

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

Hiroshi Nishida

Abstract—This paper introduces a simple and fast $GF(2^{16})$ library that uses two step memory lookups, and illustrates its mechanism. Our benchmark shows its superior performance over other available open source $GF(2^{n \geq 16})$ libraries. The source code of gf-nishida-16 is released under a BSD license.

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

I. INTRODUCTION

In some programs, the calculation in Galois (or Finite) Field greatly affects their overall performance. For example, encoding and decoding in Random Network Coding (RNC) repeatedly multiplies and divides given data in GF , and therefore the calculation in a GF determines the processing speeds of all programs based on it. Our distributed data system that uses RNC was not exceptional; our prototype was very slow and far from practical because of a slow GF library. This urged us to develop a fast 16 or greater bit GF library, and as a result a sufficiently fast 16bit library gf-nishida-16 was born. We have decided to release it under a BSD license which is one of the least restricted licenses so that it helps many developers write their own code.

II. GENERAL $GF(2^n)$ LIBRARIES

Fast 8bit GF multiplication and division can be easily achieved by simple memory lookups in the following way:

```
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];
}
```

assuming entries in GF8mulTbl and GF8divTbl are calculated beforehand. This requires only 2^{17} (128k) byte memory for the tables and is 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 in $GF(2^{16})$. The reason why we needed a 16 or greater bit GF library is simple. RNC encodes data by

creating linear equations like below and decodes by solving a system of those linear equations.

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

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. Hence, we sought for a GF library with 16 or greater bits for our distributed data system. Note that randomly created $GF(2^{16})$ linear equations still produce duplications at a low probability. In our experiment, we had approximately four duplications out of randomly created 200,000 patterns of three linear equations. Although we regard it as low enough, our distributed data system uses by way of caution the combinations of coefficients that do not generate any duplications. The same approach is applicable to $GF(2^8)$ but there is still a much higher chance of duplication in regenerating a new linear equation from existing linear equations than $GF(2^{16})$ [1].

SSE provides probably the most ideal performance and parallel processing environment for GF calculation especially in high bits like 64. In fact, a 64bit multiplication function from GF-Complete created by Plank, et al. [2], which utilizes SSE, records superb benchmark results in our experiment. However unfortunately, their program has been removed due to potential patent violation [3]. We regret it and at the same time decided to create an alternative library that does not use SSE.

Intel added Carry-less Multiplication (CLMUL) instruction set specialized in $GF(2^n)$ multiplication to their CPUs in 2010. There is also a patent on processing erasure coding by CLMUL [4]. However our benchmark does not show sufficient performance (gf-solaris-128).

There is another GF library by Plank [5] which covers 8 to 32bits. I have just noticed that its log table based functions use the same approach as gf-nishida-16, but the biggest difference is that gf-nishida-16 is more tweaked as a computer program and minimizes the code. The benchmark results on Plank's library appear as gf-plank and gf-plank-logtable.

III. MECHANISM 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 $2^i \wedge P$ instead,

where P is a polynomial primitive like $2^{16} + 2^{12} + 2^3 + 2^1 + 1 = 0x1100b$ and \wedge 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

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

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

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

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. 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

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

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 hits one of 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 $a = 0 \parallel b = 0$. GF16mul(0, b), GF16mul($a, 0$) and GF16div(0, b) for $a, b \neq 0$ should return 0, while GF16div($a, 0$) is undefined. To satisfy it, 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

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

and

$$0 \leq GF16memIdx[b] \leq 65534,$$

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

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

outputs one of the numbers between GF16memL[131072–196606], which is also 0. Thus, our multiplication and division functions fulfill the above condition for 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 $a \neq 0$ causes segmentation violation because

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

and

$$-65534 \leq GF16memIdx[a] - 65534 \leq 0.$$

Programmers need to be aware of it.

IV. FASTER COMPUTATION

Because of its simple mechanism, gf-nishida-16 can be faster at multiplying a region of data by a single number. For instance, RNC encoding repeats multiplication like:

```
uint16_t a, x[NUM], b[NUM];
for (i = 0; i < NUM; i++) {
    b[i] = GF16mul(a, x[i]);
}
```

Since

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

we can replace $(\text{GF16memL} + \text{GF16memIdx}[a])$ with a static value like:

```
|| 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.

V. BENCHMARK RESULTS

We measured elapsed times for multiplication and division with various open source *GF* libraries based on the program described in Section IV. The libraries used are:

- gf-nishida-8 is a 8bit memory lookup type library described in Section II. gf-nishida-region-16 is a region computation version of gf-nishida-16 as explained in Section IV.
- gf-complete [3] [2], as mentioned in Section II, is a library which uses SSE.
- gf-sensor608-8 is a 8bit library similar to gf-nishida-8.
- gf-plank is based on [5].
- gf-plank-logtable-16 [5] is a 16bit library similar to gf-nishida-16 which looks up two memory tables. See Section II.
- gf-solaris-128 is a 128bit library retrieved from Solaris source code and uses CLMUL. See Section II.
- gf-ff downloaded from [6] 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 retrieved from FreeBSD source code is a basic $GF(2^{128})$ calculation program that does not use any 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 this time.

The benchmark results are shown in Fig. 1 and Fig. 2. In the graphs, the shorter, the faster. As a result, gf-nishida-16, especially gf-nishida-region-16, shows excellent performance. We believe gf-nishida-16 is one of the best solutions for *GF* computation in Random Network Coding.

REFERENCES

- [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

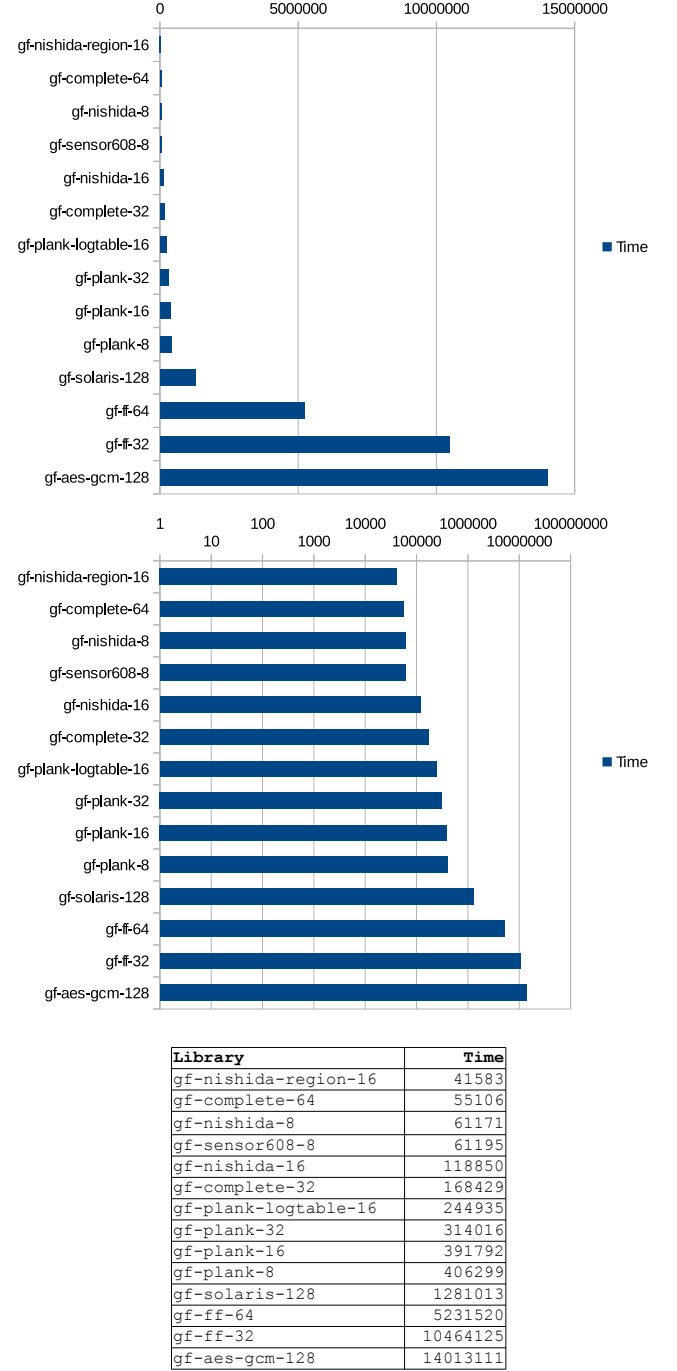


Fig. 1. Benchmark results in multiplication (standard and logarithm scales)

- [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 Technologies, "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>

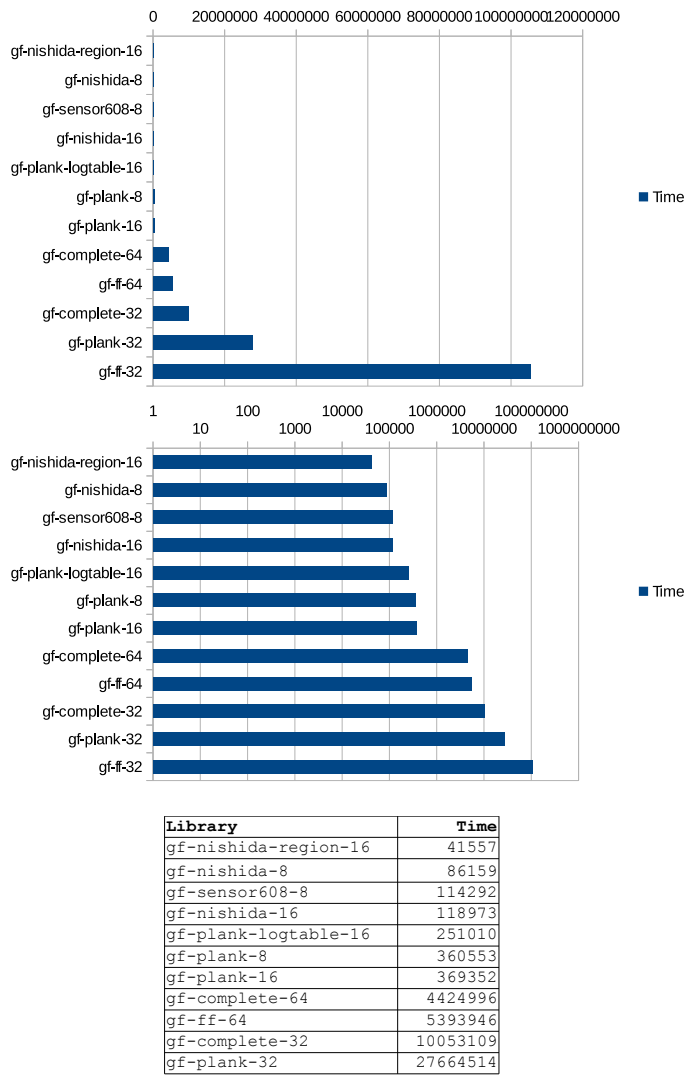


Fig. 2. Benchmark results in division (standard and logarithm scales)