GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic

Version 1.02

James S. Plank* Ethan L. Miller Kevin M. Greenan Benjamin A. Arnold
John A. Burnum Adam W. Disney Allen C. McBride

January 29, 2014

`https://bitbucket.org/jimplank/gf-complete`

`http://web.eecs.utk.edu/~plank/plank/papers/GF-Complete-Manual-1.02.pdf`

This is a user's manual for GF-Complete, version 1.02. This release supersedes version 0.1 and represents the first major release of GF-Complete. To our knowledge, this library implements every Galois Field multiplication technique applicable to erasure coding for storage, which is why we named it GF-Complete. The primary goal of this library is to allow storage system researchers and implementors to utilize very fast Galois Field arithmetic for Reed-Solomon coding and the like in their storage installations. The secondary goal is to allow those who want to explore different ways to perform Galois Field arithmetic to be able to do so effectively.

If you wish to cite GF-Complete, please cite technical report UT-CS-13-716: [PMG+13].

If You Use This Library or Document

Please send me an email to let me know how it goes. Or send me an email just to let me know you are using the library. One of the ways in which we are evaluated both internally and externally is by the impact of our work, and if you have found this library and/or this document useful, we would like to be able to document it. Please send mail to *plank@cs.utk.edu*. Please send bug reports to that address as well.

The library itself is protected by the New BSD License. It is free to use and modify within the bounds of this license. To the authors' knowledge, none of the techniques implemented in this library have been patented, and the authors are not pursing patents.

*`plank@cs.utk.edu` (University of Tennessee), `elm@cs.ucsc.edu` (UC Santa Cruz), `kmgreen2@gmail.com` (Box). This material is based upon work supported by the National Science Foundation under grants CNS-0917396, IIP-0934401 and CSR-1016636, plus REU supplements CNS-1034216, CSR-1128847 and CSR-1246277. Thanks to Jens Gregor for helping us wade through compilation issues, and for Will Houston for his initial work on this library.

Finding the Code

This code is actively maintained on bitbucket: `https://bitbucket.org/jimplank/gf-complete`. There are previous versions on my UTK site as a technical report; however, that it too hard to maintain, so the main version is on bitbucket.

Two Related Papers

This software acccompanies a large paper that describes these implementation techniques in detail [PGM13a]. We will refer to this as "*The Paper.*" You do not have to read The Paper to use the software. However, if you want to start exploring the various implementations, then The Paper is where you'll want to go to learn about the techniques in detail.

This library implements the techniques described in the paper "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions," [PGM13b]. The Paper describes all of those techniques as well.

If You Would Like Help With the Software

Please contact the first author of this manual.

Changes from Revision 1.01

The major change is that we are using autoconf to aid with compilation, thus obviating the need for the old **flag_tester** code. Additionally, we have added a quick timing tool, and we have modified **gf_methods** so that it may be used to run the timing tool and the unit tester.

Contents

1 Introduction

Galois Field arithmetic forms the backbone of erasure-coded storage systems, most famously the Reed-Solomon erasure code. A Galois Field is defined over w-bit words and is termed $GF(2^w)$. As such, the elements of a Galois Field are the integers 0, 1, . . ., $2^w - 1$. Galois Field arithmetic defines addition and multiplication over these closed sets of integers in such a way that they work as you would hope they would work. Specifically, every number has a unique multiplicative inverse. Moreover, there is a value, typically the value 2, which has the property that you can enumerate all of the non-zero elements of the field by taking that value to successively higher powers.

Addition in a Galois Field is equal to the bitwise exclusive-or operation. That's nice and convenient. Multiplication is a little more complex, and there are many, many ways to implement it. The Paper describes them all, and the following references provide more supporting material: [Anv09, GMS08, LHy08, LD00, LBOX12, Pla97]. The intent of this library is to implement *all* of the techniques. That way, their performance may be compared, and their tradeoffs may be analyzed.

When used for erasure codes, there are typically five important operations:

1. **Adding two numbers in** $GF(2^w)$. That's bitwise exclusive-or.
2. **Multiplying two numbers in** $GF(2^w)$. Erasure codes are usually based on matrices in $GF(2^w)$, and constructing these matrices requires both addition and multiplication.
3. **Dividing two numbers in** $GF(2^w)$. Sometimes you need to divide to construct matrices (for example, Cauchy Reed-Solomon codes [BKK$^+$95, Rab89]). More often, though, you use division to invert matrices for decoding. Sometimes it is easier to find a number's inverse than it is to divide. In that case, you can divide by multiplying by an inverse.
4. **Adding two regions of numbers in** $GF(2^w)$, which will be explained along with...
5. **Mutiplying a region of numbers in** $GF(2^w)$ **by a constant in** $GF(2^w)$. Erasure coding typically boils down to performing dot products in $GF(2^w)$. For example, you may define a coding disk using the equation:

   $$c_0 = d_0 + 2d_1 + 4d_2 + 8d_3.$$

   That looks like three multiplications and three additions However, the way that's implemented in a disk system looks as in Figure 1. Large regions of disks are partitioned into w-bit words in $GF(2^w)$. In the example, let us suppose that $w = 8$, and therefore that words are bytes. Then the regions pictured are 1 KB from each disk. The bytes on disk D_i are labeled $d_{i,0}, d_{i,1}, \ldots, d_{i,1023}$, and the equation above is replicated 1024 times. For $0 \leq j < 1024$:

   $$c_{0,j} = d_{0,j} + 2d_{1,j} + 4d_{2,j} + 8d_{3,j}.$$

   While it's possible to implement each of these 1024 equations independently, using the single multiplication and addition operations above, it is often much more efficient to aggregate. For example, most computer architectures support bitwise exclusive-or of 64 and 128 bit words. Thus, it makes much more sense to add regions of numbers in 64 or 128 bit chunks rather than as words in $GF(2^w)$. Multiplying a region by a constant can leverage similar optimizations.

GF-Complete supports multiplication and division of single values for all values of $w \leq 32$, plus $w = 64$ and $w = 128$. It also supports adding two regions of memory (for any value of w, since addition equals XOR), and multiplying a region by a constant in $GF(2^4)$, $GF(2^8)$, $GF(2^{16})$, $GF(2^{32})$, $GF(2^{64})$ and $GF(2^{128})$. These values are chosen because words in $GF(2^w)$ fit into machine words with these values of w. Other values of w don't lend themselves to efficient multiplication of regions by constants (although see the **"CAUCHY"** option in section 6.1.5 for a way to multiply regions for other values of w).

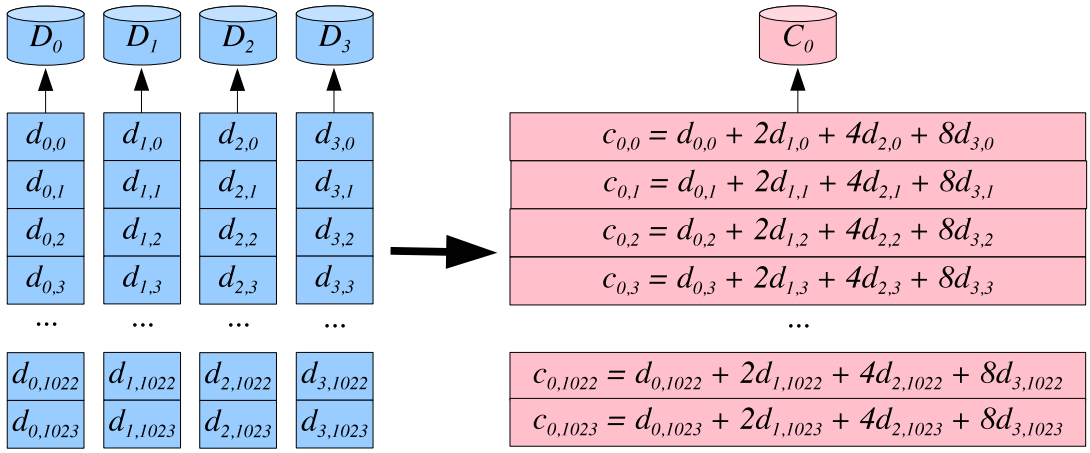

Figure 1: An example of adding two regions of numbers, and multiplying a region of numbers by a constant in $GF(2^w)$. In this example, $w = 8$, and each disk is holding a 1KB region. The same coding equation — $c_{0,j} = d_{0,j} + ad_{1,j} + a^2d_{2,j} + a^3d_{3,j}$ is applied 1024 times. However, rather than executing this equation 1024 times, it is more efficient to implement this with three region-constant multiplications and three region-region additions.

2 Files in the Library

This section provides an overview of the files that compose GF-Complete. They are partitioned among multiple directories.

2.1 Header files in the directory "include"

The following header files are part of GF-Complete.

- **gf_complete.h**: This is the header file that applications should include. It defines the **gf_t** type, which holds all of the data that you need to perform the various operations in $GF(2^w)$. It also defines all of the arithmetic operations. For an application to use this library, you should include **gf_complete.h** and then compile with the library **src/libgf_complete.la**.
- **gf_method.h**: If you are wanting to modify the implementation techniques from the defaults, this file provides a "helper" function so that you can do it from the Unix command line.
- **gf_general.h**: This file has helper routines for doing basic Galois Field operations with any legal value of w. The problem is that $w \leq 32$, $w = 64$ and $w = 128$ all have different data types, which is a pain. The procedures in this file try to alleviate that pain. They are used in **gf_mult**, **gf_unit** and **gf_time**. I'm guessing that most applications won't use them, as most applications use $w \leq 32$.
- **gf_rand.h**: I've learned that **srand48()** and its kin are not supported in all C installations. Therefore, this file defines some random number generators to help test the programs. The random number generator is the "Mother

of All" random number generator [Mar94] which we've selected because it has no patent issues. **gf_unit** and **gf_time** use these random number generators.

- **gf_int.h**: This is an internal header file that the various source files use. This is *not* intended for applications to include.
- **config.*xx*** and **stamp-h1** are created by autoconf, and should be ignored by applications.

2.2 Source files in the "src" directory

The following C files compose **gf_complete.a**, and they are in the direcoty **src**. You shouldn't have to mess with these files, but we include them in case you have to:

- **gf.c**: This implements all of the procedures in both **gf_complete.h** and **gf_int.h**.
- **gf_w4.c**: Procedures specific to $w = 4$.
- **gf_w8.c**: Procedures specific to $w = 8$.
- **gf_w16.c**: Procedures specific to $w = 16$.
- **gf_w32.c**: Procedures specific to $w = 32$.
- **gf_w64.c**: Procedures specific to $w = 64$.
- **gf_w128.c**: Procedures specific to $w = 128$.
- **gf_wgen.c**: Procedures specific to other values of w between 1 and 31.
- **gf_general.c**: Procedures that let you manipulate general values, regardless of whether $w \leq 32$, $w = 64$ or $w = 128$. (I.e. the procedures defined in **gf_general.h**).
- **gf_method.c**: Procedures to help you switch between the various implementation techniques. (I.e. the procedures defined in **gf_method.h**).
- **gf_rand.c**: The "Mother of all" random number generator. (I.e. the procedures defined in **gf_rand.h**).

2.3 Library tools files in the "tools" directory

The following are tools to help you with Galois Field arithmetic, and with the library. They are explained in greater detail elsewhere in this manual.

- **gf_mult.c**, **gf_div.c** and **gf_add**: Command line tools to do multiplication, division and addition by single numbers.
- **gf_time.c**: A program that times the procedures for given values of w and implementation options.
- **time_tool.sh**: A shell script that helps perform rough timings of the various multiplication, division and region operations in GF-Complete.
- **gf_methods.c**: A program that enumerates most of the implementation methods supported by GF-Complete.
- **gf_poly.c**: A program to identify irreducible polynomials in regular and composite Galois Fields.

2.4 The unit tester in the "test" directory

The **test** directory contains the proram **gf_unit.c**, which performs a battery of unit tests on GF-Complete. This is explained in more detail in section 6.3.

2.5 Example programs in the "examples" directory

There are seven example programs to help you understand various facets of GF-Complete. They are in the files **gf_example_x.c** in the **examples** directory. They are explained in sections 4.2 through 4.5, and section 7.9.

3 Compilation

From revision 1.02 forward, we are using autoconf. The old "flag_tester" directory is now gone, as it is no longer in use.

To compile and install, you should do the standard operations that you do with most open source Unix code:

```
UNIX> ./configure
...
UNIX> make
...
UNIX> sudo make install
```

If you perform the **install**, then the header, source, tool, and library files will be moved to system locations. In particular, you may then compile the library by linking with the flag **-lgf_complete**, and you may use the tools from a global executable directory (like **/usr/local/bin**).

If you don't perform the install, then the header and tool files will be in their respective directories, and the library will be in **src/libgf_complete.la**.

If your system supports the various Intel SIMD instructions, the compiler will find them, and GF-Complete will use them by default.

4 Some Tools and Examples to Get You Started

4.1 Three Simple Command Line Tools: gf_mult, gf_div and gf_add

Before delving into the library, it may be helpful to explore Galois Field arithmetic with the command line tools: **gf_mult**, **gf_div** and **gf_add**. These perform multiplication, division and addition on elements in $GF(2^w)$. If these are not installed on your system, then you may find them in the **tools** directory. Their syntax is:

- **gf_mult** a b w - Multiplies a and b in $GF(2^w)$.
- **gf_div** a b w - Divides a by b in $GF(2^w)$.
- **gf_add** a b w - Adds a and b in $GF(2^w)$.

You may use any value of w from 1 to 32, plus 64 and 128. By default, the values are read and printed in decimal; however, if you append an 'h' to w, then a, b and the result will be printed in hexadecimal. For $w = 128$, the 'h' is mandatory, and all values will be printed in hexadecimal.

Try them out on some examples like the ones below. You of course don't need to know that, for example, $5 * 4 = 7$ in $GF(2^4)$; however, once you know that, you know that $\frac{7}{5} = 4$ and $\frac{7}{4} = 5$. You should be able to verify the **gf_add** statements below in your head. As for the other **gf_mult**'s, you can simply verify that division and multiplication work with each other as you hope they would.

```
UNIX> gf_mult 5 4 4
7
UNIX> gf_div 7 5 4
4
UNIX> gf_div 7 4 4
5
UNIX> gf_mult 8000 2 16h
100b
UNIX> gf_add f0f0f0f0f0f0f0f0 1313131313131313 64h
e3e3e3e3e3e3e3e3
UNIX> gf_mult f0f0f0f0f0f0f0f0 1313131313131313 64h
8da08da08da08da0
UNIX> gf_div 8da08da08da08da0 1313131313131313 64h
f0f0f0f0f0f0f0f0
UNIX> gf_add f0f0f0f0f0f0f0f001313131313131313 13131313131313f0f0f0f0f0f0f0f0f0 128h
e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3
UNIX> gf_mult f0f0f0f0f0f0f0f001313131313131313 13131313131313f0f0f0f0f0f0f0f0 128h
786278627862784982d782d782d7816e
UNIX> gf_div 786278627862784982d782d782d7816e f0f0f0f0f0f0f0f001313131313131313 128h
13131313131313f0f0f0f0f0f0f0f0
UNIX>
```

Don't bother trying to read the source code of these programs yet. Start with some simpler examples like the ones below.

4.2 Quick Starting Example #1: Simple multiplication and division

The source files for these examples are in the **examples** directory.

These two examples are intended for those who just want to use the library without getting too complex. The first example is **gf_example_1**, and it takes one command line argument – w, which must be between 1 and 32. It generates two random non-zero numbers in $GF(2^w)$ and multiplies them. After doing that, it divides the product by each number.

To perform multiplication and division in $GF(2^w)$, you must declare an instance of the **gf_t** type, and then initialize it for $GF(2^w)$ by calling **gf_init_easy()**. This is done in **gf_example_1.c** with the following lines:

```
gf_t gf;

...

if (!gf_init_easy(&gf, w)) {
  fprintf(stderr, "Couldn't initialize GF structure.\n");
  exit(0);
}
```

Once **gf** is initialized, you may use it for multiplication and division with the function pointers **multiply.w32** and **divide.w32**. These work for any element of $GF(2^w)$ so long as $w \leq 32$.

```
c = gf.multiply.w32(&gf, a, b);
printf("%u * %u = %u\n", a, b, c);

printf("%u / %u = %u\n", c, a, gf.divide.w32(&gf, c, a));
printf("%u / %u = %u\n", c, b, gf.divide.w32(&gf, c, b));
```

Go ahead and test this program out. You can use **gf_mult** and **gf_div** to verify the results:

```
UNIX> gf_example_1 4
12 * 4 = 5
5 / 12 = 4
5 / 4 = 12
UNIX> gf_mult 12 4 4
5
UNIX> gf_example_1 16
14411 * 60911 = 44568
44568 / 14411 = 60911
44568 / 60911 = 14411
UNIX> gf_mult 14411 60911 16
44568
UNIX>
```

gf_init_easy() (and later **gf_init_hard()**) do call **malloc()** to implement internal structures. To release memory, call **gf_free()**. Please see section 6.4 to see how to call **gf_init_hard()** in such a way that it doesn't call **malloc()**.

4.3 Quick Starting Example #2: Multiplying a region by a constant

The program **gf_example_2** expands on **gf_example_1**. If w is equal to 4, 8, 16 or 32, it performs a region multiply operation. It allocates two sixteen byte regions, **r1** and **r2**, and then multiples **r1** by **a** and puts the result in **r2** using the **multiply_region.w32** function pointer:

```
gf.multiply_region.w32(&gf, r1, r2, a, 16, 0);
```

That last argument specifies whether to simply place the product into **r2** or to XOR it with the contents that are already in **r2**. Zero means to place the product there. When we run it, it prints the results of the **multiply_region.w32** in hexadecimal. Again, you can verify it using **gf_mult**:

```
UNIX> gf_example_2 4
12 * 2 = 11
11 / 12 = 2
11 / 2 = 12

multiply_region by 0xc (12)

R1 (the source):   0 2 d 9 d 6 8 a 8 d b 3 5 c 1 8 8 e b 0 6 1 5 a 2 c 4 b 3 9 3 6
R2 (the product):  0 b 3 6 3 e a 1 a 3 d 7 9 f c a a 4 d 0 e c 9 1 b f 5 d 7 6 7 e
```

```
UNIX> gf_example_2 16
49598 * 35999 = 19867
19867 / 49598 = 35999
19867 / 35999 = 49598

multiply_region by 0xc1be (49598)

R1 (the source):   8c9f b30e 5bf3 7cbb 16a9 105d 9368 4bbe
R2 (the product):  4d9b 992d 02f2 c95c 228e ec82 324e 35e4
UNIX> gf_mult c1be 8c9f 16h
4d9b
UNIX> gf_mult c1be b30e 16h
992d
UNIX>
```

4.4 Quick Starting Example #3: Using $w = 64$

The program in **gf_example_3.c** is identical to the previous program, except it uses $GF(2^{64})$. Now **a**, **b** and **c** are **uint64_t**'s, and you have to use the function pointers that have **w64** extensions so that the larger types may be employed.

```
UNIX> gf_example_3
a9af3adef0d23242 * 61fd8433b25fe7cd = bf5acdde4c41ee0c
bf5acdde4c41ee0c / a9af3adef0d23242 = 61fd8433b25fe7cd
bf5acdde4c41ee0c / 61fd8433b25fe7cd = a9af3adef0d23242

multiply_region by a9af3adef0d23242

R1 (the source):   61fd8433b25fe7cd 272d5d4b19ca44b7 3870bf7e63c3451a 08992149b3e2f8b7
R2 (the product):  bf5acdde4c41ee0c ad2d786c6e4d66b7 43a7d857503fd261 d3d29c7be46b1f7c
UNIX> gf_mult a9af3adef0d23242 61fd8433b25fe7cd 64h
bf5acdde4c41ee0c
UNIX>
```

4.5 Quick Starting Example #4: Using $w = 128$

Finally, the program in **gf_example_4.c** uses $GF(2^{128})$. Since there is not universal support for **uint128_t**, the library represents 128-bit numbers as arrays of two **uint64_t**'s. The function pointers for multiplication, division and region multiplication now accept the return values as arguments:

```
  gf.multiply.w128(&gf, a, b, c);
```

Again, we can use **gf_mult** and **gf_div** to verify the results:

```
UNIX> gf_example_4
e252d9c145c0bf29b85b21a1ae2921fa * b23044e7f45daf4d70695fb7bf249432 =
7883669ef3001d7fabf83784d52eb414
```

```
multiply_region by e252d9c145c0bf29b85b21a1ae2921fa

R1 (the source):   f4f56f08fa92494c5faa57ddcd874149 b4c06a61adbbec2f4b0ffc68e43008cb
R2 (the product):  b1e34d34b031660676965b868b892043 382f12719ffe3978385f5d97540a13a1
UNIX> gf_mult e252d9c145c0bf29b85b21a1ae2921fa f4f56f08fa92494c5faa57ddcd874149 128h
b1e34d34b031660676965b868b892043
UNIX> gf_div 382f12719ffe3978385f5d97540a13a1 b4c06a61adbbec2f4b0ffc68e43008cb 128h
e252d9c145c0bf29b85b21a1ae2921fa
UNIX>
```

5 Important Information on Alignment when Multiplying Regions

In order to make multiplication of regions fast, we often employ 64 and 128 bit instructions. This has ramifications for pointer alignment, because we want to avoid bus errors, and because on many machines, loading and manipulating aligned quantities is much faster than unalinged quantities.

When you perform **multiply_region.w*xx*(**gf**,** $source$**,** $dest$**,** $value$**,** $size$**,** add**)**, there are three requirements:

1. The pointers $source$ and $dest$ must be aligned for w-bit words. For $w = 4$ and $w = 8$, there is no restriction; however for $w = 16$, the pointers must be multiples of 2, for $w = 32$, they must be multiples of 4, and for $w \in \{64, 128\}$, they must be multiples of 8.

2. The $size$ must be a multiple of $\lceil \frac{w}{8} \rceil$. With $w = 4$ and $w = 8$, $\lceil \frac{w}{8} \rceil = 1$ and there is no restriction. The other sizes must be multiples of $\lceil \frac{w}{8} \rceil$ because you have to be multiplying whole elements of $GF(2^w)$.

3. The $source$ and $dest$ pointers must be aligned identically with respect to each other for the implementation chosen. This is subtle, and we explain it in detail in the next few paragraphs. However, if you'd rather not figure it out, the following recommendation will *always* work in GF-Complete:

   > **If you want to be safe, make sure that** $source$ **and** $dest$ **are both multiples of 16. That is not a strict requirement, but it will always work!**

If you want to relax the above recommendation, please read further.

When performing **multiply_region.w*xx*()**, the implementation is typically optimized for a region of bytes whose size must be a multiple of a variable s, and which must be aligned to a multiple of another variable t. For example, when doing **multiply_region.w32()** in $GF(2^{16})$ with SSE enabled, the implementation is optimized for regions of 32 bytes, which must be aligned on a 16-byte quantity. Thus, $s = 32$ and $t = 16$. However, we don't want **multiply_region.w32()** to be too restrictive, so instead of requiring $source$ and $dest$ to be aligned to 16-byte regions, we require that ($source$ mod 16) equal ($dest$ mod 16). Or, in general, that ($source$ mod t) equal ($dest$ mod t).

Then, **multiply_region.w*xx*()** proceeds in three phases. In the first phase, **multiply.w*xx*()** is called on successive words until ($source$ mod t) equals zero. The second phase then performs the optimized region multiplication on chunks of s bytes, until the remaining part of the region is less than s bytes. At that point, the third phase calls **multiply.w*xx*()** on the last part of the region.

A detailed example helps to illustrate. Suppose we make the following call in $GF(2^{16})$ with SSE enabled:

multiply_region.w32(gf**, 0x10006, 0x20006,** a**,** 274**,** 0**)**

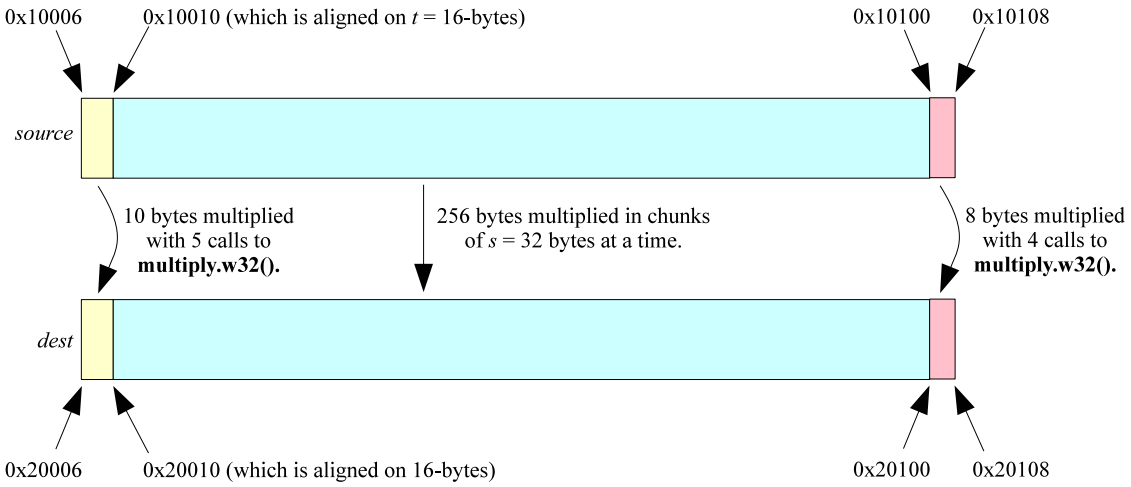

Figure 2: Example of multiplying a region of 274 bytes in $GF(2^{16})$ when ($source$ mod 16) = ($dest$ mod 16) = 6. The alignment parameters are $s = 32$ and $t = 16$. The multiplication is in three phases, which correspond to the initial unaligned region (10 bytes), the aligned region of s-byte chunks (256 bytes), and the final leftover region (8 bytes).

First, note that $source$ and $dest$ are aligned on two-byte quantities, which they must be in $GF(2^{16})$. Second, note that $size$ is a multiple of $\lceil \frac{16}{8} \rceil = 2$. And last, note that ($source$ mod 16) equals ($dest$ mod 16). We illustrate the three phases of region multiplication in Figure 2. Because ($source$ mod 16) = 6, there are 10 bytes of unaligned words that are multiplied with five calls to **multiply.w32()** in the first phase. The second phase multiplies 256 bytes (eight chunks of $s = 32$ bytes) using the SSE instructions. That leaves 8 bytes remaining for the third phase.

When we describe the defaults and the various implementation options, we specify s and t as "alignment parameters."

One of the advanced region options is using an alternate mapping of words to memory ("ALTMAP"). These interact in a more subtle manner with alignment. Please see Section 7.9 for details.

6 The Defaults

GF-Complete implements a wide variety of techniques for multiplication, division and region multiplication. We have set the defaults with three considerations in mind:

1. **Speed:** Obviously, we want the implementations to be fast. Therefore, we choose the fastest implementations that don't violate the other considerations. The compilation environment is considered. For example, if SSE is enabled, region multiplication in $GF(2^4)$ employs a single multiplication table. If SSE is not enabled, then a "double" table is employed that performs table lookup two bytes at a time.

2. **Memory Consumption:** We try to keep the memory footprint of GF-Complete low. For example, the fastest way to perform **multiply.w32()** in $GF(2^{32})$ is to employ 1.75 MB of multiplication tables (see Section 7.4 below). We do not include this as a default, however, because we want to keep the default memory consumption of GF-Complete low.

3. **Compatibility with "standard" implementations:** While there is no *de facto* standard of Galois Field arithmetic, most libraries implement the same fields. For that reason, we have not selected composite fields, alternate polynomials or memory layouts for the defaults, even though these would be faster. Again, see section 7.7 for more information.

Table 1 shows the default methods used for each power-of-two word size, their alignment parameters s and t, their memory consumption and their rough performance. The performance tests are on an Intel Core i7-3770 running at 3.40 GHz, and are included solely to give a flavor of performance on a standard microprocessor. Some processors will be faster with some techniques and others will be slower, so we only put numbers in so that you can ballpark it. For other values of w between 1 and 31, we use table lookup when $w \leq 8$, discrete logarithms when $w \leq 16$ and "Bytwo$_p$" for $w \leq 32$.

With SSE

w	Memory Usage	**multiply()** Implementation	Performance (Mega Ops / s)	**multiply_region()** Implementation	s	t	Performance (MB/s)
4	$<$ 1K	Table	501	Table	16	16	11,659
8	136K	Table	501	Split Table (8,4)	16	16	11,824
16	896K	Log	260	Split Table (16,4)	32	16	7,749
32	$<$ 1K	Carry-Free	48	Split Table (32,4)	64	16	5,011
64	2K	Carry-Free	84	Split Table (64,4)	128	16	2,402
128	64K	Carry-Free	48	Split Table (128,8)	16	16	833

Without SSE

w	Memory Usage	**multiply()** Implementation	Performance (Mega Ops / s)	**multiply_region()** Implementation	s	t	Performance (MB/s)
4	4K	Table	501	Double Table	1	1	1,982
8	128K	Table	501	Table	1	1	1,397
16	896K	Log	266	Split Table (16,8)	2	2	2,135
32	4K	Bytwo$_p$	19	Split Table (32,8)	4	4	1,149
64	16K	Bytwo$_p$	9	Split Table (64,8)	8	8	987
128	64K	Bytwo$_p$	1.4	Split Table (128,8)	16	8	833

Table 1: The default implementations, memory consumption and rough performance when w is a power of two. The variables s and t are alignment variables described in Section 5.

A few comments on Table 1 are in order. First, with SSE, the performance of **multiply()** is faster when $w = 64$ than when $w = 32$. That is because the primitive polynomial for $w = 32$, that has historically been used in Galois Field implementations, is sub-ideal for using carry-free multiplication (PCLMUL). You can change this polynomial (see section 7.7) so that the performance matches $w = 64$.

The region operations for $w = 4$ and $w = 8$ without SSE have been selected to have a low memory footprint. There are better options that consume more memory, or that only work on large memory regions (see section 6.1.5).

6.1 Changing the Defaults

There are times that you may want to stray from the defaults. For example:

- You may want better performance.

- You may want a lower memory footprint.
- You may want to use a different Galois Field or even a ring.
- You only care about multiplying a region by the value two.

Our command line tools allow you to deviate from the defaults, and we have two C functions — **gf_init_hard()** and **create_gf_from_argv()** — that can be called from application code to override the default methods. There are six command-line tools that can be used to explore the many techniques implemented in GF-Complete:

- **gf_methods** is a tool that enumerates most of the possible command-line arguments that can be sent to the other tools.
- **gf_mult** and **gf_div** are explained above. You may change the multiplication and division technique in these tools if you desire.
- **gf_unit** performs unit tests on a set of techniques to verify correctness.
- **gf_time** measures the performance of a particular set of techniques.
- **time_tool.sh** makes some quick calls to **gf_time** so that you may gauge rough performance..
- **gf_poly** tests the irreducibility of polynomials in a Galois Field.

To change the default behavior in application code, you need to call **gf_init_hard()** rather than **gf_init_easy()**. Alternatively, you can use **create_gf_from_argv()**, included from **gf_method.h**, which uses an **argv**-style array of strings to specify the options that you want. The procedure in **gf_method.c** parses the array and makes the proper **gf_init_hard()** procedure call. This is the technique used to parse the command line in **gf_mult**, **gf_div**, **gf_unit** *et al*.

6.1.1 Changing the Components of a Galois Field with create_gf_from_argv()

There are five main components to every Galois Field instance:

- w
- Multiplication technique
- Division technique
- Region technique(s)
- Polynomial

The procedures **gf_init_hard()** and **create_gf_from_argv()** allow you to specify these parameters when you create your Galois Field instance. We focus first on **create_gf_from_argv()**, because that is how the tools allow you to specify the components. The prototype of **create_gf_from_argv()** is as follows:

```
int create_gf_from_argv(gf_t *gf, int w, int argc, char **argv, int starting);
```

You pass it a pointer to a **gf_t**, which it will initialize. You specify the word size with the parameter **w**, and then you pass it an **argc**/**argv** pair as in any C or C++ program. You also specify a **starting** argument, which is where in **argv** the specifications begin. If it successfully parses **argc** and **argv**, then it creates the **gf_t** using **gf_init_hard()** (described below in section 6.4). It returns one past the last index of **argv** that it considered when creating the **gf_t**. If it fails, then it returns zero, and the **gf_t** is unmodified.

For example, **gf_mult.c** calls **create_gf_from_argv()** by simply passing **argc** and **argv** from its **main()** declaration, and setting **starting** to 4.

To choose defaults, **argv[starting]** should equal "-". Otherwise, you specify the component that you are changing with "-m" for multiplication technique, "-d" for division technique, "-r" for region technique, and "-p" for the polynomial. You may change multiple components. You end your specification with a single dash. For example, the following call multiplies 6 and 5 in $GF(2^4)$ with polynomial 0x19 using the "SHIFT" technique for multiplication (we'll explain these parameters later):

```
UNIX> ./gf_mult 6 5 4 -p 0x19 -m SHIFT -
7
UNIX>
```

If **create_gf_from_argv()** fails, then you can call the procedure **gf_error()**, which prints out the reason why **create_gf_from_argv()** failed.

6.1.2 Changing the Polynomial

Galois Fields are typically implemented by representing numbers as polynomials with binary coefficients, and then using the properties of polynomials to define addition and multiplication. You do not need to understand any of that to use this library. However, if you want to learn more about polynomial representations and how they construct fields, please refer to The Paper.

Multiplication is based on a special polynomial that we will refer to here as the "defining polynomial." This polynomial has binary coefficients and is of degree w. You may change the polynomial with "-p" and then a number in hexadecimal (the leading "0x" is optional). It is assumed that the w-th bit of the polynomial is set – you may include it or omit it. For example, if you wish to set the polynomial for $GF(2^{16})$ to $x^{16} + x^5 + x^3 + x^2 + 1$, rather than its default of $x^{16} + x^{12} + x^3 + x + 1$, you may say "-p 0x1002d," "-p 1002d," "-p 0x2d" or "-p 2d."

We discuss changing the polynomial for three reasons in other sections:

- Leveraging carry-free multiplication (section 7.7).
- Defining composite fields (section 7.6).
- Implementing rings (section 7.8.1).

Some words about nomenclature with respect to the polynomial. A Galois Field requires the polynomial to be *irreducible.*. That means that it cannot be factored. For example, when the coefficients are binary, the polynomial $x^5 + x^4 + x + 1$ may be factored as $(x^4 + 1)(x + 1)$. Therefore it is not irreducible and cannot be used to define a Galois Field. It may, however, be used to define a ring. Please see section 7.8.1 for a discussion of ring support in GF-Complete.

There is a subset of irreducible polynomials called *primitive*. These have an important property that one may enumerate all of the elements of the field by raising 2 to successive posers. All of the default polynomials in GF-Complete are primitive. However, so long as a polynomial is irreducible, it defines a Galois Field. Please see section 7.7 for a further discussion of the polynomial.

One thing that we want to stress here is that changing the polynomial changes the field, so fields with different polynomials may not be used interchangeably. So long as the polynomial is irreducible, it generates a Galois Field that is isomorphic to all other Galois Fields; however the multiplication and division of elements will differ. For example, the polynomials 0x13 (the default) and 0x19 in $GF(2^4)$ are both irreducible, so both generate valid Galois Fields. However, their multiplication differs:

```
UNIX> gf_mult 8 2 4 -p 0x13 -
3
UNIX> gf_mult 8 2 4 -p 0x19 -
9
```

```
UNIX> gf_div 3 8 4 -p 0x13 -
2
UNIX> gf_div 9 8 4 -p 0x19 -
2
UNIX>
```

6.1.3 Changing the Multiplication Technique

The following list describes the multiplication techinques that may be changed with "-m". We keep the description here brief. Please refer to The Paper for detailed descriptions of these techniques.

- "**TABLE**:" Multiplication and division are implemented with tables. The tables consume quite a bit of memory ($2^w \times 2^w \times \lceil \frac{w}{8} \rceil$ bytes), so they are most useful when w is small. Please see "**SSE**," "**LAZY**," "**DOUBLE**" and "**QUAD**" under region techniques below for further modifications to "**TABLE**" to perform **multiply_region()**.

- "**LOG**:" This employs discrete (or "Zeph") logarithm tables to implement multiplication and division. The memory usage is roughly ($3 \times 2^w \times \lceil \frac{w}{8} \rceil$ bytes), so they are most useful when w is small, but they tolerate larger w than "**TABLE**." If the polynomial is not primitive (see section 6.1.2), then you cannot use "**LOG**" as an implementation. In that case, **gf_init_hard()** or **create_gf_from_argv()** will fail.

- "**LOG_ZERO**:" Discrete logarithm tables which include extra room for zero entries. This more than doubles the memory consumption to remove an **if** statement (please see [GMS08] or The Paper for more description). It doesn't really make a huge deal of difference in performance.

- "**LOG_ZERO_EXT**:" This expends even more memory to remove another **if** statement. Again, please see The Paper for an explanation. As with "**LOG_ZERO**," the performance difference is negligible.

- "**SHIFT**:" Implementation straight from the definition of Galois Field multiplication, by shifting and XOR-ing, then reducing the product using the polynomial. This is *slooooooooow*, so we don't recommend you use it.

- "**CARRY_FREE**:" This is identical to "**SHIFT**," however it leverages the SSE instruction PCLMUL to perform carry-free multiplications in single instructions. As such, it is the fastest way to perform multiplication for large values of w when that instruction is available. Its performance depends on the polynomial used. See The Paper for details, and see section 7.7 below for the speedups available when $w = 16$ and $w = 32$ if you use a different polynomial than the default one.

- "**BYTWO_p**:" This implements multiplication by successively multiplying the product by two and selectively XOR-ing the multiplicand. See The Paper for more detail. It can leverage Anvin's optimization that multiplies 64 and 128 bits of numbers in $GF(2^w)$ by two with just a few instructions. The SSE version requires SSE2.

- "**BYTWO_b**:" This implements multiplication by successively multiplying the multiplicand by two and selectively XOR-ing it into the product. It can also leverage Anvin's optimization, and it has the feature that when you're multiplying a region by a very small constant (like 2), it can terminate the multiplication early. As such, if you are multiplying regions of bytes by two (as in the Linux RAID-6 Reed-Solomon code [Anv09]), this is the fastest of the techniques, regardless of the value of w. The SSE version requires SSE2.

- "**SPLIT**:" Split multiplication tables (like the LR tables in [GMS08], or the SIMD tables for $w \geq 8$ in [LHy08, Anv09, PGM13b]). This argument must be followed by two more arguments, w_a and w_b, which are the index sizes of the sub-tables. This implementation reduces the size of the table from "**TABLE**," but requires multiple

table lookups. For example, the following multiplies 100 and 200 in $GF(2^8)$ using two 4K tables, as opposed to one 64K table when you use "**TABLE**:"

```
UNIX> ./gf_mult 100 200 8 -m SPLIT 8 4 -
79
UNIX>
```

See section 7.4 for additional information on the arguments to "**SPLIT**." The SSE version typically requires SSSE3.

- "**GROUP**:" This implements the "left-to-right comb" technique [LBOX12]. I'm afraid we don't like that name, so we call it "**GROUP**," because it performs table lookup on groups of bits for shifting (left) and reducing (right). It takes two additional arguments – g_s, which is the number of bits you use while shifting (left) and g_r, which is the number of bits you use while reducing (right). Increasing these arguments can you higher computational speed, but requires more memory. SSE version exists only for $w = 128$ and it requires SSE4. For more description on the arguments g_s and g_r, see section 7.5. For a full description of "**GROUP**" algorithm, please see The Paper.

- "**COMPOSITE**:" This allows you to perform operations on a composite Galois Field, $GF((2^l)^k)$ as described in [GMS08], [LBOX12] and The Paper. The field size w is equal to lk. It takes one argument, which is k, and then a specification of the base field. Currently, the only value of k that is supported is two. However, that may change in a future revision of the library.

  In order to specify the base field, put appropriate flags after specifying k. The single dash ends the base field, and after that, you may continue making specifications for the composite field. This process can be continued for multiple layers of "**COMPOSITE**." As an example, the following multiplies 1000000 and 2000000 in $GF((2^{16})^2)$, where the base field uses **BYTWO_p** for multiplication:

```
./gf_mult 1000000 2000000 32 -m COMPOSITE 2 -m BYTWO_p - -
```

  In the above example, the red text applies to the base field, and the black text applies to the composite field.

  Composite fields have two defining polynomials – one for the composite field, and one for the base field. Thus, if you want to change polynomials, you should change both. The polynomial for the composite field must be of the form x^2+sx+1, where s is an element of $GF(2^k)$. To change it, you specify s (in hexadecimal) with "-p." In the example below, we multiply 20000 and 30000 in $GF((2^8)^2)$, setting s to three, and using $x^8+x^4+x^3+x^2+1$ as the polynomial for the base field:

```
./gf_mult 20000 30000 16 -m COMPOSITE 2 -p 0x11d - -p 0x3 -
```

  If you use composite fields, you should consider using "**ALTMAP**" as well. The reason is that the region operations will go much faster. Please see section 7.6.

  As with changing the polynomial, when you use a composite field, $GF((2^l)^k)$, you are using a different field than the "standard" field for $GF(2^{lk})$. All Galois Fields are isomorphic to each other, so they all have the desired properties; however, the fields themselves change when you use composite fields.

With the exception of **"COMPOSITE"**, only one multiplication technique can be provided for a given Galois Field instance. Composite fields may use composite fields as their base fields, in which case the specification will be recursive.

6.1.4 Changing the Division Technique

There are two techniques for division that may be set with "-d". If "-d" is not specified, then appropriate defaults are employed. For example, when the multiplication technique is "**TABLE**," a table is created for division as well as multiplication. When "**LOG**" is specified, the logarithm tables are used for division. With "**COMPOSITE**," a special variant of Euclid's algorithm is employed that performs division using multiplication and division in the base field. Otherwise, Euclid's algorithm is used. Please see The Paper for a description of Euclid's algorithm applied to Galois Fields.

If you use "-d", you must also specify the multiplication technique with "-m."

To force Euclid's algorithm instead of the defaults, you may specify it with "-d EUCLID." If instead, you would rather convert elements of a Galois Field to a binary matrix and find an element's inverse by inverting the matrix, then specify "-d MATRIX." In all of our tests, "**MATRIX**" is slower than "**EUCLID**." "**MATRIX**" is also not defined for $w > 32$.

6.1.5 Changing the Region Technique

The following are the region multiplication options ("-r"):

- "**SSE**:" Use SSE instructions. Initialization will fail if the instructions aren't supported. Table 2 details the multiplication techniques which can leverage SSE instructions and which versions of SSE are required.

Multiplication Technique	**multiply()**	**multiply_region()**	SSE Version	Comments
"**TABLE**"	-	Yes	SSSE3	Only for $GF(2^4)$.
"**SPLIT**"	-	Yes	SSSE3	Only when the second argument equals 4.
"**SPLIT**"	-	Yes	SSE4	When $w = 64$ and not using "**ALTMAP**".
"**BYTWO_p**"	-	Yes	SSE2	
"**BYTWO_b**"	-	Yes	SSE2	

Table 2: Multiplication techniques which can leverage SSE instructions when they are available.

- "**NOSSE**:" Force non-SSE version.
- "**DOUBLE**:" Use a table that is indexed on two words rather than one. This applies only to $w = 4$, where the table is indexed on bytes rather than 4-bit quantities, and to $w = 8$, where the table is indexed on shorts rather than bytes. In each case, the table lookup performs two multiplications at a time, which makes region multiplication faster. It doubles the size of the lookup table.
- "**QUAD**:" Use a table that is indexed on four words rather than two or one. This only applies to $w = 4$, where the table is indexed on shorts. The "Quad" table may be lazily created or created ahead of time (the default). If the latter, then it consumes $2^4 \times 2^{16} \times 2$ = 2 MB of memory.
- "**LAZY**:" Typically it's clear whether tables are constructed upon initialization or lazily when a region operation is performed. There are two times where it is ambiguous: "**QUAD**" when $w = 4$ and "**DOUBLE**" when $w = 8$. If you don't specify anything, these tables are created upon initialization, consuming a lot of memory. If you specify "**LAZY**," then the necessary row of the table is created lazily when you call **"multiply_region()**.

- "**ALTMAP**:" Use an alternate mapping, where words are split across different subregions of memory. There are two places where this matters. The first is when implementing "**SPLIT** w **4**" using SSE when $w > 8$. In these cases, each byte of the word is stored in a different 128-bit vector, which allows the implementation to better leverage 16-byte table lookups. See section 7.4 for examples, and The Paper or [PGM13b] for detailed explanations.

  The second place where it matters is when using "**COMPOSITE**." In this case, it is advantageous to split each memory region into two chunks, and to store half of each word in a different chunk. This allows us to call **region_multiply()** recursively on the base field, which is *much* faster than the alternative. See Section 7.6 for examples, and The Paper for an explanation.

  It is important to note that with "**ALTMAP**," the words are not "converted" from a standard mapping to an alternate mapping and back again. They are assumed to always be in the alternate mapping. This typically doesn't matter, so long as you always use the same "**ALTMAP**" calls. Please see section 7.9 for further details on "**ALTMAP**," especially with respect to alignment.

- "**CAUCHY**:" Break memory into w subregions and perform only XOR's as in Cauchy Reed-Solomon coding [BKK+95] (also described in The Paper). This works for *any* value of $w \leq 32$, even those that are not powers of two. If SSE2 is available, then XOR's work 128 bits at a time. For "**CAUCHY**" to work correctly, *size* must be a multiple of w.

It is possible to combine region multiplication options. This is fully supported as long as **gf_methods** has the combination listed. If multiple region options are required, they should be specified independently (as flags for **gf_init_hard()** and independent options for command-line tools and **create_gf_from_argv()**).

6.2 Determining Supported Techniques with gf_methods

The program **gf_methods** prints a list of supported methods on standard output. It is called as follows:

```
./gf_methods w -BADC -LUMDRB
```

The first argument is w, which may be any legal value of w. The second argument has the following flags:

- "**B**:" This only prints out "basic" methods that are useful for the given value of w. It omits "**SHIFT**" and other methods that are never really going to be useful.
- "**A**:" In constrast, this specifies to print "all" methods.
- "**D**:" This includes the "**EUCLID**" and "**MATRIX**" methods for division. By default, they are not included.
- "**C**:" This includes the "**CAUCHY**" methods for region multiplication. By default, it is not included.

You may specify multiple of these as the second argument. If you include both "**B**" and "**A**," then it uses the last one specified.

The last argument determines the output format of **gf_methods**. If it is "**L**," then it simply lists methods. If it is "**U**," then the output contains **gf_unit** commands for each of the methods. For the others, the output contains **gf_time_tool.sh** commands for **M**ultiplication, **D**ivision, **R**egion multiplications with multiple buffer sizes, and the **B**est region multiplication.

gf_methods enumerates combinations of flags, and calls **create_gf_from_argv()** to see if the combinations are supported. Although it enumerates a large number of combinations, it doesn't enumerate all possible parameters for "**SPLIT**," "**GROUP**" or "**COMPOSITE**."

Some examples of calling **gf_methods** are shown below in section 6.3.2.

6.3 Testing with gf_unit, gf_time, and time_tool.sh

gf_unit and **gf_time** may be used to verify that a combination of arguments works correctly and efficiently on your platform. If you plan to stray from the defaults, it is probably best to run both tools to ensure there are no issues with your environment. **gf_unit** will run a set of unit tests based on the arguments provided to the tool, and **gf_time** will time Galois Field methods based on the provided arguments.

The usage of **gf_unit** is:

```
gf_unit w tests seed method
```

The usage of **gf_time** is:

```
gf_time w tests seed buffer-size iterations method
```

The **seed** is an integer — negative one uses the current time. The tests are specified by a listing of characters. The following tests are supported (All are supported by **gf_time**. Only ', 'S' and 'R' are supported by **gf_unit**):

- **'M'**: Single multiplications.
- **'D'**: Single divisions.
- **'I'**: Single inverses.
- **'G'**: Region multiplication of a buffer by a random constant.
- **'0'**: Region multiplication of a buffer by zero (does nothing and **bzero()**).
- **'1'**: Region multiplication of a buffer by one (does **memcpy() and XOR**).
- **'2'**: Region multiplication of a buffer by two – sometimes this is faster than general multiplication.
- **'S'**: All three single tests.
- **'R'**: All four region tests.
- **'A'**: All seven tests.

Here are some examples of calling **gf_unit** and **gf_time** to verify that "**-m SPLIT 32 4 -r ALTMAP -**" works in $GF(2^{32})$, and to get a feel for its performance. First, we go to the **test** directory and call **gf_unit**:

```
UNIX> cd test
UNIX> ./gf_unit 32 A -1 -m SPLIT 32 4 -r ALTMAP -
Args: 32 A -1 -m SPLIT 32 4 -r ALTMAP - / size (bytes): 684
UNIX>
```

gf_unit reports on the arguments and how may bytes the **gf_t** consumes. If it discovers any problems or inconsistencies with multiplication, division or region multiplication, it will report them. Here, there are no problems.

Next, we move to the **tools** directory and run performance tests on a 10K buffer, with 10,000 iterations of each test:

```
UNIX> cd ../tools
UNIX> ./gf_time 32 A -1 10240 10000 -m SPLIT 32 4 -r ALTMAP -
Seed: 1388435794
      Multiply:            4.090548 s   Mops:     24.414        5.968 Mega-ops/s
        Divide:           37.794962 s   Mops:     24.414        0.646 Mega-ops/s
       Inverse:           33.709875 s   Mops:     24.414        0.724 Mega-ops/s
 Region-Random: XOR: 0     0.035210 s     MB:     97.656     2773.527 MB/s
 Region-Random: XOR: 1     0.036081 s     MB:     97.656     2706.578 MB/s
Region-By-Zero: XOR: 0     0.003199 s     MB:     97.656    30523.884 MB/s
Region-By-Zero: XOR: 1     0.000626 s     MB:     97.656   156038.095 MB/s
```

```
 Region-By-One: XOR: 0      0.003810 s     MB:     97.656     25628.832 MB/s
 Region-By-One: XOR: 1      0.008363 s     MB:     97.656     11677.500 MB/s
 Region-By-Two: XOR: 0      0.032942 s     MB:     97.656      2964.486 MB/s
 Region-By-Two: XOR: 1      0.033488 s     MB:     97.656      2916.153 MB/s
UNIX>
```

The first column of output displays the name of the test performed. Region tests will test with and without the XOR flag being set (see Section 4.3 for an example). The second column displays the total time the test took to complete measured in seconds (s). The third column displays the size of the test measured in millions of operations (Mops) for single tests and in Megabytes (MB) for the region tests. The final column displays the speed of the tests calculated from the second and third columns, and is where you should look to get an idea of a method's performance.

If the output of **gf_unit** and **gf_time** are to your satisfaction, you can incorporate the method into application code using **create_gf_from_argv()** or **gf_init_hard()**.

The performance of "Region-By-Zero" and "Region-By-One" will not change from test to test, as all methods make the same calls for these. "Region-By-Zero" with "XOR: 1" does nothing except set up the tests. Therefore, you may use it as a control.

6.3.1 time_tool.sh

Finally, the shell script **time_tool.sh** makes a bunch of calls to **gf_time** to give a rough estimate of performance. It is called as follows:

```
usage sh time_tool.sh M|D|R|B w method
```

The values for the first argument are **MDRB**, for **M**ultiplication, **D**ivision, **R**egion multiplications with multiple buffer sizes, and the **B**est region multiplication. For the example above, let's call **time_tool.sh** to get a rough idea of performance:

```
UNIX> sh time_tool.sh M 32 -m SPLIT 32 4 -r ALTMAP -
M speed (MB/s):     6.03   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
UNIX> sh time_tool.sh D 32 -m SPLIT 32 4 -r ALTMAP -
D speed (MB/s):     0.65   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
UNIX> sh time_tool.sh R 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size:  16K (MB/s):  3082.91   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size:  32K (MB/s):  3529.07   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size:  64K (MB/s):  3749.94   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size: 128K (MB/s):  3861.27   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size: 256K (MB/s):  3754.97   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size: 512K (MB/s):  3820.82   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size:   1M (MB/s):  3737.41   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size:   2M (MB/s):  3002.90   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Buffer-Size:   4M (MB/s):  2760.77   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Best (MB/s):  3861.27   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
UNIX> sh time_tool.sh B 32 -m SPLIT 32 4 -r ALTMAP -
Region Best (MB/s):  3929.09   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
UNIX>
```

We say that **time_tool.sh** is "rough" because it tries to limit each test to 5 ms or less. Thus, the time granularity is fine, which means that the numbers may not be as precise as they could be were the time granularity to be course. When in doubt, you should make your own calls to **gf_time** with a lot of iterations, so that startup costs and roundoff error may be minimized.

6.3.2 An example of gf_methods and time_tool.sh

Let's give an example of how some of these components fit together. Suppose we want to explore the basic techniques in $GF(2^{32})$. First, let's take a look at what **gf_methods** suggests as "basic" methods:

```
UNIX> gf_methods 32 -B -L
w=32: -
w=32: -m GROUP 4 8 -
w=32: -m SPLIT 32 4 -
w=32: -m SPLIT 32 4 -r ALTMAP -
w=32: -m SPLIT 32 8 -
w=32: -m SPLIT 8 8 -
w=32: -m COMPOSITE 2 - -
w=32: -m COMPOSITE 2 - -r ALTMAP -
UNIX>
```

You'll note, this is on my old Macbook Pro, which doesn't support (PCLMUL), so "**CARRY_FREE**" is not included as an option. Now, let's run the unit tester on these to make sure they work, and to see their memory consumption:

```
UNIX> gf_methods 32 -B -U
../test/gf_unit 32 A -1 -
../test/gf_unit 32 A -1 -m GROUP 4 8 -
../test/gf_unit 32 A -1 -m SPLIT 32 4 -
../test/gf_unit 32 A -1 -m SPLIT 32 4 -r ALTMAP -
../test/gf_unit 32 A -1 -m SPLIT 32 8 -
../test/gf_unit 32 A -1 -m SPLIT 8 8 -
../test/gf_unit 32 A -1 -m COMPOSITE 2 - -
../test/gf_unit 32 A -1 -m COMPOSITE 2 - -r ALTMAP -
UNIX> gf_methods 32 -B -U | sh
Args: 32 A -1 - / size (bytes): 684
Args: 32 A -1 -m GROUP 4 8 - / size (bytes): 1296
Args: 32 A -1 -m SPLIT 32 4 - / size (bytes): 684
Args: 32 A -1 -m SPLIT 32 4 -r ALTMAP - / size (bytes): 684
Args: 32 A -1 -m SPLIT 32 8 - / size (bytes): 4268
Args: 32 A -1 -m SPLIT 8 8 - / size (bytes): 1839276
Args: 32 A -1 -m COMPOSITE 2 - - / size (bytes): 524648
Args: 32 A -1 -m COMPOSITE 2 - -r ALTMAP - / size (bytes): 524648
UNIX>
```

As anticipated, "**SPLIT 8 8**" consumes quite a bit of memory! Now, let's see how well they perform with both single multiplications and region multiplications:

```
UNIX> gf_methods 32 -B -M
sh time_tool.sh M 32 -
sh time_tool.sh M 32 -m GROUP 4 8 -
sh time_tool.sh M 32 -m SPLIT 32 4 -
sh time_tool.sh M 32 -m SPLIT 32 4 -r ALTMAP -
sh time_tool.sh M 32 -m SPLIT 32 8 -
sh time_tool.sh M 32 -m SPLIT 8 8 -
```

```
sh time_tool.sh M 32 -m COMPOSITE 2 - -
sh time_tool.sh M 32 -m COMPOSITE 2 - -r ALTMAP -
UNIX> gf_methods 32 -B -M | sh
M speed (MB/s):     5.90   W-Method: 32 -
M speed (MB/s):    14.09   W-Method: 32 -m GROUP 4 8 -
M speed (MB/s):     5.60   W-Method: 32 -m SPLIT 32 4 -
M speed (MB/s):     5.19   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
M speed (MB/s):     5.98   W-Method: 32 -m SPLIT 32 8 -
M speed (MB/s):    22.10   W-Method: 32 -m SPLIT 8 8 -
M speed (MB/s):    34.98   W-Method: 32 -m COMPOSITE 2 - -
M speed (MB/s):    34.16   W-Method: 32 -m COMPOSITE 2 - -r ALTMAP -
UNIX> gf_methods 32 -B -B | sh
Region Best (MB/s):  2746.76   W-Method: 32 -
Region Best (MB/s):   177.06   W-Method: 32 -m GROUP 4 8 -
Region Best (MB/s):  2818.75   W-Method: 32 -m SPLIT 32 4 -
Region Best (MB/s):  3818.21   W-Method: 32 -m SPLIT 32 4 -r ALTMAP -
Region Best (MB/s):   728.68   W-Method: 32 -m SPLIT 32 8 -
Region Best (MB/s):   730.97   W-Method: 32 -m SPLIT 8 8 -
Region Best (MB/s):   190.20   W-Method: 32 -m COMPOSITE 2 - -
Region Best (MB/s):  1837.99   W-Method: 32 -m COMPOSITE 2 - -r ALTMAP -
UNIX>
```

The default is quite a bit slower than the best performing methods for both single and region multiplication. So why are the defaults the way that they are? As detailed at the beginning of this chapter, we strive for lower memory consumption, so we don't use "**SPLIT 8 8**," which consumes 1.75 MB. We don't implement alternate fields by default, which is why we don't use "**COMPOSITE**." Finally, we don't implement alternate mappings of memory by default, which is why we don't use "**-m SPLIT 32 4 -r ALTMAP -**."

Of course, you may change these defaults if you please.

Test question: *Given the numbers above, it would appear that "***COMPOSITE***" yields the fastest performance of single multiplication, while "***SPLIT 32 4***" yields the fastest performance of region multiplication. Should I use two* **gf_t***'s in my application – one for single multiplication that uses "***COMPOSITE***," and one for region multiplication that uses "***SPLIT 32 4***?"*

The answer to this is "no." Why? Because composite fields are different from the "standard" fields, and if you mix these two **gf_t**'s, then you are using different fields for single multiplication and region multiplication. Please read section 7.2 for a little more information on this.

6.4 Calling gf_init_hard()

We recommend that you use **create_gf_from_argv()** instead of **gf_init_hard()**. However, there are extra things that you can do with **gf_init_hard()**. Here's the prototype:

```
int gf_init_hard(gf_t *gf,
                 int w,
                 int mult_type,
                 int region_type,
                 int divide_type,
                 uint64_t prim_poly,
                 int arg1,
                 int arg2,
```

```
                GFP base_gf,
                void *scratch_memory);
```

The arguments **mult_type**, **region_type** and **divide_type** allow for the same specifications as above, except the types are integer constants defined in **gf_complete.h**:

```
typedef enum {GF_MULT_DEFAULT,
              GF_MULT_SHIFT,
              GF_MULT_CARRY_FREE,
              GF_MULT_GROUP,
              GF_MULT_BYTWO_p,
              GF_MULT_BYTWO_b,
              GF_MULT_TABLE,
              GF_MULT_LOG_TABLE,
              GF_MULT_LOG_ZERO,
              GF_MULT_LOG_ZERO_EXT,
              GF_MULT_SPLIT_TABLE,
              GF_MULT_COMPOSITE } gf_mult_type_t;

#define GF_REGION_DEFAULT      (0x0)
#define GF_REGION_DOUBLE_TABLE (0x1)
#define GF_REGION_QUAD_TABLE   (0x2)
#define GF_REGION_LAZY         (0x4)
#define GF_REGION_SSE          (0x8)
#define GF_REGION_NOSSE        (0x10)
#define GF_REGION_ALTMAP       (0x20)
#define GF_REGION_CAUCHY       (0x40)

typedef enum { GF_DIVIDE_DEFAULT,
               GF_DIVIDE_MATRIX,
               GF_DIVIDE_EUCLID } gf_division_type_t;
```

You can mix the region types with bitwise or. The arguments to **GF_MULT_GROUP**, **GF_MULT_SPLIT_TABLE** and **GF_MULT_COMPOSITE** are specified in **arg1** and **arg2**. **GF_MULT_COMPOSITE** also takes a base field in **base_gf**. The base field is itself a **gf_t**, which should have been created previously with **create_gf_from_argv()**, **gf_init_easy()** or **gf_init_hard()**. Note that this **base_gf** has its own **base_gf** member and can be a composite field itself.

You can specify an alternate polynomial in **prim_poly**. For $w \leq 32$, the leftmost one (the one in bit position w) is optional. If you omit it, it will be added for you. For $w = 64$, there's no room for that one, so you have to leave it off. For $w = 128$, your polynomial can only use the bottom-most 64 bits. Fortunately, the standard polynomial only uses those bits. If you set **prim_poly** to zero, the library selects the "standard" polynomial.

Finally, **scratch_memory** is there in case you don't want **gf_init_hard()** to call **malloc()**. You may call **gf_scratch_size()** to find out how much extra memory each technique uses, and then you may pass it a pointer for it to use in **scratch_memory**. If you set **scratch_memory** to **NULL**, then the extra memory is allocated for you with **malloc()**. If you use **gf_init_easy()** or **create_gf_from_argv()**, or you use **gf_init_hard()** and set **scratch_memory** to **NULL**, then you should call **gf_free()** to free memory. If you use **gf_init_hard()** and use your own **scratch_memory** you can still call **gf_free()**, and it will not do anything.

Both **gf_init_hard()** and **gf_scratch_size()** return zero if the arguments don't specify a valid **gf_t**. When that happens, you can call **gf_error()** to print why the call failed.

We'll give you one example of calling **gf_init_hard()**. Suppose you want to make a **gf_init_hard()** call to be equivalent to "-m SPLIT 16 4 -r SSE -r ALTMAP -" and you want to allocate the scratch space yourself. Then you'd do the following:

```
gf_t gf;
void *scratch;
int size;

size = gf_scratch_size(16, GF_MULT_SPLIT_TABLE,
                           GF_REGION_SSE | GF_REGION_ALTMAP,
                           GF_DIVIDE_DEFAULT,
                           16, 4);
if (size == 0) { gf_error(); exit(1); } /* It failed. That shouldn't happen */
scratch = (void *) malloc(size);
if (scratch == NULL) { perror("malloc"); exit(1); }
if (!gf_init_hard(&gf, 16, GF_MULT_SPLIT_TABLE,
                           GF_REGION_SSE | GF_REGION_ALTMAP,
                           GF_DIVIDE_DEFAULT,
                           0, 16, 4, NULL, scratch)) {
  gf_error();
  exit(1);
}
```

6.5 gf_size()

You can call **gf_size(gf_t *gf)** to learn the memory consumption of the **gf_t**. It returns all memory consumed by the **gf_t**, including the **gf_t** itself, any scratch memory required by the **gf_t**, and the memory consumed by the sub-field if the field is "**COMPOSITE**." If you provided your own memory to **gf_init_hard()**, it does not report the size of this memory, but what the size should be, as determined by **gf_scratch_size()**. **gf_unit()** prints out the return value of **gf_size()** on the given field.

7 Further Information on Options and Algorithms

7.1 Inlining Single Multiplication and Division for Speed

Obviously, procedure calls are more expensive than single instructions, and the mechanics of multiplication in "**TABLE**" and "**LOG**" are pretty simple. For that reason, we support inlining for "**TABLE**" when $w = 4$ and $w = 8$, and for "**LOG**" when $w = 16$. We elaborate below.

When $w = 4$, you may inline multiplication and division as follows. The following procedures return pointers to the multiplication and division tables respectively:

```
uint8_t *gf_w4_get_mult_table(gf_t * gf);
uint8_t *gf_w4_get_div_table(gf_t * gf);
```

The macro **GF_W4_INLINE_MULTDIV**($table$, a, b) then multiplies or divides a by b using the given $table$. This of course only works if the multiplication technique is "**TABLE**," which is the default for $w = 4$. If the multiplication technique is not "**TABLE**," then **gf_w4_get_mult_table()** will return **NULL**.

When $w = 8$, the procedures **gf_w8_get_mult_table()** and **gf_w8_get_div_table()**, and the macro **GF_W8_INLINE_MULTDIV**($table$, a, b) work identically to the $w = 4$ case.

When $w = 16$, the following procedures return pointers to the logarithm table, and the two inverse logarithm tables respectively:

```
uint16_t *gf_w16_get_log_table(gf_t * gf);
uint16_t *gf_w16_get_mult_alog_table(gf_t * gf);
uint16_t *gf_w16_get_div_alog_table(gf_t * gf);
```

The first inverse logarithm table works for multiplication, and the second works for division. They actually point to the same table, but to different places in the table. You may then use the macro **GF_W16_INLINE_MULT**(log, $alog$, a, b) to multiply a and b, and the macro **GF_W16_INLINE_DIV**(log, $alog$, a, b) to divide a and b. Make sure you use the $alog$ table returned by **gf_w16_get_mult_alog_table()** for multiplication and the one returned by **gf_w16_get_div_alog_table()** for division. Here are some timings:

```
UNIX> gf_time 4 M 0 10240 10240 -
Seed: 0
      Multiply:             0.228860 s   Mops:    100.000       436.949 Mega-ops/s
UNIX> gf_inline_time 4 0 10240 10240
Seed: 0
Inline mult:     0.096859 s   Mops:    100.000      1032.424 Mega-ops/s
UNIX> gf_time 8 M 0 10240 10240 -
Seed: 0
      Multiply:             0.228931 s   Mops:    100.000       436.812 Mega-ops/s
UNIX> gf_inline_time 8 0 10240 10240
Seed: 0
Inline mult:     0.114300 s   Mops:    100.000       874.889 Mega-ops/s
UNIX> gf_time 16 M 0 10240 10240 -
Seed: 0
      Multiply:             0.193626 s   Mops:     50.000       258.229 Mega-ops/s
UNIX> gf_inline_time 16 0 10240 10240
Seed: 0
Inline mult:     0.310229 s   Mops:    100.000       322.342 Mega-ops/s
UNIX>
```

7.2 Using different techniques for single and region multiplication

You may want to "mix and match" the techniques. For example, suppose you'd like to use "-m SPLIT 8 8" for **multiply()** in $GF(2^{32})$, because it's fast, and you don't mind consuming all of that space for tables. However, for **multiply_region()**, you'd like to use "-m SPLIT 32 4 -r ALTMAP," because that's the fastest way to implement **multiply_region()**. Unfortunately, There is no way to create a **gf_t** that does this combination. In this case, you should simply create two **gf_t**'s, and use one for **multiply()** and the other for **multiply_region()**. All of the implementations may be used interchangably with the following exceptions:

- "**COMPOSITE**" implements a different Galois Field.
- If you change a field's polynomial, then the resulting Galois Field will be different.

- If you are using "**ALTMAP**" to multiply regions, then the contents of the resulting regions of memory will depend on the multiplication technique, the size of the region and its alignment. Please see section 7.9 for a detailed explanation of this.
- If you are using "**CAUCHY**" to multiply regions, then like "**ALTMAP**," the contents of the result regions of memory the multiplication technique and the size of the region. You don't have to worry about alignment.

7.3 General w

The library supports Galois Field arithmetic with $2 < w \leq 32$. Values of w which are not whole number powers of 2 are handled by the functions in **gf_wgen.c** . For these values of w, the available multiplication types are "**SHIFT**," "**BYTWO_p**," "**BYTWO_b**," "**GROUP**," "**TABLE**" and "**LOG**." "**LOG**" is only valid for $w < 28$ and "**TABLE**" is only valid for $w < 15$. The defaults for these values of w are "**TABLE**" for $w < 8$, "**LOG**" for $w < 16$, and "**BYTWO_p**" for $w < 32$.

7.4 Arguments to "SPLIT"

The "**SPLIT**" technique is based on the distributive property of multiplication and addition:

$$a * (b + c) = (a * b) + (a * c).$$

This property allows us to, for example, split an eight bit word into two four-bit components and calculate the product by performing two table lookups in 16-element tables on each of the compoents, and adding the result. There is much more information on "**SPLIT**" in The Paper. Here we describe the version of "**SPLIT**" implemented in GF-Complete.

"**SPLIT**" takes two arguments, which are the number of bits in each component of a, which we call w_a, and the number of bits in each component of b, which we call w_b. If the two differ, it does not matter which is bigger – the library recognizes this and performs the correct implementation. The legal values of w_a and w_b fall into five categories:

1. w_a is equal to w and w_b is equal to four. In this case, b is broken up into $\frac{w}{4}$ four-bit words which are used in 16-element lookup tables. The tables are created on demand in **multiply_region()** and the SSSE3 instruction **mm_shuffle_epi8()** is leveraged to perform 16 lookups in parallel. Thus, these are very fast implementations. When $w \geq 16$, you should combine this with "**ALTMAP**" to get the best performance (see The Paper or [PGM13b] for explanation). If you do this please see section 7.9 for information about "**ALTMAP**" and alignment.

   If you don't use "**ALTMAP**," the implementations for $w \in \{16, 32, 64\}$ convert the standard representation into "**ALTMAP**," perform the multiplication with "**ALTMAP**" and then convert back to the standard representation. The performance difference using "**ALTMAP**" can be significant:

`gf_time 16 G 0 1048576 100 -m SPLIT 16 4 -`	Speed = 8,389 MB/s
`gf_time 16 G 0 1048576 100 -m SPLIT 16 4 -r ALTMAP -`	Speed = 9,212 MB/s
`gf_time 32 G 0 1048576 100 -m SPLIT 32 4 -`	Speed = 5,304 MB/s
`gf_time 32 G 0 1048576 100 -m SPLIT 32 4 -r ALTMAP -`	Speed = 7,146 MB/s
`gf_time 64 G 0 1048576 100 -m SPLIT 64 4 -`	Speed = 2,595 MB/s
`gf_time 64 G 0 1048576 100 -m SPLIT 64 4 -r ALTMAP -`	Speed = 3,436 MB/s

2. w_a is equal to w and w_b is equal to eight. Now, b is broken into bytes, each of these is used in its own 256-element lookup table. This is typically the best way to perform **multiply_region()** without SSE.

   Because this is a region optimization, when you specify these options, you get a default **multiply()** — see Table 1 for a listing of the defaults. See section 7.2 for using a different **multiply()** than the defaults.

3. w_a is equal to w and w_b is equal to 16. This is only valid for $w = 32$ and $w = 64$. Now, b is broken into shorts, each of these is used in its own 64K-element lookup table. This is typically slower than when w_b equals 8, and requires more amortization (larger buffer sizes) to be effective.

4. w_a and w_b are both equal to eight. Now both a and b are broken into bytes, and the products of the various bytes are looked up in multiple 256×256 tables. In $GF(2^{16})$, there are three of these tables. In $GF(2^{32})$, there are seven, and in $GF(2^{64})$ there are fifteen. Thus, this implementation can be a space hog. However, for $w = 32$, this is the fastest way to perform **multiply()** on some machines.

   When this option is employed, **multiply_region()** is implemented in an identical fashion to when $w_a = w$ and $w_b = 8$.

5. $w_a = 32$ and $w_b = 2$. (w = 32 only). I was playing with a different way to use **mm_shuffle_epi8()**. It works, but it's slower than when $w_b = 4$.

7.5 Arguments to "GROUP"

The "**GROUP**" multiplication option takes two arguments, g_s and g_r. It implements multiplication in the same manner as "**SHIFT**," except it uses a table of size 2^{g_s} to perform g_s shifts at a time, and a table of size 2^{g_r} to perform g_r reductions at at time. The program **gf_methods** only prints the options 4 4 and 4 8 as arguments for "**GROUP**." However, other values of g_s and g_r are legal and sometimes desirable:

- For $w \leq 32$ and $w = 64$, any values of g_s and g_r may be used, so long as they are less than or equal to w and so long as the tables fit into memory. There are four exceptions to this, listed below.
- For $w = 4$, "**GROUP**" is not supported.
- For $w = 8$, "**GROUP**" is not supported.
- For $w = 16$, "**GROUP**" is only supported for $g_s = g_r = 4$.
- For $w = 128$ "**GROUP**" only supports $g_s = 4$ and $g_r \in \{4, 8, 16\}$.

The way that g_s and g_r impact performance is as follows. The "**SHIFT**" implementation works by performing a carry-free multiplication in w steps, and then performing reduction in w steps. In "**GROUP**," the carry-free multiplication is reduced to $\lceil \frac{w}{g_s} \rceil$ steps, and the reduction is reduced to $\lceil \frac{w}{g_r} \rceil$. Both require tables. The table for the carry-free multiplication must be created at the beginning of each **multiply()** or **multiply_region()**, while the table for reduction is created when the **gf_t** is initialized. For that reason, it makes sense for g_r to be bigger than g_s.

To give a flavor for the impact of these arguments, Figure 3 shows the performance of varying g_s and g_r for single multiplication and region multiplication respectively, in $GF(2^{32})$ and $GF(2^{64})$. As the graphs demonstrate, **multiply()** performs better with smaller values of g_s, while **multiply_region()** amortizes the creation of the shifting table, and can tolerate larger values of g_s. When g_s equals g_r, there are some optimizations that we hand-encode. These can be seen clearly in the **multiply_region()** graphs.

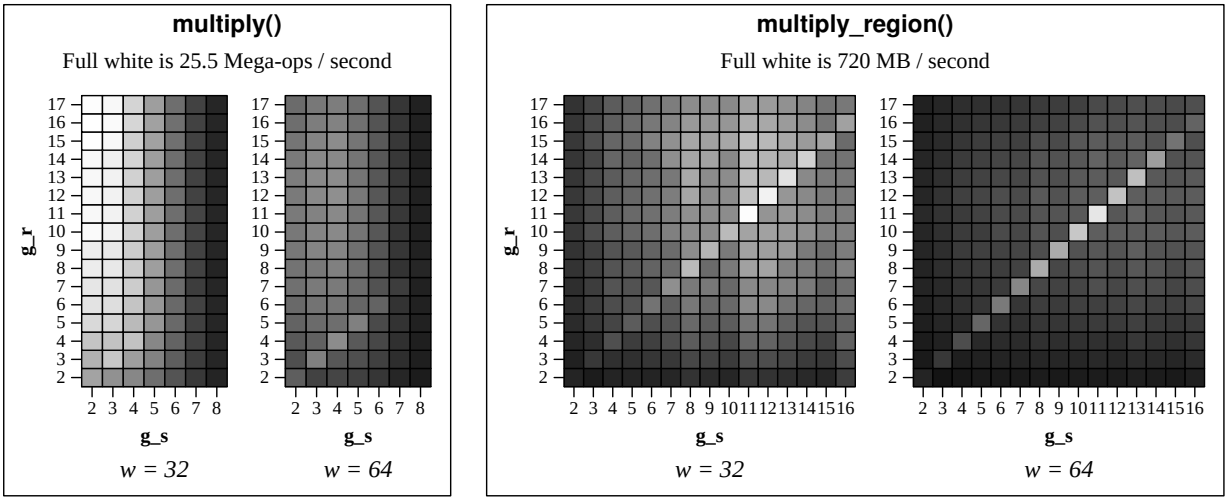

Figure 3: The performance of **multiply()** and **multiply_region()** using "**GROUP**," and varying the arguments g_s and g_r. All graphs are heat maps with black equaling zero. The region size is 100KB.

7.6 Considerations with "COMPOSITE"

As mentioned above, using "**ALTMAP**" with "**COMPOSITE**" allows **multiply_region()** to recursively call **multiply_region()**, rather than simply calling **multiply()** on every word in the region. The difference can be pronounced:

`gf_time 32 G 0 10240 10240 -m COMPOSITE 2 - -` Speed = 322 MB/s
`gf_time 32 G 0 10240 10240 -m COMPOSITE 2 - -r ALTMAP -` Speed = 3,368 MB/s
`gf_time 32 G 0 10240 10240 -m COMPOSITE 2 -m SPLIT 16 4 -r ALTMAP - -r ALTMAP -` Speed = 3,925 MB/s

There is support for performing **multiply()** inline for the "**TABLE**" implementations for $w \in \{4, 8\}$ and for the "**LOG**" implementation for $w = 16$ (see section 7.1). These are leveraged by **multiply()** in "**COMPOSITE**," and by **multiply_region()** if you are not using "**ALTMAP**." To demonstrate this, in the table below, you can see that the performance of **multiply()** with "**SPLIT 8 4**" is 88 percent as fast than the default in $w = 8$ (which is "**TABLE**"). When you use each as a base field for "**COMPOSITE**" with $w = 16$, the one with "**SPLIT 8 4**" is now just 37 percent as fast. The difference is the inlining of multiplication in the base field when "**TABLE**" is employed:

`gf_time 8 M 0 1048576 100 -`	Speed = 501 Mega-ops/s
`gf_time 8 M 0 1048576 100 -m SPLIT 8 4 -`	Speed = 439 Mega-ops/s
`gf_time 8 M 0 1048576 100 -m COMPOSITE 2 - -`	Speed = 207 Mega-ops/s
`gf_time 8 M 0 1048576 100 -m COMPOSITE 2 -m SPLIT 8 4 - -`	Speed = 77 Mega-ops/s

You can keep making recursive definitions of composites field if you want. For example, this one's not too slow for region operations (641 MB/s):

```
gf_time 128 G 0 1048576 100 -m COMPOSITE 2 -m COMPOSITE 2 -m COMPOSITE 2
      -m SPLIT 16 4 -r ALTMAP - -r ALTMAP - -r ALTMAP - -r ALTMAP -
```

Please see section 7.8.1 for a discussion of polynomials in composite fields.

7.7 "CARRY_FREE" and the Primitive Polynomial

If your machine supports the PCLMUL instruction, then we leverage that in "**CARRY_FREE**." This implementation first performs a carry free multiplication of two w-bit numbers, which yields a $2w$-bit number. It does this with one PCLMUL instruction. To reduce the $2w$-bit number back to a w-bit number requires some manipulation of the polynomial. As it turns out, if the polynomial has a lot of contiguous zeroes following its leftmost one, the number of reduction steps may be minimized. For example, with $w = 32$, we employ the polynomial 0x100400007, because that is what other libraries employ. This only has 9 contiguous zeros following the one, which means that the reduction takes four steps. If we instead use 0x1000000c5, which has 24 contiguous zeros, the reduction takes just two steps. You can see the difference in performance:

`gf_time 32 M 0 1048576 100 -m CARRY_FREE -`	Speed = 48 Mega-ops/s
`gf_time 32 M 0 1048576 100 -m CARRY_FREE -p 0xc5 -`	Speed = 81 Mega-ops/s

This is relevant for $w = 16$ and $w = 32$, where the "standard" polynomials are sub-optimal with respect to "**CARRY_FREE**." For $w = 16$, the polynomial 0x1002d has the desired property. It's less important, of course, with $w = 16$, because "**LOG**" is so much faster than "**CARRY_FREE**."

7.8 More on Primitive Polynomials

7.8.1 Primitive Polynomials that are not Primitive

The library is willing to work with most polynomials, even if they are not primitive or irreducible. For example, the polynomial $x^4 + x^3 + x^2 + x + 1$ is irreducible, and therefore generates a valid Galois Field for $GF(2^4)$. However, it is not primitive, because $2^5 = 1$. For that reason, if you use this polynomial, you cannot use the "**LOG**" method. The other methods will work fine:

```
UNIX> gf_mult 2 2 4 -p 0xf -
4
UNIX> gf_mult 4 2 4 -p 0xf -
8
UNIX> gf_mult 8 2 4 -p 0xf -
15
UNIX> gf_mult 15 2 4 -p 0xf -
1
UNIX> gf_div 1 15 4 -p 0xf -
2
UNIX> gf_div 1 15 4 -p 0xf -m LOG -
usage: gf_div a b w [method] - does division of a and b in GF(2^w)
Bad Method Specification: Cannot use Log tables because the polynomial is not primitive.
UNIX>
```

If a polynomial is reducible, then it does not define a Galois Field, but instead a ring. GF-Complete attempts to work here where it can; however certain parts of the library will not work:

1. Division is a best effort service. The problem is that often quotients are not unique. If **divide()** returns a non-zero number, then that number will be a valid quotient, but it may be one of many. If the multiplication technique is "**TABLE**," then if a quotient exists, one is returned. Otherwise, zero is returned. Here are some examples – the polynomial $x^4 + 1$ is reducible, and therefore produces a ring. Below, we see that with this polynomal, 1*6 = 6 and 14*6 = 6. Therefore, $\frac{6}{6}$ has two valid quotients: 1 and 14. GF-Complete returns 14 as the quotient:

```
UNIX> gf_mult 1 6 4 -p 0x1 -
6
UNIX> gf_mult 14 6 4 -p 0x1 -
6
UNIX> gf_div 6 6 4 -p 0x1 -
14
UNIX>
```

2. When "**EUCLID**" is employed for division, it uses the extended Euclidean algorithm for GCD to find a number's inverse, and then it multiplies by the inverse. The problem is that not all numbers in a ring have inverses. For example, in the above ring, there is no number a such that $6a = 1$. Thus, 6 has no inverse. This means that even though $\frac{6}{6}$ has quotients in this ring, "**EUCLID**" will fail on it because it is unable to find the inverse of 6. It will return 0:

```
UNIX> gf_div 6 6 4 -p 0x1 -m TABLE -d EUCLID -
0
UNIX>
```

3. Inverses only work if a number has an inverse. Inverses may not be unique.
4. "**LOG**" will not work. In cases where the default would be "**LOG**," "**SHIFT**" is used instead.

Due to problems with division, **gf_unit** may fail on a reducible polynomial. If you are determined to use such a polynomial, don't let this error discourage you.

7.8.2 Default Polynomials for Composite Fields

GF-Complete will successfully select a default polynomial in the following composite fields:

- $w = 8$ and the default polynomial (0x13) is employed for $GF(2^4)$.
- $w = 16$ and the default polynomial (0x11d) is employed for $GF(2^8)$.
- $w = 32$ and the default polynomial (0x1100b) is employed for $GF(2^{16})$.
- $w = 32$ and 0x1002d is employed for $GF(2^{16})$.
- $w = 32$ and the base field for $GF(w^{16})$ is a composite field that uses a default polynomial.
- $w = 64$ and the default polynomial (0x100400007) is employed for $GF(2^{32})$.
- $w = 64$ and 0x1000000c5 is employed for $GF(2^{32})$.
- $w = 64$ and the base field for $GF(w^{32})$ is a composite field that uses a default polynomial.
- $w = 128$ and the default polynomial (0x1b) is employed for $GF(2^{64})$.
- $w = 128$ and the base field for $GF(w^{64})$ is a composite field that uses a default polynomial.

7.8.3 The Program gf_poly for Verifying Irreducibility of Polynomials

The program **gf_poly** uses the Ben-Or algorithm [GP97] to determine whether a polynomial with coefficients in $GF(2^w)$ is reducible. Its syntax is:

```
gf_poly w method power:coef power:coef ...
```

You can use it to test for irreducible polynomials with binary coefficients by specifying $w = 1$. For example, from the discussion above, we know that $x^4 + x + 1$ and $x^4 + x^3 + x^2 + x + 1$ are both irreducible, but $x^4 + 1$ is reducible. **gf_poly** confirms:

```
UNIX> gf_poly 1 - 4:1 1:1 0:1
Poly: x^4 + x + 1
Irreducible.
UNIX> gf_poly 1 - 4:1 3:1 2:1 1:1 0:1
Poly: x^4 + x^3 + x^2 + x + 1
Irreducible.
UNIX> gf_poly 1 - 4:1 0:1
Poly: x^4 + 1
Reducible.
UNIX>
```

For composite fields $GF((2^l)^2)$, we are looking for a value s such that $x^2 + sx + 1$ is irreducible. That value depends on the base field. For example, for the default field $GF(2^{32})$, a value of $s = 2$ makes the polynomial irreducible. However, if the polynomial 0xc5 is used (so that PCLMUL is fast – see section 7.7), then $s = 2$ yields a reducible polynomial, but $s = 3$ yields an irreducible one. You can use **gf_poly** to help verify these things, and to help define s if you need to stray from the defaults:

```
UNIX> gf_poly 32 - 2:1 1:2 0:1
Poly: x^2 + (0x2)x + 1
Irreducible.
UNIX> gf_poly 32 -p 0xc5 - 2:1 1:2 0:1
Poly: x^2 + (0x2)x + 1
Reducible.
UNIX> gf_poly 32 -p 0xc5 - 2:1 1:3 0:1
Poly: x^2 + (0x3)x + 1
Irreducible.
UNIX>
```

gf_unit does random sampling to test for problems. In particular, it chooses a random a and a random b, multiplies them, and then tests the result by dividing it by a and b. When w is large, this sampling does not come close to providing complete coverage to check for problems. In particular, if the polynomial is reducible, there is a good chance that **gf_unit** won't discover any problems. For example, the following **gf_unit** call does not flag any problems, even though the polynomial is reducible.

```
UNIX> gf_unit 64 A 0 -m COMPOSITE 2 -p 0xc5 - -p 2 -
UNIX>
```

How can we demonstrate that this particular field has a problem? Well, when the polynomial is 0xc5, we can factor $x^2 + 2x + 1$ as $(x + 0\text{x}7\text{f}6\text{f}95\text{f}9)(x + 0\text{x}7\text{f}6\text{f}95\text{fb})$. Thus, in the composite field, when we multiply 0x17f6f95f9 by 0x17f6f95fb, we get zero. That's the problem:

```
UNIX> gf_mult 7f6f95f9 7f6f95fb 32h -p 0xc5 -
1
UNIX> gf_mult 17f6f95f9 17f6f95fb 64h -m COMPOSITE 2 -p 0xc5 - -p 2 -
0
UNIX>
```

7.9 "ALTMAP" considerations and extract_word()

There are two times when you may employ alternate memory mappings:

1. When using "**SPLIT**" and $w_b = 4$.
2. When using "**COMPOSITE**."

Additionally, by default, the "**CAUCHY**" region option also employs an alternate memory mapping.

When you use alternate memory mappings, the exact mapping of words in $GF(2^w)$ to memory depends on the situation, the size of the region, and the alignment of the pointers. To help you figure things out, we have included the procedures **extract_word.w*xx*()** as part of the **gf_t** struct. This procedure takes four parameters:

- A pointer to the **gf_t**.
- The beginning of the memory region.
- The number of bytes in the memory region.
- The desired word number: n.

It then returns the n-th word in memory. When the standard mapping is employed, this simply returns the n-th contiguous word in memory. With alternate mappings, each word may be split over several memory regions, so **extract_word()** grabs the relevant parts of each memory region to extract the word. Below, we go over each of the above situations in detail. Please refer to Figure 2 in Section 5 for reference.

7.9.1 Alternate mappings with "SPLIT"

The alternate mapping with "**SPLIT**" is employed so that we can best leverage **mm_shuffle_epi8()**. Please read [PGM13b] for details as to why. Consider an example when $w = 16$. In the main region of memory (the middle region in Figure 2), multiplication proceeds in units of 32 bytes, which are each broken into two 16-byte regions. The first region holds the high bytes of each word in $GF(2^{16})$, and the second region holds the low bytes.

Let's look at a very detailed example, from **gf_example_5.c**. This program makes the following call, where **gf** has been initialized for $w = 16$, using "**SPLIT**" and "**ALTMAP**:"

```
gf.multiply_region.w32(&gf, a, b, 0x1234, 30*2, 0);
```

In other words, it is multiplying a region a of 60 bytes (30 words) by the constant 0x1234 in $GF(2^{16})$, and placing the result into b. The pointers a and b have been set up so that they are not multiples of 16. The first line of output prints a and b:

```
a: 0x10010008c    b: 0x10010015c
```

As described in Section 5, the regions of memory are split into three parts:

1. 4 bytes starting at 0x1001008c / 0x10010015c.
2. 32 bytes starting at 0x10010090 / 0x100100160.
3. 24 bytes starting at 0x100100b0 / 0x100100180.

In the first and third parts, the bytes are laid out according to the standard mapping. However, the second part is split into two 16-byte regions — one that holds the high bytes of each word and one that holds the low bytes. To help illustrate, the remainder of the output prints the 30 words of a and b as they appear in memory, and then the 30 return values of **extract_word.w32()**:

```
      0    1    2    3    4    5    6    7    8    9
a: 640b 07e5 2fba ce5d f1f9 3ab8 c518 1d97 45a7 0160
b: 1ba3 644e 84f8 be3c 4318 4905 b2fb 46eb ef01 a503


     10   11   12   13   14   15   16   17   18   19
a: 3759 b107 9660 3fde b3ea 8a53 75ff 46dc c504 72c2
b: da27 e166 a0d2 b3a2 1699 3a3e 47fb 39af 1314 8e76


     20   21   22   23   24   25   26   27   28   29
a: b469 1b97 e91d 1dbc 131e 47e0 c11a 7f07 76e0 fe86
b: 937c a5db 01b7 7f5f 8974 05e1 cff3 a09c de3c 4ac0

Word  0: 0x640b * 0x1234 = 0x1ba3    Word 15: 0x4575 * 0x1234 = 0xef47
Word  1: 0x07e5 * 0x1234 = 0x644e    Word 16: 0x60dc * 0x1234 = 0x03af
Word  2: 0xba59 * 0x1234 = 0xf827    Word 17: 0x0146 * 0x1234 = 0xa539
Word  3: 0x2f37 * 0x1234 = 0x84da    Word 18: 0xc504 * 0x1234 = 0x1314
Word  4: 0x5d07 * 0x1234 = 0x3c66    Word 19: 0x72c2 * 0x1234 = 0x8e76
Word  5: 0xceb1 * 0x1234 = 0xbee1    Word 20: 0xb469 * 0x1234 = 0x937c
Word  6: 0xf960 * 0x1234 = 0x18d2    Word 21: 0x1b97 * 0x1234 = 0xa5db
Word  7: 0xf196 * 0x1234 = 0x43a0    Word 22: 0xe91d * 0x1234 = 0x01b7
Word  8: 0xb8de * 0x1234 = 0x05a2    Word 23: 0x1dbc * 0x1234 = 0x7f5f
Word  9: 0x3a3f * 0x1234 = 0x49b3    Word 24: 0x131e * 0x1234 = 0x8974
Word 10: 0x18ea * 0x1234 = 0xfb99    Word 25: 0x47e0 * 0x1234 = 0x05e1
Word 11: 0xc5b3 * 0x1234 = 0xb216    Word 26: 0xc11a * 0x1234 = 0xcff3
Word 12: 0x9753 * 0x1234 = 0xeb3e    Word 27: 0x7f07 * 0x1234 = 0xa09c
Word 13: 0x1d8a * 0x1234 = 0x463a    Word 28: 0x76e0 * 0x1234 = 0xde3c
Word 14: 0xa7ff * 0x1234 = 0x01fb    Word 29: 0xfe86 * 0x1234 = 0x4ac0
```

In the first region are words 0 and 1, which are identical to how they appear in memory: 0x640b and 0x07e5. In the second region are words 2 through 17. These words are split among the two sixteen-byte regions. For example, word 2, which **extract_word()** reports is 0xba59, is constructed from the low byte in word 2 (0xba) and the low byte in word 10 (0x59). Since 0xba59 * 0x1234 = 0xf827, we see that the low byte in word 2 of b is 0xf8, and the low byte in word 10 is 0x27.

When we reach word 22, we are in the third region of memory, and words are once again identical to how they appear in memory.

While this is confusing, we stress that that so long as you call **multiply_region()** with pointers of the same alignment and regions of the same size, your results with **ALTMAP** will be consistent. If you call it with pointers of

different alignments, or with different region sizes, then the results will not be consistent. To reiterate, if you don't use **ALTMAP**, you don't have to worry about any of this – words will always be laid out contiguously in memory.

When $w = 32$, the middle region is a multiple of 64, and each word in the middle region is broken into bytes, each of which is in a different 16-byte region. When $w = 64$, the middle region is a multiple of 128, and each word is stored in eight 16-byte regions. And finally, when $w = 128$, the middle region is a multiple of 128, and each word is stored in 16 16-byte regions.

7.9.2 Alternate mappings with "COMPOSITE"

With "**COMPOSITE**," the alternate mapping divides the middle region in half. The lower half of each word is stored in the first half of the middle region, and the higher half is stored in the second half. To illustrate, **gf_example_6** performs the same example as **gf_example_5**, except it is using "**COMPOSITE**" in $GF((2^{16})^2)$, and it is multiplying a region of 120 bytes rather than 60. As before, the pointers are not aligned on 16-bit quantities, so the region is broken into three regions of 4 bytes, 96 bytes, and 20 bytes. In the first and third region, each consecutive four byte word is a word in $GF(2^{32})$. For example, word 0 is 0x562c640b, and word 25 is 0x46bc47e0. In the middle region, the low two bytes of each word come from the first half, and the high two bytes come from the second half. For example, word 1 as reported by **extract_word()** is composed of the lower two bytes of word 1 of memory (0x07e5), and the lower two bytes of word 13 (0x3fde). The product of 0x3fde07e5 and 0x12345678 is 0x211c880d, which is stored in the lower two bytes of words 1 and 13 of b.

```
a: 0x10010011c    b: 0x1001001ec

          0        1        2        3        4        5        6        7        8        9
a: 562c640b 959407e5 56592fba cbadce5d 1d1cf1f9 35d73ab8 6493c518 b37c1d97 8e4545a7 c0d80160
b: f589f36c f146880d 74f7b349 7ea7c5c6 34827c1a 93cc3746 bfd9288b 763941d1 bcd33a5d da695e64


         10       11       12       13       14       15       16       17       18       19
a: 965b3759 cb3eb107 1b129660 95a33fde 95a7b3ea d16c8a53 153375ff f74646dc 35aac504 98f972c2
b: fd70f125 3274fa8f d9dd34ee c01a211c d4402403 8b55c08b da45f0ad 90992e18 b65e0902 d91069b5


         20       21       22       23       24       25       26       27       28       29
a: 5509b469 7f8a1b97 3472e91d 9ee71dbc de4e131e 46bc47e0 5bc9c11a 931d7f07 d40676e0 c85cfe86
b: fc92b8f5 edd59668 b4bc0d90 a679e4ce 1a98f7d0 6038765f b2ff333f e7937e49 fa5a5867 79c00ea2

Word  0: 0x562c640b * 0x12345678 = 0xf589f36c    Word 15: 0xb46945a7 * 0x12345678 = 0xb8f53a5d
Word  1: 0x3fde07e5 * 0x12345678 = 0x211c880d    Word 16: 0x55098e45 * 0x12345678 = 0xfc92bcd3
Word  2: 0x95a39594 * 0x12345678 = 0xc01af146    Word 17: 0x1b970160 * 0x12345678 = 0x96685e64
Word  3: 0xb3ea2fba * 0x12345678 = 0x2403b349    Word 18: 0x7f8ac0d8 * 0x12345678 = 0xedd5da69
Word  4: 0x95a75659 * 0x12345678 = 0xd44074f7    Word 19: 0xe91d3759 * 0x12345678 = 0x0d90f125
Word  5: 0x8a53ce5d * 0x12345678 = 0xc08bc5c6    Word 20: 0x3472965b * 0x12345678 = 0xb4bcfd70
Word  6: 0xd16ccbad * 0x12345678 = 0x8b557ea7    Word 21: 0x1dbcb107 * 0x12345678 = 0xe4cefa8f
Word  7: 0x75fff1f9 * 0x12345678 = 0xf0ad7c1a    Word 22: 0x9ee7cb3e * 0x12345678 = 0xa6793274
Word  8: 0x15331d1c * 0x12345678 = 0xda453482    Word 23: 0x131e9660 * 0x12345678 = 0xf7d034ee
Word  9: 0x46dc3ab8 * 0x12345678 = 0x2e183746    Word 24: 0xde4e1b12 * 0x12345678 = 0x1a98d9dd
Word 10: 0xf74635d7 * 0x12345678 = 0x909993cc    Word 25: 0x46bc47e0 * 0x12345678 = 0x6038765f
Word 11: 0xc504c518 * 0x12345678 = 0x0902288b    Word 26: 0x5bc9c11a * 0x12345678 = 0xb2ff333f
Word 12: 0x35aa6493 * 0x12345678 = 0xb65ebfd9    Word 27: 0x931d7f07 * 0x12345678 = 0xe7937e49
```

```
Word 13: 0x72c21d97 * 0x12345678 = 0x69b541d1    Word 28: 0xd40676e0 * 0x12345678 = 0xfa5a5867
Word 14: 0x98f9b37c * 0x12345678 = 0xd9107639    Word 29: 0xc85cfe86 * 0x12345678 = 0x79c00ea2
```

As with "**SPLIT**," using **multiply_region()** with "**COMPOSITE**" and "**ALTMAP**" will be consistent only if the alignment of pointers and region sizes are identical.

7.9.3 The mapping of "CAUCHY"

With "**CAUCHY**," the region is partitioned into w subregions, and each word in the region is broken into w bits, each of which is stored in a different subregion. To illustrate, **gf_example_7** multiplies a region of three bytes by 5 in $GF(2^3)$ using "**CAUCHY**:"

```
UNIX> gf_example_7
a: 0x100100190    b: 0x1001001a0

a: 0x0b 0xe5 0xba
b: 0xee 0xba 0x0b

a bits: 00001011 11100101 10111010
b bits: 11101110 10111010 00001011

Word  0: 3 * 5 = 4
Word  1: 5 * 5 = 7
Word  2: 2 * 5 = 1
Word  3: 5 * 5 = 7
Word  4: 4 * 5 = 2
Word  5: 6 * 5 = 3
Word  6: 2 * 5 = 1
Word  7: 6 * 5 = 3
UNIX>
```

The program prints the three bytes of a and b in hexadecimal and in binary. To see how words are broken up, consider word 0, which is the lowest bit of each of the three bytes of a (and b). These are the bits 1, 1 and 0 in a, and 0, 0, and 1 in b. Accordingly, the word is 3 in a, and 3*5 = 4 in b. Similarly, word 7 is the high bit in each byte: 0, 1, 1 (6) in a, and 1, 1, 0 (3) in b.

With "**CAUCHY**," **multiply_region()** may be implemented exclusively with XOR operations. Please see [BKK+95] for more information on the motivation behind "**CAUCHY**."

8 Thread Safety

Once you initialize a **gf_t**, you may use it wontonly in multiple threads for all operations except for the ones below. With the implementations listed below, the scratch space in the **gf_t** is used for temporary tables, and therefore you cannot call **region_multiply**, and in some cases **multiply** from multiple threads because they will overwrite each others' tables. In these cases, if you want to call the procedures from multiple threads, you should allocate a separate **gf_t** for each thread:

- All "**GROUP**" implementations are not thread safe for either **region_multiply()** or **multiply()**. Other than "**GROUP**," **multiply()** is always thread-safe.

- For $w = 4$, **region_multiply.w32()** is unsafe in in "-m TABLE -r QUAD -r LAZY."
- For $w = 8$, **region_multiply.w32()** is unsafe in in "-m TABLE -r DOUBLE -r LAZY."
- For $w = 16$, **region_multiply.w32()** is unsafe in in "-m TABLE."
- For $w \in \{32, 64, 128\}$, all "**SPLIT**" implementations are unsafe for **region_multiply()**. This means that if the default uses "**SPLIT**" (see Table 1 for when that occurs), then **region_multiply()** is not thread safe.
- The "**COMPOSITE**" operations are only safe if the implementations of the underlying fields are safe.

9 Listing of Procedures

The following is an alphabetical listing of the procedures, data types and global variables for users to employ in GF-complete.

- **GF_W16_INLINE_DIV()** in **gf_complete.h**: This is a macro for inline division when $w = 16$. See section 7.1.
- **GF_W16_INLINE_MULT()** in **gf_complete.h**: This is a macro for inline multiplication when $w = 16$. See section 7.1.
- **GF_W4_INLINE_MULTDIV()** in **gf_complete.h**: This is a macro for inline multiplication/division when $w = 4$. See section 7.1.
- **GF_W8_INLINE_MULTDIV()** in **gf_complete.h**: This is a macro for inline multiplication/division when $w = 8$. See section 7.1.
- **MOA_Fill_Random_Region()** in **gf_rand.h**: Fills a region with random numbers.
- **MOA_Random_128()** in **gf_rand.h**: Creates a random 128-bit number.
- **MOA_Random_32()** in **gf_rand.h**: Creates a random 32-bit number.
- **MOA_Random_64()** in **gf_rand.h**: Creates a random 64-bit number.
- **MOA_Random_W()** in **gf_rand.h**: Creates a random w-bit number, where $w \leq 32$.
- **MOA_Seed()** in **gf_rand.h**: Sets the seed for the random number generator.
- **_gf_errno** in **gf_complete.h**: This is to help figure out why an initialization call failed. See section 6.1.
- **gf_create_gf_from_argv()** in **gf_method.h**: Creates a **gf_t** using C style argc/argv. See section 6.1.1.
- **gf_division_type_t** in **gf_complete.h**: the different ways to specify division when using **gf_init_hard()**. See section 6.4.
- **gf_error()** in **gf_complete.h**: This prints out why an initialization call failed. See section 6.1.
- **gf_extract** in **gf_complete.h**: This is the data type of **extract_word()** in a **gf_t**. See section 7.9 for an example of how to use **extract_word()**.

- **gf_free()** in **gf_complete.h**: If **gf_init_easy()**, **gf_init_hard()** or **create_gf_from_argv()** allocated memory, this frees it. See section 6.4.
- **gf_func_a_b** in **gf_complete.h**: This is the data type of **multiply()** and **divide()** in a **gf_t**. See section 4.2 for examples of how to use **multiply()** and **divide()**.
- **gf_func_a_b** in **gf_complete.h**: This is the data type of **multiply()** and **divide()** in a **gf_t**. See section 4.2 for examples of how to use **multiply()** and **divide()**.
- **gf_func_a** in **gf_complete.h**: This is the data type of **inverse()** in a **gf_t**.
- **gf_general_add()** in **gf_general.h**: This adds two **gf_general_t**'s.
- **gf_general_divide()** in **gf_general.h**: This divides two **gf_general_t**'s.
- **gf_general_do_region_check()** in **gf_general.h**: This checks a region multiply of **gf_general_t**'s.
- **gf_general_do_region_multiply()** in **gf_general.h**: This does a region multiply of **gf_general_t**'s.
- **gf_general_do_single_timing_test()** in **gf_general.h**: Used in **gf_time.c**.
- **gf_general_inverse()** in **gf_general.h**: This takes the inverse of a **gf_general_t**.
- **gf_general_is_one()** in **gf_general.h**: This tests whether a **gf_general_t** is one.
- **gf_general_is_two()** in **gf_general.h**: This tests whether a **gf_general_t** is two.
- **gf_general_is_zero()** in **gf_general.h**: This tests whether a **gf_general_t** is zero.
- **gf_general_multiply()** in **gf_general.h**: This multiplies two **gf_general_t**'s. See the implementation of **gf_mult.c** for an example.
- **gf_general_s_to_val()** in **gf_general.h**: This converts a string to a **gf_general_t**. See the implementation of **gf_mult.c** for an example.
- **gf_general_set_one()** in **gf_general.h**: This sets a **gf_general_t** to one.
- **gf_general_set_random()** in **gf_general.h**: This sets a **gf_general_t** to a random number.
- **gf_general_set_two()** in **gf_general.h**: This sets a **gf_general_t** to two.
- **gf_general_set_up_single_timing_test()** in **gf_general.h**: Used in **gf_time.c**.
- **gf_general_set_zero()** in **gf_general.h**: This sets a **gf_general_t** to zero.
- **gf_general_t** in **gf_general.h**: This is a general data type for all values of w. See the implementation of **gf_mult.c** for examples of using these.
- **gf_general_val_to_s()** in **gf_general.h**: This converts a **gf_general_t** to a string. See the implementation of **gf_mult.c** for an example.
- **gf_init_easy()** in **gf_complete.h**: This is how you initialize a default **gf_t**. See 4.2 through 4.5 for examples of calling **gf_init_easy()**.

- **gf_init_hard()** in **gf_complete.h**: This allows you to initialize a **gf_t** without using the defaults. See 6.4. We recommend calling **create_gf_from_argv()** when you can, instead of **gf_init_hard()**.
- **gf_mult_type_t** in **gf_complete.h**: the different ways to specify multiplication when using **gf_init_hard()**. See section 6.4.
- **gf_region_type_t** in **gf_complete.h**: the different ways to specify region multiplication when using **gf_init_hard()**. See section 6.4.
- **gf_region** in **gf_complete.h**: This is the data type of **multiply_region()** in a **gf_t**. See section 4.3 for an example of how to use **multiply_region()**.
- **gf_scratch_size()** in **gf_complete.h**: This is how you calculate how much memory a **gf_t** needs. See section 6.4.
- **gf_size()** in (**gf_complete.h**: Returns the memory consumption of a **gf_t**. See section 6.5.
- **gf_val_128_t** in **gf_complete.h**: This is how you store a value where $w \leq 128$. It is a pointer to two 64-bit unsigned integers. See section 4.4.
- **gf_val_32_t** in **gf_complete.h**: This is how you store a value where $w \leq 32$. It is equivalent to a 32-bit unsigned integer. See section 4.2.
- **gf_val_64_t** in **gf_complete.h**: This is how you store a value where $w \leq 64$. It is equivalent to a 64-bit unsigned integer. See section 4.5.
- **gf_w16_get_div_alog_table()** in **gf_complete.h**: This returns a pointer to an inverse logarithm table that can be used for inlining division when $w = 16$. See section 7.1.
- **gf_w16_get_log_table()** in **gf_complete.h**: This returns a pointer to a logarithm table that can be used for inlining when $w = 16$. See section 7.1.
- **gf_w16_get_mult_alog_table()** in **gf_complete.h**: This returns a pointer to an inverse logarithm table that can be used for inlining multiplication when $w = 16$. See section 7.1.
- **gf_w4_get_div_table()** in **gf_complete.h**: This returns a pointer to a division table that can be used for inlining when $w = 4$. See section 7.1.
- **gf_w4_get_mult_table()** in **gf_complete.h**: This returns a pointer to a multiplication table that can be used for inlining when $w = 4$. See section 7.1.
- **gf_w8_get_div_table()** in **gf_complete.h**: This returns a pointer to a division table that can be used for inlining when $w = 8$. See section 7.1.
- **gf_w8_get_mult_table()** in **gf_complete.h**: This returns a pointer to a multiplication table that can be used for inlining when $w = 8$. See section 7.1.

10 Troubleshooting

- **SSE support.** Leveraging SSE instructions requires processor support as well as compiler support. For example, the Mac OS 10.8.4 (and possibly earlier versions) default compile environment fails to properly compile PCLMUL instructions. This issue can be fixed by installing an alternative compiler; see Section 3 for details.
- **Initialization segfaults.** You have to already have allocated your **gf_t** before you pass a pointer to it in **gf_init_easy()**, **create_gf_from_argv()**, or **gf_init_hard()**.
- **GF-Complete is slower than it should be.** Perhaps your machine has SSE, but you haven't specified the SSE compilation flags. See section 3 for how to compile using the proper flags.
- **Bad alignment.** If you get alignment errors, see Section 5.
- **Mutually exclusive region types.** Some combinations of region types are invalid. All valid and implemented combinations are printed by **gf_methods.c**.
- **Incompatible division types.** Some choices of multiplication type constrain choice of divide type. For example, "**COMPOSITE**" methods only allow the default division type, which divides by finding inverses (*i.e.*, neither "**EUCLID**" nor "**MATRIX**" are allowed). For each multiplication method printed by **gf_methods.c**, the corresponding valid division types are also printed.
- **Arbitrary "GROUP" arguments.** The legal arguments to "**GROUP**" are specified in section 7.5.
- **Arbitrary "SPLIT" arguments.** The legal arguments to "**SPLIT**" are specified in section 7.4.
- **Threading problems.** For threading questions, see Section 8.
- **No default polynomial.** If you change the polynomial in a base field using "**COMPOSITE**," then unless it is a special case for which GF-Complete finds a default polynomial, you'll need to specify the polynomial of the composite field too. See 7.8.2 for the fields where GF-Complete will support default polynomials.
- **Encoding/decoding with different fields.** Certain fields are not compatible. Please see section 7.2 for an explanation.
- **"ALTMAP" is confusing.** We agree. Please see section 7.9 for more explanation.
- **I used "ALTMAP" and it doesn't appear to be functioning correctly.** With 7.9, the size of the region and its alignment both matter in terms of how "**ALTMAP**" performs **multiply_region()**. Please see section 7.9 for detailed explanation.
- **Where are the erasure codes?.** This library only implements Galois Field arithmetic, which is an underlying component for erasure coding. Jerasure will eventually be ported to this library, so that you can have fast erasure coding.

11 Timings

We don't want to get too detailed with timing, because it is quite machine specific. However, here are the timings on an Intel Core i7-3770 CPU running at 3.40 GHz, with 4 $\times$ 256 KB L2 caches and an 8MB L3 cache. All timings are obtained with **gf_time** or **gf_inline_time**, in user mode with the machine dedicated solely to running these jobs.

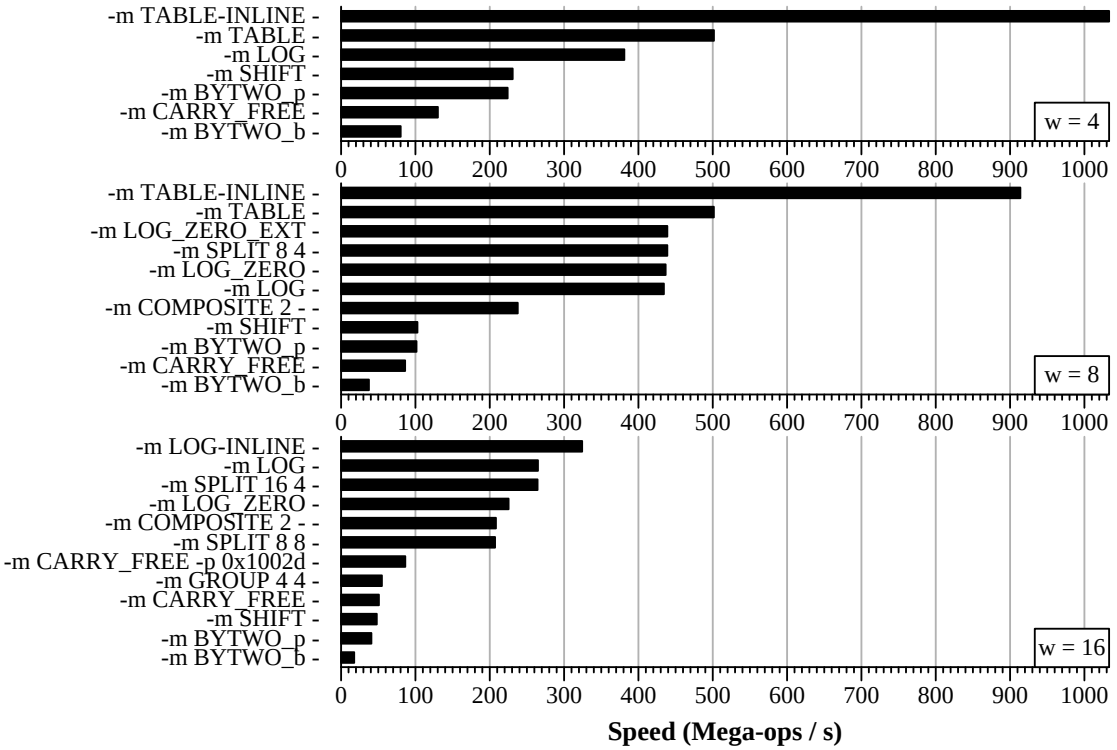

Figure 4: Speed of doing single multiplications for $w \in \{4, 8, 16\}$.

11.1 Multiply()

The performance of **multiply()** is displayed in Figures 4 for $w \in \{4, 8, 16\}$ and 5 for $w \in \{32, 64, 128\}$. These numbers were obtained by calling **gf_time** with the size and iterations both set to 10240. We plot the speed in mega-ops per second.

As would be anticipated, the inlined operations (see section 7.1) outperform the others. Additionally, in all cases with the exception of $w = 32$, the defaults are the fastest performing implementations. With $w = 32$, "**CARRY_FREE**" is the fastest with an alternate polynomial (see section 7.7). Because we require the defaults to use a "standard" polynomial, we cannot use this implementation as the default.

11.2 Divide()

For the "**TABLE**" and "**LOG**" implementations, the performance of division is the same as multiplication. This means that for $w \in \{4, 8, 16\}$, it is very fast indeed. For the other implementations, division is implemented with Euclid's method, and is several factors slower than multiplication.

In Figure 6, we plot the speed of a few implementations of the larger word sizes. Compared to the "**TABLE**" and "**LOG**" implemenations for the smaller word sizes, where the speeds are in the hundreds of mega-ops per second, these are very slow. Of note is the "**COMPOSITE**" implementation for $w = 32$, which is much faster than the others

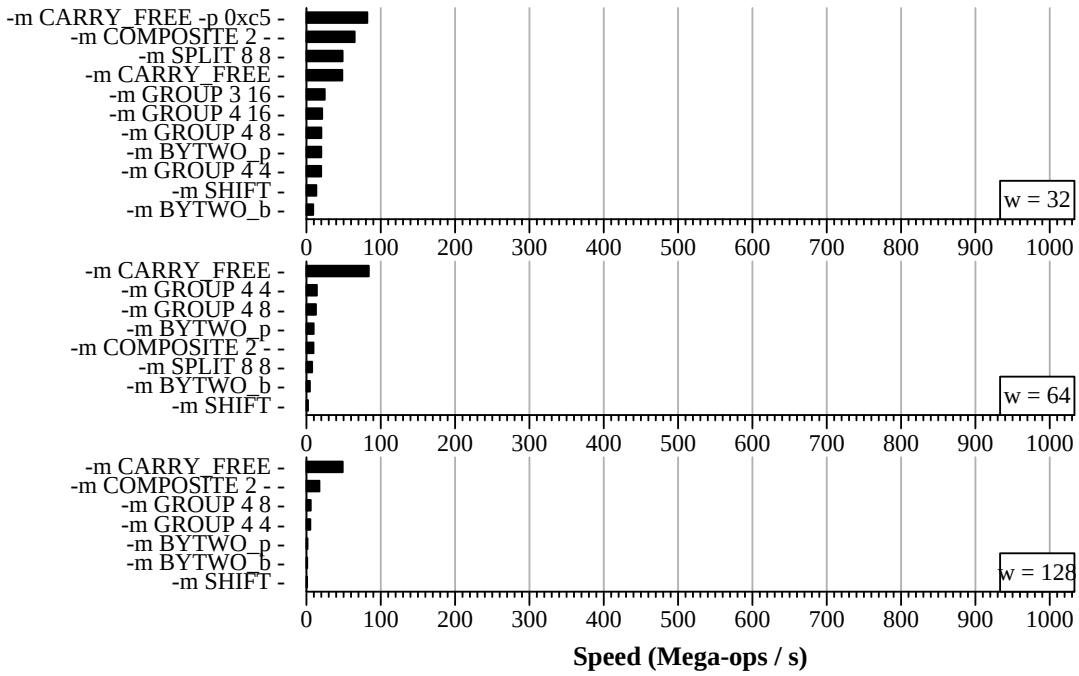

Figure 5: Speed of doing single multiplications for $w \in \{32, 64, 128\}$.

because it uses a special application of Euclid's method, which relies on division in $GF(2^{16})$, which is very fast.

11.3 Multiply_Region()

Tables 3 through 8 show the performance of the various region operations. It should be noted that for $GF(2^{16})$ through $GF(2^{128})$, the default is *not* the fastest implementation of **multiply_region()**. The reasons for this are outlined in section 6.

For these tables, we performed 1GB worth of **multiply_region()** calls for all regions of size 2^i bytes for $10 \leq i \leq 30$. In the table, we plot the fastest speed obtained.

We note that the performance of "**CAUCHY**" can be improved with techniques from [LSXP13] and [PSR12].

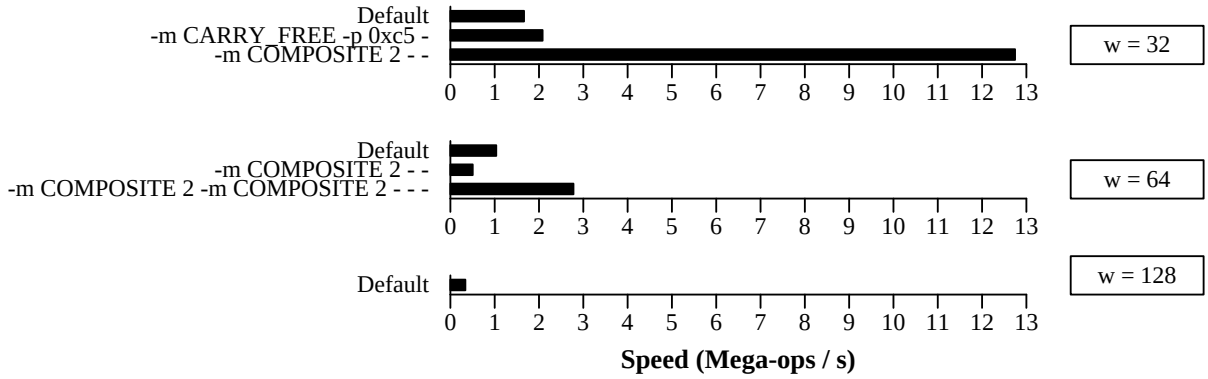

Figure 6: Speed of doing single divisions for $w \in \{32, 64, 128\}$.

Method	Speed (MB/s)
-m TABLE (Default) -	11879.909
-m TABLE -r CAUCHY -	9079.712
-m BYTWO_b -	5242.400
-m BYTWO_p -	4078.431
-m BYTWO_b -r NOSSE -	3799.699
-m TABLE -r QUAD -	3014.315
-m TABLE -r DOUBLE -	2253.627
-m BYTWO_p -r NOSSE -	2021.237
-m TABLE -r NOSSE -	1061.497
-m LOG -	503.310
-m SHIFT -	157.749
-m CARRY_FREE -	86.202

Table 3: Speed of various calls to **multiply_region()** for $w = 4$.

References

[Anv09] H. P. Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2009.

[BKK+95] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[GMS08] K. Greenan, E. Miller, and T. J. Schwartz. Optimizing Galois Field arithmetic for diverse processor architectures and applications. In *MASCOTS 2008: 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Baltimore, MD, September 2008.

[GP97] S. Gao and D. Panario. Tests and constructions of irreducible polynomials over finite fields. In *Foundations of Computational Mathematics*, pages 346–361. Springer Verlag, 1997.

Method	Speed (MB/s)
-m SPLIT 8 4 (Default) -	13279.146
-m COMPOSITE 2 - -r ALTMAP -	5516.588
-m TABLE -r CAUCHY -	4968.721
-m BYTWO_b -	2656.463
-m TABLE -r DOUBLE -	2561.225
-m TABLE -	1408.577
-m BYTWO_b -r NOSSE -	1382.409
-m BYTWO_p -	1376.661
-m LOG_ZERO_EXT -	1175.739
-m LOG_ZERO -	1174.694
-m LOG -	997.838
-m SPLIT 8 4 -r NOSSE -	885.897
-m BYTWO_p -r NOSSE -	589.520
-m COMPOSITE 2 - -	327.039
-m SHIFT -	106.115
-m CARRY_FREE -	104.299

Table 4: Speed of various calls to **multiply_region()** for $w = 8$.

[LBOX12] J. Luo, K. D. Bowers, A. Oprea, and L. Xu. Efficient software implementations of large finite fields $GF(2^n)$ for secure storage applications. *ACM Transactions on Storage*, 8(2), February 2012.

[LD00] J. Lopez and R. Dahab. High-speed software multiplication in f_{2^m}. In *Annual International Conference on Cryptology in India*, 2000.

[LHy08] H. Li and Q. Huan-yan. Parallelized network coding with SIMD instruction sets. In *International Symposium on Computer Science and Computational Technology*, pages 364–369. IEEE, December 2008.

[LSXP13] J. Luo, M. Shrestha, L. Xu, and J. S. Plank. Efficient encoding schedules for XOR-based erasure codes. *IEEE Transactions on Computing*, May 2013.

[Mar94] G. Marsaglia. The mother of all random generators. `ftp://ftp.taygeta.com/pub/c/mother.c`, October 1994.

[PGM13a] J. S. Plank, K. M. Greenan, and E. L. Miller. A complete treatment of software implementations of finite field arithmetic for erasure coding applications. Technical Report UT-CS-13-717, University of Tennessee, September 2013.

[PGM13b] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *FAST-2013: 11th Usenix Conference on File and Storage Technologies*, San Jose, February 2013.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

Method	Speed (MB/s)
-m SPLIT 16 4 -r ALTMAP -	10460.834
-m SPLIT 16 4 -r SSE (Default) -	8473.793
-m COMPOSITE 2 - -r ALTMAP -	5215.073
-m LOG -r CAUCHY -	2428.824
-m TABLE -	2319.129
-m SPLIT 16 8 -	2164.111
-m SPLIT 8 8 -	2163.993
-m SPLIT 16 4 -r NOSSE -	1148.810
-m LOG -	1019.896
-m LOG_ZERO -	1016.814
-m BYTWO_b -	738.879
-m COMPOSITE 2 - -	596.819
-m BYTWO_p -	560.972
-m GROUP 4 4 -	450.815
-m BYTWO_b -r NOSSE -	332.967
-m BYTWO_p -r NOSSE -	249.849
-m CARRY_FREE -	111.582
-m SHIFT -	95.813

Table 5: Speed of various calls to **multiply_region()** for $w = 16$.

[PMG+13] J. S. Plank, E. L. Miller, K. M. Greenan, B. A. Arnold, J. A. Burnum, A. W. Disney, and A. C. McBride. GF-Complete: A comprehensive open source library for Galois Field arithmetic. version 1.0. Technical Report UT-CS-13-716, University of Tennessee, September 2013.

[PSR12] J. S. Plank, C. D. Schuman, and B. D. Robison. Heuristics for optimizing matrix-based erasure codes for fault-tolerant storage systems. In *DSN-2012: The International Conference on Dependable Systems and Networks*, Boston, MA, June 2012. IEEE.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

Method	Speed (MB/s)
-m SPLIT 32 4 -r SSE -r ALTMAP -	7185.440
-m SPLIT 32 4 (Default)	5063.966
-m COMPOSITE 2 -m SPLIT 16 4 -r ALTMAP - -r ALTMAP -	4176.440
-m COMPOSITE 2 - -r ALTMAP -	3360.860
-m SPLIT 8 8 -	1345.678
-m SPLIT 32 8 -	1340.656
-m SPLIT 32 16 -	1262.676
-m SPLIT 8 8 -r CAUCHY -	1143.263
-m SPLIT 32 4 -r NOSSE -	480.859
-m CARRY_FREE -p 0xc5 -	393.185
-m COMPOSITE 2 - -	332.964
-m BYTWO_b -	309.971
-m BYTWO_p -	258.623
-m GROUP 4 8 -	242.076
-m GROUP 4 4 -	227.399
-m CARRY_FREE -	226.785
-m BYTWO_b -r NOSSE -	143.403
-m BYTWO_p -r NOSSE -	111.956
-m SHIFT -	52.295

Table 6: Speed of various calls to **multiply_region()** for $w = 32$.

Method	Speed (MB/s)
-m SPLIT 64 4 -r ALTMAP -	3522.798
-m SPLIT 64 4 -r SSE (Default) -	2647.862
-m COMPOSITE 2 -m SPLIT 32 4 -r ALTMAP - -r ALTMAP -	2461.572
-m COMPOSITE 2 - -r ALTMAP -	1860.921
-m SPLIT 64 16 -	1066.490
-m SPLIT 64 8 -	998.461
-m CARRY_FREE -	975.290
-m SPLIT 64 4 -r NOSSE -	545.479
-m GROUP 4 4 -	230.137
-m GROUP 4 8 -	153.947
-m BYTWO_b -	144.052
-m BYTWO_p -	124.538
-m SPLIT 8 8 -	98.892
-m BYTWO_p -r NOSSE -	77.912
-m COMPOSITE 2 - -	77.522
-m BYTWO_b -r NOSSE -	36.391
-m SHIFT -	25.282

Table 7: Speed of various calls to **multiply_region()** for $w = 64$.

Method	Speed (MB/s)
-m SPLIT 128 4 -r ALTMAP -	1727.683
-m COMPOSITE 2 -m SPLIT 64 4 -r ALTMAP - -r ALTMAP -	1385.693
-m COMPOSITE 2 - -r ALTMAP -	1041.456
-m SPLIT 128 8 (Default)	872.619
-m CARRY_FREE -	814.030
-m SPLIT 128 4 -	500.133
-m COMPOSITE 2 - -	289.207
-m GROUP 4 8 -	133.583
-m GROUP 4 4 -	116.187
-m BYTWO_p -	25.162
-m BYTWO_b -	25.157
-m SHIFT -	14.183

Table 8: Speed of various calls to **multiply_region()** for $w = 128$.