# Practical Improvements to Complex Divide

## An Experimental Approach

By Patrick McGehearty

Linux Plumbers Conference
Sept 20-24, 2021

# Complex Divide Naive Formula

(a + bi) / (c + di) =

( (a*c + b*d) / (c*c + d*d)

+ ( (b*c – a*d) / (c*c + d*d) ) i

- Denominator: **(c*c + d*d)** may overflow when **c** or **d** is large.

# Complex Divide Smith's Method (1962)

```
if (fabs(c) < fabs(d))
  {
      ratio = c / d;
      denom = (c * ratio) + d;
      xx = ((a * ratio) + b) / denom;
      yy = ((b * ratio) - a) / denom;
  } else {
      ratio = d / c;
      denom = (d * ratio) + c;
      xx = ((b * ratio) + a) / denom;
      yy = (b - (a * ratio)) / denom;
  }
```

Reducing (c*c + d*d) by the larger of c,d significantly reduces overflow/underflow
Smith's method was used by libgcc prior to gcc 11.2 (changed in Spring 2021).

# Complex Divide Research

- "The Mathematical-Function Computation Handbook" (Beebe 2017) describes four newer methods for computing complex divide:
  - C99 Style - scales by power of 2 of fmax(fabs(c),fabs(d))
    - similar to Smith's method for reducing overflow/underflow.
  - Doug Priest's method – test exponents and scale when too large/small
    - Seemed promising but not substantially better than C99 Style
  - G.W. Stewart's method
    - Adds multiple tests with complicated reordering to always multiply largest with smallest values before multiplying middle value.
      - Substantial improvement – a factor of 50 fewer extreme failures.

# Complex Divide Research (continued)

- Baudin&Smith's method
  - Test ratio of Smith's method for zero (likely underflow),
    reorder computation accordingly.
    - Results for basic method nearly as good as Stewart's method.
    - Less complicated.
    - Showed potential for more direct improvements.

- All further discussion derives from Baudin&Smith starting point.

# Complex Divide Testing Methods

- Early exploration was based on manual creation of test case.
  - Over 100 individual tests developed but approach was limited and biased.
- Developed code to generate IEEE-754 64-bit values with uniform distribution over full exponent range. Mantissa was uniformly distributed.
  - Primary tests used 10 million pairs of complex values.
  - Random number generation was somewhat different from Baudin&Smith
    - My method showed higher error rates than B&S method.
- Note: Floating point numbers are shown in hex-float format (%a).
  - Format lends itself to seeing specific/interesting bit patterns and exponents.

# Complex Divide Baudin and Smith's Method

```
if (fabs(c) < fabs(d))    {
    ratio = c / d;
    denom = (c * ratio) + d;
    if ( ratio != 0)        {
        xx = ((a * ratio) + b) / denom;
        yy = ((b * ratio) - a) / denom;
      }      else        {
        xx = (c * (a / d) + b) / denom;
        yy = (c * (b / d) - a) / denom;
      }
  }  else {
...
```

# Complex Divide Initial Results

- Count of Errors (> 20 bits in mantissa) out of 10M tests
  - Smith's method 157,144
  - B&S's basic method 28,571
  - 5.5x reduction in gross errors.
- Now what?
  - Experimental method: examine input values that fail and intermediate calculations for sources of failure.
  - Propose additional algorithm adjustments.

# Complex Divide Improvements – First Iteration

- Input: (a+bi) / (c+di)  Output: Quad = 128-bit result;  New= new result

  a  = 0x1.73a3dac1d2f1fp+509  b = -0x1.c4dba4ba1ee79p-620 i

  c  = -0x1.adf526c249cf0p+353  d = 0x1.98b3fbc1677bbp-697  i

  quad = -0x1.ba8df8075bceep+155  -0x1.a4ad6**28da5b74**p-895 i

  new  = -0x1.ba8df8075bceep+155  -0x1.a4ad6**329485f0**p-895 i

  - Key intermediate result: **ratio** = -0x**0.0000000**f3584ap-1022

    ratio is a subnormal  =  fabs(ratio) < DBL_MIN (= 0x1.0p-1022) && != 0

- Solution: test ratio for (< DBL_MIN) instead of  (== 0)

# Complex Divide First Iteration Results

- Count of Errors (> 20 bits in mantissa) out of 10M tests
  - Smith's method 157,144
  - B&S's basic method 28,571
  - Add DBL_MIN check 2,248 !!
  - Additional 12x reduction in gross errors.

# Complex Divide Improvements – Second Iteration

- Sample failure: a = -0x0.000000008e4f8p-1022 b = 0x0.0000060366ba7p-1022 i

  c = -0x1.605b467369526p-245 d = 0x1.417bd33105808p-256 i

  quad = 0x1.cde59**3daa4ffe**p-810 -0x1.179b9a6**3df6d3**p-799 i

  new = 0x1.cde59**176d317d**p-810 -0x1.179b9a6**2f3a97**p-799 i

- Key intermediate values:

  a/c = 0x1.9d934db079f74p-811 and **d*(a/c)** = 0x0.000000000104p-1022

  b/c = -0x1.179ba0f993deap-799 and **d*(b/c)** = -0x0.00000000af90bp-1022

- Solution: If **fmax(c,d) < DBL_EPSILON**; scale a,b,c,d up by 1/DBL_EPSILON
  - DBL_EPSILON = 0x1.0p-52 (found by trial and error)

# Complex Divide Second Iteration Results

- Count of Errors (> 20 bits in mantissa) out of 10M tests
  - Smith's method 157,144
  - B&S's basic method 28,571
  - Add DBL_MIN check 2,248
  - Add DBL_EPSILON scaling 100
  - Additional 22x reduction in gross errors.

# Complex Divide Improvements – Third Iteration

- Sample failure: a = 0x1.cb27eece7c585p-355 **b = 0x0.000000223b8a8p-1022**

  c = -0x1.74e7ed2b9189fp-22 d = 0x1.3d80439e9a119p-731

  good -0x1.3b35ed806ae5ap-333 -0x0.05e01bc**bfd9f6**p-1022

  new -0x1.3b35ed806ae5ap-333 -0x0.05e01bc**cb0923**p-1022

- Key values: b is subnormal and no value is “too big”
- Solution:

```
if (((fabs (a) < DBL_MIN) && (fabs (b) < RMAX2) && (fabs (d) < RMAX2))
 || ((fabs (b) < DBL_MIN) && (fabs (a) < RMAX2) && (fabs (d) < RMAX2)))
 scale a,b,c,d by 1/DBL_EPSILON
```

# Complex Divide Third Iteration Results

- Count of Errors (> 20 bits in mantissa) out of 10M tests
  - Smith's method 157,144
  - B&S's basic method 28,571
  - Add DBL_MIN check (c,d) 2,248
  - Add DBL_EPSILON scaling 100
  - Add DBL_MIN check (a,b) 4
  - Additional 25x reduction in gross errors.

# Complex Divide Improvements – Fourth Iteration

- Sample failure: a = -0x1.f5c75c69829f0p-530 b= -0x1.e73b1fde6b909p+316 i

  c = -0x1.ff96c3957742bp+1023 d= 0x1.5bd78c9335899p+1021 i

  quad = -0x1.423c6ce00c73bp-710 0x1.d9edcf45bcb0ep-708 i

  new = 0x0.000000000000p+0 0x0.000000000000p+0 i

  Difference is 53 bits in error (i.e. completely wrong)
- Key values: c and d are near DBL_MAX (= 0x1.fffffffffffffp+1023)
- Solution: if (larger of c,d) > DBL_MAX/2) then scale a,b,c,d down by 2

# Complex Divide Fourth Iteration Results

- Count of Errors (> 20 bits in mantissa) out of 10M tests
  - Smith's method 157,144
  - B&S's basic method 28,571
  - Add DBL_MIN check (c,d) 2,248
  - Add DBL_EPSILON scaling 100
  - Add DBL_MIN check (a,b) 4
  - Add DBL_MAX check (c,d) 0
  - Eliminates almost all gross errors.... Are we done??

# Complex Divide Details Error Report

| Bits of Error | Smith | Baudin&Smith | New |
|---|---|---|---|
| 2 | 188,189 | 50,354 | 900 |
| 4 | 184,095 | 47,117 | 143 |
| 6 | 180,353 | 44,558 | 72 |
| 8 | 176,822 | 42,168 | 11 |
| 10 | 173,417 | 39,768 | 3 |
| 12 | 170,107 | 37,469 | 1 |
| 16 | 163,461 | 32,879 | 1 |
| 20 | 157,144 | 28,571 | 0 |
| 52 | 113,449 | 350 | 0 |

Count of Errors in 10 Million Tests
Remaining errors due to 'catastrophic subtraction' (when b*c and a*d are nearly equal).

# Baudin&Smith – Offers 10 Difficult Values

| Case | Naive | Smith | C99 | Priest | Stewart | B&S | New |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 53 | 53 | 53 | 53 | 53 | 53 |
| 2 | 0 | 53 | 53 | 53 | 53 | 53 | 53 |
| 3 | 0 | 0 | 0 | 0 | 53 | 53 | 53 |
| 4 | 0 | 0 | 0 | 53 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 53 | 53 | 53 |
| 6 | 0 | 53 | 53 | 53 | 53 | 53 | 53 |
| 7 | 0 | 41 | 53 | 53 | 0 | 0 | 53 |
| 8 | 0 | 0 | 0 | 52 | 0 | 0 | 53 |
| 9 | 0 | 0 | 53 | 53 | 0 | 0 | 53 |
| 10 | 0 | 5 | 5 | 53 | 5 | 5 | 53 |

Remaining failure is when a+bi = 0x1.0p1023 + 0x1.0p1023i; c+di = 1.0+1.0i

# Complex Divide 64-bit Performance Comparison

| | Older X86 64 bit | Older Arm 64 bit |
|---|---|---|
| Relative Slowdown | 1.38 | 1.56 |

Relative Slowdown will vary with different HW implementations, even within a single architecture family.

The purpose of this sample comparison is to show that the loss of performance is not extreme.

Applications with heavy use of complex divide with well behaved input ranges can choose to use gcc compile switches to select faster but less stable methods.

# Complex Divide Experimental Summary

- Method for improving Math Libraries:
  - Start with investigation of current work
  - Measure with wide range of input values.
  - Study failures of current methods very closely.
  - Improve incrementally, repeat.
- Be mindful of performance impact of changes
  - If it's too slow, it won't be accepted, used.

# Complex Divide Discussion/References

- Discussion
  - Other precisions (float16, float, long double have been improved)
  - Further work:  use 80-bit format for intermediate computation when available?
  - Comments? Questions?
- References:
  - [1] Nelson H.F. Beebe, "The Mathematical-Function Computation Handbook. Springer International Publishing AG, 2017.  (Highly recommended, through treatment of many, many topics)
  - [2] Robert L. Smith. Algorithm 116: Complex division.  Commun. ACM, 5(8):435, 1962.
  - [3] Michael Baudin and Robert L. Smith. "A robust complex division in Scilab," October 2012, available at http://arxiv.org/abs/1210.4539.
  - Src code:  https://gcc.gnu.org/git/gitweb.cgi?p=gcc.git;h=54f0224d55a1b56dde092460ddf76913670e6efc