Algoritmos de divisão de números complexos para aplicações de alto desempenho

Marcelo Daronco Ribas[*], Lucas Leandro Nesi, Lucas Mello Schnorr

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{mdribas,lucas.nesi,schnorr}@inf.ufrgs.br

Resumo. *Os números complexos são elementos importantes em aplicações de alto desempenho. No entanto, a divisão desses números pode não ser exata na representação em ponto flutuante. Este trabalho avalia o desempenho e a acurácia de quatro algoritmos de divisão complexa, com três diferentes datasets em processadores AMD e Intel. O principal resultado é que a escolha do algoritmo depende dos dados da aplicação.*

1. Introdução

Números complexos são fundamentais para a modelagem de fenômenos físicos de aplicações de alto desempenho, por exemplo, aplicações geofísicas [Key 2009]. Neste contexto, a representação de números reais e, por consequência, de números complexos pode ser feita com ponto flutuante [Goldberg 1991]. Entretanto, este é um modelo discreto de representação, ou seja, alguns números reais não têm um ponto flutuante equivalente. Ainda, algumas precisões podem ser utilizadas para ponto flutuante, como 32 bits (float), 64 bits (double), ou precisões arbitrárias com n bits. O problema é que as operações algébricas que utilizam ponto flutuante podem não ser exatas, já que seus resultados intermediários ou finais não são representáveis, requerendo uma aproximação. Neste contexto, as operações elementares de números complexos são desafiadoras, por necessitarem de diversas operações de números reais. Uma operação extensamente usada no contexto destas aplicações geofísicas é a divisão complexa, que possui diversos algoritmos para lidar com o problema de acurácia. O objetivo deste trabalho é analisar quatro algoritmos de divisão de números complexos, comparando tempo de execução e acurácia dos resultados. A Seção 2 apresenta os algoritmos a serem estudados, seguida pela Seção 3 que descreve a metodologia adotada. A Seção 4 apresenta os resultados de desempenho e acurácia. Finalmente, a Seção 5 apresenta uma discussão e conclusão do trabalho. Material complementar deste artigo está disponível em um repositório público[1].

2. Algoritmos de divisão complexa e trabalhos relacionados

Números complexos possuem números reais e a unidade imaginária *i*. A representação algébrica destes números é da forma $x = a + bi$. A divisão algébrica de dois números complexos $\frac{x}{y} = e + fi$ com $y = c + di$ pode ser feita multiplicando-se o numerador e o denominador pelo conjugado do denominador [Yaglom 2014]:

$$e + fi = \frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \left(\frac{ac + bd}{c^2 + d^2}\right) + \left(\frac{bc - ad}{c^2 + d^2}\right) i \qquad (1)$$

[*]Este estudo foi financiado pelo projeto Petrobras (2018/00263-5) e pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

[1]https://gitlab.com/marcelodr/erad2024/

No entanto, a realização desta operação em um sistema operacional utilizando a representação de ponto flutuante apresenta desafios. Ao selecionar uma precisão do ponto flutuante, como *double*, a divisão algébrica não garante que resultados intermediários sejam representáveis nesta precisão, podendo, ainda, haver casos de *overflow* e *underflow* na representação numérica. Para evitar esse problema, existem outros algoritmos que podem ser usados a fim de obter resultados mais confiáveis.

Para tentar evitar *underflows*, o Algoritmo 1 proposto por Smith [Smith 1962], compara o valor absoluto das partes real e imaginária do denominador e calcula um termo para redimensionar os cálculos, tentanto evitar resultados não representáveis.

```
1 if(fabs(c) < fabs(d)){
2   r = c / d;
3   den = (c * r) + d;
4   e = ((a * r) + b)/den;
5   f = ((b * r) - a)/den;
6 }else{
7   r = d / c;
8   den = c + (d * r);
9   e = (a + (b * r))/den;
10  f = (b - (a * r))/den;
11 }
```

Algoritmo 1. Smith

```
1 s = fabs(c) + fabs(d);
2 os = 1.0 / s;
3 ars = a * os;
4 ais = b * os;
5 brs = c * os;
6 bis = d * os;
7 s = (brs * brs) + (bis * bis);
8 os = 1.0 / s;
9 e = ((ars * brs) + (ais * bis)) * os;
10 f = ((ais * brs) - (ars * bis)) * os;
```

Algoritmo 2. cucomplex

Havendo a possibilidade de executar aplicações em GPU, existe o Algoritmo 2 de divisão de números complexos implementado pela biblioteca cucomplex do *framework* CUDA [NVIDIA Corporation 2024] que, assim como o algoritmo de Smith, usa um termo adicional para evitar *overflows* e *undeflows*. Veja que esta implementação não utiliza instruções de controle de fluxo (*if*), que são usualmente inadequadas nas GPUs.

Ainda na busca por minimizar erros, um algoritmo [McGehearty 2021] baseado no de Smith foi desenvolvido e proposto para inclusão no compilador GCC para tratar expoentes grandes e pequenos, além dos *denormals*, números pequenos necessários para a correta operação do ponto flutuante. Este algoritmo é o atualmente utilizado no GCC com as diretivas de configuração padrão.

Outros trabalhos analisam a acurácia de operações com números flutuantes [Gladman et al. 2023, NVIDIA Corporation 2024], utilizando ulp (*Unit in the Last Place*) [Gladman et al. 2023], que indica a quantidade de números de ponto flutuante daquela precisão representáveis entre os valores comparados. Entretanto, estes trabalhos não analisam operações de números complexos. Uma comparação da divisão complexa se encontra no patch proposto para o algoritmo do GCC [McGehearty 2021]. Diferentemente, este trabalho utiliza ulp para medir o erro, adiciona o algoritmo cucomplex, e compara o desempenho e a acurácia com intervalos de números ponto-flutuante.

3. Metodologia experimental

Os experimentos foram realizados a partir da implementação dos algoritmos de divisão de números complexos em C, utilizando a precisão *double* e o compilador *gcc* (versão 11.4.0) com a *flag O2*. Os códigos para medição de tempo e acurácia foram executados dez vezes em duas máquinas distintas, uma com processador Intel Xeon Gold 5317 e outra

com processador AMD Ryzen 9 3950X, sendo considerada a média dos tempos em cada máquina para a análise de desempenho e todos os resultados para a acurácia. Os cálculos foram feitos realizando a divisão entre todos os números de três diferentes *datasets*, um deles com números aleatórios sem restrição alguma, outro apenas com *denormals* e o terceiro com números "normais" no intervalo $[3 \times 10^{-5}, 256]$.

O cálculo dos erros foi baseado no resultado obtido usando a biblioteca *mpfr* [Fousse et al. 2007] que permite a representação do ponto flutuante com n bits de precisão. Os experimentos de referência utilizam uma precisão de 200 bits. Para calcular o valor de referência foram implementados os algoritmos *algébrico* e *cucomplex*, e o resultado foi arredondado para *double* com uma função da própria biblioteca. Os resultados de cada algoritmo foram comparados com o valor de referência por ulp's.

4. Resultados experimentais

Esta seção apresenta os resultados de tempo de execução e acurácia.

4.1. Tempo de execução

A Figura 1 apresenta a comparação de tempo de execução médio com intervalo de confiança de 98.5% para cada algoritmo, com diferentes *datasets* de tamanho 40000, com os processadores AMD e Intel.

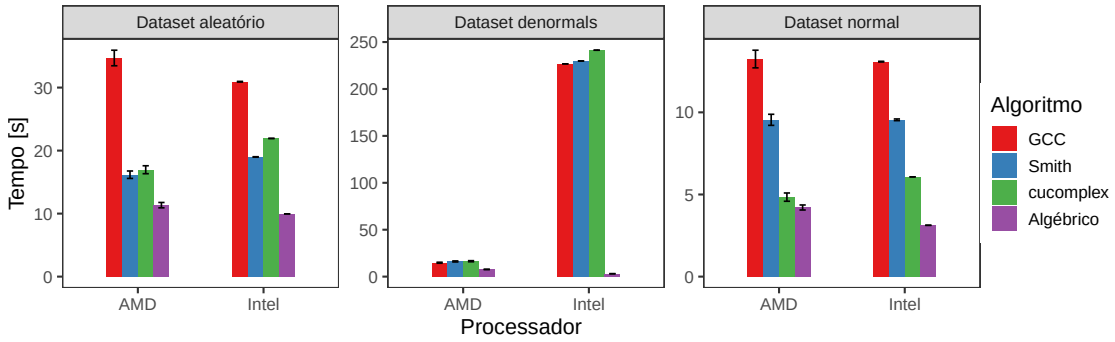

Figura 1. Comparação de tempo de execução

Para os *datasets* aleatório e normal os dois processadores tiveram desempenho parecido, com algum processador tendo leve vantagem em relação ao outro para dado algoritmo, sendo o GCC o mais lento para ambos. A diferença mais expressiva é com o *dataset* denormals, em que o AMD é muito mais rápido que o Intel para todos os algoritmos, menos o algébrico. Ainda, o algoritmo algébrico é o mais rápido, com o *Smith* e *cucomplex* alterando posições dependendo do *dataset* utilizado.

4.2. Erro por ulp

A Figura 2 apresenta os resultados de diferença por ulp de cada algoritmo em relação ao valor de referência para a parte real (facetas superiores) e para a imaginária (inferiores), com os diferentes *datasets* de tamanho 2000. Os resultados agrupam o erro de ulp em 6 categorias (representadas pelas cores): 0 ulp de diferença (verde), 1 ulp (amarelo), entre 2 e 100 ulp's (lilás), e onde o cálculo de ulp não foi possível já que o resultado final foi infinito ou *Not a Number* (NaN) (laranja). Os cálculos foram realizados somente com o processador Intel, porque a diferença em relação ao AMD é apenas o tempo de execução.

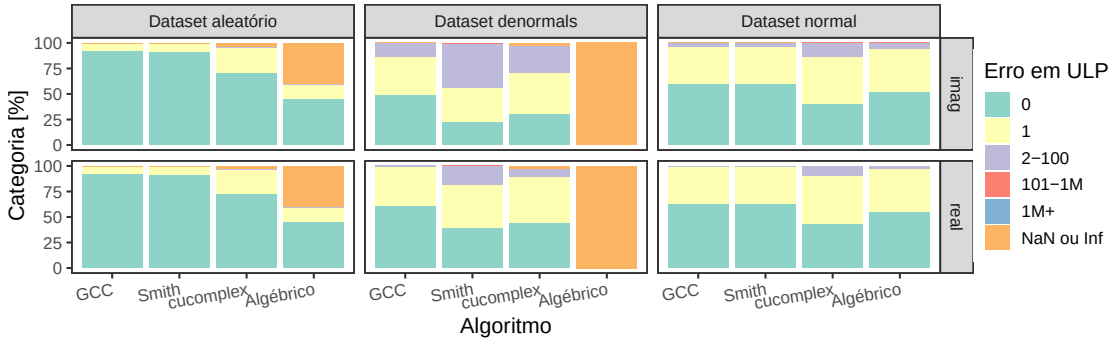

Figura 2. Comparação de acurácia

Com o *dataset* aleatório os algoritmos *Smith* e *GCC* ficaram com acurácia similar e baixa taxa de erro. O algoritmo *cucomplex* teve boa acurácia, com mais erros por 1 ulp em relação aos anteriores. Por outro lado, o algoritmo algébrico teve a pior acurácia com quase 50% dos resultados infinitos ou NaN. Para o *dataset denormals*, o *GCC* segue com menos erros, o *cucomplex* teve mais acertos, comparado com o *Smith*, mas continua apresentando resultados infinitos ou NaN, o que também é visto para o algoritmo algébrico, que teve 100% de resultados infinitos ou NaN. O *dataset normal* não gerou erros infinitos ou NaN para nenhum dos quatro algoritmos. Os algoritmos *Smith* e *GCC* apresentaram resultados idênticos e mais de 50% de acerto. Aqui, o *cucomplex* teve a menor taxa de acerto e foi superado pelo algoritmo algébrico, que teve acerto pouco acima de 50%.

5. Discussão e conclusão

De modo geral, não é possível definir qual a melhor combinação de processador e algoritmo para qualquer aplicação. A escolha a ser feita depende, principalmente, das exigências da aplicação, por exemplo, para operar com *denormals* o algoritmo *GCC* é o mais preciso e o processador AMD é o mais rápido. Por outro lado, se for utilizado um *dataset normal*, ambos os processadores têm desempenho similar e a escolha do algoritmo é feita de acordo com a precisão necessária. Trabalhos futuros incluem a avaliação de outras operações de números complexos e a análise em GPUs.

Referências

Fousse, L. et al. (2007). MPFR: a multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2).

Gladman, B. et al. (2023). Accuracy of mathematical functions in single, double, double extended, and quadruple precision. Relatório técnico, Université de Lorraine.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1).

Key, K. (2009). 1D inversion of multicomponent, multifrequency marine csem data: Methodology and synthetic studies for resolving thin resistive layers. *GEOPHYSICS*.

McGehearty, P. (2021). [PATCH v9] Practical improvement to libgcc complex divide.

NVIDIA Corporation (2024). CUDA Toolkit Documentation 12.

Smith, R. L. (1962). Algorithm 116: complex division. *Comm. of the ACM*, 5(8):435.

Yaglom, I. M. (2014). *Complex numbers in geometry*. Academic Press.