gf-nishida-16: シンプルかつ高速な $GF(2^{16})$ 演算ライブラリ

西田 博史
ASUSA Corporation, Salem Oregon, USA
E-mail: nishida at asusa.net *

2016/1/5

概要

本報告書ではシンプルで高速な $GF(2^{16})$ 演算ライブラリである gf-nishida-16 の紹介とその詳細について述べる。演算速度の計測では他のオープンソースのガロア体演算ライブラリと比較し、非常に優れた結果を出している。gf-nishida-16 のソースコードは C で書かれており、2-Clause BSD ライセンスの下に公開されている。

1 初めに

ガロア体での演算を多用するプログラムにおいて、その演算速度がプログラム全体の処理速度を大きく左右することがある。例えば Random Network Coding (RNC) においてエンコーディング、デコーディングに消費される処理量は膨大であるが、その際にはガロア体ににおける加減乗除算が用いられるため、 RNC をベースにしたプログラムの処理速度はガロア体での演算速度に大きく依存する。我々の開発した RNC による分散データシステム RNCDDS も例外ではなく、開発初期段階で効率の良くないガロア体演算ライブラリを使用したため、実用的な処理速度に到達させることが困難であった。このため我々は 16bit 以上で高速なガロア体ライブラリの開発を余儀なくされ、gf-nishida-16 を産み出す結果となった。我々はこの gf-nishida-16 を最も制限の少ない 2-Clause BSD (FreeBSD) ライセンスの下に公開することを決め、大勢の開発者達の役に立つことを願っている。

2 一般的な $GF(2^n)$ 演算ライブラリ

8bit GF における高速な乗除算は、事前にメモリテーブルに演算結果を保存しておき、以下のような単純なメモリ検索で求めることができる。

```
uint8_t GF8mulTbl[256][256];
uint8_t GF8divTbl[256][256];

// Returns a * b in GF(2^8)
void GF8mul(uint8_t a, uint8_t b)
{
    return GF8mulTbl[a][b];
```

* gf-nishida-16 は ASUSA Corporation における Random Network Coding Project の一環として開発されたものです

```
}

// Returns a / b in GF(2^8)
void GF8div(uint8_t a, uint8_t b)
{
    return GF8divTbl[a][b];
}
```

ここで GF8mulTbl および GF8divTbl はそれぞれ乗算、除算の演算結果を蓄えたメモリテーブルである。この手法で使用されるメモリ量はわずか 2^{17} (128k) バイトで、原始多項式が定数であれば非常に効率の良い手法と言える。しかしながら同等の手法を $GF(2^{16})$ に適用すれば 2^{34} (16G) バイトのメモリが必要となり実用性の範疇を超える。

RNC や Erasure Coding (EC) では以下の理由で少なくとも 16bit のガロア体演算ライブラリを用いるのが好ましい。RNC では下記のような連立一次方程式を作ることによってエンコーディングを行い、その連立一次方程式を解くことによってデコーディングを行う。

$$\begin{cases} [l]a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 = b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 = b_1 \\ \cdots \end{cases} \tag{1}$$

その際に係数 $a_{i,j}$ を乱数により生成した場合、$GF(2^8)$ ではガロア体での空間が狭すぎるためにこれらの方程式の独立性を常に維持することが困難となり、結果的に高確率で連立一次方程式の解が求められなくなる。しかし 16bit 以上ではその確率が格段に低くなり、少数のエンコードされたファイルからのデコーディング時の失敗の可能性も大きく減る。ただし $GF(2^{16})$ においても依然として連立一次方程式における方程式の重複がある程度の確率で起こり得ることに留意する必要がある。我々の分散データシステムにおける実験では、約 20 万個のファイルに対して平均 4 から 5 個のファイルに方程式の 重複が見つかり、デコーディング（ファイルの復元）ができない状態であったことが判明している。この確率は高いものではないが、実際には更に多数のファイルを扱うことを考慮し、我々のシステムでは方程式の重複がないよう係数を選択している。なお同等の手法が $GF(2^8)$ に対しても適用し得るが、既存の方程式から新たなる方程式を発生させる際に高い確率で重複を産み出す。詳細は [1] に記されている。

SSE を使った $GF(2^n)$ での演算はその処理速度を考えると理想的であると言える。実際 Plank らによって作られた GF-Complete [2] は 64bit のガロア体における乗算で非常に優れた結果を出している（第 5 章参照）。しかし残念ながら GF-Complete は StreamScale, Inc 社に特許を侵害していると指摘され、すでに作者により削除されている。Plank は自身のホームページ [3] で「今後 GF-Complete を商用目的で使用することは米国特許に抵触するため GF-Complete を保守しない」旨を記述している。これにより我々は SSE を使用したガロア体における乗除算は特許侵害の可能性があるとして、使用不可と認識している。憂うべきことではあると思われるが、これが我々の代替手段の模索へと繋がった一つの理由である。

Intel は 2010 年、自社 CPU に Carry-less Multiplication (CLMUL) という $GF(2^n)$ の演算に特化した命令セットを搭載した。この命令セットを使用した Erasure Coding の処理に関する特許も存在するが [4]、我々の計測では充分な高速性を見出せていない（第 5 章の gf-solaris-128 参照）。

Plank は [5] で 8 から 32bit までのガロア体に対応した別のライブラリを公開している。今回の調査の中でこのライブラリに含まれている対数テーブルを使った関数が我々の gf-nishida-16 と同等の手法を使ってることが判明した。しかしながら gf-nishida-16 はプログラミングにおいても高速化による工夫がなされており、

Plank のものよりも良い計測結果を残している。このライブラリの計測結果は gf-plank と gf-plank-logtable として第 5 章に掲載されている。

3 gf-nishida-16 の仕組み

この章では gf-nishida-16 の仕組みについて説明する。まず次のような二つのメモリ空間を用意する。

```
uint16_t GF16memL[65536 * 4];
uint32_t GF16memIdx[65536];
```

そして $0 \leq i \leq 65534$ の範囲で `GF16memL[i]` に $GF(2^{16})$ 方式で 2^i の値を入力する。ここで言う $GF(2^{16})$ 方式とは「もし 2^i が 65536 以上であれば $2^i \hat{} P$ （P は $2^{16}+2^{12}+2^3+2^1+1 = 0\text{x}1100\text{b}$ のような原始多項式で、^ は XOR）を代わりに入力する」を意味する。結果的に `GF16memL` の初め 1/4 は

```
uint16_t t;
uint32_t i, n;

GF16memL[0] = t = 1;
for (i = 1; i <= 65534; i++) {
    n = (uint32_t)t << 1;
    GF16memL[i] = t = (uint16_t)((n >= 65536) ? n ^ P : n);
}
```

のように値を入力することになる。またこのループの中で `GF16memIdx` にも以下のように数値を入れていく。

```
uint16_t t;
uint32_t i, n;

GF16memL[0] = t = 1;
for (i = 1; i <= 65534; i++) {
    n = (uint32_t)t << 1;
    GF16memL[i] = t =
          (uint16_t)((n >= 65536) ? n ^ P : n);
    GF16memIdx[t] = i;
}
```

これにより $1 \leq a \leq 65535$ を満たす任意の数値 a は

$$a = 2^{\texttt{GF16memIdx}[a]} = \texttt{GF16memL[GF16memIdx}[a]] \tag{2}$$

となり、同じく任意の $1 \leq b \leq 65535$ な b との乗算 $a \times b$ は

$$\begin{aligned}[l] a \times b &= 2^{\texttt{GF16memIdx}[a]} \times 2^{\texttt{GF16memIdx}[b]} \\ &= 2^{\texttt{GF16memIdx}[a]+\texttt{GF16memIdx}[b]} \\ &= \texttt{GF16memL[GF16memIdx}[a] + \texttt{GF16memIdx}[b]] \end{aligned} \tag{3}$$

と表すことが可能である。結果的に $GF(2^{16})$ における乗算を行う関数 `GF16mul(a, b)` は

```
#define GF16mul(a, b)   GF16memL[GF16memIdx[a] + GF16memIdx[b]]
```

と定義することができる。しかしながらここで `GF16memIdx[a] + GF16memIdx[b]` が 65534 を超える可能性があることに注意しなくてはならない。この時点では `GF16memL[65535]` 以降は未定義のため誤った値が返されてしまう。このような事象に対処する際に多くのプログラムは

```
int idx = (GF16memL[GF16memIdx[a] + GF16memIdx[b]]) % 65534;
return GF16memL[idx];
```

のように 65534 との剰余を計算しそれを用いるが、これではプログラムに余分なコードを足してしまうため処理速度の劣化を招く原因となる。これを回避するため我々は GF16memL[65535] 以降の空間を以下のように GF16memL[0-65534] の値を複製して GF16memH として利用する。

```
uint16_t *GF16memH = GF8memL + 65535;
memcpy(GF16memH, GF16memL, sizeof(uint16_t) * 65535);
```

かくして GF16memIdx[a] + GF16memIdx[b] > 65534 においても上で定義された GF16mul(a, b) は正しい値を返すこととなる。

GF16memH は除算にも使用できる。除算は

$$\begin{aligned}[l]a/b &= 2^{\texttt{GF16memIdx}[a]}/2^{\texttt{GF16memIdx}[b]} \\ &= 2^{\texttt{GF16memIdx}[a]-\texttt{GF16memIdx}[b]} \\ &= \texttt{GF16memL}[\texttt{GF16memIdx}[a] - \texttt{GF16memIdx}[b]],\end{aligned} \tag{4}$$

と表すことができ、$GF(2^{16})$ における除算の関数は

```
#define GF16div(a, b)   GF16memL[GF16memIdx[a] - GF16memIdx[b]]
```

のように定義することができる。しかしながら GF16memIdx[a] - GF16memIdx[b] < 0 となる可能性があり、その際にはプログラムは segmentation violation を引き起こす。解決策として GF16memL を GF16memH に置き換える。これにより GF16memH[[GF16memIdx[a] - GF16memIdx[b]] は GF16memL[0-65534] または GF16memH[0-65534] のどれか一つを指すこととなり segmentation violation を防ぐことができる。よって GF16div(a, b) は以下のように再定義する。

```
#define GF16div(a, b)   GF16memH[GF16memIdx[a] - GF16memIdx[b]]
```

ここまで a = 0 || b = 0 のケースに言及してこなかったが、a, b != 0 における GF16mul(0, b)、GF16mul(a, 0) および GF16div(0, b) は 0 を返さなくてはならない（GF16div(a, 0) は未定義）。この条件を満たすため GF16memL の残りの空間 GF16memL[13170-262143] を 0 で埋め、GF16memIdx[0] に $65536 * 2 - 1$ を入れる。

```
memset(GF16memL + (65535 * 2) - 2, 0, sizeof(uint16_t) * 65536 * 2 + 2);
GF16memIdx[0] = 65536 * 2 - 1
```

これを用いると

$$\begin{aligned}[l]\texttt{GF16mul}(0,b) &= \texttt{GF16memL}[\texttt{GF16memIdx}[0] + \texttt{GF16memIdx}[b]] \\ &= \texttt{GF16memL}[(65536 * 2 - 1) + \texttt{GF16memIdx}[b]]\end{aligned} \tag{5}$$

となり、

$$0 \leq \texttt{GF16memIdx}[b] \leq 65534,$$

であるため、 GF16mul(0, b) は GF16memL[131071-196605] のうちいずれか一つを指すこととなるが、その値は常に 0 となる。同様に

$$\begin{aligned}[l]\texttt{GF16div}(0,b) &= \texttt{GF16memH}[\texttt{GF16memIdx}[0] - \texttt{GF16memIdx}[b]] \\ &= \texttt{GF16memL}[65535 + \texttt{GF16memIdx}[0] - \texttt{GF16memIdx}[b]] \\ &= \texttt{GF16memL}[196606 - \texttt{GF16memIdx}[b]]\end{aligned} \tag{6}$$

で、GF16memL[131072-196606] のうち一つを指すこととなり、その値も常に 0 となる。結果、我々の GF16mul() および GF16div() は前述した条件を満たすこととなる。参考までに GF16memL[196607] 以降は GF16mul(0, 0) = GF16memL[262142] のためだけに必要である。

注意して欲しいのは、gf-nishida-16 における a != 0 の際の除算 GF16div(a, 0) は

$$\begin{aligned}[l]\texttt{GF16div}(a,0) &= \texttt{GF16memH}[\texttt{GF16memIdx}[a] - \texttt{GF16memIdx}[0]] \\ &= \texttt{GF16memL}[65535 + \texttt{GF16memIdx}[a] - (65535 * 2 - 1)] \\ &= \texttt{GF16memL}[\texttt{GF16memIdx}[a] - 65534]\end{aligned} \tag{7}$$

および

$$-65534 \leq \texttt{GF16memIdx}[a] - 65534 \leq 0$$

となるため segmentation violation を起こす。プログラマは GF16div(a, 0) の使用を避けるべきである。

4 更なる高速化

Random Network Coding や Erasure Coding で最もよく用いられる計算条件では gf-nishida-16 は更に高速な乗除算を実行することができる。例えば RNC や EC では

```
uint16_t a, x[NUM], b[NUM];

for (i = 0; i < NUM; i++) {
        b[i] = GF16mul(a, x[i]);
}
```

のように一定の a に対しての繰り返し乗算が頻繁に行われる。ここで

$$\begin{aligned}[r]\texttt{GF16mul}(a,x[i]) &= \texttt{GF16memL}[\texttt{GF16memIdx}[a] + \texttt{GF16memIdx}[x[i]]] \\ &= (\texttt{GF16memL + GF16memIdx}[a])[\texttt{GF16memIdx}[x[i]]]\end{aligned} \tag{8}$$

であるため、(GF16memL + GF16memIdx[a]) を以下のように一定値に置換することができ、

```
uint16_t *gf_a = GF16memL + GF16memIdx[a];
```

上記プログラムを

```
uint16_t a, x[NUM], b[NUM];
uint16_t *gf_a = GF16memL + GF16memIdx[a];

for (i = 0; i < NUM; i++) {
        b[i] = gf_a[GF16memIdx[x[i]]];
}
```

のように書き換えることができる。これは演算速度を二倍以上にする（第 5 章の gf-nishida-region-16 を参照）。

5 計測結果

我々は第 4 章で述べたプログラムをもとに、様々な GF 演算ライブラリで乗算と除算における演算時間を計測した。使用したライブラリは以下の通りである。

- **gf-nishida-8**: 第 2 章で述べた 8bit のメモリ検索型ライブラリ。
- **gf-nishida-16**: 第 3 章で詳解した本報告書の主要ライブラリ。
- **gf-nishida-region-16**: 第 4 章で紹介した gf-nishida-16 の高速版。
- **gf-complete**: 第 2 章で紹介した SSE を使用したライブラリ（ [3] [2] 参照)。前述の通り、特許抵触のため現在は使用不可である。
- **gf-sensor608-8**: gf-nishida-8 と同等の手法を使った 8bit ライブラリ。
- **gf-plank**: [5] に基づいたライブラリ。
- **gf-plank-logtable-16**: [5] に紹介されたライブラリで、gf-nishida-16 と同等の手法を用いている（第 2 章参照)。
- **gf-solaris-128**: Solaris のソースコードから抜粋したもので CLMUL を利用した $GF(2^{128})$ プログラム（第 2 章参照)。
- **gf-ff**: [6] からダウンロードされた $GF(2^n)$ 演算ライブラリであるが、詳細を理解していない。
- **gf-aes-gcm-128**: FreeBSD のソースコードから抜粋した特別な手法を用いない $GF(2^{128})$ プログラム。

ライブラリによっては除算の関数を含まないものもあり、gf-nishida-region-16 と同様に一定の値を使った乗除算を対象にした関数を含むものもあったが、それらに対しては計測を行わなかった。計測結果は図 1、2 の通りで、値が小さいほど高速であることを示している。

結果として gf-nishida-16、特に gf-nishida-region-16 は乗算、除算両方において非常に高いパフォーマンスを示している。第 2 章で述べたように Random Network Coding や Erasure Coding では 8bit の利用に問題があること、並びに SSE を使った $GF(2^n)$ での演算は特許抵触の恐れがあることから、RNC には gf-nishida-16 の利用が最適な選択であると言える。

参考文献

[1] K. Nguyen, T. Nguyen, Y. Kovchegov, and V. Le, "Distributed data replenishment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2, pp. 275–287, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2012.115

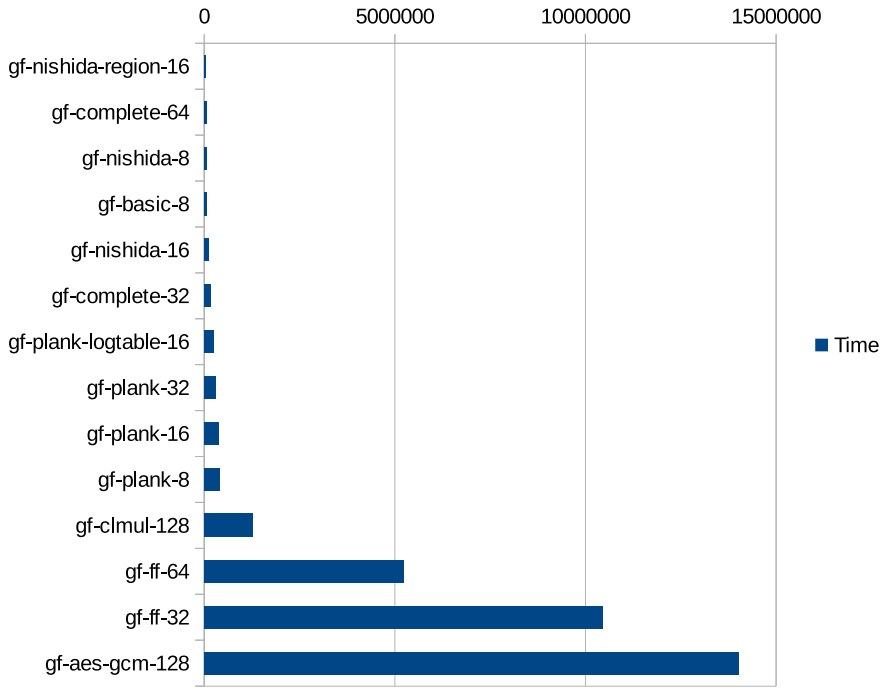
[2] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel simd instructions," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, Feb. 2013, pp. 298–306. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/plank_james_simd

[3] e. a. James S. Plank, "Gf-complete: A comprehensive open source library for galois field arithmetic, version 1.0." [Online]. Available: http://web.eecs.utk.edu/~plank/plank/papers/CS-13-716.html

[4] I. James Hughes, Futurewei Techologies, "Using Carry-less Multiplication (CLMUL) to implement erasure code." [Online]. Available: http://www.google.com/patents/US20140317162

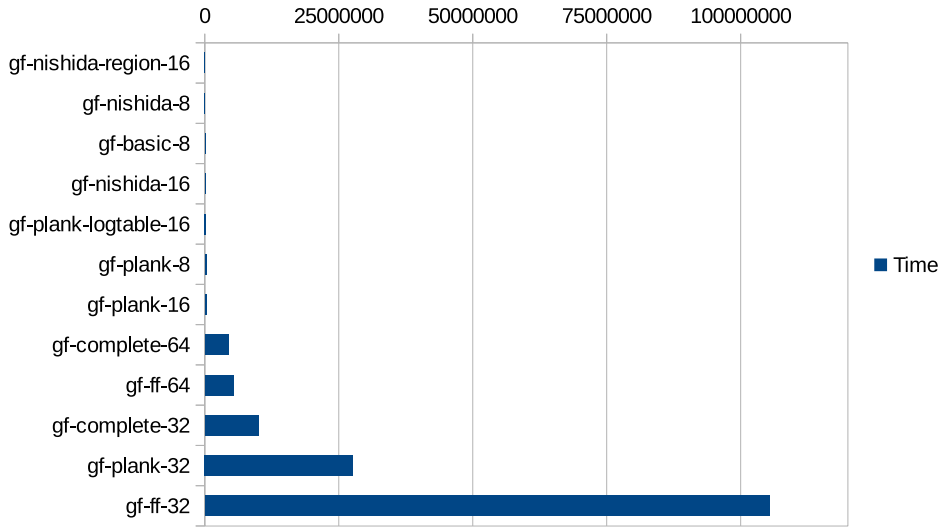
[5] J. S. Plank., "Fast galois field arithmetic library in c/c++." [Online]. Available: http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593/

[6] A. Bellezza, "Binary finite field library." [Online]. Available: http://www.beautylabs.net/software/finitefields.html



Library	Time
gf-nishida-region-16	41583
gf-complete-64	55106
gf-nishida-8	61171
gf-sensor608-8	61195
gf-nishida-16	118850
gf-complete-32	168429
gf-plank-logtable-16	244935
gf-plank-32	314016
gf-plank-16	391792
gf-plank-8	406299
gf-solaris-128	1281013
gf-ff-64	5231520
gf-ff-32	10464125
gf-aes-gcm-128	14013111

図 1　乗算における計測結果（値が小さいほど高速）



Library	Time
gf-nishida-region-16	41557
gf-nishida-8	86159
gf-sensor608-8	114292
gf-nishida-16	118973
gf-plank-logtable-16	251010
gf-plank-8	360553
gf-plank-16	369352
gf-complete-64	4424996
gf-ff-64	5393946
gf-complete-32	10053109
gf-plank-32	27664514
gf-ff-32	105356429

図 2　除算における計測結果（値が小さいほど高速）