A Loop Structure Optimization Targeting High-level Synthesis of Fast Number Theoretic Transform

Kazushi Kawamura*†, Masao Yanagisawa‡, and Nozomu Togawa§
*Waseda Research Institute for Science and Engineering, Waseda University
†kazushi.kawamura@togawa.cs.waseda.ac.jp
‡Dept. of Electronic and Physical Systems, Waseda University
§Dept. of Computer Science and Communications Engineering, Waseda University

***Abstract*—Multiplication with a large number of digits is heavily used when processing data encrypted by a fully homomorphic encryption, which is a bottleneck in computation time. An algorithm utilizing fast number theoretic transform (FNTT) is known as a high-speed multiplication algorithm and the further speeding up is expected by implementing the FNTT process on an FPGA. A high-level synthesis tool enables efficient hardware implementation even for FNTT with a large number of points. In this paper, we propose a methodology for optimizing the loop structure included in a software description of FNTT so that the performance of the synthesized FNTT processor can be maximized. The loop structure optimization is considered in terms of loop flattening and trip count reduction. We implement a 65,536-point FNTT processor with the loop structure optimization on an FPGA, and demonstrate that it can be executed 6.9 times faster than the execution on a CPU.**

***Index Terms*—fully homomorphic encryption (FHE), number theoretic transform (NTT), high-level synthesis (HLS), loop optimization, FPGA**

I. Introduction

To promote the use of big data, it is required to provide processing and analysis services of cloud data. Secure computation is important for realizing such services because we need to consider the protection of private information. Fully homomorphic encryption (FHE) techniques [1]–[3] are now attracting attention since arbitrary processings of encrypted data can be performed without decrypting them by using an FHE technique and hence high-security cloud services can be realized. However, the data encrypted by an FHE is very huge in general, which makes it difficult to put into practical use. In particular, multiplication with a large number of digits is heavily used when processing the encrypted data, which is a bottleneck in computation time.

A multiplication algorithm utilizing the fast number theoretic transform (FNTT) [4] is one of the methods which enable us to perform multiplication with a large number of digits at high speed. Some research groups have studied hardware design of FNTT [5], [6] since the further speeding up can be achieved by implementing the FNTT process on an FPGA.

From now on, it is expected that acceleration of a wide variety of applications using FHE techniques will be strongly required. Since FPGA implementation based on a conventional RTL design flow is problematic in terms of design efficiency, hardware design utilizing a high level synthesis (HLS) tool becomes important. By utilizing an HLS tool, various hardware processors can be automatically synthesized from a software description. As a hardware accelerator design for FHEs utilizing an HLS tool, we attempt, in this paper, to synthesize the processors of FNTT with a large number of points efficiently and automatically.

The contributions of this paper are as follows:

1) We propose a methodology for optimizing the loop structure included in a software code of FNTT so that the performance of the synthesized FNTT processor can be maximized. The loop structure optimization is considered in terms of *loop flattening* and *trip count reduction*.
2) We implement a 65,536-point FNTT processor with the loop structure optimization on a Xilinx FPGA, and demonstrate that it can be executed 6.9 times faster than the execution on a CPU.

The rest of this paper is organized as follows: In Section II, we introduce algorithms of multiplication and FNTT, which are used for multiplication with a large number of digits; In Section III, we propose a methodology for optimizing the loop structure targeting HLS of FNTT; Section IV demonstrates the experimental results and gives conclusions.

II. Multiplication with a Large Number of Digits and an FNTT Algorithm

A large decimal N-digit integer A can be expressed as a polynomial with radix of 10 as follows:[1]

$$A = a_0 \times 10^0 + a_1 \times 10^1 + \cdots + a_{N-1} \times 10^{N-1} \quad (1)$$

We convert the integer A into time-domain signals $\{x(0), x(1), \ldots, x(2N-1)\}$ focusing on the coefficient series of Eq. (1). The relationship between Eq. (1) and $x(t)$ ($0 \le t \le 2N-1$, $t \in \mathbb{Z}$) is as follows:[2]

$$x(t) = \begin{cases} a_t & (0 \le t < N) \\ 0 & (N \le t < 2N) \end{cases} \quad (2)$$

In this section, we consider a method for multiplication of two integers $A = x(t)$ and $B = y(t)$. Note that A and B

[1]We can use an arbitrary natural number as the radix for expressing the integer A. For simplicity, the radix is limited to 10 in this paper.

[2]Since the product of two N-digit integers has a maximum of $2N$ digits, an N-digit integer is represented by a set of time-domain signals whose number of elements is $2N$.

are both N-digit integers and each of them is expressed as a set of time-domain signals whose number of elements is M ($= 2N$). We first introduce, in Section II-A, a multiplication method based on the convolution operation which is the simplest method to multiply two integers. Then, a multiplication algorithm utilizing the number theoretic transform (NTT) is introduced in Section II-B. An FNTT algorithm, whose computational complexity is lower than the NTT, is finally introduced in Section II-C [4].

A. A Multiplication Method based on the Convolution Operation

The convolution for the time-domain signals $x(t)$ and $y(t)$ is defined by the following equation:

$$(x * y)(t) = \sum_{u=0}^{t} x(u)y(t-u) \tag{3}$$

The convolution value $(x * y)(t)$ $(t = 0, 1, \ldots, M-1)$ shows the sum of the partial products generated at the t-th digit in the multiplication $A \times B$. Therefore, the multiplication result is finally obtained by the carry processing for the convolution value. For high-speed multiplication, it is important to speed up the process of obtaining convolution values since its computational complexity is much higher than that of the carry processing. The multiplication method based on the convolution operation has a computational complexity of $O(M^2)$.

B. A Multiplication Algorithm utilizing the NTT

As a high-speed multiplication method for integers with a large number of digits, an algorithm utilizing the fast Fourier transform (FFT) [7] is widely known. The FFT can be realized by modifying the computational process of the discrete Fourier transform (DFT). In the multiplication algorithm, the following relationship (called the convolution theorem), which holds between the convolution operation in Eq. (3) and the Fourier transform F, is used:

$$F\left[(x * y)(t)\right] = F\left[x(t)\right] \times F\left[y(t)\right] \tag{4}$$

Since an inverse transform F^{-1} of F exists, the convolution value $(x * y)(t)$ can be obtained as follows:

$$(x * y)(t) = F^{-1}\left[F\left[x(t)\right] \times F\left[y(t)\right]\right] \tag{5}$$

Although it is possible to obtain convolution values at high speed by utilizing the FFT algorithm, floating point arithmetic operations are required in the algorithm, which makes it unsuitable for hardware implementation.

In this paper, we focus on the number theoretic transform (NTT) which can obtain convolution values in the same framework as the Fourier transform. In the number theoretic transform, time-domain signals are transformed on a finite field, whereas in the Fourier transform, on a complex number field. That is, the NTT can be implemented only with integer arithmetic operations, and is suitable for hardware implementation.

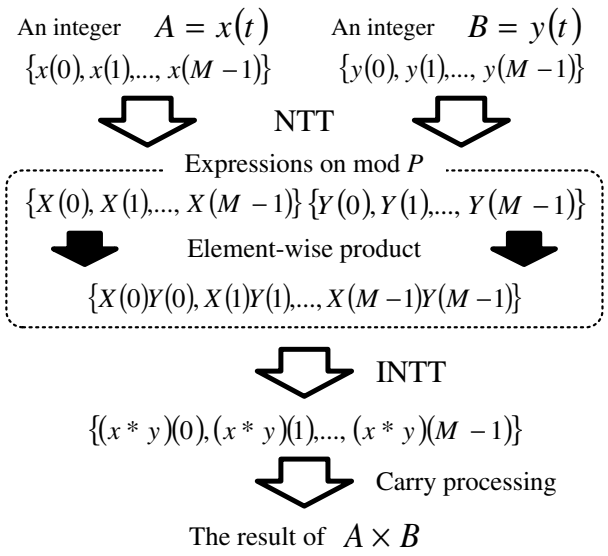

Fig. 1. A multiplication algorithm utilizing the NTT.

In the NTT, time-domain signals are converted to an expression $X(k)$ on $\bmod P$ by using Eq. (6):

$$X(k) = \sum_{t=0}^{M-1} x(t)\alpha^{tk} \pmod{P} \tag{6}$$

Eq. (6) shows M-point NTT and we calculate $X(k)$ for $k = 0, 1, \ldots, M-1$. Note that α and P are a natural number and a prime number, respectively, and they satisfy the following conditions (Eqs. (7)–(9)):[3]

$$\begin{cases} \alpha^n \equiv 1 & (n = M) \\ \alpha^n \not\equiv 1 & (0 \le n \le M-1, n \in \mathbb{Z}) \end{cases} \pmod{P} \tag{7}$$

$$P \ge 9 \times 9 \times M \tag{8}$$

$$P - 1 = s \times M \quad (s \in \mathbb{N}) \tag{9}$$

Eq. (7) is a necessary condition for satisfying the convolution theorem (Eq. (4)). The right side of Eq. (8) shows the maximum convolution value $conv_{max}$, and a convolution value may not be uniquely specified if P is less than $conv_{max}$. We can guarantee by Eq. (9) the existence of α that satisfies Eq. (7).

As with the Fourier transform, there is an inverse transform for the NTT. The following equation shows M-point inverse NTT (INTT):

$$x(t) = \frac{1}{M} \sum_{k=0}^{M-1} X(k)\alpha^{-kt} \pmod{P} \tag{10}$$

Fig. 1 shows a multiplication algorithm utilizing the NTT. In the algorithm, Eqs. (6), (10), (5) are used. This algorithm has a computational complexity of $O(M^2)$.

[3]In the experiments of this paper, we use the smallest prime number P that satisfies Eqs. (7)–(9) for a point number M.

C. An FNTT Algorithm

In the multiplication algorithm utilizing the NTT shown in Section II-B, the computational complexity cannot be reduced compared with the method based on the convolution operation. Note that, in the NTT, the following property holds between α and P when the point number M is an even:

$$\alpha^{\frac{M}{2}} \equiv -1 \pmod{P} \tag{11}$$

By utilizing this property, M-point NTT can be transformed when M is an even. For $k = 0, 1, \ldots, M-1$, we can obtain Eq. (12) from Eq. (6):

$$\begin{aligned}
X(k) &= \begin{cases} X(2l) \\ X(2l+1) \end{cases} \quad \left(0 \le l \le \frac{M}{2} - 1\right) \\
&= \begin{cases} \sum_{t=0}^{M-1} x(t)\alpha^{t(2l)} \\ \sum_{t=0}^{M-1} x(t)\alpha^{t(2l+1)} \end{cases} \\
&= \begin{cases} \sum_{t=0}^{\frac{M}{2}-1} \left\{ x(t)\alpha^{t(2l)} + x\left(t+\frac{M}{2}\right)\alpha^{\left(t+\frac{M}{2}\right)(2l)} \right\} \\ \sum_{t=0}^{\frac{M}{2}-1} \left\{ x(t)\alpha^{t(2l+1)} + x\left(t+\frac{M}{2}\right)\alpha^{\left(t+\frac{M}{2}\right)(2l+1)} \right\} \end{cases} \\
&= \begin{cases} \sum_{t=0}^{\frac{M}{2}-1} \left\{ x(t)\alpha^{t(2l)} + x\left(t+\frac{M}{2}\right)\alpha^{t(2l)}\alpha^{lM} \right\} \\ \sum_{t=0}^{\frac{M}{2}-1} \left\{ x(t)\alpha^{t(2l+1)} + x\left(t+\frac{M}{2}\right)\alpha^{t(2l+1)}\alpha^{lM}\alpha^{\frac{M}{2}} \right\} \end{cases} \\
&= \begin{cases} \sum_{t=0}^{\frac{M}{2}-1} \left\{ x(t) + x\left(t+\frac{M}{2}\right) \right\} \alpha^{t(2l)} \\ \sum_{t=0}^{\frac{M}{2}-1} \left\{ x(t) - x\left(t+\frac{M}{2}\right) \right\} \alpha^{t(2l+1)} \end{cases} \\
&= \begin{cases} \sum_{t=0}^{\frac{M}{2}-1} \underline{\left\{ x(t) + x\left(t+\frac{M}{2}\right) \right\} (\alpha^2)^{tl}} \\ \sum_{t=0}^{\frac{M}{2}-1} \underline{\left\{ x(t) - x\left(t+\frac{M}{2}\right) \right\} \alpha^t (\alpha^2)^{tl}} \end{cases}
\end{aligned} \tag{12}$$

As a result, the point number can be reduced to $M/2$. Eq. (12) shows an expression on $\bmod P$, but for simplicity, it is not shown in this equation. In the case of 8-point NTT, for example, the point number can be reduced to 4 as shown in Fig. 2. As in Fig. 2, each input of $M/2$-point NTT is calculated by the underline part in Eq. (12).

When M is a power of 2, by recursively applying the transformation of Eq. (12), M-point NTT can be transformed to 2-point NTT. By solving the NTT in this way, an FNTT algorithm can be derived. Fig. 3 shows the FNTT algorithm with 8 points. M-point FNTT is calculated by dividing it into $\log_2 M$ parts (called stages) as shown in Fig. 3. In the first stage, we execute $M/2$ butterfly operations in one group, where each of them consists of an addition, a subtraction and a multiplication, and the number of butterfly operations halves as the stage progresses while the number of groups is doubled. By using the FNTT algorithm, the computational complexity of the multiplication in Fig. 1 can be reduced to $O(M \log M)$.

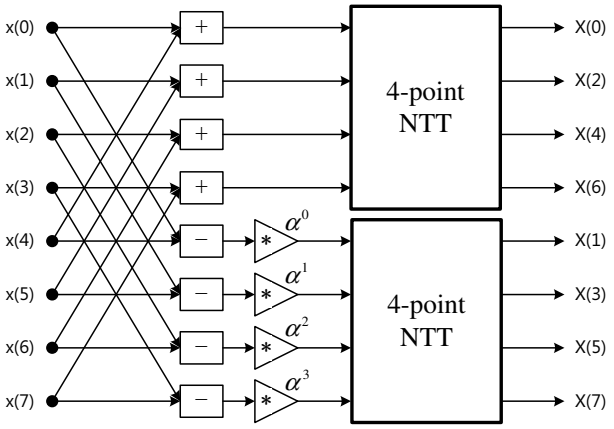

Fig. 2. A transformation of 8-point NTT to 4-point NTT.

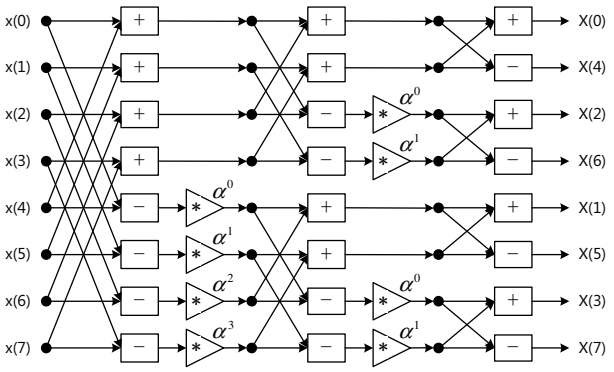

Fig. 3. An 8-point FNTT algorithm.

III. A Loop Structure Optimization Methodology targeting HLS of FNTT

We discuss, in this section, the way to synthesize a processor of FNTT so that the performance can be maximized. As mentioned before, we attempt to automatically synthesize the FNTT processor by using an HLS tool, and it is assumed to be implemented on an FPGA.

M-point FNTT, where $M = 2^m$ ($m \in \mathbb{N}$), can be written in C as shown in Fig. 4. Since the number of stages is m, we first allocate an array having $(m+1) \times M$ elements and initialize the first M elements with input data (INIT). The main body of this code is a triple for loop with three iterators s, g and n which denotes a stage, a group and a butterfly operation, respectively. For every iteration of the innermost loop, one butterfly operation is executed (BF). In BF, $data[s][idx1]$ and $data[s][idx2]$ are read from the array, and the operation results are stored in $data[s+1][idx1]$ and $data[s+1][idx2]$.

In a practical HLS tool, a function to add directives for optimization can be used, which enables us to obtain different hardware processors from a software code. For example, loop pipelining, loop expansion and array partitioning are typical directives for optimization. When given a software code, one of the most important steps in HLS is appropriately

```
int data[m+1][M]; INIT( data[0] );
for( s=0; s < m; s++ ){
        group = 1 << s;  // # of groups
        op = M >> (s+1); // # of ops. in a group
        point = op << 1;
        for( g=0; g < group; g++ ){
                for( n=0; n < op; n++ ){
                        idx1 = point * g + n;
                        idx2 = idx1 + op;
                        BF( s, idx1, idx2 );
                }
        }
}
```

Fig. 4. A software code of M-point FNTT, where $M = 2^m$ ($m \in \mathbb{N}$).

```
int data[m+1][M]; INIT( data[0] );
for( s=0; s < m; s++ ){
        shift_idx = m - 1 - s;
        op = M >> (s+1);
        point = op << 1;
        for( k=0; k < M/2; k++ ){
                g = k >> shift_idx;        // group
                n = k - (g << shift_idx); // op.
                idx1 = point * g + n;
                idx2 = idx1 + op;
                BF( s, idx1, idx2 );
        }
}
```

Fig. 5. Loop flattening of inner double loop.

adding these directives to the code so that a high-performance hardware processor can be synthesized. On the other hand, it may be possible to achieve a performance improvement by rewriting the original code [8], [9]. Therefore, in order to maximize the performance of the synthesized processor, it is required to appropriately rewrite the original code and then appropriately add directives for optimization.

In this section, we propose a methodology for optimizing the loop structure included in the software code shown in Fig. 4 so that the performance of the synthesized FNTT processor can be maximized. The loop structure optimization is considered in terms of *loop flattening* and *trip count reduction*.

A. Loop Flattening

Given a software code including loop structure, the performance of the synthesized processor can be improved by adding pipeline directives to the loops. When synthesizing a multiple loop with a pipeline directive, the structure of the loop greatly affects the synthesized result. It may not be possible to achieve a sufficient performance improvement when pipelining a loop having some outer loops since extra clock cycles are required for entering a loop and exiting from a loop. One of the methods to solve this problem is the *loop flattening*.

The loop flattening is a method to transform a multiple loop with deeply-nested structure into a fewer nested loop [8]. For example, for a double loop whose trip counts of the inner and the outer loops are i and j, respectively, it may be possible to transform it into a single loop whose trip count is $i \times j$, which enables us to reduce the overhead related to entering a loop and exiting from a loop.

For the description of Fig. 4, by applying the loop flattening to the inner double loop, we obtain the description shown in Fig. 5. Although, in the description of Fig. 4, the number of groups $group$ and the number of butterfly operations op vary depending on the stage s, we can apply the loop flattening by focusing on the fact that the product of $group$ and op is always $M/2$. Furthermore, we obtain the description shown in Fig. 6 by applying the loop flattening to the double loop included in Fig. 5. Note that the values of s, g and n are common among the descriptions of Figs. 4–6.

```
int data[m+1][M]; INIT( data[0] );
for( r=0; r < m * M/2; r++ ){
        s = r >> (m - 1);                  // stage
        k = r - (s << (m - 1));
        shift_idx = m - 1 - s;
        op = M >> (s+1);
        point = op << 1;

        g = k >> shift_idx;                // group
        n = k - (g << shift_idx);          // op.
        idx1 = point * g + n;
        idx2 = idx1 + op;
        BF( s, idx1, idx2 );
}
```

Fig. 6. Loop flattening of triple loop.

Experiments and Discussions: In order to verify the effectiveness of *loop flattening* for HLS of FNTT, we have synthesized the descriptions of Figs. 4–6 using an HLS tool, Vivado-HLS 2015.2 [10]. The FPGA board has been assumed to be Xilinx Virtex-7 (xc7vx690tffg1926-2). For $M = 16$ and $M = 1,024$, we have tried to synthesize these descriptions with the following six approaches:

(1) **For the description of Fig. 4, perform the HLS without adding directives.**
(2) **For the description of Fig. 4, add a pipeline directive to the innermost loop and then perform the HLS.**
(3) **For the description of Fig. 4, add a pipeline directive to the innermost loop, expand the outermost loop, and then perform the HLS.**
(4) **For the description of Fig. 5, add a pipeline directive to the inner loop and then perform the HLS.**
(5) **For the description of Fig. 5, add a pipeline directive to the inner loop, expand the outer loop, and then perform the HLS.**
(6) **For the description of Fig. 6, add a pipeline directive to the loop and then perform the HLS.**

The results are shown in tables I and II. The 2nd and 3rd columns of this table show the clock period and the number of clock cycles necessary for executing the FNTT process, and their product shows the performance. The 4th to 7th columns show the number of hardware resources.

TABLE I
EVALUATIONS OF LOOP FLATTENING ($M = 16$).

	Clock [ns]	#Steps	BRAM	DSP48	FF	LUT
(1)	10.0	2,858	0	3	1,114	1,256
(2)	20.0	2,777	0	2	4,586	4,880
(3)	10.0	324	4	4	8,554	8,536
(4)	20.0	2,740	3	2	4,492	4,861
(5)	10.0	196	4	4	8,541	8,520
(6)	20.0	2,726	3	2	4,490	4,860

TABLE II
EVALUATIONS OF LOOP FLATTENING ($M = 1,024$).

	Clock [ns]	#Steps	BRAM	DSP48	FF	LUT
(1)	10.0	485,398	23	3	1,269	2,079
(2)	20.0	471,061	23	2	5,963	6,945
(3)	10.0	29,701	23	10	34,268	34,758
(4)	20.0	468,012	23	2	5,923	7,039
(5)	10.0	7,597	23	10	34,192	34,699
(6)	20.0	467,974	23	2	5,920	7,039

Compared to the approach (1), the performance is deteriorated in the approaches (2), (4) and (6), whereas is improved in (3) and (5). The former group has constructed a pipeline structure across all stages. The latter group, on the other hand, has constructed a separate pipeline structure for each stage. In the FNTT process, since a read and a write operations to the same array occur in adjacent stages, constructing a pipeline structure across multiple stages increases its initiation interval (II) due to memory access competition. In other words, it is important to construct a separate pipeline structure for each stage in synthesizing an FNTT processor, and hence the description shown in Fig. 6 is not valid.

Compared to the approach (3), a higher performance FNTT processor can be synthesized by (5), especially in the case of $1,024$-point FNTT. In stages close to the output, the number of groups is more than the number of butterfly operations, which means that, in the description of Fig. 4, the inner double loop has a large trip count for the outer loop and a small trip count for the inner loop. In such stages, loop pipelining causes large overhead related to entering a loop and exiting from a loop. The results in table II show that the overhead can be reduced efficiently by applying the loop flattening to the inner double loop in Fig. 4. From the above discussions, we find that the software code shown in Fig. 5 has an optimal loop structure and (5) is the most appropreate approach at this point.

B. Trip Count Reduction

In the approach (5) in Section III-B, the number of clock cycles $L_t(s)$ necessary for executing a stage s can be formulated as the following equation:

$$L_t(s) = IL(s) + II(s) \times (TC(s) - 1) + C \qquad (13)$$

where $IL(s)$, $II(s)$ and $TC(s)$ show, in the stage s, the number of clock cycles necessary for executing a single iteration, the initiation interval and the trip count, respectively. C shows a constant which includes clock cycles for entering and exiting from the loop.

```
int data[m+1][M]; INIT( data[0] );
for( s=0; s < m-1; s++ ){
        shift_idx = m - 1 - s;
        op = M >> (s+1);
        point = op << 1;
        for( k=0; k < M/2; k+=2 ){
                g = k >> shift_idx;          // group
                n = k - (g << shift_idx); // op.
                idx1 = point * g + n;
                idx2 = idx1 + op;
                idx3 = idx1 + 1;
                idx4 = idx3 + op;
                BF( s, idx1, idx2 );
                BF( s, idx3, idx4 );
        }
}
... // Code of the stage m is omitted.
```

Fig. 7. Trip count reduction of inner loop.

TABLE III
EVALUATIONS OF TRIP COUNT REDUCTION ($M = 16$).

	Clock [ns]	#Steps	BRAM	DSP48	FF	LUT
(5)	10.0	196	4	4	8,541	8,520
(7)	10.0	189	4	6	14,970	15,226
(8)	10.0	178	6	6	15,087	15,116

TABLE IV
EVALUATIONS OF TRIP COUNT REDUCTION ($M = 1,024$).

	Clock [ns]	#Steps	BRAM	DSP48	FF	LUT
(5)	10.0	7,597	23	10	34,192	34,699
(7)	10.0	7,594	23	19	64,542	65,753
(8)	10.0	5,291	43	19	64,206	65,125

In FNTT with a large number of points, the second term of Eq. (13) becomes dominant. The approach (5) derives, in all stages, $II(s) = 1$ and $TC(s) = M/2$, and it may be possible to further reduce $L_t(s)$ by reducing the trip count $TC(s)$ in each stage. In order to realize the *trip count reduction*, we rewrite the description of Fig. 5 so that two butterfly operations are executed for every iteration of the inner loop. Note that we need to remove the description of the final stage (the stage m) from the loop structure because there is only one butterfly operation in a group. By applying the trip count reduction to the inner loop included in Fig. 5, we obtain the description shown in Fig. 7. In Fig. 7, $L_t(s)$ is expected to be reduced thanks to the reduction of $TC(s)$.

Experiments and Discussions: In order to verify the effectiveness of *trip count reduction* for HLS of FNTT, we have synthesized the descriptions of Fig. 7 using an HLS tool, Vivado-HLS 2015.2 [10]. The FPGA board has been assumed to be Xilinx Virtex-7 (xc7vx690tffg1926-2). For $M = 16$ and $M = 1,024$, we have tried to synthesize this description with the following two approaches:

(7) **For the description of Fig. 7, add a pipeline directive to the inner loop, expand the outer loop, and then perform the HLS.**
(8) **For the description of Fig. 7, add a pipeline directive to the inner loop and an array partition directive**

TABLE V
EXPERIMENTAL RESULTS.

#Points M	BRAM	DSP48	FF	LUT	Slack [ns]	#Steps	Latency [ms]	Execution time on a CPU [ms]
1,024	11.5	10	16,402	21,167	4.295	7,597	0.076 (3.347x)	0.254
	21.5	19	30,498	38,984	1.983	5,291	0.053 (4.806x)	
2,048	19.0	11	20,421	25,236	2.171	15,852	0.159 (3.513x)	0.557
	24.5	21	38,224	46,738	1.718	10,731	0.107 (5.190x)	
4,096	35.5	12	23,802	30,517	2.877	33,337	0.333 (3.617x)	1.206
	41.5	22	44,767	58,082	2.311	22,072	0.221 (5.463x)	
8,192	82.0	13	28,178	35,965	2.429	70,273	0.703 (3.769x)	2.649
	75.5	44	52,552	65,143	1.603	45,695	0.457 (5.797x)	
16,384	182.5	26	31,961	40,899	1.933	148,173	1.482 (3.801x)	5.632
	188.5	48	60,592	76,667	1.574	94,924	0.949 (5.933x)	
32,768	403.0	28	36,792	46,977	1.352	312,093	3.121 (3.884x)	12.122
	402.0	53	69,476	87,477	1.632	197,398	1.974 (6.141x)	
65,536	887.0	30	42,175	55,103	0.062	656,241	6.562 (4.336x)	28.455
	885.0	56	80,275	102,584	0.835	410,480	4.105 (6.932x)	

to the array $data$ **(so that the number of elements becomes** $M/2$**), expand the outer loop, and then perform the HLS.**

The results are shown in tables III and IV. Compared to the approach (5), the performance is hardly improved in (7), whereas is improved in (8). In the description of Fig. 7, the number of accesses to the array in each iteration is doubled as compared with Fig. 5. This means that the approach (7), where the array is not partitioned, increases the II of the inner loop due to memory access competition. In the approach (8), the access competition is resolved by array partitioning, and the effect of the trip count reduction can be demonstrated. From the above discussions, we find that the loop structure in the software code of FNTT can be optimized as in Fig. 7 by the trip count reduction, and the highest performance of the FNTT processor can be achieved by the approach (8).

IV. EXPERIMENTAL RESULTS AND CONCLUSIONS

In this section, the effectiveness of our methodology is verified through HLS of FNTT with a large number of points. For $M = 2^m$ ($10 \leq m \leq 16$), we have synthesized M-point FNTT processors using an HLS tool, Vivado-HLS 2015.2 [10]. In the HLS step, we have adopted the approach (5) (presented in Section III-A) and the approach (8) (presented in Section III-B). In this experiment, we have assumed Xilinx Virtex-7 (xc7vx690tffg1926-2) as the FPGA board. After the HLS step, we have implemented the FNTT processors on the FPGA using an implementation tool, Vivado 2015.2 [10]. In the implementation step, the clock period constraint has assumed to be 10 ns.

Table V shows the implementation results of the FNTT processors. For each point number, the upper and lower rows show the results of the approaches (5) and (8), respectively. The 2nd to 5th columns show the number of hardware resources and the 6th column shows the slack. Latency shown in the 8th column is calculated by multiplying the number of clock cycles (in the 7th column) by the clock period (10 ns). In this experiment, we have also executed FNTT processes on a CPU (Intel Corei7-5600U@2.6GHz) and have measured the execution times as shown in the 9th column.

The improved rate of the performance compared to the CPU is also shown in the 8th column of Table V. Experimental results show that a $65,536$-point FNTT processor, optimized by our methodology and implemented on an FPGA, can be executed 6.9 times faster than the execution on a CPU.

ACKNOWLEDGMENT

This work was supported by JST CREST Grant Number JPMJCR1503, Japan.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. of the 41st annual ACM Symposium on Theory of Computing*, 2009, pp. 169–178.
[2] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012.
[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. on Computation Theory*, vol. 6, no. 3, pp. 13.1–13.36, 2014.
[4] J. M. Pollard, "The fast fourier transform in a finite field," *Mathematics of Computation*, vol. 25, no. 114, pp. 365–374, 1971.
[5] W. Wang, X. Huang, N. Emmart, and C. Weems, "Vlsi design of a large-number multiplier for fully homomorphic encryption," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 22, no. 9, pp. 1879–1887, 2014.
[6] E. Ozturk, Y. Doroz, B. Sunar, and E. Savas, "Accelerating somewhat homomorphic evaluation using fpgas," Cryptology ePrint Archive, Report 2015/294, 2015.
[7] A. Schonhage and V. Strassen, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, 1971.
[8] H. Sim, A. Rahman, and J. Lee, "Efficient high-level synthesis for nested loops of nonrectangular iteration spaces," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 24, no. 8, pp. 2799–2802, 2016.
[9] M. Lattuada and F. Ferrandi, "Exploiting outer loops vectorization in high level synthesis," in *Proc. of the 28th International Conference on Architecture of Computing Systems*, 2015.
[10] Vivado Design Suite, http://www.xilinx.com/products/design-tools/vivado/index.htm.