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

2022/06/2 改訂

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べ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、3-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 Erasure Coding (EC) や 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、その際にガロア体ににおける加減乗除算が用いられるため、EC や 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 [1] 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。我々は大勢の研究者や開発者達の役に立つことを願い、この gf-nishida-16 を最も制限の少ない 3-Clause BSD ライセンスの下に公開することを決断した。

2. 一般的な $GF(2^N)$ 演算ライブラリ

$GF(2^8)$ (8bit) における高速な乗除算は、事前にメモリー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テーブル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];
}

// 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ーテーブルである。この手法で使用されるメモリー量はわずか 2^{17} (128k) bytes 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) bytes のメモリーが必要となり実用性の範疇を超える。

RNC や 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。

$$\begin{cases} a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 = b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 = b_1 \\ \cdots \end{cases} \quad (1)$$

その際、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、 デコーディングの失敗の確率も大きく減少する。([2] 参照)

Intel の SSE や ARM の NEON などの SIMD 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 [3] は 32bit と 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 [4] 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。これにより我々は計算ベースで SIMD 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、自由に利用することは不可能と認識している。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 EC [5]、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 [6] 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。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は我々の gf-nishida-16 と同様の手法を使ってるが、gf-nishida-16 はプログラミングにおいても高速化のための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ー空間を用意する。

```
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 $2^{16}+2^{12}+2^3+2^1+1 = 0$x1100b のような原始多項式で、$\hat{}$ 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 $1 \leq \mathtt{a} \leq 65535$ を満たす任意の数値 a は

$$\mathtt{a} = 2^{\mathtt{GF16memId}[\mathtt{a}]} = \mathtt{GF16memL}[\mathtt{GF16memIdx}[\mathtt{a}]] \quad (2)$$

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

$$\begin{aligned} \mathtt{a} \times \mathtt{b} &= 2^{\mathtt{GF16memIdx}[\mathtt{a}]} \times 2^{\mathtt{GF16memIdx}[\mathtt{b}]} \\ &= 2^{\mathtt{GF16memIdx}[\mathtt{a}]+\mathtt{GF16memIdx}[\mathtt{b}]} \\ &= \mathtt{GF16memL}[\mathtt{GF16memIdx}[\mathtt{a}] + \mathtt{GF16memIdx}[\mathtt{b}]] \end{aligned} \quad (3)$$

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 = (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

$$\begin{aligned} \mathtt{a}/\mathtt{b} &= 2^{\mathtt{GF16memIdx}[\mathtt{a}]}/2^{\mathtt{GF16memIdx}[\mathtt{b}]} \\ &= 2^{\mathtt{GF16memIdx}[\mathtt{a}]-\mathtt{GF16memIdx}[\mathtt{b}]} \\ &= \mathtt{GF16memL}[\mathtt{GF16memIdx}[\mathtt{a}] - \mathtt{GF16memIdx}[\mathtt{b}]], \end{aligned} \quad (4)$$

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

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

となり、

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

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 GF16div(0, b) も

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

で、GF16memL[131072-196606] のうちの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] のためだけに必要となる。

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

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

および

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

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. 領域計算

EC や RNC など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現する。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]);
}
```

のように係数である定数 a とデータ配列 x の各要素に対しての繰り返し乗算が頻繁に使用される。このような計算を**領域計算**と呼び、これに対して gf-nishida-16 は以下 4 つの高速処理手法を用意している。

1. 二段階テーブル検索
2. 一段階テーブル検索
3. 二つの小さなテーブルを使用した一段階テーブル検索
4. 八つの極小テーブルと SIMD を使用した一段階テーブル検索

4.1. 二段階テーブル検索法

この手法は gf-nishida-16 が提供する領域計算アルゴリズムの中で最もシンプルであるが、それでも GF16mul() や GF16div() より高速である。まず GF16mul() において

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

であり、領域計算においては (GF16memL + GF16memIdx[a]) は一定であるため、それを以下のように置換することができる。

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

そのため上記プログラムを

```
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。ここで gf_a[GF16memIdx[x[i]]] と GF16mul(a, x[i]) を比較すると、GF16mul(a, x[i]) は

```
#define GF16mul(a, x[i]) \
    GF16memL[GF16memIdx[a] + GF16memIdx[x[i]]]
```

と定義されており、gf_a[GF16memIdx[x[i]]] では GF16memIdx[a] とそれを加算する処理が省かれているため、より高速な処理を実現する。（第 5 章 **gf-nishida-region-16-1** 参照）

4.2. 一段階テーブル検索法

上記手法は、GF16memIdx[] でテーブル検索を行った後に、gf_a[] で再度テーブル検索を行うという、二段階のテーブル検索を必要とする。またメモリーも gf_a (GF16memL) に sizeof(uint16_t) * 65536 * 4 bytes、GF16memIdx に sizeof(uint32_t) * 65536 bytes、計 768kB 要する。ここで紹介する手法はその両者を減らすことが可能で、より高速な処理を実現する。

まず以下のように新たなテーブル tbl を設け、値を入力する。

```
uint16_t tbl[65536];

for (n = 0; n < 65536; n++) {
        tbl[n] = gf_a[GF16memIdx[n]];
        // = GF16mul(a, n)
}
```

そしてこれを以下のように用いれば、これまで述べてきた領域乗算と同様の結果を得ることが出来る。

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

この手法では tbl[] の一段階のテーブル検索しか利用せず、またメモリーも tbl の sizeof(uint16_t) * 65536 bytes = 128kB しか使用しない。このため、大半の CPU の L2 キャッシュで動作することが想定され、実際にこれまで述べてきた手法よりも高速に処理することが確認されている。（第 5 章 **gf-nishida-region-16-2** 参照）

4.3. 二つの小さなテーブルを使用した一段階テーブル検索法

上記手法は L2 キャッシュ内で動作する確率が高いものの、多くの CPU において L1 キャッシュ内に収まることは困難だと思われる。ここで紹介する手法は計算量が増加するものの、使用メモリー量を減らし、L1 キャッシュ内で走らせることを前提としたものである。一般的に L1 キャッシュ内で動作させることは高速化に不可欠で、L2 内で動作させる場合よりも大きな高速化が得られることが多い。なお本手法は [7] に紹介されているものを著者の助言とともに採用している。

乗算 $\mathtt{a} \times \mathtt{b}$ において、b は以下のように 2 つの 8bit 符号なし整数 $\mathtt{b_h}$ と $\mathtt{b_l}$ に分けることができる。

$$\begin{aligned} \mathtt{b_h} &= \mathtt{b} >> 8 \\ \mathtt{b_l} &= \mathtt{b} \,\&\, \mathtt{0x00ff} \\ \mathtt{b} &= (\mathtt{b_h} << 8) \,\hat{}\, \mathtt{b_l} \end{aligned} \tag{9}$$

そのため、$\mathtt{a} \times \mathtt{b}$ は

$$\begin{aligned} \mathtt{a} \times \mathtt{b} &= \mathtt{a}\,((\mathtt{b_h} << 8) \,\hat{}\, \mathtt{b_l}) \\ &= \mathtt{a}\,(\mathtt{b_h} << 8) \,\hat{}\, \mathtt{a}\,\mathtt{b_l}. \end{aligned} \tag{10}$$

と表すことができる。ここで以下のように 16bit の要素を 256 個持つ 2 つのテーブル $\mathtt{tbl_h}$ と $\mathtt{tbl_l}$ を用意する。

```
uint16_t tbl_h[256], tbl_l[256];
uint16_t *gf_a = GF16memL + GF16memIdx[a];

for (n = 0; n < 256; n++) {
        tbl_h[n] = gf_a[GF16memIdx[n << 8]];
                // = GF16mul(a, n << 8)
        tbl_l[n] = gf_a[GF16memIdx[n]];
                // = GF16mul(a, n)
}
```

すると

$$\begin{aligned} \mathtt{a} \times \mathtt{b} &= \mathtt{tbl_h}[\mathtt{b_h}]) \,\hat{}\, \mathtt{tbl_l}[\mathtt{b_l}] \\ &= \mathtt{tbl_h}[\mathtt{b} >> 8] \,\hat{}\, \mathtt{tbl_l}[\mathtt{b} \,\&\, \mathtt{0xff}] \end{aligned} \tag{11}$$

となるため、これまで使用してきた領域乗算は

```
uint16_t xi;

for (i = 0; i < NUM; i++) {
        xi = x[i];
        b[i] = tbl_h[xi >> 8] ^
               tbl_l[xi & 0xff];
                    // = GF16mul(a, x[i])
}
```

と置き換えることができる。この手法は前述の通り計算量を増加させるが、$\mathtt{tbl_h}$ と $\mathtt{tbl_l}$ に計 1kB しか使用しないため、多くの CPU の L1 キャッシュ内で走らせることが可能であり、CPU によっては第 4.2 章の手法よりも高速な処理を実現する。（第 5 章 **gf-nishida-region-16-3** 参照）また注目して頂きたいのは、本手法は $GF(2^{32}), GF(2^{64})$ などの高ビットでも使用できることである。実際に $GF(2^{32}), GF(2^{64})$ に実装し試験することを今後の課題としている。

4.4. 八つの極小テーブルと SIMD を使用した一段階テーブル検索

Intel SSE と ARM NEON には 1 byte の要素を 16 個持ったテーブル（計 128bit）の検索を複数同時に行う命令が実装されており、[7] はそれを使った手法についても言及している。これとこれまで述べた一段階検索用テーブルとを組み合わせることにより、gf-nishida-16 は非常に高速な領域計算処理を可能にしている。第 4.3 章では b を 2 つの符号なし整数に分割したが、本手法では同様に以下の方法で b を $\mathtt{b_3}, \mathtt{b_2}, \mathtt{b_1}, \mathtt{b_0}$ の 4 つに分割する。

$$\begin{aligned} \mathtt{b_3} &= \mathtt{b} >> 12 \\ \mathtt{b_2} &= (\mathtt{b} >> 8) \,\&\, \mathtt{0x0f} \\ \mathtt{b_1} &= (\mathtt{b} >> 4) \,\&\, \mathtt{0x0f} \\ \mathtt{b_0} &= \mathtt{b} \,\&\, \mathtt{0x0f} \\ \mathtt{b} &= (\mathtt{b_3} << 12) \,\hat{}\, (\mathtt{b_2} << 8) \,\hat{}\, (\mathtt{b_1} << 8) \,\hat{}\, \mathtt{b_0} \end{aligned} \tag{12}$$

そして以下のように 16 個の要素を持った 8 つのテーブルを作る。

```
uint8_t  tbl_0_l[16], tbl_0_h[16];
uint8_t  tbl_1_l[16], tbl_1_h[16];
uint8_t  tbl_2_l[16], tbl_2_h[16];
uint8_t  tbl_3_l[16], tbl_3_h[16];
uint16_t *gf_a = GF16memL + GF16memIdx[a], tmp;

for (n = 0; n < 16; n++) {
        tmp = gf_a[GF16memIdx[n << 12]];
                // = GF16mul(a, n << 12)
        tbl_3_h[n] = tmp >> 8; // Upper 8bit
        tbl_3_l[n] = tmp & 0xff; // Lower 8bit

        tmp = gf_a[GF16memIdx[n << 8]];
                // = GF16mul(a, n << 8)
        tbl_2_h[n] = tmp >> 8; // Upper 8bit
        tbl_2_l[n] = tmp & 0xff; // Lower 8bit

        tmp = gf_a[GF16memIdx[n << 4]];
                // = GF16mul(a, n << 4)
        tbl_1_h[n] = tmp >> 8; // Upper 8bit
        tbl_1_l[n] = tmp & 0xff; // Lower 8bit

        tmp = gf_a[GF16memIdx[n]];
                // = GF16mul(a, n)
        tbl_0_h[n] = tmp >> 8; // Upper 8bit
        tbl_0_l[n] = tmp & 0xff; // Lower 8bit
}
```

これらを使用すれば、領域乗算を以下のように表すことができる。

```
uint16_t xi;

for (i = 0; i < NUM; i++) {
        xi = x[i];
        b[i] =
            (tbl_3_h[(xi >> 12) & 0xf] << 8) ^
            (tbl_3_l[(xi >> 12) & 0xf]) ^
            (tbl_2_h[(xi >> 8) & 0xf] << 8) ^
            (tbl_2_l[(xi >> 8) & 0xf]) ^
            (tbl_1_h[(xi >> 4) & 0xf] << 8) ^
            (tbl_1_l[(xi >> 4) & 0xf]) ^
            (tbl_0_h[xi & 0xf] << 8) ^
            (tbl_0_l[xi & 0xf]);
                    // = GF16mul(a, x[i])
}
```

ここで Intel SSE の PSHUFB 命令や ARM NEON の TBL 命令などを使用すれば、各々のテーブル検索を大量かつ高速に処理することができ、また NUM 回繰り返すループの回数を大幅に減らすことができる。詳細についてはソースコードの gf.h 及び gf-bench/multiplication/gf-nishida-region-16/gf-bench.c を参照して頂きたい。我々は Intel SSE, AVX と ARM NEON で処理速度を測定し、結果を第 5 章の **gf-nishida-region-16-4*** として示している。なお本方法は計算ではなくテーブル検索に SIMD を使用しているので、我々の知りうるいかなる特許も侵害していないものと確信している。また本手法も前章で述べた手法と同じく、$GF(2^{32})$, $GF(2^{64})$ などの高ビットに適用できる。実際にそれらに実装し試験することを今後の課題としている。

5. 計測結果

我々は第 4 章の先頭で述べた領域乗算プログラムをもとに、様々なオープンソース $GF(2^n)$ 演算ライブラリで乗算と除算における演算速度を計測した。計測には Intel Core i7-10710U と ARM Cortex-A72 の 2 つの CPU を使用し、各測定を 10 回繰り返し平均値を取った。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 のテーブル検索型ライブラリ。
- **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 4.1 から 4.4 章で紹介した gf-nishida-16 の領域計算版。
- **gf-complete**: 第 2 章で紹介した SIMD を使用したライブラリ（ [4] [3] 参照）。前述の通り、特許抵触のおそれのため現在は使用不可。
- **gf-complete-region**: 上記 gf-complete の領域計算版で、乗算のみに対応。
- **gf-sensor608-8**: gf-nishida-8 と同等の手法を使った 8bit ライブラリ。
- **gf-plank**: [6] に基づいたライブラリ。
- **gf-plank-logtable-16**: [6] 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**: [8] からダウンロードされた $GF(2^n)$ 演算ライブラリ。
- **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。

なおライブラリによっては除算の関数を含まないものもある。

計測結果は表 1 から 4 および図 1 から 4 の通りである。全体的に gf-nishida-region-16-4、特にその AVX 版が際立ったパフォーマンスを見せており、gf-complete-region-16 がこれに続いている（乗算のみ）。gf-nishida-region-16-3 は SIMD を使用しないアルゴリズムとしては最速と見られ、gf-complete-region-64 をも凌駕している。gf-nishida-region-16-2 はそのシンプルなアルゴリズムにもかかわらず、Intel の CPU では gf-nishida-region-16-3 と同等の演算速度を示しており、ARM の CPU でも十分な速度を見せている。一方、Intel の $GF(2^n)$ 乗算用の命令セットである CLMUL を使った gf-solaris-128 では、期待したほどの速度が得られていない。また多くのライブラリーで除算の速度が乗算よりも遅くなっているケースが見られるが、gf-nishida-16 では基本的にそのようなものは観測されていない。興味深いのは、Intel の CPU で gf-nishida-16 が gf-nishida-region-16-1 と非常に近い結果を残していることである。これはコンパイラーの最適化によるものであると考えられ、実際に最適化オプション-Ofast を取り除くと、それぞれ異なった結果になる。

6. 今後の課題

第 4.3 と 4.4 章で述べた手法は、前述した通り $GF(2^{32})$ や $GF(2^{64})$ にも使用できる。これらに対して実際に実装し、試験を行うことが課題として残っている。

7. 結論

本稿では gf-nishida-16 について詳解し、gf-nishida-region-16-4 を始めとしたアルゴリズムがいかにガロア体の演算に適しているかを示してきた。その演算速度と特許侵害に対する懸念のなさから、我々は gf-nishida-16 が非常に優れたライブラリーであると確信し、世界中の研究者や開発者の役に立つことを願っている。

8. 参考文献

[1] H. Nishida and T. Nguyen, “Rncdds - random network coded distributed data system,” in *2017 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, 2017, pp. 297–302.

[2] 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

[3] 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

[4] J. S. Plank and et al., “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

[5] I. James Hughes, Futurewei Techologies, “Using Carryless Multiplication (CLMUL) to implement erasure code.” [Online]. Available: http://www.google.com/patents/US20140317162

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

[7] A. Tosho, “Fast galois field region multiplication techniques.” [Online]. Available: https://github.com/animetosho/ParPar/blob/master/fast-gf-multiplication.md

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

	Speed (MB/s)
gf-nishida-region-16-4-AVX	13900.28
gf-nishida-region-16-4-SSE	9197.07
gf-complete-region-16	8339.05
gf-complete-region-32	4913.23
gf-nishida-region-16-2	3847.44
gf-nishida-region-16-3	3727.36
gf-complete-64	2540.20
gf-complete-region-64	2470.43
gf-sensor608-8	2405.05
gf-nishida-8	2386.06
gf-nishida-region-16-1	1398.93
gf-nishida-16	1397.35
gf-complete-32	1094.45
gf-plank-logtable-16	577.85
gf-plank-32	551.07
gf-plank-16	357.86
gf-ff-32	314.00
gf-ff-64	287.13
gf-plank-8	252.39
gf-solaris-128	97.75
gf-aes-gcm-128	15.36

Table 1. Intel Core i7-10710U における乗算速度 (MB/s)（値が大きいほど高速）

	Speed (MB/s)
gf-nishida-region-16-4-AVX	14613.00
gf-nishida-region-16-4-SSE	9454.99
gf-nishida-region-16-2	3754.73
gf-nishida-region-16-3	3625.82
gf-nishida-8	1667.44
gf-nishida-16	1369.20
gf-nishida-region-16-1	1347.87
gf-sensor608-8	935.44
gf-plank-logtable-16	586.56
gf-plank-16	357.81
gf-plank-8	301.56
gf-ff-32	34.86
gf-complete-64	29.50
gf-complete-32	20.85
gf-ff-64	17.68
gf-plank-32	6.25

Table 2. Intel Core i7-10710U における除算速度 (MB/s)（値が大きいほど高速）

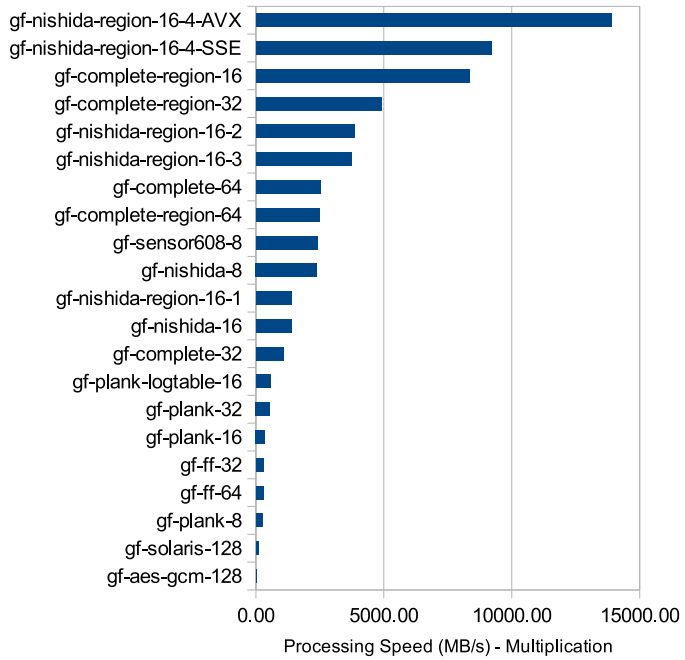

Fig. 1. Intel Core i7-10710U における乗算速度 (MB/s)（値が大きいほど高速）

	Speed (MB/s)
gf-nishida-region-16-4	3209.97
gf-complete-region-16	2939.03
gf-complete-region-32	1708.17
gf-nishida-region-16-3	1075.70
gf-complete-region-64	816.56
gf-nishida-region-16-2	656.68
gf-nishida-region-16-1	330.86
gf-sensor608-8	196.97
gf-nishida-8	196.84
gf-complete-32	69.92
gf-complete-64	67.24
gf-nishida-16	63.01
gf-plank-logtable-16	45.99
gf-plank-32	40.58
gf-ff-32	40.32
gf-ff-64	39.22
gf-plank-8	34.79
gf-plank-16	26.80
gf-aes-gcm-128	2.78

Table 3. ARM Cortex-A72 における乗算速度 (MB/s)（値が大きいほど高速）

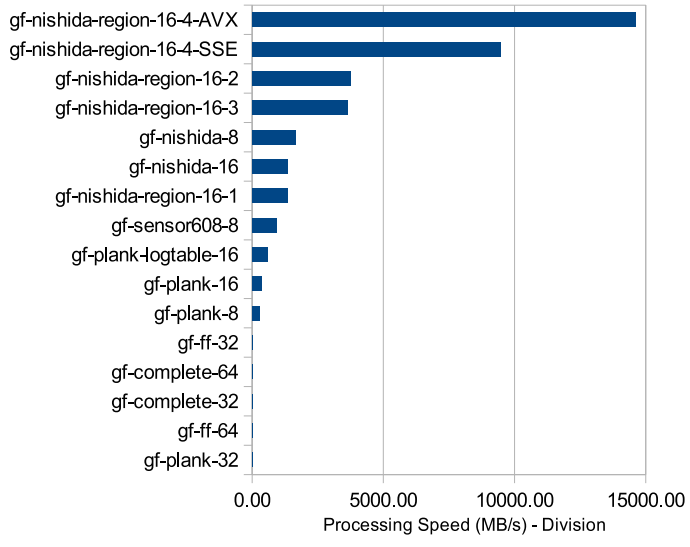

Fig. 2. Intel Core i7-10710U における除算速度 (MB/s)（値が大きいほど高速）

	Speed (MB/s)
gf-nishida-region-16-4	3205.83
gf-nishida-region-16-3	1075.20
gf-nishida-region-16-2	656.66
gf-nishida-region-16-1	330.53
gf-nishida-8	63.26
gf-nishida-16	55.48
gf-sensor608-8	53.98
gf-plank-logtable-16	46.74
gf-plank-16	26.25
gf-plank-8	23.65
gf-complete-32	3.16
gf-ff-64	2.26
gf-ff-32	2.25
gf-complete-64	1.81
gf-plank-32	0.30

Table 4. ARM Cortex-A72 における除算速度 (MB/s)（値が大きいほど高速）

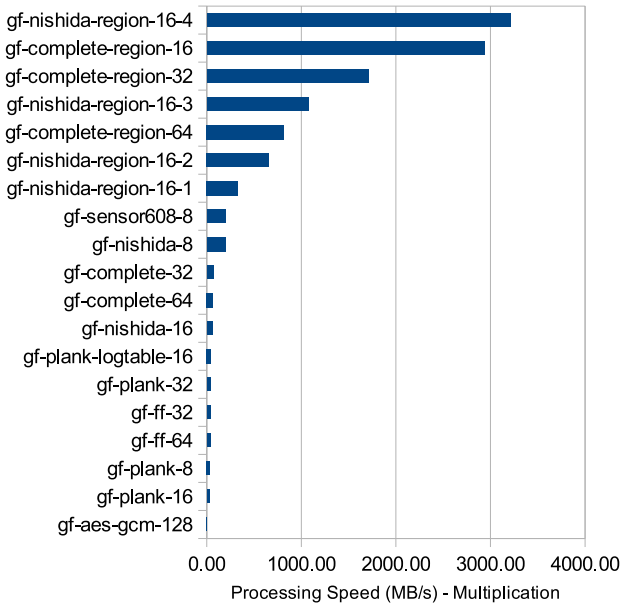

Fig. 3. ARM Cortex-A72 における乗算速度 (MB/s)（値が大きいほど高速）

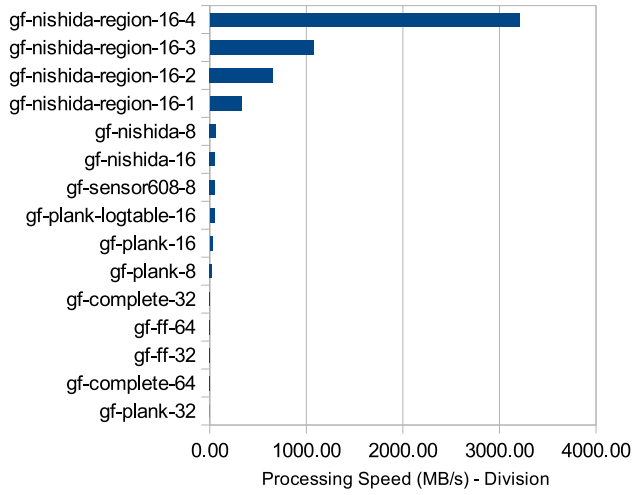

Fig. 4. ARM Cortex-A72 における除算速度 (MB/s)（値が大きいほど高速）