高ポイント高速数論変換に対する高位合成のためのループ構造最適化

川村 一志[1,a)]　柳澤 政生[2]　戸川 望[2]

概要：秘匿計算の実用化に向け，完全準同型暗号が注目を集めている．完全準同型暗号を用いた暗号文演算においては桁数の大きな乗算が多用され，演算時間のボトルネックとなる．高速数論変換を用いた乗算アルゴリズムにより桁数の大きな乗算を高速に実行可能であるが，高速数論変換処理の FPGA 実装によりさらなる高速化が期待される．実装にあたり，高位合成ツールを活用することでポイント数の大きな高速数論変換処理に対しても効率的なハードウェア設計が可能となる．本稿では，合成後ハードウェアの性能を最大限引き出すため，ソフトウェアコードに含まれるループ構造を二つの観点（Loop flattening, Trip count reduction）で最適化する．ループ構造最適化を施した 65,536 ポイントの高速数論変換処理を高位合成し，FPGA 上に実装した結果，CPU での実行に比べ 6.9 倍高速化できることを確認した．

1. はじめに

ビッグデータの利活用を推進する目的から，クラウドに収集されたデータを処理・解析するサービスの提供が求められている．このようなサービスを実現するにあたっては秘匿計算の基盤を構築することが重要となる．完全準同型暗号（FHE: Fully Homomorphic Encryption）[1–3] を用いると，暗号化されたデータに対して復号することなく任意の処理を施すことが可能となり，秘匿計算による安全性の高いクラウドサービスを実現できる．一方，FHE により暗号化されたデータは非常に巨大であることから暗号文演算にかかる時間が大きく，実用化には難がある．特に，暗号文演算中では桁数の大きな乗算が多用され，演算時間のボトルネックとなっている．

桁数の大きな乗算を高速に実行可能な手法のひとつに高速数論変換（FNTT: Fast Number Theoretic Transform）[4] を用いた乗算アルゴリズムがある．FNTT 処理を FPGA 実装することで乗算のさらなる高速化が期待されることから，これまでに FNTT のハードウェア設計に関する研究が進められてきた [5,6]．

今後，FHE を用いる多種多様なアプリケーションのアクセラレーションが必要になると予想される．従来の RTL 設計にもとづく FPGA 実装は設計効率の面で問題があるため，高位合成ツールを活用したハードウェア設計の必要性が高まっている．高位合成ツールを活用した FHE 向けハードウェアアクセラレータ設計として，本稿では FNTT 処理ハードウェアの自動合成を試みる．高位合成ツールの活用によりソフトウェアコードからハードウェアを自動合成できるため，ポイント数の大きな FNTT 処理に対しても効率的なハードウェア設計が可能となる．

本稿では，合成後ハードウェアの性能を最大化することを目的とし，ソフトウェアコードに含まれるループ構造を最適化する．ループ構造の最適化は Loop flattening 及び Trip count reduction の観点で実施される．ループ構造最適化を施した 65,536 ポイントの FNTT 処理を高位合成し，FPGA 上に実装した結果，CPU での実行に比べ 6.9 倍高速化できることを確認した．

本稿は以下のように構成される．2 章では，桁数の大きな乗算に用いられるアルゴリズム及び FNTT を紹介する．3 章では，FNTT 処理の高位合成を対象に最適なループ構造を検討し，合成後ハードウェアの性能向上を図る．4 章に計算機実験結果を示し，5 章で結論を述べる．

2. 桁数の大きな乗算と高速数論変換

桁数が N の非常に大きな 10 進数整数 A は，基数が 10 の多項式として以下のように表現できる．[*1]

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

本稿では，式 (1) の係数系列に着目し，整数 A を時間領域の信号 $\{x(0), x(1), \ldots, x(2N-1)\}$ として表現する．式 (1) と $x(t)$ $(0 \le t \le 2N-1,\ t$ は整数$)$ との関係は以下の通りである．[*2]

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

1　早稲田大学理工学術院総合研究所
2　早稲田大学大学院基幹理工学研究科情報理工・情報通信専攻
a)　kazushi.kawamura@togawa.cs.waseda.ac.jp

*1　整数 A を表現するための基数として任意の自然数を用いることができる．本稿では，簡単のために基数を 10 に限定する．
*2　N 桁の整数どうしの積は最大で $2N$ 桁となるため，N 桁の整数は要素数が $2N$ の時間信号で表現する．

本章では，時間領域の信号として表現された 2 つの整数 $A = x(t)$，$B = y(t)$ の乗算手法を検討する．整数 A，B は N 桁の整数であり，要素数が M $(= 2N)$ の時間信号で表現される．最も単純な乗算手法である畳み込み演算による乗算を 2.1 節で紹介し，2.2 節で数論変換を用いた乗算アルゴリズムを紹介する．2.3 節では，数論変換よりも計算量の少ない高速数論変換 [4] を紹介する．

2.1 畳み込み演算による乗算

時間領域の信号 $x(t)$，$y(t)$ に対する畳み込みは，以下の式で定義される．

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

畳み込み値 $(x * y)(t)$ $(t = 0, 1, \ldots, M-1)$ は乗算 $A \times B$ において t 桁目に生成される部分積の和を表す．従って，畳み込み値に桁上げ処理を施すことで乗算結果を得ることができる．桁上げ処理にかかる時間は畳み込み値を求める時間に比べて極めて小さいため，高速な乗算処理のためには畳み込み値を求める処理の高速化が必要となる．畳み込み演算による乗算の計算量は $O(M^2)$ である．

2.2 数論変換を用いた乗算アルゴリズム

桁数の大きな整数に対する高速な乗算アルゴリズムとして，高速フーリエ変換（FFT: Fast Fourier Transform）を用いたアルゴリズム [7] が広く知られている．FFT は離散フーリエ変換（DFT）の計算方法に工夫を加え，計算量を削減したものである．FFT を用いた乗算アルゴリズムでは，フーリエ変換 F と式 (3) に示す畳み込み演算の間に成立する以下の関係（畳み込み定理）を利用する．

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

加えて，フーリエ変換 F には逆変換 F^{-1} が存在することから，$x(t)$，$y(t)$ に対する畳み込みを以下の式により求めることができる．

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

FFT を用いることで畳み込み値を高速に求めることが可能となるが，一般的な FFT は浮動小数点演算を用いて実装されるため，ハードウェア実装に向かない．

本稿では，畳み込み値をフーリエ変換と同様の枠組みで求めることのできる数論変換（NTT: Number Theoretic Transform）に着目する．フーリエ変換では時間領域の信号を複素数体上で変換するのに対し，数論変換では時間領域の信号を有限体上で変換する．すなわち，NTT は整数演算のみで実装可能であり，ハードウェア実装に向く．

数論変換では，変換式 (6) を用いて時間信号 $x(t)$ を mod P 上での表現 $X(k)$ に変換する．

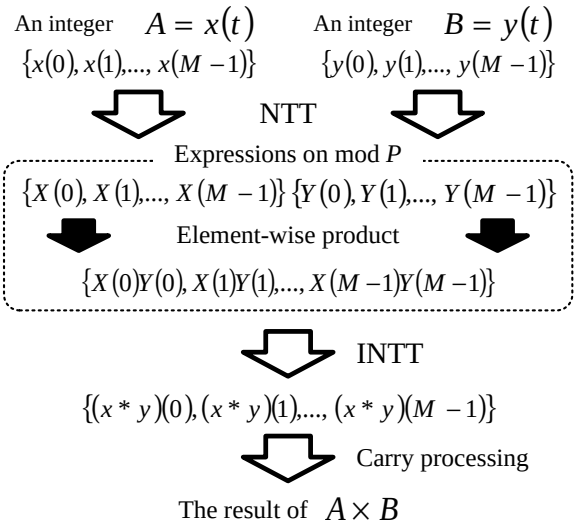

図 1 数論変換を用いた乗算アルゴリズム．

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

式 (6) はポイント数 M の NTT を表し，$k = 0, 1, \ldots, M-1$ に対し $X(k)$ を導出する．ここで，α は自然数，P は素数であり，α と P は以下の条件式 (7)〜(9) を満たす．[*3]

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

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

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

式 (7) は式 (4) に示す畳み込み定理を成り立たせるために必要な条件である．式 (8) の右辺は畳み込み値の最大値 $conv_{max}$ を表し，素数 P が $conv_{max}$ よりも小さいと畳み込み値を一意に特定できなくなる可能性がある．式 (9) により式 (7) を満たす α が存在することを保証できる．

フーリエ変換同様，数論変換にも逆変換が存在し，ポイント数が M の逆数論変換（INTT: Inverse NTT）は以下の式で表される．

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

数論変換を用いた乗算アルゴリズムを図 1 に示す．乗算アルゴリズムには式 (6)，式 (10) に示す NTT，INTT，及び式 (5) の関係式が用いられている．数論変換を用いた乗算アルゴリズムの計算量は $O(M^2)$ である．

2.3 高速数論変換

2.2 節に示す数論変換を用いた乗算アルゴリズムでは畳み込み演算による乗算と比較して計算量を削減できない．

[*3] 本稿の計算機実験では，ポイント数 M に対し，条件式 (7)〜(9) を満たす素数 P の中で最小の値を用いる．

ポイント数 M が偶数のとき，α と P の間に以下の性質が成り立つ．

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

この性質を利用し，ポイント数 M が偶数の NTT を以下のように変形する．$k = 0, 1, \ldots, M-1$ に対し，

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

と変形すると $M/2$ ポイント NTT へと帰着させることができる．ここで，式 (12) は mod P 上での演算を表すが，記述を簡単にするため表記していない．例えば，$M = 8$ とした場合に $M/2 = 4$ ポイント NTT へと帰着させると，図 2 のようになる．図 2 にあるように，$M/2$ ポイント NTT の入力は式 (12) の下線部で表される．

ポイント数 M が 2 のべき乗で表されるとき，式 (12) の変形を再帰的に適用することで最終的に 2 ポイント NTT へと帰着させることができる．このようにして計算する数論変換は高速数論変換（FNTT: Fast NTT）と呼ばれる．8 ポイント FNTT の計算の流れを図 3 に示す．図 3 からも分かる通り，M ポイントの FNTT は $\log_2 M$ のステージに分けて計算される．最初のステージでは $M/2$ 個のバタフライ演算（加算，減算，乗算から成る 2 入力 2 出力の演算）を 1 グループで実行し，ステージがひとつ進むとバタフライ演算の個数が半減し，グループ数が倍増する．高速数論変換を用いることで図 1 の乗算アルゴリズムの計算量を $O(M^2)$ から $O(M \log M)$ に削減可能である．

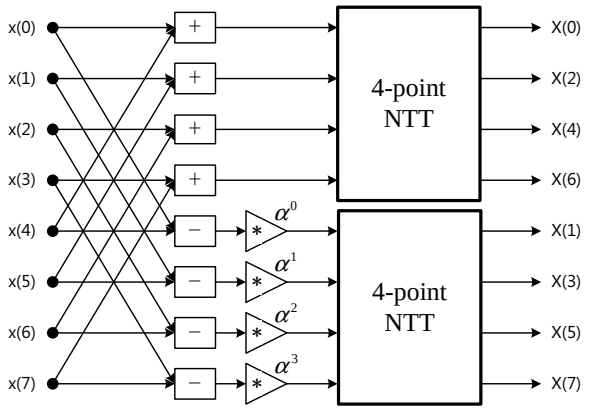

図 2　4 ポイント数論変換への帰着．

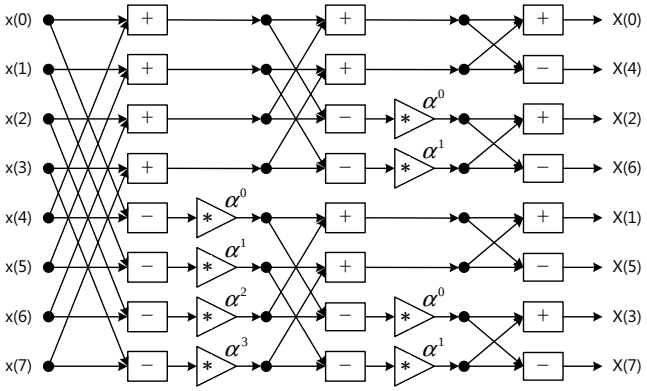

図 3　8 ポイント高速数論変換．

3. 高速数論変換に対する高位合成のためのループ構造最適化

2 章では，高速数論変換（FNTT）を用いることで桁数の大きな乗算の計算量が削減可能であることを示した．本章にて FNTT 処理の FPGA 実装を検討し，さらなる高速化を図る．前述の通り，本稿では高位合成ツールを活用したハードウェアの自動合成に着目する．

ポイント数 $M = 2^m$ $(m \in \mathbb{N})$ の FNTT は C 言語で図 4 のように記述できる．ポイント数 2^m に対し m ステージの計算が必要となるため，最初に要素数 $(m+1) \times M$ の配列 $data$ を確保し，先頭の M 個を入力データで初期化（INIT）する．FNTT 処理の本体はステージ s，グループ g，バタフライ演算 n をイテレータとする三重ループを成し，最内ループのイテレーション毎にバタフライ演算（BF）を 1 回実行する．BF 部分では，$data[s][idx1]$，$data[s][idx2]$ のデータを読み出し，演算結果を $data[s+1][idx1]$，$data[s+1][idx2]$ に格納している．

実用的な高位合成ツールには最適化指示子を付加する機能が備わっており，ソフトウェアコードに最適化指示子を付加して高位合成することで異なるハードウェアを得ることができる．代表的な最適化指示子として，ループパイプライン，ループ展開，メモリ分割があり，適切な最適化指

```
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 );
        }
    }
}
```

図 4 ソフトウェアコード (C) による FNTT の記述.

```
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 );
    }
}
```

図 5 内側二重ループの平坦化.

示子を適切に付加することで高性能なハードウェアの合成が可能となる．一方，ソフトウェアコード自体の書き換えにより性能向上が達成される事例も報告されている [8,9]．従って，合成後ハードウェアの性能を最大限引き出すためには，コードを適切に書き換えた上で適切に最適化指示子を付加することが求められる．

本章では，図 4 の FNTT 記述を対象に，Loop flattening 及び Trip count reduction に着目したソフトウェアコードの書き換えを検討する．その上で，高ポイント FNTT 処理のハードウェア自動合成を想定した場合に性能を最大化するループ構造，及び最適化指示子を提示する．

3.1 Loop flattening

ループを含むコードを高位合成する場合，パイプライン指示子を付加することで合成後ハードウェアの性能を大幅に向上させることができる．パイプライン化対象が多重ループのとき，そのループ構造は合成後ハードウェアの性能を大きく左右する．ループへの進入及びループからの脱出には余分なクロックサイクルを必要とするため，外側にループを持つループをパイプライン化すると十分な性能向上を達成できない可能性がある．この問題を解決する方法としてループ平坦化（Loop flattening）がある．

ループ平坦化とは，多重ループをネストの浅いループに変換することを指す [8]．例として外側と内側の trip count がそれぞれ i，j の二重ループを考えると，ループ平坦化により trip count が $i \times j$ の一重ループで表現できる可能性があり，ループの進入・脱出にかかるオーバーヘッドの軽減が期待される．

図 4 の記述に対し，内側二重ループを平坦化すると図 5 の記述を得る．変換前の記述においてグループ数 *group* とバタフライ演算の個数 *op* はステージ s によって変化するが，*group* と *op* の積が必ず $M/2$ となる点に着目すると平坦化することができる．さらに，図 5 の記述に含まれる二重ループを平坦化すると図 6 の記述を得る．

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

図 6 三重ループの平坦化.

評価実験・考察

ループ平坦化の効果を検証するため，図 4〜6 の記述を Vivado-HLS 2015.2 [10] を用いて高位合成した．FPGA ボードは Xilinx 社 Virtex-7（xc7vx690tffg1926-2）を想定した．16 ポイント及び 1,024 ポイントの FNTT 処理を対象に，合成方法（コードと最適化指示子の組み合わせ）として以下の 6 通りを検証した．

(1) 図 4 の記述を最適化指示子なしで高位合成する．
(2) 図 4 の記述の最内ループにパイプライン指示子を付加して高位合成する．
(3) 図 4 の記述の最内ループにパイプライン指示子を付加し，最外ループを展開して高位合成する．
(4) 図 5 の記述の内側ループにパイプライン指示子を付加して高位合成する．
(5) 図 5 の記述の内側ループにパイプライン指示子を付加し，外側ループを展開して高位合成する．
(6) 図 6 の記述にパイプライン指示子を付加して高位合成する．

結果を表 1，表 2 に示す．表の 2, 3 列目はクロック周期と実行に必要なサイクル数を表し，これらの積が性能の目安となる．表の 4〜7 列目は必要な資源数を表す．

表 1　Loop flattening の評価（16 ポイント FNTT）.

合成方法	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

表 2　Loop flattening の評価（1,024 ポイント FNTT）.

合成方法	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

合成方法 (1) を基準としたとき，(2)，(4)，(6) の合成方法では性能が悪化する一方，(3)，(5) の合成方法では性能が向上する．前者はすべてのステージでひとつのパイプラインを構成するのに対し，後者はステージ毎に個別のパイプラインを構成する．FNTT 処理では隣接するステージで同一の配列に対する読み出し・書き込みが発生するため，ステージを跨ぐパイプライン化はメモリへのアクセス競合により実行間隔（II: Initiation Interval）を増加させる．すなわち，FNTT のハードウェア合成においてはステージ毎に個別のパイプラインを構成させることが重要であり，図 6 の記述は有効でない．

続いて，(3) と (5) の合成方法を比較すると，特に 1,024 ポイントの場合に (5) の方法で高性能なハードウェアを合成できる．出力に近いステージではバタフライ演算の個数に比べてグループ数が多くなり，図 4 の記述において内側二重ループの内側の trip count が小さくなる反面，外側の trip count が大きくなる．そのため，出力に近いステージでループの進入・脱出にかかるオーバーヘッドが大きく軽減でき，性能が向上したと考えられる．以上により，適切な合成方法が (5) であることを確認できる．

3.2　Trip count reduction

3.1 節の合成方法 (5) において，ステージ s の実行に必要なサイクル数 $L_t(s)$ は以下のように定式化できる．

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

ここで，$IL(s)$，$II(s)$，$TC(s)$ は s の関数であり，それぞれイテレーション 1 回分の実行サイクル数，イテレーションの II 及び trip count を表す．C は定数であり，ループの進入・脱出に必要なサイクル数を含む．

ポイント数の大きな FNTT では式 (13) の第二項が支配的となる．合成方法 (5) ではすべてのステージを通して $II(s) = 1$ かつ $TC(s) = M/2$ となっており，$TC(s)$ を削減することでハードウェア性能をさらに向上できる可能性がある．そこで，内側ループのイテレーション毎に 2 回のバタフライ演算を実行するよう，図 5 の記述を書き換え，図 7 の記述とする．ただし，最終ステージはグループ内に 1 個のバタフライ演算しか存在しないため，図 5 の記述のまま for 文の外に記述する．この書き換えにより，最終ステージを除くすべてのステージで $TC(s)$ が削減され，$L_t(s)$ が約半分となることが期待される．

```
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.
```

図 7　trip count の削減.

表 3　Trip count reduction の評価（16 ポイント FNTT）.

合成方法	Clock [ns]	#Steps	BRAM	DSP48	FF	LUT
(7)	10.0	189	4	6	14,970	15,226
(8)	10.0	178	6	6	15,087	15,116

表 4　Trip count reduction の評価（1,024 ポイント FNTT）.

合成方法	Clock [ns]	#Steps	BRAM	DSP48	FF	LUT
(7)	10.0	7,594	23	19	64,542	65,753
(8)	10.0	5,291	43	19	64,206	65,125

評価実験・考察

trip count 削減の効果を検証するため，図 7 の記述を Vivado-HLS 2015.2 [10] を用いて高位合成した．FPGA ボードは Xilinx 社 Virtex-7（xc7vx690tffg1926-2）を想定した．16 ポイント及び 1,024 ポイントの FNTT 処理を対象に，合成方法として以下の 6 通りを検証した．

(7) 図 7 の記述の内側ループにパイプライン指示子を付加し，外側ループを展開して高位合成する．

(8) 図 7 の記述の内側ループにパイプライン指示子を付加し，外側ループを展開して高位合成する．このとき，配列 $data$ の要素数が $M/2$ 個となるように配列分割する．

結果を表 3，表 4 に示す．

合成方法 (5) を基準としたとき，(7) の合成方法では性能向上が見られないのに対し，(8) の合成方法では性能向上が確認された．trip count の削減を試みた図 7 の記述では，図 5 の記述と比較してイテレーション毎の配列へのアクセス回数が倍増する．そのため，配列を分割しない (7)

表 5　計算機実験結果.

ポイント数 M	BRAM	DSP48	FF	LUT	Slack [ns]	#Steps	Latency [ms]	CPU での実行時間 [ms]
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)	

の合成方法では配列へのアクセス競合により，II が増加する．一方，(8) の合成方法では配列分割によりアクセス競合を解消し，trip count 削減の効果を発揮できる．以上により，適切な合成方法が (8) であることを確認できる．

4. 計算機実験結果

3 章で検討したループ構造，及び最適化指示子にもとづきポイント数の大きな FNTT 処理を高位合成し，FPGA 上に実装することで, ハードウェア性能を検証する．本実験では，ポイント数 $M = 2^m$ $(12 \leq m \leq 16)$ の FNTT 処理を対象とし，3.1 節の合成方法 (5) 及び 3.2 節の合成方法 (8) を採用した．高位合成ツールとして Vivado-HLS 2015.2 [10] を用い，FPGA ボードは Xilinx 社 Virtex-7（xc7vx690tffg1926-2）を想定した．FPGA 実装のための論理合成・レイアウトツールとして Vivado 2015.2 [10] を用いた．実験を通して，クロック周期制約は 10 ns とした．

結果を表 5 に示す．各ポイント数に対し，上段が合成方法 (5) による結果を表し，下段が合成方法 (8) による結果を表す．また，右端の列は CPU（Intel Corei7-5600U@2.6Ghz）上での FNTT 処理の実行時間を表す．表の 2〜5 列目は必要な資源数を表し，6 列目はスラックの最悪値を表す．スラックの最悪値がすべて正の値となっているため，すべてのポイント数の FNTT 処理を FPGA 上に実装できていることが確認できる．表の 7 列目は実行にかかるクロックサイクル数を表し，これにクロック周期（10 ns）を掛け合わせることで 8 列目に示すレイテンシが導出される．

表 5 の 8 列目の括弧内には CPU 上での実行と比較したときの速度向上率を示した．この結果から，FPGA 上に実装した 65,536 ポイント FNTT 処理ハードウェアは CPU での実行に比べ 6.9 倍高速に実行可能であることが確認できた．

5. おわりに

本稿では，ソフトウェアコードとして記述された FNTT 処理に含まれるループ構造を最適化し，ハードウェア性能の最大化を図った．ループ構造の最適化は Loop flattening 及び Trip count reduction の観点で実施された．ループ構造最適化を施した 65,536 ポイントの FNTT 処理を高位合成し，FPGA 上に実装した結果，CPU での実行に比べ 6.9 倍高速化できることを確認した．

今後の研究では，FHE を用いる種々のアプリケーションを対象に FPGA 上で動作可能なハードウェアアクセラレータを設計し，性能評価することが求められる．

謝辞　本研究は JST CREST JPMJCR1503 の支援を受けたものである．

参考文献

[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.