社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＣＳＥＣ－22　(6)
2003／7／17

太田乗算法にレジスタブロッキングやキャリーセーブ手法の適用に関する研究

テイ　チョユウ†　太田　昌孝†　荒木　純道‡

†東京工業大学情報理工学研究科　〒152-8552 東京都目黒区大岡山 2-12-1
‡東京工業大学理工学研究科　〒152-8552 東京都目黒区大岡山 2-12-1

E-mail: †zheng@mobile.ee.titech.ac.jp, mohta@necom830.hpcl.titech.ac.jp ‡araki@mobile.ss.titech.ac.jp

あらまし　キャリーの伝搬をまとめて行うキャリーセーブ手法は高速ハードウェア乗算器で使われる伝統的な技法であり、レジスタブロッキングは行列の乗算の高速なコーディング手法であるが、太田は両手法が多倍長乗算にも有効であることを示した。本研究では多倍長乗算に基づく剰余乗算法である太田乗算法にキャリーセーブとレジスタブロッキング手法を適用して実験し、一方，剰余乗算のモンゴメリ乗算法にも従来知られてきた最適の方式に基づく効率的なコーディングを採用し Intel 社の Pentium4(1.8GHz)のプロセッサ上で両者の性能を比較した．C 言語によるコーディングで 220bit の剰余乗算に対して，太田乗算法は 1.74μs 計算時間，モンゴメリ乗算法は 1.79μs の計算時間が得られ、太田乗算法はモンゴメリ乗算法より約 3%高速になった．作成したプログラムは，公開鍵暗号特に楕円暗号の計算に必要な程度の桁数に対して有効性と期待できる。

キーワード　太田乗算法，モンゴメリ乗算法，剰余乗算，レジスタブロッキング，キャリーセーブ，ハイパフォーマンスコンピューティング，楕円暗号，RSA 暗号

Research on the applications of Register Blocking Technique and Carry Save Technique to Ohta Method

ZHENG Chuyu†　Masataka OHTA†　and　Kiyomichi ARAKI‡

† Graduate School of Information Science and Engineering, Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo, 152-8552 Japan
‡ Graduate School of Science and Engineering, Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo, 152-8552 Japan

E-mail: †zheng@mobile.ee.titech.ac.jp, mohta@necom830.hpcl.titech.ac.jp ‡araki@mobile.ss.titech.ac.jp

Abstract　Carry save technique, which delays propagations of carries, is a classical technique for fast hardware multiplication and register blocking technique is a efficient coding technique for matrix multiply, both of which was shown by Ohta to be applicable to multiprecision multiplication. In this paper, effectiveness of carry save and register blocking techniques for Ohta Multiplication, which is a modular multiplication using multiprecision multiplications, is compared against Montgomery Multiplication with most efficient coding known so far on Intel Pentium4 (1.8GHz) processor. For 220 bits of modular multiplication coded with C, Ohta Multiplication and Montogomery one take 1.74us and 1.79us, correspondingly, that Ohta Multiplication is about 3% faster than Montgomery one. The program is expected to be applicable to computations of public key cryptography, esiecially Elliptic one.

Keyword　Ohta method, Montgomery method, modular multiplication, Register Blocking, Carry Save, High Performance Computing, Elliptic Curve Cryptography, RSA Cryptography

1. はじめに

近年公開鍵暗号の分野では，RSA 暗号[1]や楕円曲線暗号 ECC[2][3]の暗号化・復号化における基本演算である剰余乗算は，高速計算の要求が高まっている．剰余乗算を行う方法は色々があるが，その中では除算を用いずにモンゴメリ乗算法[4]は，高速アルゴリズムとして広く使用されている．しかし，モンゴメリ乗算法の演算では，２重ループ内に存在する二つのループには依存関係があるため、その順序を入れ替えることはできず、また、途中結果を正規化された形で持つ必要

がある。そのため，多倍長乗算の２重ループによる積和演算で、加算の順序を入れ替えられることを利用したレジスタブロッキング手法や、加算の結果を正規化しないまま加算を続けられることを利用したキャリーセーブ手法[5] という高速なコーディング手法はモンゴメリ乗算法に適用できない．一方，以前我々が提案したもう一種の高速な剰余乗算の太田乗算法[6]には，多倍長の乗算が完全な２重ループ積和演算であるため，これらの高速なコーディング手法は適用できる．

そこで本論文では，レジスタブロッキング手法とキャリーセーブ手法を適用した太田乗算法と，ソースコードが最適化されたモンゴメリ乗算法を実験した．Intel 社の Pentium4(1.8GHz)のプロセッサ上，C 言語の実装では，220 ビットの剰余乗算に対して，太田乗算法が約 1.74 μ s，モンゴメリ乗算法が約 1.79 μ s の計算時間が得られた．太田乗算法がモンゴメリ乗算法より3%高速のことは示された．

次章以下，第２章ではモンゴメリ乗算法を述べる．第３章では太田乗算法を述べる．第４章ではレジスタブロックング手法を述べる．第５章ではキャリーセーブ手法を述べる．第６章ではソフトウェア実装による速度比較結果を示す．第７章ではまとめと今後の課題について述べる．

2. モンゴメリ乗算法

p を $n$bit の素数として，素体 $GF(p)$ 上の整数 a,b の積 c は次式で表される．

$$c = ab \bmod p \quad (0 \le a,b < p < 2^n) \tag{2.1}$$

試行乗算を大量に含む通常の除算を用いればすなおに c を計算できるが，膨大な計算時間がかかる．

そこでモンゴメリ乗算法は，除算を用いずに加減乗算とシフト演算のみによる高速剰余乗算を提供する．

2.1. モンゴメリ乗算法アルゴリズム

モンゴメリ乗算の重要部分は $REDC$ 関数から構成される．　　(x,y : $n$bit の整数、 p :素数)

$0 \le x,y < p < 2^n, R = 2^n$ に対して

$$z = REDC(x,y) = xyR^{-1} \bmod p \tag{2.2}$$

を計算する．

m 個のブロックに分割したモンゴメリ乗算法のアルゴリズム[7][8]は以下のようになる．(ただし，$B = 2^l$)

[Algorithm 1]

$X = \{x_{m-1}, \cdots, x_1, x_0\}_B, Y = \{y_{m-1}, \cdots, y_1, y_0\}_B$
$P = \{p_{m-1}, \cdots, p_1, p_0\}_B, 0 \le x,y < p,$
$R = B^m, \gcd(p,B) = 1, p' = -p_0^{-1} \bmod B$
$Z = XYR^{-1} \bmod P \quad (Z = (z_{m-1}, \cdots, z_1, z_0))$
1. $T = 0$ $(t = (t_m, \cdots, t_1, t_0))$
2. for i from 0 to $(m-1)$ do:
　　$T = T + y_i X$
　　$u = t_0 p' \bmod B$
　　$T = (T + uP)/B$
3. if $(T \ge P)$ then $Z = T - P$
　else $Z = T$

2.2. モンゴメリ乗算法の最適化プログラム

ANSI の C 言語では，int 宣言した整数のメモリサイズは 4 バイトである．そのため，二つの整数の積を一つの 4 バイト整数に収める観点から，実際使用したメモリサイズは 2 バイトとなる．つまり，１ブロッサイズは 16 ビットである．本実験では，220bit の多倍長整数が使用されるため，ブロック数は 14 となる．

ここで，[Algorithm 1]のステップ 2 に対して，最適されたソースコードは以下のように示される．

```
#define LOMASK 0X0000FFFF
void redc(x, y, p, p0, z)
unsigned int *x, *y, *p, p0, z
{
  省略
  for (i=0; i<14; i++)
  {
  //z = z + y_i * x
    ec = y[i] * x[0] + z[0];
    z[0] = ec & LOMASK;
    ec = y[i] * x[1] + z[1] + (ec >> 16);
    z[1] = ec & LOMASK;
    ec = y[i] * x[2] + z[2] + (ec >> 16);
    z[2] = ec & LOMASK;
    ...
    ec = y[i] * x[13] + z[13] + (ec >> 16);
    z[13] = ec & LOMASK;
    z[14] = ec >> 16;

    u =(z[0] * p0) & LOMASK;

    ec =u * p[0] + z[0];
    ec =u * p[1] + z[1] + (ec >> 16);
    z[0] = ec & LOMASK;
    ec = u * p[2] + z[2] + (ec >> 16);
    z[1] = ec & LOMASK;
    ...
    ec = u * p[13] + z[13] + (ec >> 16);
    z[12] = ec & LOMASK;
```

```
    z[13] =z[14]+ (ec >> 16);
  }
  省略
}
```

3. 太田乗算法

モンゴメリ乗算法と同様に，太田乗算法[6]は除算を用いずに，多倍長の乗算と加減算やシフト演算のみで剰余乗算を高速に行えるアルゴリズムである．

3.1. 太田乗算法のアルゴリズム

剰余乗算式　$c = ab \bmod p$　　(3.1)

$0 \le a, b < p$ において，$\mathrm{int}(R^2/p)$を太田定数Kとし，Kをあらかじめ準備しておくことにより式(3.1)を以下のように計算することができる．

[Algorithm 2]

$c = ab \bmod p$

1. $H = ab$
2. $d = \mathrm{int}(HK/R^2)$
3. $f = H - dp$
4. if $f \ge p$ then $c = f - p$
   else $c = f$

$R = 2^n$とすると，[Algorithm 2]のステップ2において，$\mathrm{int}(HK/R^2)$の計算は単にHKの上位2nビットを取り出す操作となる．

3.2. 省略型の太田乗算法

[Algorithm 2]のステップ2において，$R/2 < p < R$を満たすRを選べば，$\mathrm{int}(HK/R^2)$の計算が削減できる．さらに，$\mathrm{int}(HK/R^2)$に1の誤差を許容すれば，HKの計算において下位$2n - \log_2 n$ビットの計算を省略できる．

[Algorithm 2]のステップ3において，太田乗算法での誤差と省略計算$\mathrm{int}(HK/R^2)$で生じた誤差から，dpとHの差は0~3pの間となる．

よって，省略型の太田乗算法のアルゴリズム[6]は以下のように示される．

[Algorithm 3]

1. $H = a*b$

2'. $d = \mathrm{int}(HK/R^2)$ or $d = \mathrm{int}(HK/R^2) - 1$

3'. $f = H - dp$

4. if $f \ge d$ then $c = f - d$ else $c = f$

5. if $c \ge d$ then $c = c - d$

3.3. モンゴメリ乗算法との比較

mブロック分割のモンゴメリ乗算法と省略型太田乗算法において，1回の剰余乗算に必要な$GF(p)$上の加減算と乗算の数を表3.1[6]にまとめた．

ただし，場合により発生しないことのある演算回数もすべて計算されている．

表3.1 各手法における必要な演算回数一覧表

演算	乗算回数	加減算回数
モンゴメリ法 REDC 関数	$2m^2 + m$	$4m^2 + 3m$
省略型太田法	$2m^2 + 3m + 2$	$4m^2 + 9m + 4$

表3.1から，省略型太田乗算法はモンゴメリ乗算法(REDC 関数)より若干演算回数が多いが，第4章のレジスタブロッキング手法と第5章のキャリーセーブ手法を適用したら，省略型太田乗算法がモンゴメリ乗算法(REDC 関数)より高速のことは可能となる．

4. レジスタブロックング手法

コンピュータのアーキテクチャーでは，演算対象はメモリからレジスタ内に貯蓄されてから，プロセッサに演算させる．演算を行う度に，毎回メモリからデータを呼び出すことは処理速度が非常に遅くなる．そのため，レジスタの再更新までレジスタに貯蓄したデータをできる限り利用する．つまり，できるだけメモリのアクセス回数を減らす．この観点に応じて，レジスタブロックング手法[5]というコーディング手法が存在する．特に，この手法が多倍長の乗算に対して有効であると知られている．

多倍長の乗算では，素朴なプログラムは以下のように示される．

```
void mul(a, b, c)
unsigned int *a, *b, *c
{
  for ( i=0; i<14; i++)
  {
    ec = 0;
    for ( j=0; j<16;j++)
    {
      ec = a[i] * b[j] + c[i+j] + (ec >> 16);
      c[i+j] = ec & LOMASK;
    }
    c[i+j] = (ec >> 16);
  }
}
```

このプログラムを 3×3 レジスタブロックングに変更して，以下のようになる．

```
void mul(a, b, c)
unsigned int *a, *b, *c
{
  unsigned int a0,a1,a2,b0,b1,b2;
  a0=a[0]; a1=a[1]; a2=a[2];
```

```
b0=b[0]; b1=b[1]; b2=b[2];
c[0]=a0*b0;
c[1]=a1*b0+a0*b1;
c[2]=a2*b0+a1*b1+a0*b2;
c[3]=a2*b1+a1*b2;
c[4]=a2*b2;
b0=b[3]; b1=b[4]; b2=b[5];
c[3]+=a0*b0;
c[4]+=a1*b0+a0*b1;
c[5]=a2*b0+a1*b1+a0*b2;
c[6]=a2*b1+a1*b2;
c[7]=a2*b2;
...
}
```

モンゴメリ乗算法では，ループ内に依存関係が存在しており，計算結果の順序をかえられないため，レジスタブロッキング手法が適用できない．一方，太田乗算法では，[Algorithm 3]のステップ 1,2,3 の乗算はすべて２重ループの多倍長乗算であるため，レジスタブロッキング手法の適用が可能となる．

5. キャリーセーブ手法

２重ループの多倍長乗算において，毎回通常の乗算を行った直後，キャリー上げ伝搬の作業を行う．そのため，処理時間が遅くなる．そこで，キャリーを毎回繰り上げずに，計算の最後段階にだけキャリー上げ伝搬するように，アルゴリズムに改良できる．この手法はキャリーセーブ手法[5]と言われている．キャリーセーブは、途中結果の総和がハードウェアで加算できる整数のサイズ(32 ビット)を超えない限り行える。

本実験では，使用した多倍長の整数サイズが 220 ビットであるため，ブロックサイズを 16 ビットから 14 ビットに変更すれば,乗算の結果はちょうど 32 ビットのメモリサイズに収められる．ブロックサイズは 14 ビットに変更されたため，ブロック数は 14 から 16 になった．

キャリーセーブ手法のプログラムは以下のように示される．

```
void mul(a, b, c)
unsigned int *a, *b, *c
{
    for ( i=0; i<16; i++)
    {
      for ( j=0; j<16; j++)
      {
          c[i+j] += a[i] * b-[j];
      }
    }
    ec=0;
    for ( i=0; i<32; i++)
    {
        ec +=c[i];
        c[i]=ec & 0X00003FFFF;
        ec=ec>>14;
    }
}
```

レジスタブロックング手法と同様に，モンゴメリ乗算法はループ間の依存関係のため前の結果が正規化されている必要があり，キャリーセーブ手法も適用できない．太田乗算法において，すべての乗算が多倍長乗算であるため，キャリーセーブ手法は適用できる．

6. ソフトウェアの実装評価

レジスタブロッキング手法とキャリーセーブ手法を繰り合わせて，太田乗算法を改良した．一方，モンゴメリ乗算法も第２章のようにプログラムを最適化した．

実験環境は以下のように示される．

プロセッサ：Intel 社の Pentium4(1.8GHz)
メモリ：256MB
OS: redhat 9.0
開発言語：C 言語
コンパイラ：GCC 3.2.2
コンパイラオプション：
　　-O1
　　-fomit-frame-pointer
　　-fschedule-insns2
　　-fno-force-mem
　　-fno-force-addr
多倍長の整数サイズ：220 ビット

レジスタブロッキング手法において，2×2，3×3 と 4×4 のブロック化計算を実験してみた．3×3 ブロック化計算は最も高速であることがわかった．したがって，今回の実験では，3×3 ブロック化計算を使用した．

省略型太田乗算法とモンゴメリ乗算法の乱数を入力とする 10 回の平均測定時間の実験結果は以下のように示される．

省略型太田乗算法：1.74 μ s
モンゴメリ乗算法：1.79 μ s

7. むすび

本論文では，レジスタブロッキング手法とキャリーセーブ手法が適用した省略型太田乗算法はモンゴメリ乗算法より 3%高速であることは示された．

レジスタブロッキングのブロックは大きくすればするほど効率的だが、それだけ多数のレジスタを消費

する。本実験で利用した Pentium は整数型レジスタの数が少ないが、浮動小数点レジスタと浮動小数点演算を利用したり、多くのレジスタをもつのが当然の RISC では、より効率的な実装が期待できる。また，RSA 暗号を念頭に、1024 ビットや 2048 ビットの多倍長乗算鍵サイズを実装すれば，モンゴメリ乗算法よりさらに高速になると期待できる。

文　献

[1] R.L.Rivest, A.Shamir and L.Adleman, "A method for obtaining digital signatures and public key cryptosystems," Commun. ACM , vol.21, pp120-126, 1978

[2] V.S. Miller, "Use of elliptic curve in cryptography," Advances in Cryptology (Crypto'85). LNCS 218, pp417-426, Springer-Verlag, 1986

[3] N.Koblitz, "Elliptic curve cryptosystems," Math. Comp, vol.48, pp203-209, 1987.

[4] P.L. Montgomery, "Modular Multiplication without Trial Division", Mathematics of Computation, vol. 44, pp.519-521, 1985

[5] 太田昌孝，前野年紀，"多倍長計算の HPC 技術，" ハイパフォーマンスコンピューティング，54-9, pp.61-65, Dec.1994

[6] テイ　チョユウ，太田昌孝，荒木純道，"GF(p)上の楕円曲線暗号における剰余乗算の高速化に関する研究"，電子情報通信学会信学技報，pp.41-47,2003-05

[7] 秋下徹，飯塚健，佐藤英雄，"非接続型ＩＣカード用の楕円曲線ハードウェア実装，" 2002 年暗号と情報セキュリティシンポジウム，SCIS02-15B1, pp.1107-1112, Jan. 2002

[8] 佐藤証，高野光司，大庭信之，"ＧＦ（Ｐ）上の楕円曲線暗号回路のスケーラブルアーキテクチャ，" 電子情報通信学会論文誌 (A),Vol.J85-A, No.11,pp.1264-1272,Nov.2002.