High Performance Integer Multiplier on FPGA with Radix-4 Number Theoretic Transform

Boon-Chiao Chang[1], Wai-Kong Lee[2], Bok-Min Goi[1] and Seong Oun Hwang[2]
[1]Lee Kong Chian Faculty of Engineering and Science, Universiti Tunku Abdul Rahman (UTAR), Sungai Long, Malaysia.
[2]Department of Computer Engineering, Gachon University, Seongnam, South Korea.
[e-mail: sohwang@gachon.ac.kr]
*Corresponding author: Seong Oun Hwang

Received March 30, 2022; revised July 11, 2022; accepted July 19, 2022; published August 31, 2022

Abstract

Number Theoretic Transform (NTT) is a method to design efficient multiplier for large integer multiplication, which is widely used in cryptography and scientific computation. On top of that, it has also received wide attention from the research community to design efficient hardware architecture for large size RSA, fully homomorphic encryption, and lattice-based cryptography. Existing NTT hardware architecture reported in the literature are mainly designed based on radix-2 NTT, due to its small area consumption. However, NTT with larger radix (e.g., radix-4) may achieve faster speed performance in the expense of larger hardware resources. In this paper, we present the performance evaluation on NTT architecture in terms of hardware resource consumption and the latency, based on the proposed radix-2 and radix-4 technique. Our experimental results show that the 16-point radix-4 architecture is 2× faster than radix-2 architecture in expense of approximately 4× additional hardware. The proposed architecture can be extended to support the large integer multiplication in cryptography applications (e.g., RSA). The experimental results show that the proposed 3072-bit multiplier outperformed the best 3k-multiplier from Chen et al. [16] by 3.06%, but it also costs about 40% more LUTs and 77.8% more DSPs resources.

Keywords: Cryptography, FPGA, Number Theoretic Transform, Homomorphic Encryption and Lattice based Cryptosystem.

A preliminary version of this paper was presented at ICONI 2021 and was selected as an outstanding paper.

http://doi.org/10.3837/tiis.2022.08.020 **ISSN : 1976-7277**

1. Introduction

Large integer multiplication is required for many applications, including scientific computation [1] and cryptography [2-4]. However, standard schoolbook multiplication algorithm is difficult to scale when the size of integer grows. The computational complexity of schoolbook multiplication is $O(n^2)$, where n refers to the length of input data, indicating that this is a quadratic relationship. A widely used algorithm to reduce the complexity of multiplication is Schonhage-Strassen Multiplication Algorithm (SSMA) [5]. This algorithm is able to improve the complexity from $O(n^2)$ to $O(n \log n (\log \log (n)))$ by performing the multiplication in the frequency domain, which is also known as convolution. The speedup gained by SSMA is due to the fast algorithm to convert between time and frequency domain. Discrete Fourier Transform (DFT) is a popular technique to transform the data from time domain to its frequency domain, or vice versa. DFT operates on complex domain, while Number Theoretic Transform (NTT) operates over a finite field GF(p). The modulus p needs to be specifically chosen to allow NTT to operate correctly. Since DFT is using floating point arithmetic to compute the complex numbers, it is not suitable to be used in cryptography. This is because floating point arithmetic contains round-off errors, and the error analysis is difficult to handle correctly. Hence, NTT is more widely used in cryptography since it only involves integer arithmetic.

Algorithm 1 Pseudocode for SSMA

Input: x_i *and* y_i, the coefficients of the multiplier and multiplicand.

Output: z_i, the coefficients of the multiplication product,

$z_i = x_i \text{ X } y_i$

$X \leftarrow NTT(x), Y \leftarrow NTT(y)$
for i from 0 to $N - 1$ **do**
    $Z[i] \leftarrow X[i] * Y[i]$
end for
z ← *inverseNTT(Z)*
$z \leftarrow Evaluation(z)$
Return z

Algorithm 1 shows the standard SSMA which involves three NTTs steps. It first performs two forward transforms on X and Y (the multiplication operands), followed by the point-wise multiplication and one inverse transform on the result of product (Z). The implementation of NTT in hardware usually employs radix-2 architecture due to the small area consumption, but radix-4 have potential to achieve faster speed performance. In this paper, we performed evaluation for radix-2 and radix-4 are NTT in terms of its speed performance and hardware area consumption. Our experimental results show that radix-4 can achieve faster performance in expense of additional hardware area.

2. Number Theoretic Transform and Fast Fourier Transform

2.1 NTT and FFT

Fast Fourier Transform (FFT) is a much more efficient way to calculate Discrete Fourier Transform (DFT), effectively reducing the $O(n^2)$ complexity to $O(n \log n)$. Cooley-Tukey FFT (CTFFT) [6] is used in this paper as it is more suitable for parallel implementation in FPGA. Note that the CTFFT can also be applied to speedup NTT. In the subsequent discussions, we denote our implementation in integer domain as CTFNT (Cooley Tukey Fast Number Theoretic Transform) to differentiate it from CTFFT (complex domain). CTFNT allows a large integer or polynomial to be computed with multiple FNTs with smaller sizes. The data is organized in two-dimensional form, which is a property that can be exploited to improve the parallelism. The steps to perform CTFNT are presented below:

1. N-point FNT is decomposed into 2D, $N_1 \times N_2$.
2. Column FNT (Perform N_1 times of N_2-point FNT).
3. Twiddle factors multiplication.
4. Row FNT (Perform N_2 times of N_1-point FNT).

2.2 Radix-2 vs Radix-4 NTT

Radix-2 FNT is most commonly used in designing hardware architecture, among other FNT variants, due to its low area consumption and simplicity. Algorithm 2 shows the steps in performing an in-place radix-2 FNT using the Cooley-Tukey technique. Note that the in-place implementation stores the results of NTT onto the input memory. This effectively eliminates the need to have a separate memory for storing the output data, which is useful in embedded systems that are constrained in memory.

Algorithm 2 In-place radix-2 CTFNT

Input: Polynomial *a* in the time domain; pre-computed twiddle factors (ω)

Output: Polynomial *A* in the frequency (NTT) domain

```
for NP=n/2; NP>0 NP=NP/2 do
    a[i] ← a[i] × Y[i]
    jf ←0; j←0; jTwi←0;                    // Initialize the indices.
     for jf=0; jf<n; jf=j+NP do
         for j=jf; j<jf+NP; j=j+1 do
              temp ← (ω[jTwi] × a[j+NP]) mod p;
              a[j+NP] ← (a[j]-temp) mod p;        // Butterfly operations
              a[j] ← (a[j]+temp) mod p;
         end for
        jTwi ++;
     end for
end for
Return z
```

A radix-R FNT factorizes an N-point NTT into N_R-point NTTs following $\log_R N$ levels of decomposition. Consider that the index of the first level is 1, the index of the top-level NTT (before any decomposition) is equals to 0. At each level of decomposition, the index is increased of 1, and the number of NTTs at each level $R^{\cdot}$. Each NTT now contains N/R^l number of points. To implement radix-R FNT for a N-point NTT, the length N must be a power of R. There is a way to avoid this limitation when N is not a power of R, which is widely known as "mixed-radix FNT". This technique uses different radices of FNT at different level. However, such technique is very complicated to implement in hardware, and it often consumes a larger hardware area since it has to accommodates separate computational modules for different radices. Radix-R FNT is essentially a technique that employs the divide-and-conquer paradigm by dividing the original N-point FNT into R number of FNTs. This process is repeated until each of the FNTs have only R data.

To compute N-point FNT, a radix-R FNT module competes R amount of data at a time. This means that a radix-4 FNT module can process more data at each level compared to radix-2 FNT. In contrast, a radix-2 FNT module has shorter latency compared to radix-4 module, but it requires more modules to compute the same amount of work. For instance, we need four radix-2 FNT modules to compute a 4-point NTT, but only one radix-4 FNT module to do the same computation. The number of radix-R FNT modules, N_R needed to compute N-point FNT is $N_R = (N/R) \times \log_R N$, where $\log_R N$ is the number of FNT level and (N/R) is the number of radix-R FNT module needed at each level.

Implementing N-point FNT module with fully N_R number of radix-R modules is too costly for resource constrained FPGA. To overcome this issue, the number of radix-R modules implemented in hardware is usually less than N_R; they are being reused at different level of FNT to reduce the hardware resource consumption. At each level of FNT, the input data are loaded from the memory for computation; the intermediate results are then written back to the memory. This process repeats until the whole FNT process is completed. Hence, the number of FNT level is a crucial for speed performance, as it determines the number of memory read/write operations required, thus affecting the memory latency of FNT process.

Table 1. N_R needed for radix-2 and radix-4 FFT for N-point FFT

N	Radix-2	Radix-4
2	1	-
4	4	1
8	12	-
16	32	8
32	80	-
64	192	48
128	448	-
256	1024	256

Although a single radix-4 FNT module has longer latency compared to radix-2 FNT module, it is advantageous due to the lesser FNT level, eventually reduced the memory latency of the whole FNT process. The number of levels required to complete an N-point FNT (N_S) is equals to $\log_R N$. This implies that FNT with higher radix (R) has smaller number of FNT level (N_S).

Radix-4 FFT has lesser stages compared to radix-2 FNT, while radix-8 FNT has lesser stages than radix-4 FNT. However, the improvement gained by reducing N_S (increase R) is diminishing when R gets larger. **Fig. 2** shows the number of levels (N_S) for different sizes in N, with R = 2, 4, 8, 16.

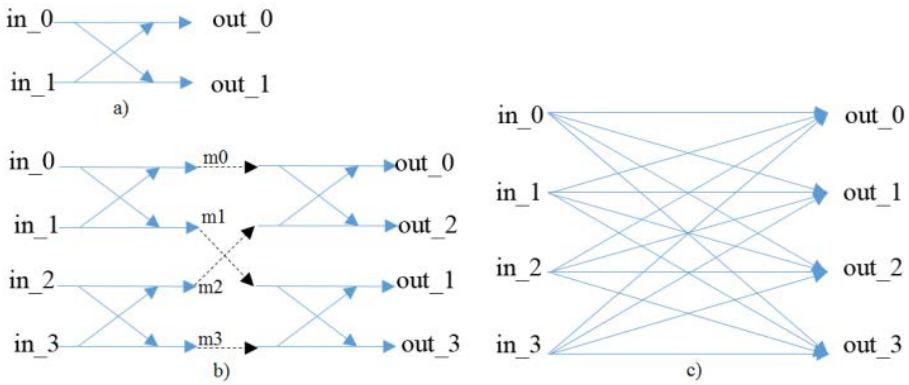

Fig. 1. Illustration of radix-2 and radix-4 FNT.

Fig. 1a) shows the construction of a radix-2 FNT module. **Fig. 1b**) illustrates how four radix-2 FNT modules can be used to form a 4-point FNT, wherein four input data are processed two-by-two and stored into a set of intermediate data (m0; m1; m2; m3), before proceeding to the second level of radix-2 FNT. On the other hand, radix-4 module (**Fig. 1c**)) allows four input data to be processed at once within one level. **Table 1** shows the number of radix-2 and radix-4 modules needed for N-point FNT, where N is in power of 2 and range from 2 to 256. Note that for a fully radix-R FNT to be computed, N must be a power of R; this explains that for cases where N = 2; 8; 32; 128, radix-4 FNT cannot be used. This is the drawback for high radix FNT where $R > 2$.

Implementing N-point FNT module with fully N_R number of radix-R modules is too costly for resource constrained FPGA. To overcome this issue, the number of radix-R modules implemented in hardware is usually less than N_R; they are being reused at different level of FNT to reduce the hardware resource consumption. At each level of FNT, the input data are loaded from the memory for computation; the intermediate results are then written back to the memory. This process repeats until the whole FNT process is completed. Hence, the number of FNT level is a crucial for speed performance, as it determines the number of memory read/write operations required, thus affecting the memory latency of FNT process.

Although a single radix-4 FNT module has longer latency compared to a single radix-2 FNT module, it is advantageous due to the lesser FNT level, eventually reduced the memory latency of the whole FNT process. The number of levels required to complete an N-point FNT (N_S) is equals to $\log_R N$. This implies that FNT with higher radix (R) has smaller number of FNT level (N_S). Radix-4 FFT has lesser stages compared to radix-2 FNT, while radix-8 FNT has lesser stages than radix-4 FNT. However, the improvement gained by reducing N_S (increase R) is diminishing when R gets larger. **Fig. 2** shows the number of levels (N_S) for different sizes in N, with R = 2, 4, 8, 16.

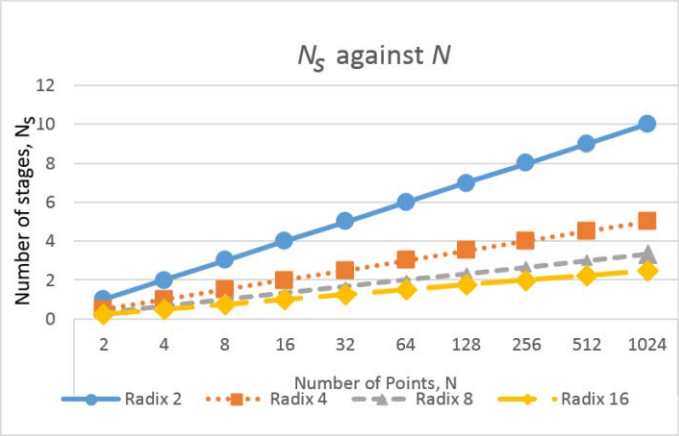

Fig. 2. Number of stages, N_S against Number of points, N.

2.3 Related Work

SSMA is widely used to speed-up the computation of large integer multiplication and polynomial multiplication in cryptography. It is mainly employed by fully homomorphic encryption and public key cryptography, since these algorithms involve a lot of heavy computation in the integer or polynomial domain. For instance, elliptic curve cryptography (ECC) needs to compute a lot of point multiplication that involve modular reduction on integer larger than 128-bit. To ensure sufficient security, RSA also needs to perform modular exponentiations on a very large integer (greater than 2048-bit).

Recently, the introduction of practical quantum computer created serious concern among the research community. The Shor's algorithm [7] executed on quantum computers can easily break the existing ECC and RSA schemes. This has a catastrophic consequence because ECC and RSA schemes are very widely used in the industry in the past decades, and they are still being used now. This stimulated the development of many new public key cryptographic algorithms to resist the potential threat from quantum computers. To avoid such problems, the United States's National Institute of Standards and Technology (NIST) had called upon a worldwide competition to select a few suitable post-quantum cryptography (PQC) schemes [8]. This competition started in 2017 and currently in its third round, wherein 15 candidates are selected for final evaluation. Among these finalists, many schemes like Kyber (key-encapsulation mechanism, KEM) [9] and Dilithium (digital signature) [10] are developed based on the lattice problem, which is known to be NP hard. Lattice-based problems are also widely used to develop advanced cryptographic protocols [11]. These lattice-based schemes are computationally expensive due to the extensive use of polynomial multiplications.

One way to improve the efficiency of computing polynomial multiplication is to offload it to hardware module. Roy and Basso [12] show that with careful design, a schoolbook polynomial multiplication technique can achieve very fast speed on FPGA hardware. This hardware module can be implemented as instruction sets to speed up the computation in embedded system. On the other hand, one can also use asymptotically fast algorithm like SSMA to speed-up the polynomial multiplication. One notable example was demonstrated by Bisheh-Niasar et al. [13] through the use of NTT, implemented efficiently on FPGA hardware.

Besides public key cryptography and PQC, fully homomorphic encryption schemes also perform computations on large integer and polynomial. For instance, Cao et al. [3] demonstrated a hardware architecture suitable for performing fully homomorphic encryption, which relies on the efficient radix-2 NTT architecture and a low-hamming weight technique to provide low latency implementation. Other low-complexity [2] and area-efficient architecture [14] were also proposed to speed-up the implementation of polynomial multiplication on hardware. Note that these hardware architectures only explore radix-2 NTT due to its low hardware area consumption. In this paper, we proposed to explore other radices to improve the speed performance of NTT.

GPU and FPGA are the two representative accelerators used by many cloud services providers like AWS and IBM. Due to this reason, there are also implementations of NTT on GPU to speed up the cryptography algorithms. For instance, Gupta et al. [20] presented the implementation of radix-2 NTT optimized for the Kyber KEM. On the other hand, Jiminez et al. [21] presented the secure implementation of RSA relying on the residue number system. Note that GPU implementation is essential hardware techniques, which are very hard to generalize to FPGA hardware architecture.

3. Evaluation of Radix-2 and Radix-4 NTT Architecture

3.1 Parameter Set

The sub-section title should be written in 11-point size using Arial font style, block color, and The modulus chosen for our NTT is 0xFFFFFFFF00000001, which is a 64-bit Solinas prime that serves several useful properties [15]. Firstly, given a 128-bit number in its polynomial form: $P_{128-bit}(X) = aX^{96} + bX^{64} + cX^{32} + d$, where a, b, c and d are the coefficients. The modular operation (over P) of this 128-bit number is equivalent to $(2^{32})(b + c)a - b + d$. This property is allowing us to handle overflow that potentially occurs when multiplying two 64-bit data. Note that the coefficients of the 128-bit number are derived from the Karatsuba multiplication algorithm.

Secondly, the root of unity, g for 4-point and 16-point are g_4 = 0x1000000000000 and g_{16} = 0x1000 respectively. Since both g are of power-of-two, the expensive twiddle factors multiplication can be replaced with simple left shifting. Each NTT point is of 24-bit size and half of the NTT points are reserved for the multiplication product.

3.2 16-point FNT Designs

Three different FNT modules are implemented to compute 16-point FNT for performance evaluation. The first design uses radix-2 FNT module, while second design employs a generic radix-4 FNT module. The third design is our proposed solution (we refer it as radix-4 CTFNT module in the subsequent discussions).

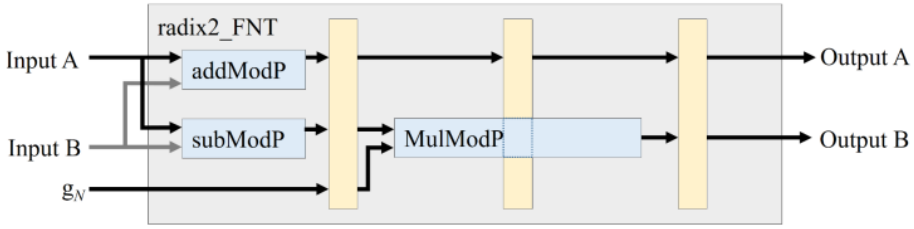

Fig. 3. Block diagram: radix-2 FNT module.

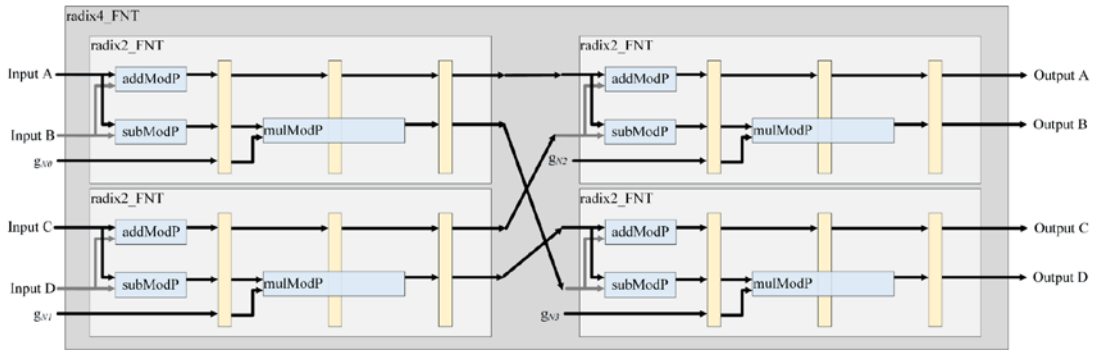

Fig. 4. Block diagram: radix-4 FNT module

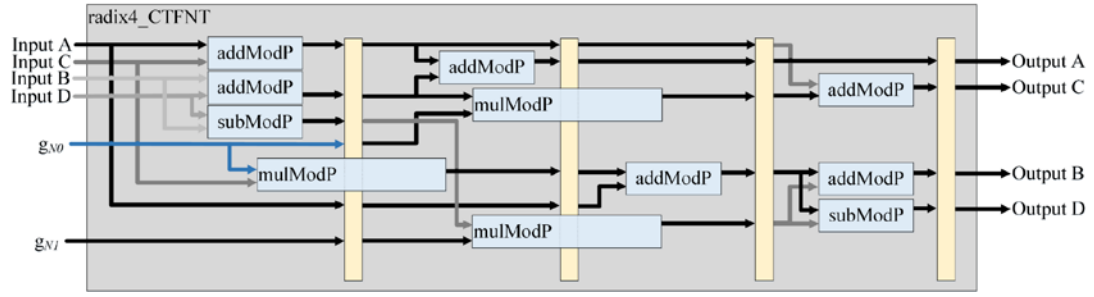

Fig. 5. Block diagram: radix-4 CTFNT module

Fig. 3 shows the block diagram of a radix-2 FNT module, which is commonly used in FNT hardware design [16]. **Fig. 4** shows the generic radix-4 FNT module that can be constructed by combining four radix-2 FNT modules in 2-by-2 manner. The block diagram of our proposed radix-4 CTFNT module with Cooley-Tukey decomposition is illustrated in **Fig. 5**. Realizing the fact that there are only two constant twiddle factors to be used in radix-4 CTFNT, and the multiplication of these two twiddle factors can be done by modular left shift, we presented an improved design in **Fig. 6**. The improved design effectively reduces the hardware resources, removing the need of various twiddle factor inputs and does not need an extra signal to choose between left shifting for forward/inverse transform.

3.3 Performance Evaluation

The proposed 16-point radix-4 CTFNT architecture is implemented in Xilinx Artix-7 (xc7a100tcsg324-1) FPGA. The result is then compared with radix-2 FNT and radix-4 FNT implementation in the same FPGA. **Table 2** shows the resources required to construct 16-point FNT with different hardware architectures (radix-2 FNT, radix-4 FNT and radix-4 CTFNT) and their respective speed performance. The results show that our proposed radix-4 CTFNT

module is able to achieve 2× speedup but consumes 4× more resources. Compared to the generic radix-4 FNT module, the radix-4 CTFNT consumes lesser resources and is 10% faster. From this evaluation, we can conclude that radix-4 CTFNT can achieve faster speed performance in expense of more hardware resources. This 16-point NTT architecture can be used to handle integer multiplication with 192-bit operands. In future, this can be extended to 256-point FNT using the proposed radix-4 CTFNT to construct a full 3072-bit SSMA multiplier, which can be useful for cryptography (e.g., RSA [16]).

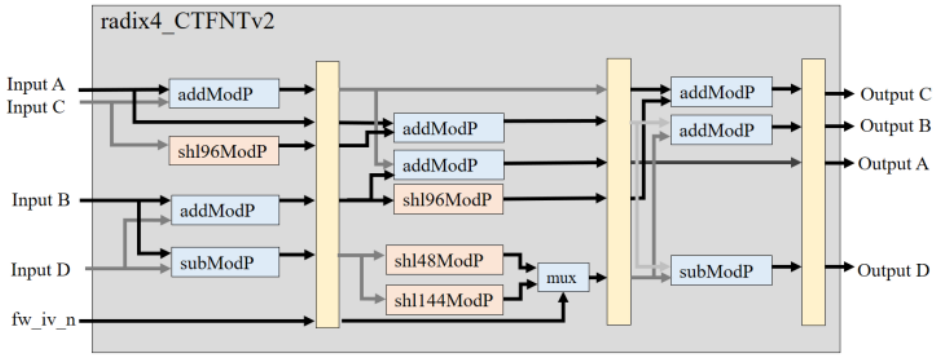

Fig. 6. Block diagram: radix-4 CTFNT module (improved)

Table 2. Resources utilization and timing performance of 16-point FNT with Radix-*R* FFT module

	Hardware Design	Resources					Timing		
		Look-up Table (LUT)	Flip-Flop (FF)	LUTRAM	BRAM	DSP	clock cycle	period (ns)	latency (ns)
16-point FNT	Radix-2 FNT	1687	456	3	4	12	36	50	1800
	Radix-4 FNT	6662	2063	68	7	48	20	50	1000
	Radix-4 CT FNT	6421	1805	35	8	45	18	50	900

4. Complete 3072-bit SSMA Multiplier

In this section, we present the design of a 256-point FNT and the construction of a 3072-bit SSMA multiplier. Each NTT point handles 24-bit integer and only half of the NTT points contain the actual data; the other half are padded with zero [3]. Hence, the maximum supported operand size for this implementation is equal to $24 \times 256=2 = 3072$-bit. The 256-point FNT is first decomposed into sixteen 16-point × 16-point FNTs; each of the 16-point FNT is then further decomposed into four 4-point FNTs. Since these two levels of CTFNT decomposition are done with symmetrical decomposition ($N_1 = N_2$), the precomputed twiddle factors can be shared by both column and row FNTs, reducing the memory costs by half.

4.1 Proposed Partially Pipelined 3K-multiplier

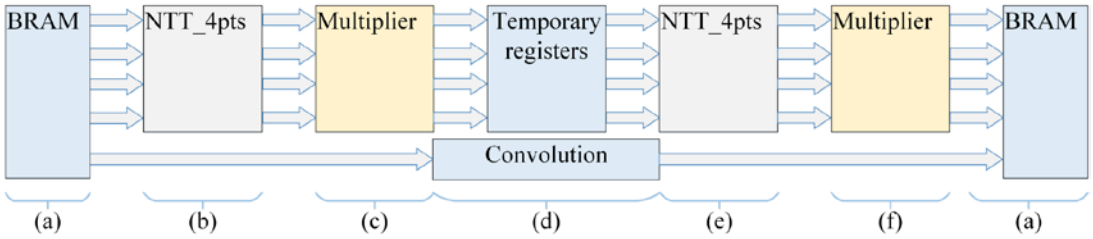

Fig. 7. Block diagram of the partially pipelined 3K-multiplier.

Fig. 7 shows the block diagram of our first 3k-multiplier design using two radix-4 CTFNT modules to perform the column and row NTT operations. The functionalities for each part of the 3k-multiplier are briefly described below:

1) Part (a): Block RAM module; this hosts the memory units to hold the input data (original, intermediate, and final data).
2) Part (b): radix-4 FNT module, perform FNT computation with four inputs and produce four FNT outputs.
3) Part (c): Multiplier module, perform multiplication with twiddle factors.
4) Part (d): Temporary registers (to store the intermediate data of FNT) and the convolution unit.
5) Part (e): radix-4 FNT module, perform FNT computation with four inputs and produce four FNT outputs.
6) Part (f): Multiplier module, perform multiplication with twiddle factors.

Part (a), (b) and (c) are the modules used to compute the column-NTT. It processes four points at a time and store the intermediate results onto part (d) (the temporary registers). All the column-NTT must complete before proceeding to the row-NTT. The row-NTT are handled by part (e), (f) and (a). Note that the BRAM in part (a) is used to store the input and intermediate data during the FNT computation. The final results are also stored in the BRAM. In this design, the column-NTT read data from the block RAM and write to the temporary registers. Conversely, we can see that the row-NTT read data from the temporary registers and write back to the block RAM. Part (d) also computes the convolution of the SSM, which reads and writes data that are stored in the block RAM. The convolution module does not need any intermediate memory. However, an efficient hardware architecture should ensure that the data flow is always fully pipelined. In this design, there are dependencies between different parts, causing it to be a partially pipeline design, which is not efficient. In particular, part (e), (f) and (a) (row-NTT) must be in idle state while waiting for blocks (a), (b) and (c) (column-NTT) to complete their computation. Likewise, block (a), (b) and (c) must be stalled when block (e), (f) and (a) are running. This drastically reduces the hardware occupancy and efficiency, which motivates us to design a fully pipelined architecture.

4.2 Proposed Fully Pipelined 3K-multiplier

Fig. 8 shows the proposed fully pipelined design improved from the partially pipelined version presented in Section 4.1. In this fully pipelined design, instead of waiting for one 16-point column FNT to complete before the 16-point row FNT, four radix-4 CTFNT modules are

instantiated and arranged in parallel. This allows the proposed architecture to compute either four 4-point column or 4-point row FNTs, and then store back to the memory before the next FNT. In other words, there will be always sufficient data to feed the pipeline in our architecture, eventually achieving a more efficient design compared to the partially pipelined version. To achieve this efficient design, there are extra pipeline registers added to the output of the multipliers. This is designed in this way to avoid the possible collision of data (race condition) when two or more modules are accessing the same BRAM. For instance, if four data from the multipliers tries to write onto the BRAM0, then the second data is delayed by one clock cycle. The remaining third and fourth data are then delayed by two and three clock cycles respectively. The overall process takes 198 clock cycles, while the delay introduced three additional clock cycles to completely fill the pipeline. This overhead is insignificant as it is only around 1.5% of the overall process, but it allows full throughput efficiency to be achieved.

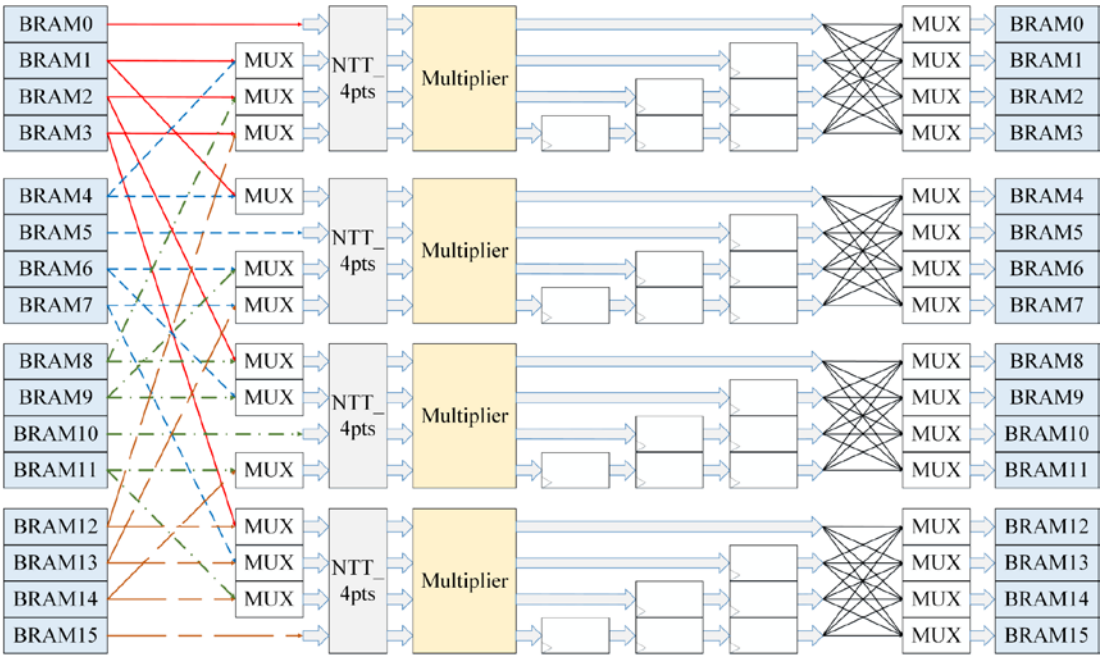

Fig. 8. Block diagram of the fully pipelined 3K-multiplier.

4.3 Experimental Results

Table 3. Resources and Performance Comparison with [16].

Architecture Design	FPGA Hardware	Bit-size (bit)	Resources			Timing		
			Look-up Table (LUT)	**BRAM**	**DSP**	**Clock Cycles**	**Period (ns)**	**Latency (ns)**
Our Work	Artix-7	3072	20129	16:1	192	198	50	9900
	Virtex-6	3072	30489	48:0	192	198	33	6534
Chen et al. [16]	Virtex-6	3100	21672	33:11	108	843	8	6740
	Virtex-6	3132	12147	22:0	27	1701	6.09	10360
	Virtex-6	3196	11728	22:0	27	3693	6.19	22860
	Virtex-6	3196	5835	11:0	9	3633	5.09	18490

The performance of proposed 3072-bit SSMA multiplier is shown in **Table 3**. Our implementation requires only 6.534µs and 9.900µs to complete the multiplication when it is executed in Virtex-6 (xc6vlx130t-1) and Artix-7 (xc7a100tcsg324-1) respectively. Compared with Virtex-6, the implementation on Artix-7 utilized about 2/3 lesser resources and run at slower clock frequency, resulting in a higher latency. This is because the Artix family FPGA is designed for low power instead of high performance, compared to Virtex-6 family. Hence, applications that requires high multiplication performance can implement the proposed 3072-bit SSMA multiplier into a high-end FPGA like Virtex-6. For other applications that puts priority on area consumption and energy efficiency, the proposed 3072-bit SSMA multiplier can be implemented in low end FPGA like Artix-7.

4.4 Discussions and Future Work

Our work can outperform the best 3k-multiplier from Chen et al. [16] by 3.06%, but it also costs about 40% more LUTs and 77.8% more DSPs resources. This is due to the parameters set used in their implementation is different from our work. In our implementation, we focus on 3k-multiplier implemented with fixed 64-bit Solinas prime and 64-bit data processing. On the other hand, Chen et al. [17] introduced multiplier with comprehensive range (covering 1k-bit to 15k-bit). These multiplier employs Pseudo-Fermat number as modulo for NTT, where the modulo ranges from 65-bit to 273-bit. Using a modulo with larger bit size allowed the multiplication operands to be broken down into lesser number of elements of larger size each. Hence, the NTT with lesser number of transformation points can be used to reduce the number of internal operations, including addition, subtraction, and multiplication between two points. Take their best timing performance 3k-multiplier as example, a 225-bit
modulo allowed the 3k-multiplier to be implemented with only 64-point NTT at 97-bit each. Compared to our 3k-multiplier, we use 64-bit modulo and requires 256-point NTT with 24-bit each. This explains why our current design cannot gain further speed performance against the results from Chen et al. [17].

On the other hand, it is believed that lesser number of NTT points with greater bit size for each of the point, is better than having more points with lesser bit size each. This is because NTT module with lesser points can be computed faster. However, point-wise mathematical operations of larger bit size require more time to compute. For instance, addition of two 24-bit numbers can be done faster than addition of two 97-bit numbers. Hence, we consider exploring the possibilities of using NTT modulo of larger bit size, with Solinas prime and other suitable numbers. We believe that there are still room for improvement for the proposed radix-4 CTFNT architecture, after considering these factors.

5. Conclusion

In this paper, we show that the proposed radix-4 CTFNT architecture outperforms the radix-2 and generic radix-4 NTT architecture for 16-point FNT computation. We also presented the design of a 3072-bit multiplier based on the proposed radix-4 CTFNT architecture to show its practicality in cryptography applications. In future, we plan to develop an efficient exponentiation hardware architecture based on the developed multiplier, to support a full RSA computation in FPGA. To achieve a better energy efficiency, we also aim to develop a reconfigurable version of this multiplier in future to suit different multiplicands sizes and performance constraints in Internet of Things (IoT) applications.

Acknowledgement

This work was supported by the Fundamental Research Grant Scheme (FRGS) Malaysia (No. FRGS/1/2021/ICT07/UTAR/01/1). The work of Wai-Kong Lee was supported by the Brain Pool Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information Communication Technology (ICT) under Grant 2019H1D3A1A01102607.

References

[1] D. Harvey, "Computing zeta functions of arithmetic schemes," *Proceedings of London Mathematical Society*, vol. 111, no. 6, pp. 1379 - 1401, 2015. Article (CrossRef Link)

[2] J. H. Ye and M.-D. Shieh, "Low-Complexity VLSI Design of Large Integer Multipliers for Fully Homomorphic Encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1727 - 1736, 2018. Article (CrossRef Link)

[3] X. Cao, C. Moore, M. O'Neill, E. O' Sullivan E. and N. Hanley, "Optimised multiplication architectures for accelerating fully homomorphic encryption," *IEEE Transactions on Computer*s, vol. 65, no. 9, pp. 2794 - 2806, 2016. Article (CrossRef Link)

[4] W. Wang, X. Hu, L. Chen, X. Huang, B. Sunar, "Exploring the Feasibility of Fully Homomorphic Encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698-706, 2015. Article (CrossRef Link)

[5] A. Schonhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," *Computing*, vol. 7, pp. 281-292, 1971. Article (CrossRef Link)

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297-301, 1965. Article (CrossRef Link)

[7] P. Shor, "Algorithms for Quantum Computation: Discrete Logarithm and Factoring," in *Proc. of IEEE FOCS '94*, pp. 124-134, 1994. Article (CrossRef Link)

[8] NIST, "Post-quantum cryptography standardization," 2017. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization [10-Nov-2021]

[9] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehl´e, "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," in *Proc. of 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, pp. 353–367, 2018. Article (CrossRef Link)

[10] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle, "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, vol. 2018, no. 1, pp. 238–268, 2018. Article (CrossRef Link)

[11] B. Mi and D. Liu, "Topology-Hiding Broadcast Based on NTRUEncrypt," *KSII Transactions on Internet and Information Systems*, vol. 10, no. 1, pp. 431-443, 2016. Article (CrossRef Link)

[12] S. S. Roy and A. Basso, "High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 443-466, 2020. Article (CrossRef Link)

[13] M. Bisheh-Niasar, R. Azarderakhsh, M. Mozaffari-Kermani, "Instruction-Set Accelerated Implementation of CRYSTALS-Kyber," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4648-4659, 2021. Article (CrossRef Link)

[14] X. Feng and S. Li, "Design of an Area-Effcient Million-Bit Integer Multiplier Using Double Modulus NTT," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 9, pp. 2658-2662, 2017. Article (CrossRef Link)

[15] E. Niall and C. C. Weems, "High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes," *Parallel Processing Letters*, vol. 21, no. 3, pp. 359-375, 2011. Article (CrossRef Link)

[16] X. Huang and W. Wang, "A Novel and Efficient Design for an RSA Cryptosystem With a Very Large Key Size," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 10, pp. 972-976, 2015. Article (CrossRef Link)
[17] D. D. Chen, G. X. Yao, R. C. C. Cheung, D. Pao and K. Ko, "Parameter Space for the Architecture of FFT-Based Montgomery Modular Multiplication," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 147-160, Jan. 1, 2016. Article (CrossRef Link)
[18] B. L. Tan, K. M. Mok, J. J. Chang, W. K. Lee, S. O. Hwang, "RISC32-LP: Low-power FPGA-based IoT Sensor Nodes with Energy Reduction Program Analyzer," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4214-4228, 2022. Article (CrossRef Link)
[19] W. P. Kiat, K. M. Mok, W. K. Lee, H. G. Goh, R. Achar, "An energy efficient FPGA partial reconfiguration based micro-architectural technique for IoT applications," *Microprocessors and Microsystems*, vol. 73, pp. 102966-102975, 2020. Article (CrossRef Link)
[20] N. Gupta, A. Jati, A. K. Chauhan, A. Chattopadhyay, "PQC Acceleration Using GPUs: FrodoKEM, NewHope, and Kyber," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, 575-586, 2021.
[21] E. O.-Jiminez, L. R.-Zamarripa, N. C.-Cortes, F. R.-Henrique, "Implementation of RSA Signatures on GPU and CPU Architectures," *IEEE Access*, pp. 9928-9941, 2020.

Boon-Chiao Chang received his Bachelor and the M.Eng.Sc degrees from Universiti of Tunku Abdul Rahman (UTAR), Malaysia in 2015 and 2019, respectively. He is currently working as Software Engineer in Infologic LTE PTD, Singapore. Mr. Chang's interests and focuses include problem solving and algorithms analysis in software development.

Wai-Kong Lee received the B.Eng. degree in electronics and the M.Sc. degree from Multimedia University in 2006 and 2009, respectively, and the Ph.D. degree in engineering from Universiti Tunku Abdul Rahman, Malaysia, in 2018. His research interests are in the areas of cryptography, numerical algorithms, GPU computing, Internet of Things, and energy harvesting. He is currently a post-doctoral researcher in Gachon University, South Korea.

Bok-Min Goi received his B.Eng degree from University of Malaya (UM) in 1998, and the M.Eng.Sc and PhD degrees from Multimedia University (MMU), Malaysia in 2002 and 2006, respectively. He is now the Vice President and a senior professor in the Lee Kong Chian Faculty of Engineering and Science, Universiti Tunku Abdul Rahman (UTAR), Malaysia. Ir. Prof. Goi was also the General Chair for ProvSec 2010 and CANS 2010, Honored Chair for ISPEC 2019, Programme Chair for IEEE-STUDENT 2012 , Cryptology 2014-2016, ICDSP 2019-2022 and the TPC members for many crypto/security conferences. He was elected as Fellow of the ASEAN Academy of Engineering & Technology (AAET) and Academy of Science Malaysia Fellow on 2015 and 2018, respectively. His research interests include cryptology, information security & biometrics, digital watermarking and embedded systems design. Prof. Goi is a senior member of the IEEE and corporate member of the Institution of Engineers, Malaysia (IEM).

Seong Oun Hwang received the B.S. degree in mathematics from SeoulNational University, in 1993, the M.S. degree in information and communications engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology, in 2004, South Korea. He worked as a Software Engineer with LG-CNS Systems, Inc., from 1994 to 1996. He worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), from 1998 to 2007. He worked as a Professor with the Department of Software and Communications Engineering, Hongik University, from 2008 to 2019. He is currently a Professor with the Department of Computer Engineering, Gachon University. His research interests include cryptography, cybersecurity, and artificial intelligence. He is an Editor of ETRI Journal.