信州大学工学部

学士論文

固定した原始多項式による標数 2 の有限体二乗演算の高速化に関する検討

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　09T2059H
氏名　土橋 遼

2013 年 2 月 21 日

目次

1 序章

1.1 研究の背景と目的

今日の社会では, 携帯電話やコンピューターを 1 人 1 台は所有しており誰もが情報という分野と深い関わりを持っている. それは, 情報化社会と言われるほど顕著である. その情報化社会において, 有限体は重要な役割を担っている. 例えば送信者から送られた情報源を暗号化し受信者がそれを復号化するという一連の作業のなかで, 暗号化と復号化をする際に有限体が用いられている. 有限体の利点は小数をとらず誤差が出ないことであり, 暗号化された情報源を誤りなく正確に復号化することができる. 有限体の演算をするプログラミングのツールとして Number Theory Library(以下 NTL)[1] がある. この NTL は任意の標数の有限体演算が可能であり, コンピュータの構造にあわせた標数 2 の有限体を必要とする情報通信分野や標数に囚われず理論体系の構築を目的とする研究者の間で主流である.

有限体をベクトル表現する場合, 乗算や除算をする過程では原始多項式を法とする多項式の剰余計算が必要となる. NTL のように汎用性のあるプログラムでは, 原始多項式やその次数は変数として宣言され, 多項式の剰余計算は乗算や除算をするたびに実行される. さらにその剰余計算には分岐命令が用いられるので, CPU のパイプライン処理による高速化が妨げられている. ところが, 標数や原始多項式を固定した場合, 剰余計算に用いる多くの値を演算済みの定数として事前に用意することが可能である. このことに着目した従来研究 [2] では, 乗算や除算を高速化できることが示されている. その高速化には，分岐命令が削減されていることも寄与していると考えられる, 従来研究 [2] では，原始多項式ごとにプログラムを用意しなくてはならないが, プログラムの自動生成によりこの問題も解決されている,

暗号化や復号化のアプリケーションでは, 有限体のべき乗や逆数を計算する頻度は高い, 特に, べき乗を計算する場合は乗算よりも 2 乗を用いた方が効率がよい. 例として 2^4 を挙げる. 乗算で考えると $2^4 = 2 \times 2 \times 2 \times 2$ の 3 回の計算となるが 2 乗を用いると, $2^4 = (2^2)^2$ となり 2 回の計算で済むようになる. 原始多項式を固定した時の 2 乗計算の特徴は乗算よりも多くの値を演算済みの定数として用意できる点である. これにより 2 乗計算を行う場合は乗算計算を行うのに比べ実行時間を短くすることが出来ると考えられる. 本研究では, 現在のコンピューターの構造に合った標数 2 に特化し, 上記の方法による有限体演算の高速化を提案する. 本研究においても，従来研究と同じくデメリットは原始多項式ごとにプログラムを用意しなくてはならない点であるがその点も自動生成プログラムにより解決することも合わせて提案する.

1.2　本論文の構成

以降, 本論文は次のように構成されている. 2 章では, 有限体の乗算と 2 乗計算について説明する. 3 章では従来法と提案法の概要とそれらの実験結果を示し, 比較・考察を行う. 4 章では, プログラムの自動生成について述べる. 5 章では本論文のまとめを記す.

2　有限体

後に説明する，$GF(2^n)$ における乗算と除算の演算方法は共通して次のような二つの多項式を用いて考える.

$$f_a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0x^0$$
$$f_b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1x^1 + b_0x^0$$

係数体として $GF(2)$ を用いる．すなわち各係数 a_i, b_i は 0 または 1 である．各多項式の係数ベクトル $(a_{n-1}, a_{n-2}, \ldots, a_0)$, $(b_{n-1}, b_{n-2}, \ldots, b_0)$ を a, b の 2 進数表現とする．上の多項式はビット列として表されるため，x^0 の係数が最下位ビット, x^1 の係数が最下位ビットの一つ上, x^{n-1} の係数ビットが最上位ビットに対応すると考える. また, x は不定元と呼ばれ, 形式的な変数で数値を代表するものではない. 以上のことをふまえて $f_a(x)$ と $f_b(x)$ を二進数表現する.

例:3 ビットのとき, 多項式 $f_a(x)$ で 5 を表す. 多項式 $f_a(x)$ は 3 次の場合であるので

$$f_a(x) = a_2x^2 + a_1x + a_0$$

と表される. 表 1 より, 5 は 2 進数, 及び係数で 101 と表される. よって

$$(a_2, a_1, a_0) = (1, 0, 1)$$

という関係から $f_a(x)$ は

$$f_a(x) = 1 \cdot x^2 + 0 \cdot x + 1$$

となり多項式で 5 を表すことが出来る.

表 1　10 進数の 2 進数, 係数における対応

10 進数	2 進数	(a_2, a_1, a_0)
0	000	0,0,0
1	001	0,0,1
2	010	0,1,0
3	011	0,1,1
4	100	1,0,0
5	101	1,0,1
6	110	1,1,0
7	111	1,1,1

2.1　有限体の乗算

ここでは有限体の乗算について $GF(2^3)$ を例に説明する. なお原始多項式は x^3+x+1 を使用する. この原始多項式から $x^3 = x+1$ という関係が導かれる. この場合に扱えるビット数は 3 ビットまでで 2 進数では 000 から 111 までしか扱うことができない. 原始多項式とは計算結果が 111 を超えた部分の次数を下げ, 3 ビットに収めるための式である.

手順 1：　$f_a(x)$ と $f_b(x)$ の二つの多項式を乗算する.

$$
\begin{aligned}
&f_a(x) \times f_b(x) \\
=&(a_2x^2 + a_1x + a_0) \times (b_2x^2 + b_1x + b_0) \\
=&a_2(b_2x^2 + b_1x + b_0)x^2 + a_1(b_2x^2 + b_1x + b_0)x^1 + a_0(b_2x^2 + b_1x + b_0)x^0 \\
=&a_2a_2x^4 + a_2b_1x^3 + a_2b_0x^2 \\
&\quad + a_1b_2x^3 + a_1b_1x^2 + a_1b_0x \\
&\quad\quad + a_0b_2x^2 + a_0b_1x + a_0b_0
\end{aligned}
$$

手順 2：次数が 3 以上の係数を原始多項式を用いて次数 2 以下にする. なお, この部分が剰余計算にあたる部分である. x^3, x^4 の項を原始多項式 x^3+x+1 で割った余りが $x+1$,x^2+x ということになり, 原始多項式を法として用いているため $x^3 = x+1$, $x^4 = x^2+x$ という関係が導かれ,

$$
a_2b_1x^3 = a_2b_1x + a_2b_1,\ a_1b_2x^3 = a_1b_2x + a_1b_2
$$
$$
a_2b_2x^4 = a_2b_2x^2 + a_2b_2x
$$

となる. よって, 手順 1 の続きより

$$
\begin{aligned}
&f_a(x) \times f_b(x) \\
=&a_2b_2x^2 + a_2b_2x + a_2b_1x + a_2b_1 + a_2b_0x^2 \\
&\qquad + a_2b_1x + a_2b_1 + a_1b_1x^2 + a_1b_0x \\
&\qquad\qquad + a_0b_2x^2 + a_0b_1x + a_0b_0 \\
=&(a_2b_2 + a_2b_0 + a_1b_1 + a_0b_2)x^2 \\
&\quad + (a_2b_2 + a_2b_1 + a_1b_2 + a_1b_0 + a_0b_1)x \\
&\qquad + a_2b_1 + a_1b_2 + a_0b_0
\end{aligned}
$$

となる. 多項式同士の乗算を行い, 3 次以上の項を原始多項式により 3 ビットで扱えるまでの次数に下げ, 各項ごとに足したものが答えとなる. 手順 3： (a_2, a_1, a_0) と (b_2, b_1, b_0) に 0 または 1 を代入する. 例として 2×2 の有限体乗算を行う. $a = 2$ より $(a_2, a_1, a_0) = (0, 1, 0)$, $b = 2$ より $(b_2, b_1, b_0) = (0, 1, 0)$ である. これらを代入することで手順 2 の続きから

$$
\begin{aligned}
&f_a(x) \times f_b(x) \\
=&(a_2b_2 + a_2b_0 + a_1b_1 + a_0b_2)x^2 + (a_2b_2 + a_2b_1 + a_1b_2 + a_1b_0 + a_0b_1)x + a_2b_1 + a_1b_2 + a_0b_0 \\
=&(0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 0)x^2 + (0 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 + 0 \cdot 1)x + 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 0 \\
=&1 \cdot x^2 + 0 \cdot x + 0
\end{aligned}
$$

となり演算結果は 4 となる. 手順 3 を繰り返すことで $GF(2^3)$ の乗算表は表 2 のとおりになる.

表 2　$GF(2^3)$ の乗算

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

2.2　3 次の場合の有限体の 2 乗

ここでは有限体の 2 乗計算について $GF(2^3)$ を例に説明する．原始多項式は x^3+x+1 を使用する．

手順 1：　$f_a(x)$ と $f_a(x)$ の多項式を乗算する．

$$
\begin{aligned}
&f_a(x)\times f_a(x)\\
=&a_2(a_2x^2+a_1x^1+a_0x^0)x^2+a_1(a_2x^2+a_1x^1+a_0x^0)x+a_0(a_2x^2+a_1x^1+a_0x^0)\\
=&a_2a_2x^4+a_2a_1x^3+a_2a_0x^2\\
&\quad+a_1a_2x^3+a_1a_1x^2+a_1a_0x^1\\
&\quad\quad+a_0a_2x^2+a_0a_1x^1\ +a_0a_0x^0
\end{aligned}
\tag{1}
$$

手順 2：各次数ごとに和をとる．ここで a_0,a_1,a_2 は $GF(2)$ 上の値を取るので同じ値を足すと 0 となることを用いると消える項が現れる．

(例 1)$a=3$ のとき，$(a_2,a_1,a_0)=(0,1,1)$ であるから

$a_1a_0\to 1\times 1=1$　　$a_2a_0\to 0\times 1=0$

$a_0a_1\to 1\times 1=1$　　$a_0a_2\to 1\times 0=0$

$a_0a_1+a_1a_0\to 1+1=0$　　$a_2a_0+a_0a_2\to 0+0=0$

となる．

(例 2)$a=7$ のとき，$(a_2,a_1,a_0)=(1,0,1)$ であるから

$a_1a_0\to 0\times 1=0$　　$a_2a_0\to 1\times 1=1$

$a_0a_1\to 1\times 0=0$　　$a_0a_2\to 1\times 1=1$

$a_0a_1+a_1a_0\to 0+0=0$　　$a_2a_0+a_0a_2\to 1+1=0$

となる．

よって，

$$f_a(x)\times f_a(x)=a_2a_2x^4+a_1a_1x^2+a_0a_0x^0$$

となる．このことから a_2a_2,a_1a_1,a_0a_0 のように 2 乗の項のみが残る．言い換えると，偶数次の項は残り，奇数次の項は 0 となるということが有限体の 2 乗の計算において言える．また，$GF(2)$ 上の値を取るためこれらの残った項は $a_2a_2=a_2$ のように計算出来る．あとは残った次数が 3 以上の係数を原始多項式を用いて次数を 2 以下にし，次数 3 以下の項と足し合わせることで計算結果となる．

原始多項式 $x^4 = x^2 + x$ より

$$a_2 a_2 x^4 = a_2 a_2 x^2 + a_2 a_2 x$$
$$a_2 x^4 = a_2 x^2 + a_2 x$$

となり, $f_a^2(x)$ の計算結果は

$$\begin{aligned} f_a(x) \times f_a(x) =& (a_2 a_2 + a_1 a_1)x^2 + a_2 a_2 x + a_0 a_0 x^0 \\ =& (a_2 + a_1)x^2 + a_2 x^1 \ + a_0 x^0 \end{aligned}$$

となる.

手順 3： (a_2, a_1, a_0) に 0 または 1 を代入する. 例えば 3^2 を求めたい時 $a = 3$ から $(a_2, a_1, a_0) = (0, 1, 1)$ より,

$$\begin{aligned} f_a(x) \times f_a(x) &= (a_2 + a_1)x^2 + a_2 x \ + a_0 x^0 \\ &= (0 + 1)x^2 + 0 \cdot x + 1 \\ &= x^2 + 0 \cdot x + 1 \end{aligned}$$

となり演算結果は 5 となる. 以降, 手順 3 を繰り返すことで $GF(2^3)$ の有限体の 2 乗計算は表 3 のとおりになる.

表 3　$GF(2^3)$ の 2 乗計算

	0	1	2	3	4	5	6	7
x^2	0	1	4	5	6	7	2	3

次に, 2 乗計算では偶数次の項と 0 が交互に並ぶと述べたことについて考えてみる. 2.1 節の式 (1) より,

$$\begin{aligned} & f_a(x) \times f_a(x) \\ =& a_2 a_2 x^4 + a_2 a_1 x^3 + a_2 a_0 x^2 \\ & + a_1 a_2 x^3 + a_1 a_1 x^2 + a_1 a_0 x^1 \\ & + a_0 a_2 x^2 + a_0 a_1 x^1 + a_0 a_0 x^0 \end{aligned}$$

である. この式を図 1 のように表す. 図のマスの中に○と書かれた項が偶数次で残る項, ×と書かれた項が消える項を表している.

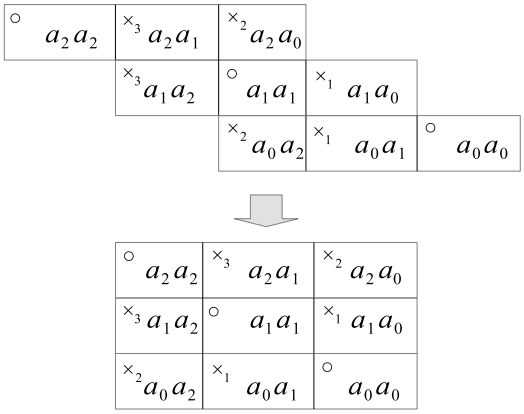

図 1　数式 (1) の図化

図 1 の上部は式 (1) を項ごと, 次数ごとに図化したものであり, 横が次数を表し, 縦同士で足し算が行われるものとなっている. 次にこの階段上に並んだ図を正方形のような形にしたものが図 1 の下部である. この図から対角線上に○のついた項が並ぶということ, 対角線以外の項は消えることがわかる.

2.3　n 次の場合の有限体の 2 乗

前節では, 3 次の 2 乗計算の方法と 2 乗計算の特徴である偶数次の項と 0 が交互に並ぶ性質について述べた. この節では, 3 次の場合の 2 乗計算だけでなく n 次の場合であってもこの性質に従うことを述べる. 多項式は

$$f_a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-1} + \cdots + a_1x^1 + a_0x^0$$

であり, この式を 2 乗すると

$$\begin{aligned}
&f_a(x) \times f_a(x) \\
=&a_{n-1}(a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0x^0)x^{n-1} \\
&+ a_{n-2}(a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0x^0)x^{n-2} \\
&+ \cdots \\
&+ a_1(a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0x^0)x^1 \\
&+ a_0(a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x^1 + a_0x^0)x^0 \qquad (2)
\end{aligned}$$

となる.

次に, 式（2）を図 1 のように正方形で表すと, 図 2 のようになり対角線上の項のみ残ることがわかる. このことから n 次の有限体の 2 乗計算においても係数が 2 乗の形をした項が残るということが言える.

$\circ\ a_{n-1}a_{n-1}$	$\times_6\ a_{n-1}a_{n-2}$	$\cdots$	$\times_5\ a_n a_1$	$\times_4\ a_n a_0$
$\times_6\ a_{n-2}a_{n-1}$	$\circ\ a_{n-2}a_{n-2}$	$\cdots$	$\times_3 a_{n-2}a_1$	$\times_2 a_{n-2}a_0$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\times_5\ a_1 a_n$	$\times_3\ a_1 a_{n-1}$	$\cdots$	$\circ\ a_1 a_1$	$\times_1\ a_1 a_0$
$\times_4\ a_0 a_n$	$\times_2\ a_0 a_{n-1}$	$\cdots$	$\times_1\ a_0 a_1$	$\circ\ a_0 a_0$

図 2　数式 (2) の図化

次にどのようにして 2 乗計算をすればよいかについて説明する. 先に述べた, 偶数次の項が 0 と交互に現れるという性質によりどの次数でも 2 乗計算では多項式が与えられたとき, 次の式のように表すことができる.

$$f_a^2(x) = \cdots + a_4x^4 + 0 \cdot x^3 + a_2x^2 + 0 \cdot x^1 + a_0x^0$$

あとは原始多項式を用いる項の次数を下げ, 残った項を次数ごとに足し合わせる. この操作で出来た式の係数に 0 か 1 を代入することで, 2 乗演算が可能になる.

3　提案法とその有効性

3.1　従来法 1 とその改善点

乗算や除算をする過程において [手順 2] のような剰余計算が必要となる. NTL ではこの剰余計算で必要となる原始多項式や次数は変数として宣言されている. 本研究では NTL のように原始多項式や次数が変数として宣言されている有限体演算プログラムを従来法 1 と呼ぶ. 原始多項式や次数は変数として宣言されているので, 多項式の剰余計算は乗算や除算をするたびに実行される. また, その剰余計算には分岐命令が用いられており, CPU のパイプライン処理による高速化が妨げられている.

3.2　従来法 2 のねらい

従来研究 [2] で提案された乗算アルゴリズムを従来法 2 と呼ぶ. 従来法 2 では, 従来法 1 で変数として扱われていた次数と原始多項式を演算済み定数として扱い, 計算量と分岐命令を減らすことで有限体演算の高速化を実現している. 次数と原始多項式を定数とした場合, 多くの

値を演算済みの定数として事前に用意することができる. 図 3 は実際に原始多項式を変数とする $a \times b$ の乗算プログラムの 3 ビットにおける例であるが, 原始多項式を $x^3 + x + 1$ に固定して演算済み定数を使うことによって $a \times b$ の乗算プログラムは図 4 のように短くなり，計算量と分岐命令が減っていることがわかる.

```
y0 = (a & 0x01)? b: 0;
y1 = 0;
for(i = 1; i < n; i++){
      y0 ^= ((a & (0x01 << i))? (b << i): 0) & mask;
      y1 ^= ((a & (0x01 << i))? (b >> (n - i)): 0) & mask;
}
y2 = 0x01 << (n - 1);
for (k = 0; k < n - 1; k++){
    y2 = ((y2 << 1) ^ ((y2 & (0x01 << (n - 1)))? p: 0)) & mask;
    if (y1 & (0x01 << k)){
        y0 ^= y2; //原子多項式で次数を落とした項を足し合わせる
    }
}
```

図 3　原始多項式を変数とした有限体乗算の演算プログラム (従来法 1)

```
int a0b = b * !!(a & 0x01); //a_0 × b
int a1b = b * !!(a & 0x02); //a_1 × b
int a2b = b * !!(a & 0x04); //a_2 × b
int y = a0b ^ (a1b << 1) ^ (a2b << 2);       //項ごとに足す(3 次以下)
int p = y ^ (0x03 * !!(y & 0x08)) ^ (0x06 * !!(y & 0x10));//原子多項式で次数を落とした項を足し合わせる
```

図 4　原始多項式を演算済み定数とした有限体乗算の演算プログラム (従来法 2)

3.3　提案法のねらい

$GF(2)$ 上の多項式を 2 乗すると係数が 2 乗の形をした項だけが残り, 偶数次の項と 0 が交互に並ぶと前章で述べた. この 2 乗計算の偶数次の項と 0 を交互に並べることは次のように簡単に実装できる. 左半分が 0 となっている 4 ビット列を例に説明する.

$$00AB$$

ここの AB の並びに対応するのが項と考えると, 偶数次の項と 0 の交互の並びは次のように表されることである.

$$0A0B$$

このビット列は次のような手順で作られる.
手順 1:ビット列を 1 ビット左にシフトする.

$$00AB \rightarrow 0AB0$$

手順 2:ビットシフトする前のビット列とビットごとに足し合わせる.(∨:ビットごとの論理和)

$$0AB0 \vee 00AB \rightarrow 0A?B$$

手順 3:使用しないビットをクリアする. (∧:ビットごとの論理積)

$$0A?B \wedge 0101 \rightarrow 0A0B$$

このようにして, 偶数次の項と 0 と交互の並びを得る. より高い次数の場合は, 手順 1 から 3 と同様の手順を繰り返すことで偶数次の項と 0 を交互に並べることができる [3].

以上のような手順を利用することによって 2 乗計算のプログラムは従来法 2 のプログラムよりもさらに計算量を減らすことができる. 図 5 は $GF(2^3)$ における変数 a の 2 乗計算をする提案法のプログラムであり, 上記の手順を利用している. 図 4 のプログラムに比べて計算量が少なくなっていることがわかる.

```
r = a & 0x3;
r = (r ^ (r << 1)) & 0x5;
r ^= (a & 0x04)? 0x06: 0; //原子多項式で次数を落とした項を足し合わせる
```

図 5　原始多項式を演算済み定数とした有限体 2 乗計算の演算プログラム (提案法)

3.4　実験結果と検証

従来法 1, 従来法 2, 提案法のプログラムを用意しそれぞれの次数に対して同じ回数プログラムを実行し 2 乗計算の演算時間を測定する実験を行った. 表 4 に従来法 1, 従来法 2, 提案法のプログラムを実行した時の演算時間と時間比の結果を示した. 従来法 1 と従来法 2 で行った計算は乗算のアルゴリズムを用いており, 先に述べた 2 乗計算では偶数次の項が残るという性質は用いていない. 表 4 で, 次数が大きくなるたびに演算時間が短くなったり長くなったりしているが, これは次数が大きくなると演算量は指数関数的に多くなり計測時間も長くなることに対処するため, プログラムを回す回数を減らしているからである.

表 4 のデータに基づき, 図 6 に従来法 1 と従来法 2 のプログラムを実行した時の演算時間の時間比を示した. 図 6 から全ての次数で従来法 2 は従来法 1 よりも演算時間は高速となっていることがわかる. これは従来研究 [2] の追実験である.

次に, 表 4 のデータに基づき, 図 7 に従来法 1 と提案法の演算時間の時間比を示した. この図 7 に示した結果は乗算のアルゴリズムを用いた 2 乗計算と, 2 乗計算法の性質を用いた提案法との比較となっている. 図 7 より全ての次数で提案法は従来法 1 よりも演算速度が高速となっていることがわかる. 次数が高くなっていくと実行時間は 0.5 倍から, 0.2 倍となり大きな改善が見られた. 分岐命令をなくしたことに加え, 計算量を従来法 2 より短くしたことが要因であると考えられる.

次に, 表 4 のデータに基づき, 図 8 に従来法 2 と提案法の演算時間の時間比を示した. 全ての次数で提案法は従来法 2 よりも演算速度が高速となっていることがわかった. また, 2 乗計算に特化することで乗算の演算時間が 0.6 倍から, 0.4 倍になることがわかった.

次に, 図 6, 7, 8, 全てに言えるが次数を大きくすると演算時間の比が小さくなっている. 次数が大きくなると演算量は指数関数的に増えるが演算量と同時に演算済みの定数も増える. このため次数が大きくなるほど演算時間の比が小さくなっていると考えられる.

最後に, 次数が 33 以上のときについてであるが, 本研究では 32 ビット計算機を用いたため工夫なしに 33 ビット以上の計算を行うことが出来ない. 33 ビット以上の計算を行うためには従来研究 [1] で用いられた多倍長リンクを実装するといったことが考えられるが, この工夫により, 32 ビット以下の演算にも影響を与えてしまうのではないかとも考えられる.

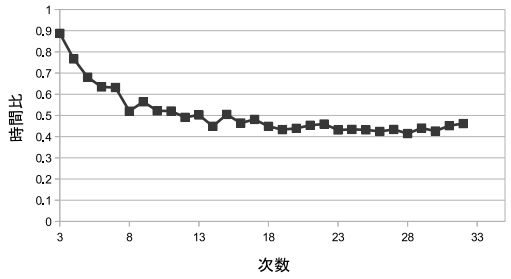

図 6　従来法 1 と従来法 2 の時間比

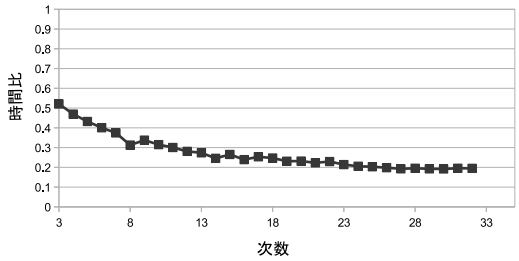

図 7　従来法 1 と提案法の時間比

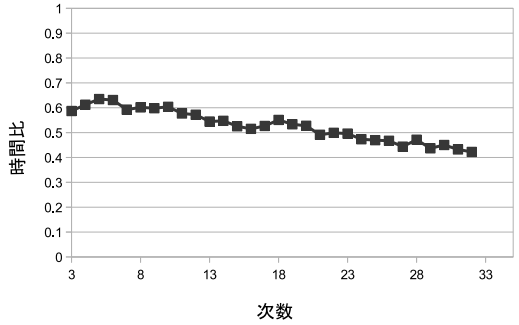

図 8　従来法 2 と提案法の時間比

表 4　提案法, 従来法の時間比

ビット	提案法 $t_1[s]$	従来法 2　$t_2[s]$	従来法 1　$t_3[s]$	t_1/t_2	t_1/t_3	t_2/t_3
3	4.477	7.628	8.596	0.587	0.521	0.887
4	5.023	8.205	10.701	0.612	0.469	0.767
5	4.368	6.879	10.108	0.635	0.432	0.680
6	4.617	7.316	11.528	0.631	0.401	0.635
7	4.602	7.768	12.292	0.592	0.374	0.632
8	4.851	8.065	15.506	0.601	0.313	0.520
9	5.179	8.658	15.334	0.598	0.338	0.565
10	4.399	7.285	13.962	0.604	0.315	0.521
11	4.383	7.597	14.586	0.577	0.300	0.521
12	3.712	6.489	13.213	0.572	0.281	0.491
13	3.712	6.817	13.572	0.545	0.274	0.502
14	3.582	7.036	15.678	0.547	0.246	0.449
15	3.853	7.332	14.523	0.526	0.265	0.504
16	3.198	6.208	13.369	0.515	0.239	0.464
17	3.369	6.396	13.291	0.527	0.253	0.481
18	6.988	12.682	28.267	0.551	0.247	0.449
19	5.584	10.467	24.164	0.533	0.231	0.433
20	5.818	11.029	25.116	0.528	0.232	0.439
21	5.803	11.809	26.037	0.491	0.229	0.454
22	4.851	9.718	21.153	0.499	0.229	0.459
23	4.836	9.75	22.589	0.496	0.214	0.432
24	4.992	10.545	24.289	0.473	0.206	0.434
25	3.993	8.502	19.671	0.470	0.203	0.423
26	4.118	8.814	20.763	0.476	0.198	0.425
27	8.221	18.532	42.697	0.444	0.193	0.434
28	6.318	13.4	32.323	0.471	0.195	0.415
29	6.333	14.493	32.916	0.437	0.192	0.440
30	6.552	14.554	34.132	0.450	0.192	0.426
31	6.537	15.116	33.446	0.432	0.195	0.432
32	6.738	15.958	34.569	0.422	0.195	0.462

4　プログラムの自動生成

本研究ではビット数と原始多項式を固定した提案法プログラムを一つ一つ手作業で書くのではなく, 与えられたビット数と原始多項式に対して自動生成を行えるようにした. 図 9 は本研究で使用した自動生成プログラムである. このプログラムは, 定数とできる部分を計算し, 繰り返しも展開された形でソースコードを生成する. このように柔軟なソースコードを生成する機能は，C++ のクラステンプレートには現在のところ実装されていない [4]. 3.4 節の実験では自動生成で得られたプログラムを用いている.

・1 段階目のプログラム

```
printf("\t" "unsigned int r;\n");
        printf("\t" "r = a & 0x%x;\n", (1 << ((n+1)/2)) - 1);
        switch (n){
        case 32: case 31: case 30: case 29: case 28: case 27: case 26: case 25:
        case 24: case 23: case 22: case 21: case 20: case 19: case 18: case 17:
                printf("\t" "r = (r ^ (r << 8)) & 0x%x;\n", 0x00FF00FF & mask);
        case 16: case 15: case 14: case 13: case 12: case 11: case 10: case 9:
                printf("\t" "r = (r ^ (r << 4)) & 0x%x;\n", 0x0F0F0F0F & mask);
        case 8: case 7: case 6: case 5:
                printf("\t" "r = (r ^ (r << 2)) & 0x%x;\n", 0x33333333 & mask);
        case 4: case 3:
                printf("\t" "r = (r ^ (r << 1)) & 0x%x;\n", 0x55555555 & mask);
        case 2: case 1:
                break;
        }
        q = p;
        if (n % 2){
                if (q & (1 << (n - 1))){
                        q = ((q << 1) ^ p) & mask;
                } else {
                        q <<= 1;
                }
          }
        for (i = (n + 1) / 2; i < n; i++){
                printf("\t" "r ^= (a & 0x%x)? 0x%x: 0;\n", 1 << i, q);
                if (q & (1 << (n - 1))){
                        q = ((q << 1) ^ p) & mask;
                   } else {
                        q <<= 1;
                }
                if (q & (1 << (n - 1))){
                        q = ((q << 1) ^ p) & mask;
                   } else {
                        q <<= 1;
                   }
          }
```

図 9　有限体 2 乗演算プログラムを自動生成するプログラム

図 10 は 1 段階目のプログラムから自動生成することで得られた演算プログラムである. また, 図 1 0 は $GF(2^3)$ の場合のものであり, プログラム中の 0x5 という部分が 101 を表すため, 3 ビットの例において偶数次の項と 0 が交互に並ぶということを表している. ビットが大きくなると, 3.3 節で述べた手順 [3] の量も多くなるため手順に応じたプログラムが自動生成されるようになっている.

・2 段階目のプログラム

```
r = a & 0x3;
r = (r ^ (r << 1)) & 0x5;
r ^= (a & 0x04)? 0x06: 0; //原子多項式で次数を落とした項を足し合わせる
```

図 10　自動生成されたプログラム

5　まとめ

NTL で公開されている有限体演算のプログラムでは原始多項式と次数が変数として宣言されており, 計算量と分岐命令が多くなる. 従来研究 [2] では, 原始多項式と次数を固定することで, 計算量と分岐命令を減少させ, 乗算と除算の高速化を実現している. 本研究では, 同じアプローチで二乗演算の高速化を実現した.

本研究で高速化が可能に出来た要因として, 計算できるのは 2 乗のみ, 標数を 2 として考えるといった特定の場合の演算方法であるということが考えられ, NTL のような汎用性の高いプログラムにはなっていない. しかし, 標数 2 としコンピュータの構造に合わせた設計となっているため実際に使える場合もあるのではないかと考えられる. 今後の課題としてはこの高速化技術を手軽に利用できるユーザ環境を考えることが挙げられる.

謝辞

本研究を行うにあたり, 終始一貫して丁寧なご指導をご指導を賜わりました指導教員の西新幹彦准教授に心より感謝の意を申し上げます.

参考文献

[1] http://shoup.net/ntl/index.html, 2012 年 5 月閲覧.

[2] 乙井良太, 「固定した原始多項式による標数 2 の有限体演算の高速化に関する検討」, 信州大学工学部, 学士論文, 2012 年.

[3] ヘンリー・S・ウォーレン, ジュニア, 「ハッカーのたのしみ 本物のプログラマはいかにして問題を解くか」, 星雲社, 2007.
[4] Steve Oualline, 「C++ 実践プログラミング第 2 版」, オライリー・ジャパン, 2006.

付録 A　本研究で用いた原始多項式

本研究で用いた次数ごとの原始多項式を表 5 に示す.

表 5　原始多項式一覧

次数 n	n 次の原始多項式
3	$x^3 + x + 1$
4	$x^4 + x + 1$
5	$x^5 + x^2 + 1$
6	$x^6 + x + 1$
7	$x^7 + x + 1$
8	$x^8 + x^7 + x^2 + x + 1$
9	$x^9 + x^4 + 1$
10	$x^{10} + x^3 + 1$
11	$x^{11} + x^2 + 1$
12	$x^{12} + x^8 + x^2 + x + 1$
13	$x^{13} + x^5 + x^2 + x + 1$
14	$x^{14} + x^{12} + x^2 + x + 1$
15	$x^{15} + x + 1$
16	$x^{16} + x^{12} + x^3 + x + 1$
17	$x^{17} + x^3 + 1$
18	$x^{18} + x^7 + 1$
19	$x^{19} + x^5 + x^2 + x + 1$
20	$x^{20} + x^3 + 1$
21	$x^{21} + x^2 + 1$
22	$x^{22} + x + 1$
23	$x^{23} + x^5 + 1$
24	$x^{24} + x^7 + x^2 + x + 1$
25	$x^{25} + x^3 + 1$
26	$x^{26} + x^6 + x^2 + x + 1$
27	$x^{27} + x^5 + x^2 + x + 1$
28	$x^{28} + x^3 + 1$
29	$x^{29} + x^2 + 1$
30	$x^{30} + x^{23} + x^2 + x + 1$
31	$x^{31} + x^3 + 1$
32	$x^{32} + x^{22} + x^2 + x + 1$