Análise e Geração de Resultados sobre Implementação de Método Runge-Kutta em CUDA*

Nórton Lopes Moreira, Claudio Schepke, Axel Manuel da Silva Cabral

[1]Ciência da Computação – Universidade Federal do Pampa (UNIPAMPA)
Alegrete – RS – Brazil

{nortonmoreira.aluno,claudioschepke,axelcabral.aluno}@unipampa.edu.br

Resumo. *O presente trabalho compreende a análise e avaliação de implementações do método Runge-Kutta em interface de programação paralela CUDA. Foram testadas 3 implementações de segunda e quarta ordem do método. Como resultados, foram feitas medições de performance e qualidade das soluções numéricas das implementações encontradas na literatura.*

1. Introdução

A maioria dos problemas matemáticos não pode ser resolvida em uma sequência finita de operações elementares, conforme demonstrado em [Chatterjee 2011]. Assim, a análise numérica entra como uma alternativa viável para calcular resultados aproximados com precisão suficiente para uso em aplicações científicas e de engenharia, além de colaborar nos estudos sobre programação de modo geral.

Dentre os diversos métodos utilizados para a resolução numérica computacional de equações matemáticas, tem-se o método de Runge-Kutta [Butcher 2016]. Além de cálculos utilizados para áreas como construção civil, por exemplo, o algoritmo pode ser utilizado para comparações e práticas em programação sequencial ou paralela, além de levantar dúvidas em como melhor constituir uma versão de código paralelo.

Assim, o objetivo principal deste trabalho é reunir e analisar implementações de versões de código do Método de Runge-Kutta paralelizados em CUDA. A computação sequencial do método pode ter um tempo computacional expressivo. Assim, usar o poder computacional de GPUs torna-se um fator atrativo, sendo CUDA uma das interfaces de programação de placas gráficas amplamente difundida.

Implementações deste tipo não estão disponibilizadas em pacotes de software clássicos, diferente de outros métodos numéricos, como os de resolução de sistemas lineares. Desta forma, o trabalho reúne algumas implementações encontradas, apresentando resultados de tempo de execução para as mesmas.

2. Algoritmo de Runge-Kutta

O Método de Runge-Kutta serve para a resolução de Equações Diferenciais Ordinárias (EDO) [Butcher 1987]. Uma EDO pode ser definida por $dy/dt = f(t, y)$. O método de Runge-Kutta é representado pela igualdade:

$$u^{i+1} = u^n + \Delta t \sum_{i=1}^{e} b_i k_i$$

*Com suporte de Bolsa PDA/UNIPAMPA Ensino e Pesquisa e PRO-IC 2021

onde,

$$k_i = F(u^n + \Delta t \sum_{i=1}^{e} a_{ij} k_j; t_n + c_i \Delta t)\ i = 1, ..., e$$

com a_{ij}, b_i, c_i constantes próprias do esquema numérico. Caso $a_{ij} = 0$ diz-se que o esquema é explícito.

Cada método de Runge-Kutta consiste em comparar um polinômio de Taylor apropriado para eliminar o cálculo das derivadas, fazendo-se várias avaliações da função a cada passo. Isso se deve porque o método de Runge-Kutta pode ser entendido como um aperfeiçoamento do método de Euler, com uma melhor estimativa da derivada da função [Novotny et al. 2012].

O método de Runge-Kutta pode usar diferentes coeficientes ou pesos para os cálculos. Assim, pode-se utilizar algoritmos de segunda ordem (RK2) e de quarta ordem (RK4), por exemplo, sendo estes os mais populares. Os esquemas destas duas implementações está comparativamente apresentadas abaixo.

(RK2)	(RK4)
$x^1 = f(t, y), y(t0) = y0$	$x^1 = f(t, y), y(t0) = y0$
$yn + 1 = yn + h/2(k1 + k2)$	$yn + 1 = yn + h/6(k1 + 2k2 + 2k3 + k4)$
$tn + 1 = tn + h$	$tn + 1 = tn + h$
$k1 = f(tn, yn)$	$k1 = f(tn, yn)$
$k2 = f(tn + k1, yn + h)$	$k2 = f(tn + h/2, yn + h/2k1)$
	$k3 = f(tn + h/2, yn + h/2k2)$
	$k4 = f(tn + h, yn + hk3)$
inclinação = $tn + (k1 + k2)/2$	inclinação = $(k1 + 2k2 + 2k3 + k4)/6$

3. Metodologia

Para a efetivação do trabalho foram buscados em repositórios de artigos. Entre os trabalhos encontrados, muitos não disponibilizam o código-fonte das implementações. Por causa disso foram pesquisados em repositórios no Github em busca de implementações do método de Runge-Kutta usando CUDA. Os 3 códigos selecionados foram:

Código A[1]: É um algoritmo eficiente, que exibe uma precisão global de quarta ordem, utilizando linguagem C++. A simulação de processos coloca altas demandas em precisão numérica, o que torna atraentes esquemas eficientes de alta ordem. O algoritmo apresentado é uma adaptação para uso de campos vetoriais. Ele exibe a precisão global esperada de quarta ordem para ambos os problemas estudados e é o mais preciso e eficiente dos métodos testados.

Código B[2]: Este código tem uma execução parecida ao do Código 1. Todas as *threads* possíveis do sistema são utilizadas. O programa é eficiente, porém, tem alguns problemas como a forma que ele trata os dados. Para isso, foi utilizada como alternativa o uso de arquivos como binários e, portanto, sempre entrarão em conflito e exigirão do usuário a intervenção a cada mesclagem. Isso faz a aplicação não ser tão boa, porém, não faz ela ser menos eficiente.

[1]Disponível em: https://github.com/rafamanzo/runge-kutta

[2]Disponível em: https://github.com/Gwzlchn/CUDA-RUNGEKUTTA

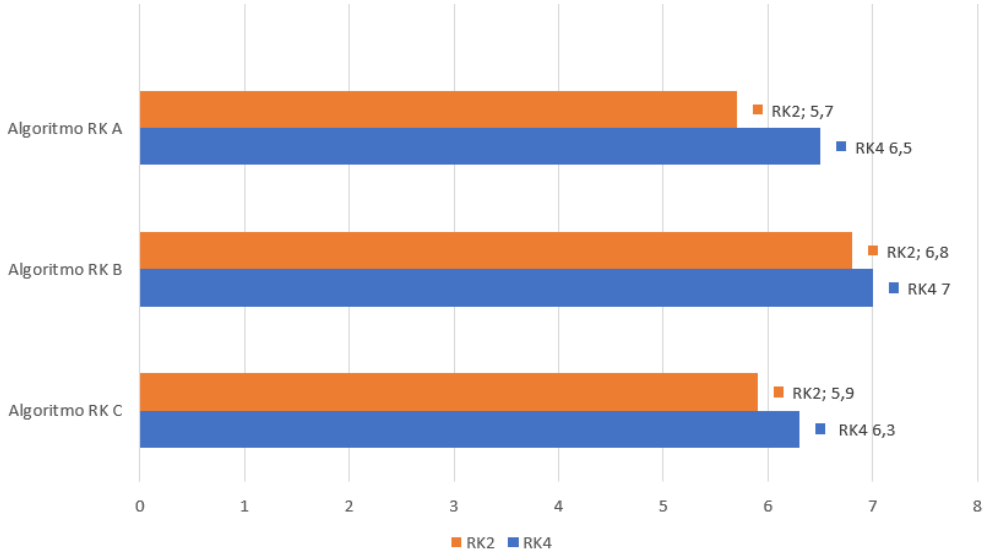

Figura 1. Tempo de execução para RK2 e RK4 de 3 diferentes implementações

Código C[3]: O código foi desenvolvido em linguagem Python. O esquema Runge-Kutta Fehlberg (RKF) foi especialmente desenvolvido para fornecer uma estimativa do principal erro de truncamento local em cada etapa, conhecido como técnica de estimativa de incorporação [Wo et al. 2013]. Dois modelos de EDOs são executados para mostrar a aceleração alcançável em comparação com a implementação da CPU. Foi utilizado inicialmente RK4, depois o EDO do Oscilador de Van der Pol [Dorodnitsyn 1953] e Atractor de Lorenz [Sparrow 2012]. A precisão e eficiência do método implementado é perceptível para cada implementação. A precisão alcançada por ambos os sistemas EDOs é altamente desejável. Portanto, pode-se concluir que a precisão não é um problema no GPU. O resultado pode variar conforme os sistemas ODE variam. O código utiliza aritmética real do tipo float para resolver $dy(i)/dt = F(t, y(1 : NEQN))$ onde são passados os valores iniciais de Y e das derivadas iniciais.

Uma vez identificados os pacotes, estes foram instalados e testados, sendo feitas comparações, a fim de obter-se resultados numéricos e de execução computacional. Em todas as aplicações foi utilizada uma GPU Quadro M5000 de 8 GB de memória RAM, onde conseguiu-se obter diversos resultados.

4. Resultados

Os resultados de tempo de execução obtidos para as três implementações testadas são mostrados na Figura 1. No gráfico é apresentado uma comparação entre as operações de RK2 e RK4 dos 3 códigos escolhidos. Para fim de maior exatidão, 20 testes foram feitos para obter-se um tempo médio de cada uma das três aplicações.

É importante lembrar que os três códigos resolvem problemas distintos embora os tempos de processamento sejam relativamente parecido entre as 3 implementações. Também observa-se RK2 possui um tempo de computação menor que RK4 para todas as implementações, uma vez que o número de cálculos a serem computados é menor.

[3]Disponível em: https://github.com/HajimeKawahara/grkf45

O código A foi executado com dimensões igual a $x = 512$, $y = 512$ e $z = 128$ usando *lines*, $x = 256$, $y = 256$ e $z = 256$ usando *gaussian* e $x = 128$, $y = 128$ e $z = 20$. usando *rotation* como campos vetoriais. O código B utiliza $x = 1000.000$ e $y = 9$, não utilizando um valor de $z =$. O código C foi testado com uma dimensão $x = 10.000$.

5. Considerações Finais

Neste trabalho foram estudados e comparados os dados provindos de implementações de Runge-Kutta paralelizados em CUDA, visando experimentar códigos diferentes e aplicações distintas aplicadas em várias formulações matemáticas. Foram apresentados, por meio de testes, resultados numéricos e visuais. Como resultado, obteve-se que a implementação de aplicações Runge-Kutta de segunda e quarta ordem, podendo assim fazer cálculos maiores em menor tempo.

Em cálculos sequenciais pesados é possível obter uma grande melhora utilizando programação paralela CUDA. A placa gráfica, por já ter um número grande de unidades de computação para carregamento, agiliza muito o tempo de execução e processamento do código, além de ter um impulso na eficácia do desempenho. Como trabalhos futuros, pretende-se aplicar estas implementações de Runge-Kutta. Neste caso poderão ser incluídas versões paralelas implementadas também em outras interfaces de programação paralela.

Referências

Butcher, J. (2016). *Numerical Methods for Ordinary Differential Equations*. Wiley.

Butcher, J. C. (1987). *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience.

Chatterjee, B. C. B. . D. (2011). *Numerical Method and Programming (WBUT), 2nd Edition*. Vikas Publishing House.

Dorodnitsyn, A. (1953). *Asymptotic Solution of Van Der Pol's Equation*. American Mathematical Society translations. American Mathematical Society.

Novotny, J., Green, M., and Boston, R. (2012). *Mathematical Modeling in Nutrition and the Health Sciences*. Advances in Experimental Medicine and Biology. Springer US.

Sparrow, C. (2012). *The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors*. Applied Mathematical Sciences. Springer New York.

Wo, M. S., Gobithaasan, R., and Miura, K. (2013). GPU Acceleration of Runge Kutta-Fehlberg and Its Comparison with Dormand-Prince Method. volume 1605.