gf-nishida-16: Simple and Efficient $GF(2^{16})$ Arithmetic Library

Hiroshi Nishida, Ph.D.
ASUSA Corporation, Salem Oregon, USA
E-mail: nishida at asusa.net
(Revised on Friday 27th May, 2022)

Abstract—This paper introduces a simple and efficient $GF(2^{16})$ arithmetic library based on table lookup and illustrates its architecture. Our benchmark results show its superior performance over other open source libraries. The source code of gf-nishida-16 is written in C and is released under a 3-Clause BSD license.

Index Terms—Galois Field, Finite Field, Erasure Coding, Random Network Coding

1 Introduction

In some programs, the calculation in *Galois (or Finite) Field* (GF) greatly affects their overall performance. For instance, encoding and decoding in *Erasure Coding* (EC) or *Random Network Coding* (RNC) repeat multiplication or division in GF and therefore influences the processing speed of all programs based on them such as RAID6, signal error correction and some distributed data systems. Our distributed data system *RNCDDS* [1] is not exceptional; our prototype used to be very slow and far from practical because of an inefficient GF library we first employed. However, this encouraged us to develop a more efficient 16 or greater bit GF library, and as a result we have succeeded in creating a 16bit GF arithmetic library *gf-nishida-16* that is simple and efficient enough for practical usage. We have decided to release it under a 3-Clause BSD license which is one of the least restricted licenses in order to help many developers and researchers who use GF arithmetic libraries. We also believe that it is patent free and that the use of it is not legally restricted. In this paper, we explain the architecture of gf-nishida-16 and its usage, and show its benchmark results by comparing it to other open source GF arithmetic libraries.

2 General $GF(2^n)$ Libraries

Fast $GF(2^8)$ multiplication and division can be easily achieved by simple table lookup as follows:

```
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];
},
```

where the entries in `GF8mulTbl` and `GF8divTbl` are calculated beforehand. This simple technique provides fast response by returning an entry in `GF8mulTbl` or `GF8divTbl` without real calculation in $GF(2^8)$. It also requires only 2^{17} (128k) byte memory for the lookup tables and is very efficient as long as the primitive polynomial is static. However, the same technique is not easily applicable to $GF(2^{16})$ as its lookup tables consume 2^{34} (16G) byte memory. The reason why 8bit GF is not sufficient and a 16 or greater bit GF library is necessary in RNC or many general cases is simple. RNC encodes data by creating linear equations as follows and decodes by solving a system of those linear equations.

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

Since $GF(2^8)$ space is too small to maintain the independency of all those linear equations, the probability of failure in solving a system of linear equations in $GF(2^8)$ is considerably high, while that of $GF(2^{16})$ is much lower, not to mention $GF(2^{n>16})$ [2]. Hence, a GF library with 16 or greater bits is necessary in RNC, and an efficient arithmetic library in $GF(2^{16})$ with a lax license was required in our development of a RNC based distributed system.

SIMD (SSE/NEON) provides probably the most ideal performance and parallel processing environment for GF calculation, especially in high bits such as 64bits. In fact, a 64bit multiplication function by *GF-Complete*

gf-nishida-16 was originally developed as part of Random Network Coding Project in ASUSA Corporation.
Many thanks to animetosho https://github.com/animetosho for introducing techniques to speed up region calculation.

[3] created by Plank, et al. records excellent benchmark results in our experiment (see Section 5). However unfortunately, its source code has been removed by the author due to potential patent infringement [4]. It is regrettable but encouraged us to create an alternative library that would not infringe any patents.

Intel added *Carry-less Multiplication* (CLMUL) instruction set specialized for $GF(2^n)$ multiplication to their CPUs in 2010, but our benchmark results do not show sufficient performance (see gf-solaris-128 in Section 5). Note there is also a patent on processing Erasure Coding by CLMUL [5].

There is another GF library by Plank [6] which covers 8 to 32bit GF arithmetic calculation, and we noticed that the technique used by its logarithmic table based functions was very similar to that of gf-nishida-16. The biggest difference seems to be the optimization on the two step table lookup; gf-nishida-16 seems to be more optimized and to access the tables faster. Our new techniques for region calculation (see Section 4) are also unique to our library and are not seen in any other libraries. The benchmark results on Plank's library are shown as gf-plank and gf-plank-logtable in Section 5.

3 Architecture of gf-nishida-16

We first prepare two memory spaces as follows:

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

then input 2^i to `GF16memL[i]` for $0 \leq i \leq 65534$ in a $GF(2^{16})$ manner. Herein, a $GF(2^{16})$ manner means: if 2^i is equal to or greater than 65536, then we input $2^i \hat{} P$ instead, where P is a polynomial primitive such as $2^{16} + 2^{12} + 2^3 + 2^1 + 1 = 0$x1100b and $\hat{}$ is XOR. Consequently, we have the following code for initializing the first quarter space of `GF16memL`:

```
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);
}
```

In this loop, we also input values to `GF16memIdx` as follows:

```
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;
}
```

so that we have

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

for $1 \leq \mathtt{a} \leq 65535$. As a result, multiplication $\mathtt{a} \times \mathtt{b}$ in $GF(2^{16})$ can be expressed as:

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

Therefore, our multiplication function `GF16mul(a, b)` can be defined as:

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

However, we need to be careful of the value of `GF16memIdx[a] + GF16memIdx[b]` as it may exceed 65534 and may cause segmentation violation for `GF16memL`. To deal with it, many programs use its residual divided by 65534 as follows:

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

which adds an extra code to the function and slows down the program. To avoid this, we utilize the second quarter space of `GF16memL`, aka `GF16memH`, by copying the content of `GF16memL` to it:

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

Thus, even if `GF16memIdx[a] + GF16memIdx[b] > 65534`, we are able to retrieve the correct value from one of `GF16memH[0-65534]`.

`GF16memH` also simplifies $GF(2^{16})$ division. Since

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

we can define a $GF(2^{16})$ division function as:

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

However, `GF16memIdx[a] - GF16memIdx[b] < 0` is possible and can cause segmentation violation. To avoid this, we use `GF16memH` instead of `GF16memL` as `GF16memH[[GF16memIdx[a] - GF16memIdx[b]]` always drops into one of the numbers in `GF16memL[0-65534]` or in `GF16memH[0-65534]`. Therefore, we can safely define `GF16div(a, b)` as:

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

Thus far, we haven't referred to the case of `a = 0 || b = 0`. `GF16mul(0, b)`, `GF16mul(a, 0)` and `GF16div(0, b)` for `a, b != 0` should return 0, while `GF16div(a, 0)` is undefined as well as general division in $a/0$. To satisfy those conditions, we fill the rest of `GF16memL` (i.e., `GF16memL[13170-262143]`) with 0 and set `GF16memIdx[0]` as follows:

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

As

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

and

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

`GF16mul(0, b)` points to one of the numbers in `GF16memL[131071-196605]`, which is always 0. Similarly,

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

outputs one of the numbers in `GF16memL[131072-196606]`, which is also 0. Thus, our multiplication and division functions fulfill the above conditions for the calculation using 0. Note the fourth quarter space of `GF16memL` (`GF16memL[196607-262143]`) is necessary only for `GF16mul(0, 0) = GF16memL[262142]`.

As for division by 0, `GF16div(a, 0)` for `a != 0` violates memory segmentation because

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

and

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

which should be prohibited by the user.

4 Region Calculation

Most GF calculation in RNC or EC is done with a combination of a static coefficient and an array of data (a region of data). That is, each element of data is multiplied or divided by a constant number as follows:

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

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

where a is a constant coefficient and x is a region of data. Our library provides the following four techniques to boost up such calculation:

1) Two step table lookup
2) One step table lookup
3) One step table lookup with two small tables
4) One step table lookup with eight tiny tables using SIMD

4.1 Two Step Table Lookup Approach

This is the simplest region calculation technique but is still faster than `GF16mul()` or `GF16div()`. Since we have

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

and `(GF16memL + GF16memIdx[a])` is static, we can replace `(GF16memL + GF16memIdx[a])` with another pointer such as:

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

Thus, the above code can be rewritten as:

```
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]]];
},
```

which removes one table lookup `GF16memIdx[a]` and one addition from (8). The benchmark results are shown as **gf-nishida-region-16-1** in Section 5.

4.2 One Step Table Lookup Approach

The approach in the last subsection requires two table lookups, that is,

```
idx = GF16memIdx[x[i]];
b[i] = gf_a[idx];
```

and consumes `sizeof(uint16_t) * 65536 * 4` bytes for `gf_a` (`GF16memL`) and `sizeof(uint32_t) * 65536` bytes for `GF16memIdx` tables, i.e., 768kB in total. However, both the number of lookup steps and memory consumption can be reduced by the following technique:

1) Create a table `tbl` such that:

```
uint16_t tbl[65536];

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

2) Then use it for multiplication of `a` and `x[i]` as follows:

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

This requires only one table lookup and `sizeof(uint16_t) * 65536` bytes = 128kB memory, which will fit L2 cache. The benchmark results are shown as **gf-nishida-region-16-2** in Section 5.

4.3 One Step Table Lookup With Two Small Tables Approach

Running a program in L1 cache is generally a great solution to maximizing the performance. However, as the aforementioned technique still allocates 128kB to the lookup table, it will not run in most L1 cache. To shrink

the size of the lookup table, we employ a split lookup table technique introduced in [7].

In multiplication $\mathtt{a} \times \mathtt{b}$, $\mathtt{b}$ can be split into two 8bit unsigned integers $\mathtt{b_h}$ and $\mathtt{b_l}$ s.t.

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

and we have

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

If we create two lookup tables $\mathtt{tbl_h}$ and $\mathtt{tbl_l}$ with 256 entries each as follows:

```
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)
}
```

then

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

holds and the region multiplication can be expressed as:

```
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])
}
```

Though this increases the amount of calculation, it runs in most L1 cache as each of $\mathtt{tbl_h}$ and $\mathtt{tbl_l}$ uses only 1kB in total and performs well with some CPUs. The benchmark results are shown as **gf-nishida-region-16-3** in Section 5. Note this approach can be applied also to $GF(2^{32})$, $GF(2^{64})$, $GF(2^{128})$, ... and we are planning to implement with those higher bits.

4.4 One Step Table Lookup With Eight Tiny Tables Using SIMD Approach

Intel SSE and ARM NEON have instructions to execute multiple table lookups for a table with sixteen 1 byte (128bits in total) entries, and [7] also explains a lookup method that utilizes them. Gf-nishida-16 uses it for the similar lookup tables to those in Section 4.2 and 4.3 and is capable of returning a $GF(2^{16})$ region multiplication/division result instantly. In Section 4.3, we split $\mathtt{b}$ into two 8bit unsigned integers $\mathtt{b_h}$ and $\mathtt{b_l}$. Similarly, we herein split $\mathtt{b}$ into four 4bit unsigned integers $\mathtt{b_3}$, $\mathtt{b_2}$, $\mathtt{b_1}$ and $\mathtt{b_0}$ as follows:

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

Then, we create the following eight lookup tables composed of 16 entries each:

```
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
}.
```

and get the region multiplication result by:

```
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])
}.
```

Intel SSE's PSHUFB and ARM NEON's TBL instructions execute multiple table lookups for a table at once and efficiently achieve the above computation with fewer loops. Please see gf.h and gf-bench/multiplication/gf-nishida-region-16/gf-bench.c for a code example. The benchmark results are shown as **gf-nishida-region-16-4** in Section 5. Note we believe this technique does not infringe any patents we've known as table lookup with SIMD is very common and widely used. Also as well as the technique described in Section 4.3, this approach is applicable to $GF(2^{32})$, $GF(2^{64})$, $GF(2^{128})$, ... and implementing with those higher bits remains as our future work.

5 Benchmark Results

We measured processing speeds (MB/s) for simple region multiplication and division based on the program described in the top of Section 4 with the following open source libraries. Each benchmark was repeated ten times and was run with Intel Core i7-10710U and ARM Cortex-A72 CPUs.

- **gf-nishida-8**: A $GF(2^8)$ table lookup based library described in Section 2.
- **gf-nishida-16**: Our main library (see Section 3).
- **gf-nishida-region-16-***: Our region calculation versions explained in Section 4.1 to 4.4.
- **gf-complete**: A library that uses SIMD and is regarded as potentially patent infringeable as mentioned in Section 2 (see [4] [3]).
- **gf-complete-region**: A region calculation version of gf-complete.
- **gf-sensor608-8**: A $GF(2^8)$ library similar to gf-nishida-8.
- **gf-plank**: A library based on [6].
- **gf-plank-logtable-16**: A $GF(2^{16})$ library similar to gf-nishida-16 which looks up two tables (see Section 2 and [6]).
- **gf-solaris-128**: A $GF(2^{128})$ program retrieved from Solaris source code that uses CLMUL (see Section 2).
- **gf-ff**: A library downloaded from [8] which provides arithmetic functions in $GF(2^n)$.
- **gf-aes-gcm-128**: A $GF(2^{128})$ program retrieved from FreeBSD source code that uses no special techniques.

Note some libraries do not provide a function for division.

The benchmark results are shown in Table 1 to 4 and Fig. 1 to 4. Overall, gf-nishida-region-16-4 exhibits outstanding performance, followed by gf-complete-region-32 (multiplication only). Gf-nishida-region-16-3 seems to be the fastest non-SIMD algorithm and even surpasses gf-complete-region-64. Despite its simpleness, gf-nishida-region-16-2 performs as well as gf-nishida-region-16-3 with an Intel CPU and shows adequate results also with an ARM CPU. On the other hand, no sufficient performance is seen with gf-solaris-128 which uses Intel CLMUL instruction set for $GF(2^n)$ multiplication. In many libraries, division is slower than multiplication but it is not the case with gf-nishida-16. Interestingly, the results of gf-nishida-region-16-1 are very similar to those of gf-nishida-16 with an Intel CPU, which we think due to the compiler's optimization. In fact, removing the optimization option (-Ofast) differentiates the results.

6 Future Work

The techniques described in Section 4.3 and 4.4 can also be used for $GF(2^{32})$ and $GF(2^{64})$. We will test with those higher bits in the near future.

	Speed (MB/s)
gf-nishida-region-16-4	8856.94
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: Multiplication speed (MB/s) with Intel Core i7-10710U (the greater, the faster)

7 Conclusion

In this paper, we showed the efficiency of gf-nishida-16, especially the high capability of gf-nishida-region-16-4 for region calculation. Considering its performance and the risklessness of patent infringement, we believe that gf-nishida-16 is the most suitable library for many coding algorithms that require GF calculation and will help many developers and researchers in the world.

References

[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 Carry-less 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	8560.24
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: Division speed (MB/s) with Intel Core i7-10710U (the greater, the faster)

	Speed (MB/s)
gf-nishida-region-16-4	3209.97
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: Multiplication speed (MB/s) with ARM Cortex-A72 (the greater, the faster)

	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: Division speed (MB/s) with ARM Cortex-A72 (the greater, the faster)

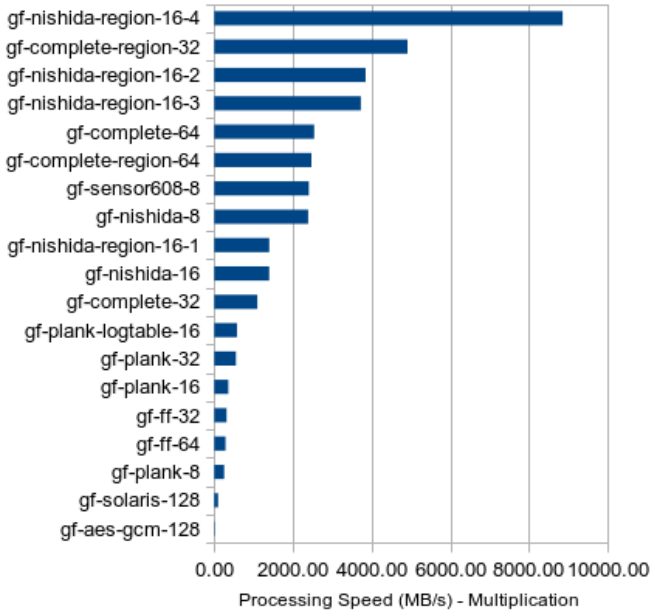

Fig. 1: Multiplication speed (MB/s) with Intel Core i7-10710U (the greater, the faster)

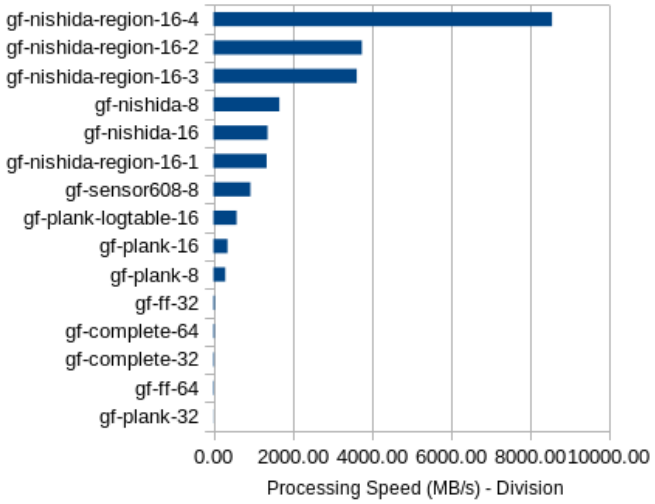

Fig. 2: Division speed (MB/s) with Intel Core i7-10710U (the greater, the faster)

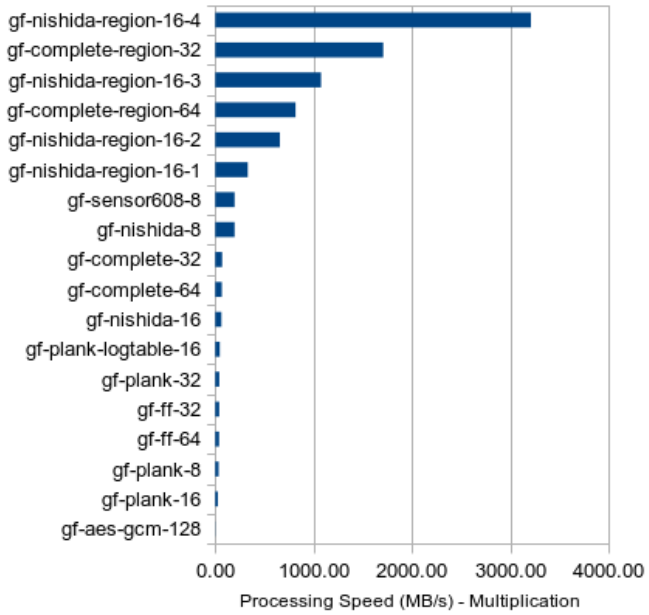

Fig. 3: Multiplication speed (MB/s) with ARM Cortex-A72 (the greater, the faster)

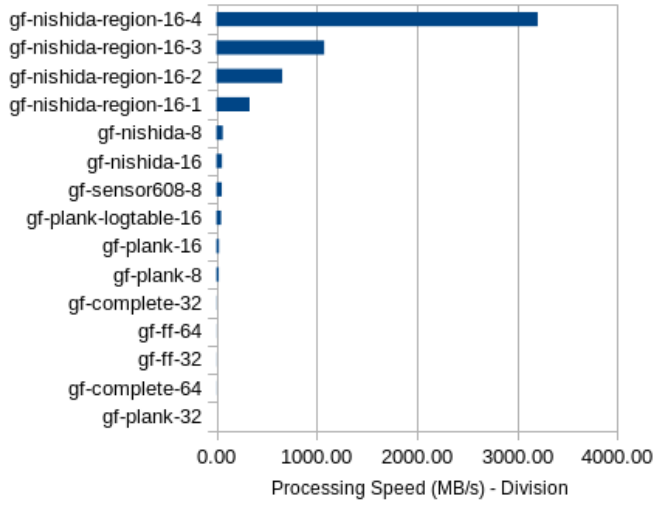

Fig. 4: Division speed (MB/s) with ARM Cortex-A72 (the greater, the faster)