mm^2. Experimental results showed that the proposed multiplier was about 2 times faster than GPU and 29 times faster than CPU, and its power consumption was less than 1 W.

References

[1] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 32, no. 4, pp. 169–178, 1978.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, Jun. 2009, pp. 169–178.

[3] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *Advances Cryptology–EUROCRYPT* (Lecture Notes in Computer Science). New York, NY, USA: Springer-Verlag, 2011, pp. 129–148.

[4] (2010). *The GNU Multiple Precision Arithmetic Library* [Online]. Available: http://gmplib.org/

[5] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. IEEE Conf. HPEC*, Sep. 2012, pp. 1–5.

[6] A. Cohen and K. Parhi, "GPU accelerated elliptic curve cryptography in GF(2^m)," in *Proc. 53rd IEEE Int. MWSCAS*, Aug. 2010, pp. 57–60.

[7] X. Zhang and K. Parhi, "High-speed VLSI architectures for the AES algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 9, pp. 957–967, Sep. 2004.

[8] M.-D. Shieh, J.-H. Chen, H.-H. Wu, and W.-C. Lin, "A new modular exponentiation architecture for efficient design of RSA cryptosystem," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 9, pp. 1151–1161, Sep. 2008.

[9] N. Emmart and C. C. Weems, "High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes," *Parallel Process. Lett.*, vol. 21, no. 3, pp. 359–375, Jul. 2011.

[10] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra, "Efficient exact geometric computation made easy," in *Proc. 15th Annu. Symp. Comput. Geometry*, 1999, pp. 341–350.

[11] C. K. Yap and V. Sharma, *Robust Geometric Computation*. New York, NY, USA: Springer-Verlag, 2008.

[12] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap, "A core library for robust numeric and geometric computation," in *Proc. 15th Annu. Symp. Comput. Geometry*, 1999, pp. 351–359.

[13] C. Gentry, "A fully homomorphic encryption scheme,"Ph.D. dissertation, Dept. Comp. Sci., Stanford Univ., Stanford, CA, USA, 2009.

[14] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Proc. USSR*, 1962, pp. 293–294.

[15] D. Knuth, *The Art of Computer Programming*, vol. 2. Reading, MA, USA: Addison-Wesley, 2006.

[16] A. Schönhage and V. Strassen, "Schnelle Multiplikation Großer Zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, 1971.

[17] P. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[18] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 3101–3109, Jul. 2011.

[19] S. Yazaki and K. Abe, "VLSI design of Karatsuba integer multipliers and its evaluation," *IEEE Trans. Electron., Inf. Syst.*, vol. 128, no. 2, pp. 220–230, Feb. 2008.

[20] S. Yazaki and K. Abe, "An optimum design of FFT multi-digit multiplier and its VLSI implementation," *Bull. Univ. Electro-Commun.*, vol. 18, no. 1, pp. 39–45, 2006.

[21] K. Kalach and J. P. David, "Hardware implementation of large number multiplication by FFT with modular arithmetic," in *Proc. 3rd Int. IEEE-NEWCAS Conf.*, Jun. 2005, pp. 267–270.

[22] J. Solinas, "Generalized mersenne numbers," Blekinge College Technol., Karlskrona, Sweden, Tech. Rep. 06/MI/006, 1999.

[23] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[24] L. Jia, Y. Gao, and H. Tenhunen, "A pipelined shared-memory architecture for FFT processors," in *Proc. 42nd IEEE Midwest Symp. Circuits Syst.*, vol. 2. Aug. 1999, pp. 804–807.

[25] L. Johnson, "Conflict free memory addressing for dedicated FFT hardware," *IEEE Trans. Circuits Syst. II, Analog Dig. Signal Process.*, vol. 39, no. 5, pp. 312–316, May 1992.

[26] H. Lo, M. Shieh, and C. Wu, "Design of an efficient FFT processor for DAB system," in *Proc. IEEE ISCAS*, vol. 4. May 2001, pp. 654–657.

[27] N. Emmart and C. Weems, "High precision integer addition, subtraction and multiplication with a graphics processing unit," *Parallel Process. Lett.*, vol. 20, no. 4, pp. 293–306, Jun. 2010.

Wei Wang received the B.E. degree from Shandong University, Jinan, China, in 2007, and the M.E. degree from Tsinghua University, Beijing, China, in 2010. He is currently pursuing the Ph.D. degree from the Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Worcester, MA, USA.

His current research interests include circuit and system designs for fully homomorphic encryption and RSA cryptosystems.

Xinming Huang (M'01–SM'09) received the Ph.D. degree in electrical engineering from the Virginia Institute of Technology, Blacksburg, VA, USA, in 2001.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA, USA. He was a Member of Technical Staff with the Wireless Advanced Technology Laboratory, Bell Labs of Lucent Technologies, Murray Hill, NJ, USA, from 2001 to 2003. His current research interests include circuits and systems, with emphasis on reconfigurable computing, wireless communications, video processing, and secure embedded systems.

Niall Emmart received the B.S. degree in pure mathematics from the University of Massachusetts, Amherst, MA, USA, in 1992, where he is currently pursuing the Ph.D. degree in computer science with a focus on GPGPU computing.

He ran Yrrid Software, from 1992 to 2012, a small firm focusing on legacy system integration with the web.

Charles Weem received the B.S. (Hons.) and M.A. degrees from Oregon State University, Portland, OR, USA, in 1977 in 1979, respectively, and the Ph.D. degree from the University of Massachusetts, Amherst, MA, USA, in 1984.

He is a Co-Director of the Architecture and Language Implementation Laboratory, University of Massachusetts, where he is an Associate Professor. His current research interests include advanced architectures for media and embedded applications, GPU computing, and high precision arithmetic.