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

Hiroshi Nishida, Ph.D.
ASUSA Corporation, Salem Oregon, USA
E-mail: nishida at asusa.net

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

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

I. 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* that uses RNC 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 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 2-Clause BSD license which is one of the least restricted licenses so that it helps many developers and researchers. 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 some other open source GF arithmetic libraries.

II. General $GF(2^n)$ Libraries

Fast 8bit GF multiplication and division can be easily achieved by simple memory 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];
},
```

gf-nishida-16 was originally developed as part of Random Network Coding Project in ASUSA Corporation

where we suppose the entries in `GF8mulTbl` and `GF8divTbl` are calculated beforehand. This simple technique provides fast processing by returning an entry in `GF8mulTbl` or `GF8divTbl` as a result of multiplication or division in GF. It also requires only 2^{17} (128k) byte memory for the lookup tables and will be safe to say very efficient as long as the primitive polynomial is static. However, the same technique cannot be easily applied to $GF(2^{16})$ as the tables consumes 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 follow and decodes by solving a system of those linear equations.

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

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 $GF(2^{n>16})$ [1]. 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.

SSE 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* [2] created by Plank, et al. records superb benchmark results in our experiment (see Section V). However unfortunately, its source code has been removed by its author due to potential patent infringement [3]. 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 V). Note there is also a patent on processing Erasure Coding by CLMUL [4].

There is another GF library by Plank [5] which covers 8 to 32bit GF arithmetic calculation. While our investigation, we noticed that the technique used by its logarithmic table based functions was very similar to that of gf-nishida-16. The only difference between them seems to be the optimization on the two step memory lookup; gf-nishida-16 seems to be more optimized and to access the memory tables faster. The benchmark results on Plank's library are shown as gf-plank

and gf-plank-logtable in Section V.

III. 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 $2^{16} + 2^{12} + 2^3 + 2^1 + 1 =$ 0x1100b 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

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

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

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

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]` because it may exceed 65534. To deal with it, many programs check/change it as follows:

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

which adds extra code to the function and slows down the program. In order to avoid it, 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 a correct value from one of `GF16memH[0-65534]`.

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

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

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 causes segmentation violation in this case. This is easily resolvable by replacing `GF16memL` with `GF16memH` because `GF16memH[[GF16memIdx[a] - GF16memIdx[b]]` always becomes one of the number in `GF16memL[0-65534]` or `GF16memH[0-65534]`. Therefore, we can 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. To satisfy the condition, 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

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

and

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

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

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

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

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

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

and

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

Programmers need to be aware of it.

IV. Faster Computation

Because of its simple architecture, gf-nishida-16 can be even faster at multiplying and dividing a region of data by a fixed number. For instance, the encoding in RNC repeats multiplication as follows:

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

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

Since we have

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

we can replace `(GF16memL + GF16memIdx[a])` with a static value 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[x[i]];
}
```

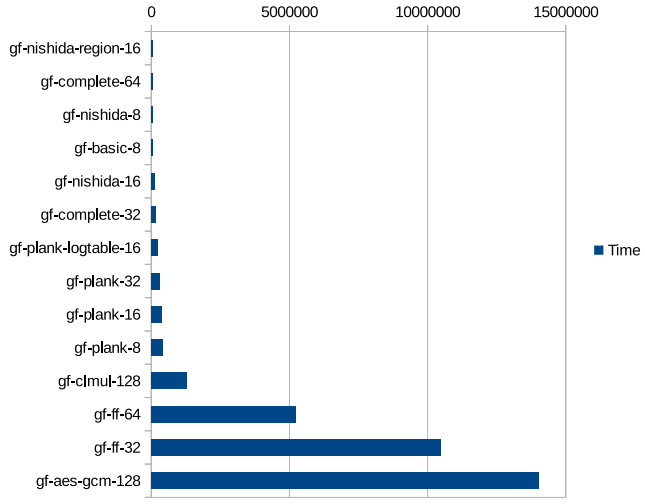
which speeds up more than double compared to the original (see gf-nishida-region-16 in Section V).

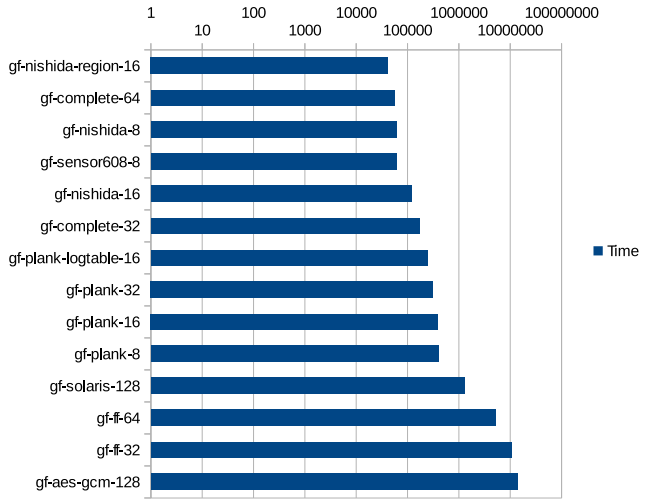
V. Benchmark Results

We measured elapsed times for multiplication and division based on the program described in Section IV with the following open source GF libraries:

- **gf-nishida-8**: An 8bit memory lookup type library described in Section II.
- **gf-nishida-16**: Our main library (see Section III).
- **gf-nishida-region-16**: A region computation version of gf-nishida-16 explained in Section IV.
- **gf-complete**: A library that uses SSE and has been removed by the author as mentioned in Section II (see [3] [2]).
- **gf-sensor608-8**: An 8bit library similar to gf-nishida-8.
- **gf-plank**: A library based on [5].
- **gf-plank-logtable-16**: A 16bit library similar to gf-nishida-16 which looks up two memory tables (see Section II and [5]).
- **gf-solaris-128**: A 128bit GF program retrieved from Solaris source code that uses CLMUL (see Section II).
- **gf-ff**: A library downloaded from [6] which provides arithmetic functions in $GF(2^n)$. Although we did not understand if it was applicable to 32 and 64bits, we benchmarked it based on the programs generated at their website.
- **gf-aes-gcm-128**: A 128bit GF program retrieved from FreeBSD source code that uses no special techniques.

Note that some libraries do not include a function for division and that some have a region computation function like gf-nishida-region-16, which we did not benchmark.





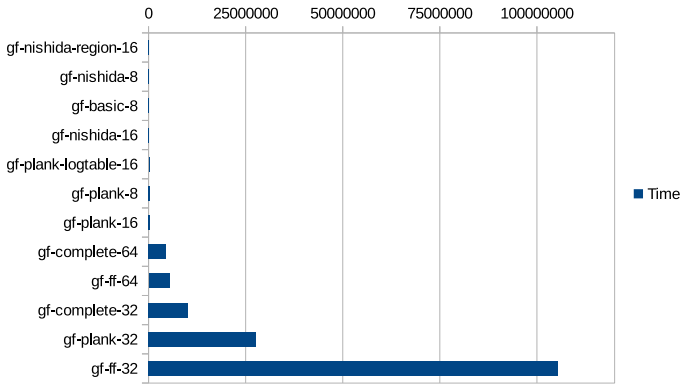
Library	Time
gf-nishida-region-16	41583
gf-complete-64	55106
gf-nishida-8	61171
gf-sensor608-8	61195
gf-nishida-16	118850
gf-complete-32	168429
gf-plank-logtable-16	244935
gf-plank-32	314016
gf-plank-16	391792
gf-plank-8	406299
gf-solaris-128	1281013
gf-ff-64	5231520
gf-ff-32	10464125
gf-aes-gcm-128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

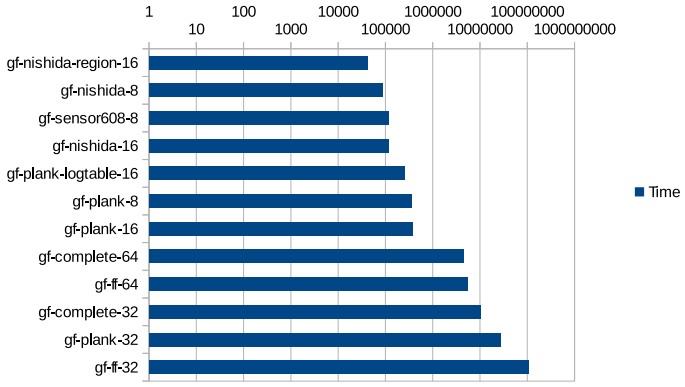
Fig. 1. Benchmark results in multiplication (standard and logarithm scales, the shorter, the faster)

The benchmark results are shown in Fig. 1 and 2 (the shorter, the faster). As a consequence, gf-nishida-16, especially gf-nishida-region-16 shows excellent performance in both multiplication and division. Considering its performance and the risklessness of patent infringement, we believe gf-nishida-16 is the best library for the GF computation in RNC.

References

[1] K. Nguyen, T. Nguyen, Y. Kovchegov, and V. Le, "Distributed data replenishment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2,





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

Fig. 2. Benchmark results in division (standard and logarithm scales, the shorter, the faster)

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