

Optimizing a reed-solomon encoder (polynomial division)

I am trying to optimize a Reed-Solomon encoder, which is in fact simply a polynomial division operation over Galois Fields 2^8 (which simply means that values wrap-around over 255). The code is in fact very very similar to what can be found here for Go: <http://research.swtch.com/field>

The algorithm for polynomial division used here is a [synthetic division](#) (also called Horner's method).

I tried everything: numpy, pypy, cython. The best performance I get is by using pypy with this simple nested loop:

```
def rsenc(msg_in, nsym, gen):
    '''Reed-Solomon encoding using polynomial division, better explained at
    http://research.swtch.com/field'''
    msg_out = bytearray(msg_in) + bytearray(len(gen)-1)
    lgen = bytearray([gf_log[gen[j]] for j in xrange(len(gen))])

    for i in xrange(len(msg_in)):
        coef = msg_out[i]
        # coef = gf_mul(msg_out[i], gf_inverse(gen[0])) // for general polynomial
        # division (when polynomials are non-monic), we need to compute: coef = msg_out[i] / gen[0]
        if coef != 0: # coef 0 is normally undefined so we manage it manually here (and it
            # also serves as an optimization btw)
                lcoef = gf_log[coef] # precaching

            for j in xrange(1, len(gen)): # optimization: can skip g0 because the first
                # coefficient of the generator is always 1! (that's why we start at position 1)
                    msg_out[i + j] ^= gf_exp[lcoef + lgen[j]] # equivalent (in Galois Field
                # 2^8) to msg_out[i+j] += msg_out[i] * gen[j]

    # Recopy the original message bytes
    msg_out[:len(msg_in)] = msg_in
    return msg_out
```

Can a Python optimization wizard guide me to some clues on how to get a speedup? My goal is to get at least a speedup of 3x, but more would be awesome. Any approach or tool is accepted, as long as it is cross-platform (works at least on Linux and Windows).

Here is a small test script with some of the other alternatives I tried (the cython attempt is not included since it was slower than native python!):

```
import random
from operator import xor

numpy_enabled = False
try:
    import numpy as np
    numpy_enabled = True
except ImportError:
    pass

# Exponent table for 3, a generator for GF(256)
gf_exp = bytearray([1, 3, 5, 15, 17, 51, 85, 255, 26, 46, 114, 150, 161, 248, 19,
53, 95, 225, 56, 72, 216, 115, 149, 164, 247, 2, 6, 10, 30, 34,
102, 170, 229, 52, 92, 228, 55, 89, 235, 38, 106, 190, 217, 112,
144, 171, 230, 49, 83, 245, 4, 12, 20, 60, 68, 204, 79, 209, 104,
184, 211, 110, 178, 205, 76, 212, 103, 169, 224, 59, 77, 215, 98,
166, 241, 8, 24, 40, 120, 136, 131, 158, 185, 208, 107, 189, 220,
127, 129, 152, 179, 206, 73, 219, 118, 154, 181, 196, 87, 249, 16,
48, 80, 240, 11, 29, 39, 105, 187, 214, 97, 163, 254, 25, 43, 125,
135, 146, 173, 236, 47, 113, 147, 174, 233, 32, 96, 160, 251, 22,
58, 78, 210, 109, 183, 194, 93, 231, 50, 86, 250, 21, 63, 65, 195,
94, 226, 61, 71, 201, 64, 192, 91, 237, 44, 116, 156, 191, 218,
117, 159, 186, 213, 100, 172, 239, 42, 126, 130, 157, 188, 223,
122, 142, 137, 128, 155, 182, 193, 88, 232, 35, 101, 175, 234, 37,
111, 177, 200, 67, 197, 84, 252, 31, 33, 99, 165, 244, 7, 9, 27,
45, 119, 153, 176, 203, 70, 202, 69, 207, 74, 222, 121, 139, 134,
145, 168, 227, 62, 66, 198, 81, 243, 14, 18, 54, 90, 238, 41, 123,
141, 140, 143, 138, 133, 148, 167, 242, 13, 23, 57, 75, 221, 124,
132, 151, 162, 253, 28, 36, 108, 180, 199, 82, 246] * 2 + [1])

# Logarithm table, base 3
gf_log = bytearray([0, 0, 25, 1, 50, 2, 26, 198, 75, 199, 27, 104, 51, 238, 223, # BEWARE:
the first entry should be None instead of 0 because it's undefined, but for a bytearray we
can't set such a value
3, 100, 4, 224, 14, 52, 141, 129, 239, 76, 113, 8, 200, 248, 105,
28, 193, 125, 194, 29, 181, 249, 185, 39, 106, 77, 228, 166, 114,
154, 201, 9, 120, 101, 47, 138, 5, 33, 15, 225, 36, 18, 240, 130,
69, 53, 147, 218, 142, 150, 143, 219, 189, 54, 208, 206, 148, 19,
92, 210, 241, 64, 70, 131, 56, 102, 221, 253, 48, 191, 6, 139, 98,
179, 37, 226, 152, 34, 136, 145, 16, 126, 110, 72, 195, 163, 182,
30, 66, 58, 107, 40, 84, 250, 133, 61, 186, 43, 121, 10, 21, 155,
159, 94, 202, 78, 212, 172, 229, 243, 115, 167, 87, 175, 88, 168,
80, 244, 234, 214, 116, 79, 174, 233, 213, 231, 230, 173, 232, 44,
215, 117, 122, 235, 22, 11, 245, 89, 203, 95, 176, 156, 169, 81,
160, 127, 12, 246, 111, 23, 196, 73, 236, 216, 67, 31, 45, 164,
118, 123, 183, 204, 187, 62, 90, 251, 96, 177, 134, 59, 82, 161,
108, 170, 85, 41, 157, 151, 178, 135, 144, 97, 190, 220, 252, 188,
149, 207, 205, 55, 63, 91, 209, 83, 57, 132, 60, 65, 162, 109, 71,
20, 42, 158, 93, 86, 242, 211, 171, 68, 17, 146, 217, 35, 32, 46,
137, 180, 124, 184, 38, 119, 153, 227, 165, 103, 74, 237, 222, 197,
49, 254, 24, 13, 99, 140, 128, 192, 247, 112, 7])
```

```

if numpy_enabled:
    np_gf_exp = np.array(gf_exp)
    np_gf_log = np.array(gf_log)

def gf_pow(x, power):
    return gf_exp[(gf_log[x] * power) % 255]

def gf_poly_mul(p, q):
    r = [0] * (len(p) + len(q) - 1)
    lp = [gf_log[p[i]] for i in xrange(len(p))]
    for j in range(len(q)):
        lq = gf_log[q[j]]
        for i in range(len(p)):
            r[i + j] ^= gf_exp[lp[i] + lq]
    return r

def rs_generator_poly_base3(nsize, fcr=0):
    g_all = {}
    g = [1]
    g_all[0] = g_all[1] = g
    for i in range(fcr+1, fcr+nsize+1):
        g = gf_poly_mul(g, [1, gf_pow(3, i)])
        g_all[nsize-i] = g
    return g_all

# Fastest way with pypy
def rsenc(msg_in, nsym, gen):
    '''Reed-Solomon encoding using polynomial division, better explained at
    http://research.swtch.com/field'''
    msg_out = bytearray(msg_in) + bytearray(len(gen)-1)
    lgen = bytearray([gf_log[gen[j]] for j in xrange(len(gen))])

    for i in xrange(len(msg_in)):
        coef = msg_out[i]
        # coef = gf_mul(msg_out[i], gf_inverse(gen[0])) # for general polynomial division
        # (when polynomials are non-monic), the usual way of using synthetic division is to divide
        # the divisor g(x) with its leading coefficient (call it a). In this implementation, this
        # means we need to compute: coef = msg_out[i] / gen[0]
        if coef != 0: # coef 0 is normally undefined so we manage it manually here (and it
        also serves as an optimization btw)
            lcoef = gf_log[coef] # precaching

            for j in xrange(1, len(gen)): # optimization: can skip g0 because the first
            coefficient of the generator is always 1! (that's why we start at position 1)
                msg_out[i + j] ^= gf_exp[lcoef + lgen[j]] # equivalent (in Galois Field
                2^8) to msg_out[i+j] += msg_out[i] * gen[j]

    # Recopy the original message bytes
    msg_out[:len(msg_in)] = msg_in
    return msg_out

# Alternative 1: the loops were completely changed, instead of fixing msg_out[i] and
# updating all subsequent i+j items, we now fixate msg_out[i+j] and compute it at once
# using all couples msg_out[i] * gen[j] - msg_out[i+1] * gen[j-1] - ... since when we fixate
# msg_out[i+j], all previous msg_out[k] with k < i+j are already fully computed.
def rsenc_alt1(msg_in, nsym, gen):
    msg_in = bytearray(msg_in)
    msg_out = bytearray(msg_in) + bytearray(len(gen)-1)
    lgen = bytearray([gf_log[gen[j]] for j in xrange(len(gen))])

    # Alternative 1
    jlist = range(1, len(gen))
    for k in xrange(1, len(msg_out)):
        for x in xrange(max(k-len(msg_in), 0), len(gen)-1):
            if k-x-1 < 0: break
            msg_out[k] ^= gf_exp[msg_out[k-x-1] + lgen[jlist[x]]]

    # Recopy the original message bytes
    msg_out[:len(msg_in)] = msg_in
    return msg_out

# Alternative 2: a rewrite of alternative 1 with generators and reduce
def rsenc_alt2(msg_in, nsym, gen):
    msg_in = bytearray(msg_in)
    msg_out = bytearray(msg_in) + bytearray(len(gen)-1)
    lgen = bytearray([gf_log[gen[j]] for j in xrange(len(gen))])

    # Alternative 1
    jlist = range(1, len(gen))
    for k in xrange(1, len(msg_out)):
        items_gen = ( gf_exp[msg_out[k-x-1] + lgen[jlist[x]]] if k-x-1 >= 0 else
        next(iter(())) for x in xrange(max(k-len(msg_in), 0), len(gen)-1) )
        msg_out[k] ^= reduce(xor, items_gen)

    # Recopy the original message bytes
    msg_out[:len(msg_in)] = msg_in
    return msg_out

# Alternative with Numpy
def rsenc_numpy(msg_in, nsym, gen):
    msg_in = np.array(bytearray(msg_in))
    msg_out = np.pad(msg_in, (0, nsym), 'constant')
    lgen = np_gf_log[gen]
    for i in xrange(msg_in.size):
        msg_out[i+1:i+lgen.size] ^= np_gf_exp[np.add(lgen[1:], msg_out[i])]

    msg_out[:len(msg_in)] = msg_in
    return msg_out

gf_mul_arr = [bytearray(256) for _ in xrange(256)]

```

```

gf_add_arr = [bytearray(256) for _ in xrange(256)]

# Precompute multiplication and addition tables
def gf_precomp_tables(gf_exp=gf_exp, gf_log=gf_log):
    global gf_mul_arr, gf_add_arr
    for i in xrange(256):
        for j in xrange(256):
            gf_mul_arr[i][j] = gf_exp[gf_log[i] + gf_log[j]]
            gf_add_arr[i][j] = i ^ j
    return gf_mul_arr, gf_add_arr

# Alternative with precomputation of multiplication and addition tables, inspired by zfec:
# https://hackage.haskell.org/package/fec-0.1.1/src/zfec/fec.c
def rsenc_precomp(msg_in, nsym, gen=None):
    msg_in = bytearray(msg_in)
    msg_out = bytearray(msg_in) + bytearray(len(gen)-1)

    for i in xrange(len(msg_in)): # [i for i in xrange(len(msg_in)) if msg_in[i] != 0]
        coef = msg_out[i]
        if coef != 0: # coef 0 is normally undefined so we manage it manually here (and
            # it also serves as an optimization btw)
            mula = gf_mul_arr[coef]
            for j in xrange(1, len(gen)): # optimization: can skip g0 because the first
                # coefficient of the generator is always 1! (that's why we start at position 1)
                #msg_out[i + j] = gf_add_arr[msg_out[i+j]][gf_mul_arr[coef][gen[j]]] #
                # slower...
                #msg_out[i + j] ^= gf_mul_arr[coef][gen[j]] # faster
                msg_out[i + j] ^= mula[gen[j]] # fastest

    # Recopy the original message bytes
    msg_out[:len(msg_in)] = msg_in # equivalent to c = mprime - b, where mprime is msg_in
    # padded with [0]*nsym
    return msg_out

def randstr(n, size):
    '''Generate very fastly a random hexadecimal string. Kudos to jcdryer
    http://stackoverflow.com/users/131084/jcdryer'''
    hexstr = '%0'+str(size)+'x'
    for _ in xrange(n):
        yield hexstr % random.randrange(16**size)

# Simple test case
if __name__ == "__main__":

    # Setup functions to test
    funcs = [rsenc, rsenc_precomp, rsenc_alt1, rsenc_alt2]
    if numpy_enabled: funcs.append(rsenc_numpy)
    gf_precomp_tables()

    # Setup RS vars
    n = 255
    k = 213

    import time

    # Init the generator polynomial
    g = rs_generator_poly_base3(n)

    # Init the ground truth
    mes = 'hello world'
    mesecc_correct = rsenc(mes, n-11, g[k])

    # Test the functions
    for func in funcs:
        # Sanity check
        if func(mes, n-11, g[k]) != mesecc_correct: print func.__name__, ": output is
incorrect!"
        # Time the function
        total_time = 0
        for m in randstr(1000, n):
            start = time.clock()
            func(m, n-k, g[k])
            total_time += time.clock() - start
        print func.__name__, ": total time elapsed %f seconds." % total_time

```

And here is the result on my machine:

```

With PyPy:
rsenc : total time elapsed 0.108183 seconds.
rsenc_alt1 : output is incorrect!
rsenc_alt1 : total time elapsed 0.164084 seconds.
rsenc_alt2 : output is incorrect!
rsenc_alt2 : total time elapsed 0.557697 seconds.

Without PyPy:
rsenc : total time elapsed 3.518857 seconds.
rsenc_alt1 : output is incorrect!
rsenc_alt1 : total time elapsed 5.630897 seconds.
rsenc_alt2 : output is incorrect!
rsenc_alt2 : total time elapsed 6.100434 seconds.
rsenc_numpy : output is incorrect!
rsenc_numpy : total time elapsed 1.631373 seconds

```

(Note: the alternatives should be correct, some index must be a bit off, but since they are slower anyway I did not try to fix them)

/UPDATE and goal of the bounty: I found a very interesting optimization trick that promises to speed up computations a lot: to [precompute the multiplication table](#). I updated the code above with the new function `rsenc_precomp()`. However, there's no gain at all in my implementation, it's even a bit slower:

rsenc : total time elapsed 0.107170 seconds.
rsenc_precomp : total time elapsed 0.108788 seconds.

How can it be that arrays lookups cost more than operations like additions or xor? Why does it work in ZFEC and not in Python?

I will attribute the bounty to whoever can show me how to make this multiplication/addition lookup-tables optimization work (faster than the xor and addition operations) or who can explain to me with references or analysis why this optimization cannot work here (using Python/PyPy/Cython/Numpy etc.. I tried them all).

python numpy optimization cython pypy

edited May 26 at 13:40

asked May 21 at 3:11



gaborous

1,933 3 17 31

Have you tried pypi.python.org/pypi/reedsolo? I'd be curious how it compares speed wise. — [figs](#) May 23 at 3:13

- 1 The code I posted is an enhancement of the reedsolo library you cite :) It's quite fast with pypy, but not as fast as the implementation I present above, but it's still quite far from the performance of zfec and other c/c++ libraries... — [gaborous](#) May 23 at 21:01

3 Answers

The following is 3x faster than pypy on my machine (0.04s vs 0.15s). Using Cython:

```
ctypedef unsigned char uint8_t # does not work with Microsoft's C Compiler: from
libc.stdint import uint8_t
import cpython.array as array

cdef uint8_t[:,1] gf_exp = bytearray([1, 3, 5, 15, 17, 51, 85, 255, 26, 46, 114, 150, 161,
248, 19,
lots of numbers omitted for space reasons
...])

cdef uint8_t[:,1] gf_log = bytearray([0, 0, 25, 1, 50, 2, 26, 198, 75, 199, 27, 104,
more numbers omitted for space reasons
...])

import cython

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.initializedcheck(False)
def rsenc(msg_in_r, nsym, gen_t):
    '''Reed-Solomon encoding using polynomial division, better explained at
    http://research.swtch.com/field'''

    cdef uint8_t[:,1] msg_in = bytearray(msg_in_r) # have to copy, unfortunately - can't
    make a memory view from a read only object
    cdef int[:,1] gen = array.array('i', gen_t) # convert list to array

    cdef uint8_t[:,1] msg_out = bytearray(msg_in) + bytearray(len(gen)-1)
    cdef int j
    cdef uint8_t[:,1] lgen = bytearray(gen.shape[0])
    for j in xrange(gen.shape[0]):
        lgen[j] = gf_log[gen[j]]

    cdef uint8_t coef, lcoef

    cdef int i
    for i in xrange(msg_in.shape[0]):
        coef = msg_out[i]
        if coef != 0: # coef 0 is normally undefined so we manage it manually here (and it
also serves as an optimization btw)
            lcoef = gf_log[coef] # precaching

        for j in xrange(1, gen.shape[0]): # optimization: can skip g0 because the
first coefficient of the generator is always 1! (that's why we start at position 1)
            msg_out[i + j] ^= gf_exp[lcoef + lgen[j]] # equivalent (in Galois Field
2^8) to msg_out[i+j] -= msg_out[i] * gen[j]

    # Recopy the original message bytes
    msg_out[:msg_in.shape[0]] = msg_in
    return msg_out
```

It is just your fastest version with static types (and checking the html from `cython -a` until the loops aren't highlighted in yellow).

A few brief notes:

- Cython prefers `x.shape[0]` to `len(shape)`
- Defining the memoryviews as `[:,1]` promises they are continuous in memory, which helps
- `initializedcheck(False)` is necessary for avoiding lots of existence checks on the globally defined `gf_exp` and `gf_log`. (You might find you can speed up your basic Python/PyPy

code by creating a local variable reference for these and using that instead)

- I had to copy a couple of the input arguments. Cython can't make a memoryview from a readonly object (in this case `msg_in`, a string. I could probably have just made it a `char*` though). Also `gen` (a list) needs to be in something with fast element access.

Other than that it's all fairly straight-forward. (I haven't tried any variations of it having got it faster). I'm really quite impressed at how well PyPy does.

edited May 21 at 21:20



gaborous

1,933 3 17 31

answered May 21 at 7:33



DavidW

1,645 3 13

Thank you very much for your answer, it seems very promising, but I can't get it to work because it crashes on the `uint8_t` type, which is part of C99 and [unluckily on Windows, Microsoft's C compiler is C89](#). Could you suggest another type please? — [gaborous](#) May 21 at 18:39

1 Does `unsigned char` work? If you replace the `cimport libc.stdint` line with `ctypedef unsigned char uint8_t` I suspect it'll work fine (haven't tested it though). — [DavidW](#) May 21 at 18:43

Thank you very much, but it fails with invalid syntax, but there's something fishy. I tried to just `ctypedef unsigned long ULong`, which is the example given in the cython doc, and it fails with invalid syntax too. Googling... — [gaborous](#) May 21 at 18:49

Fixed: I was trying to directly run the `.pyx`. I just fixed it by creating a `.py` file that calls the `.pyx`. Silly me... This code indeed runs faster, and your explanations are really useful (I wondered how to create arrays of chars in Cython), thank's a lot! However, on my real setup, I only gain about a 2x speedup (if I account the rest of the Python code that is not JITed now, I go down to only a 0.25x speedup but I will cythonize later). Can you give me clues about the variations you would do to get even more speedup? — [gaborous](#) May 21 at 19:47

I'm honestly not sure what else you could do to make it faster (short of a clever redesign of the algorithm, and that may well not be possible!). The only obvious thing I can see is to avoid copying `msg_in` at the start (you could get a `const uint8_t*` without a copy, but you need to change a few other things later), and I'd be surprised if it gained much. That's probably the best I can do. — [DavidW](#) May 21 at 20:11

Yes indeed I already timed that part of the code, the copy isn't really a bottleneck, it's just that the nested loop is huge. I'll try to see if I can find another algorithm, but for now your solution is the one I'll use. Thank you very much for your detailed advices :) — [gaborous](#) May 21 at 20:51

I found a way to get a bit more juice: by initializing directly the `gen_t` polynomial as a `bytearray()` instead of a sublist inside the generator function `rs_generator_poly_base3()`, there's no need to convert at encoding time, and in addition `gen` can then be declared as `uint8_t`. This makes timing go down from 0.042 to 0.037. — [gaborous](#) May 22 at 17:05

Tried `prange()` to parallelize the inner loop, and it works great! This boosted quite a bit more. I added it to your answer. — [gaborous](#) May 22 at 17:56

Glad to here you got it faster. They're sensible ways to speed it up. It goes a bit beyond the scope of my original answer, so it's probably better to add an answer of your own saying that you got further improvements with these methods (or just leave it in the comments) rather than editing my answer. — [DavidW](#) May 22 at 20:46

Ok fair enough, I'll post a separate answer. I just did a fix to your answer, there was still one `len(gen)` instead of `gen.shape[0]`. After fixing that, there is another great speed boost! Also, I discovered that `len()` still calls Python code, while `.shape` does not (and in fact it is mandatory to use `.shape` when releasing the gil with `with nogil:`). — [gaborous](#) May 23 at 2:41

@DavidW Nice. Try replacing `int i, j` with `Py_ssize_t i, j` that should be faster (and loop could be potentially vectorized by the compiler), as illustrated in this [question](#). — [rth](#) Jun 2 at 8:45

Alternatively, if you know C, I would recommend to rewrite this Python function in plain C and call it (say with CFFI). At least you know you reach the top performance in the inner loops of your functions without needing to be aware of either PyPy or Cython tricks.

See: <http://cffi.readthedocs.org/en/latest/overview.html#performance>

answered May 21 at 8:05



Armin Rigo

5,672 8 20

Thank you for the suggestion. CFFI is indeed very interesting and I know how to program in C, but compiling C code from the Python interpreter on Windows is always quite complicated. I could find a way to do that simply for Cython, but for CFFI it can be quite complicated from what I read, and I'd rather avoid the hassle for my users. But it could be used to make an optional high speed module for users willing to take the time to install everything that is needed, I'll give it a thought. — [gaborous](#) May 21 at 18:43

Thank's to you, I discovered that CFFI relies on [pycparser](#), a full C99 parser in pure python. Awesome! — [gaborous](#) May 21 at 18:46

2 Note that the very new CFFI 1.0 should fix most of the issues of installation (they are talking about CFFI 0.9 if the pages are more than a few days old). It is now like Cython: you need a C compiler, but you control when it is invoked—either when you run the build scripts, or in `setuptools`, or even manually on the generated C file—and you can then distribute the resulting DLL (`.pyd` file). — [Armin Rigo](#) May 21 at 23:19

Building on DavidW's answer, here's the implementation I am currently using, which is about 20% faster by u

```
from cython.parallel import parallel, prange

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.initializedcheck(False)
cdef rsenc_cython(msg_in_r, nsym, gen_t) :
    '''Reed-Solomon encoding using polynomial division, better explained at
    http://research.swtch.com/field'''

    cdef uint8_t[:,1] msg_in = bytearray(msg_in_r) # have to copy, unfortunately - can't
    make a memory view from a read only object
    #cdef int[:,1] gen = array.array('i',gen_t) # convert list to array
    cdef uint8_t[:,1] gen = gen_t

    cdef uint8_t[:,1] msg_out = bytearray(msg_in) + bytearray(len(gen)-1)
    cdef int i, j
    cdef uint8_t[:,1] lgen = bytearray(gen.shape[0])
    for j in xrange(gen.shape[0]):
        lgen[j] = gf_log_c[gen[j]]

    cdef uint8_t coef, lcoef
    with nogil:
        for i in xrange(msg_in.shape[0]):
            coef = msg_out[i]
            if coef != 0: # coef 0 is normally undefined so we manage it manually here
                (and it also serves as an optimization btw)
                lcoef = gf_log_c[coef] # precalculating

            for j in prange(1, gen.shape[0]): # optimization: can skip g0 because the
            first coefficient of the generator is always 1! (that's why we start at position 1)
                msg_out[i + j] ^= gf_exp_c[lcoef + lgen[j]] # equivalent (in Galois
                Field  $2^8$ ) to  $msg\_out[i+j] -= msg\_out[i] * gen[j]$ 

    # Recopy the original message bytes
    msg_out[:msg_in.shape[0]] = msg_in
    return msg_out
```

I would still like it to be faster (on a real implementation, data is encoded at about 6.4 MB/s with $n=255$, n being

The main lead to a faster implementation that I have found is to use a LUT (LookUp Table) approach, by previous Cython implementations, the LUT approach is slower than calculating XOR and addition operations.

There are other approaches to implement a faster RS encoder, but I don't have the abilities nor the time to try

- "Fast software implementation of finite field operations", Cheng Huang and Lihao Xu, Washington Univ <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.6595&rep=rep1&type=pdf> and a correct view=295
- Luo, Jianqiang, et al. "Efficient software implementations of large finite fields GF (2^n) for secure storage"
- "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage.", F O'Hearn, Z, FAST. Vol. 9. 2009. https://www.usenix.org/legacy/events/fast09/tech/full_papers/plank/plank.pdf of Open-Source Erasure Coding Libraries for Storage Applications", Plank and Schuman.
- Sourcecode of the ZFEC library, with multiplication LUT optimization: <https://hackage.haskell.org/pack>
- "Optimized Arithmetic for Reed-Solomon Encoders", Christof Paar (1997, June). In IEEE International ELECTRICAL ENGINEERS INC (IEEE). <https://www.emsec.rub.de/media/crypto/veroeffentlichungen>
- "A Fast Algorithm for Encoding the (255,233) Reed-Solomon Code Over GF(2^8)", R.L. Miller and T.K 56/56P.PDF
- "Optimizing Galois Field arithmetic for diverse processor architectures and applications", Greenan, Ke Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE <http://www.ssrc.ucsc.edu/Papers/greenan-mascots08.pdf>
- Anvin, H. Peter. "The mathematics of RAID-6." (2007). <https://www.kernel.org/pub/linux/kernel/people/>
- <https://github.com/catid/wirehair/>
- "A logarithmic Boolean time algorithm for parallel polynomial division", Bini, D. and Pan, V. Y. (1987), Ir Pan. "Fast parallel algorithms for polynomial division over an arbitrary field of constants." Computers & https://www.researchgate.net/profile/Dario_Bini/publication/250727103_Fast_parallel_algorithms_for_polynomial_division_over_an_arbitrary_field_of_constants/links/54c23733e815c755.pdf
- Kung, H.T. "Fast evaluation and interpolation." (1973). <http://www.eecs.harvard.edu/~htk/publication/1/>
- Cao, Zhengjun, and Hanyue Cao. "Note on fast division algorithm for polynomials using Newton iteration"
- "An Introduction to Galois Fields and Reed-Solomon Coding", James Westall and James Martin, 2010
- Mamidi, Suman, et al. "Instruction set extensions for Reed-Solomon encoding and decoding." Application IEEE International Conference on. IEEE, 2005. http://glossner.org/john/papers/2005_07_asap_reed_solomon_encoding_and_decoding.pdf
- Dumas, Jean-Guillaume, Laurent Fousse, and Bruno Salvy. "Simultaneous modular reduction and Kromer 46.7 (2011): 823-840.
- Greenan, Kevin M., Ethan L. Miller, and Thomas Schwarz. Analysis and construction of galois fields for 2007. <http://www.ssrc.ucsc.edu/Papers/ssrctr-07-09.pdf>

However, I think the best lead is to use an efficient **polynomial modular reduction** instead of polynomial division

- "Modular Reduction in GF (2ⁿ) without Pre-computational Phase". Knežević, M., et al. Arithmetic of Finite Fields, pp. 111-122. Springer, 2004.
- "On computation of polynomial modular reduction". Wu, Huapeng. Technical report, Univ. of Waterloo, 2004.
- "A fast software implementation for arithmetic operations in GF (2ⁿ)". De Win, E., Bosselaers, A., Van der Veken, B. Advances in Cryptology—Asiacrypt'96 (pp. 65-76). Springer Berlin Heidelberg. <https://www.cosic.esat.kuleuven.be/papers/DeWinBosselaersVanDerVeken.pdf>
- http://en.wikipedia.org/wiki/Barrett_reduction

/EDIT: in fact it seems "On computation of polynomial modular reduction" just uses the same approach as I did (we precompute the couples of coefficients we will need, and reduce them all at once), and unluckily it's not fast for all since it depends on the message input).

/EDIT: I found a library with really interesting optimizations, lots that are not even found in any academic paper. The fastest software implementation of Reed-Solomon: the [wirehair project](#) and the [related blog](#) for more details. Also [longhair](#) with similar optimization tricks.

edited Jun 22 at 17:09

answer



Add Another Answer