UTILIZAÇÃO DE PLACAS CUDA PARA PARALELIZAÇÃO DE PROBLEMAS NUCLEARES

Davi Ferreira Fernandes

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Nuclear, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Nuclear.

Orientadores: Alan Miranda Monteiro de Lima

Adilson Costa da Silva

Rio de Janeiro

Outubro de 2021

UTILIZAÇÃO DE PLACAS CUDA PARA PARALELIZAÇÃO DE PROBLEMAS NUCLEARES

Davi Ferreira Fernandes

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA NUCLEAR.

Orientadores: Alan Miranda Monteiro de Lima
Adilson Costa da Silva

Aprovada por: Prof. Alan Miranda Monteiro de Lima
Prof. Adilson Costa da Silva
Prof. Victor Henrique Cabral Pinheiro
Prof. Zelmo Rodrigues de Lima

RIO DE JANEIRO, RJ -BRASIL

OUTUBRO DE 2021

Fernandes, Davi Ferreira

Utilização de Placas Cuda para Paralelização de Problemas Nucleares/Davi Ferreira Fernandes.- Rio de Janeiro: UFRJ/COPPE, 2021.

XIII, 85.:il; 29,7 cm.

Orientadores: Alan Miranda Monteiro de Lima

Adilson Costa da Silva

Dissertação (mestrado) - UFRJ / COPPE/ Programa de Engenharia Nuclear.

Referências Bibliográficas: p.83-85.

1.Paralelização de Problemas Nucleares, Cinética Pontual e Método Runge-Kutta 4ª Ordem. 2. CUDA. 3. Computação Paralela I. Lima, Alan Miranda Monteiro de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Nuclear. III.Título.

Dedicatória

Dedico este trabalho à minha querida Mãe, ao meu grande pai, a minha grande amiga Rosana, aos meus professores orientadores Alan e Adilson que tanto me ajudaram ao longo do curso e ao meu grande amigo Marcelo que me ajudou desde o início do trabalho até o fim.

Agradecimentos

A Deus, que mesmo nos momentos mais difíceis da minha vida não me deixoudesistir dessa caminhada árdua.

Agradeço aos meus pais pelo amor, carinho, paciência, investimento e dedicação dados a mim.

Agradeço a todos os cientistas que me serviram de referência e inspiração para a realização deste trabalho.

Agradeço a minha grande amiga Rosana que sempre esteve ao meu lado me orientando e ajudando nos momentos mais difíceis da minha vida. Agradeço ao meu amigo Marcelo que tanto me ajudou nesse trabalho. Agradeço ao meu amigo de graduação Leandro. Agradeço ao meu amigo Carlos pelas corridas e treinos. Agradeço ao meu amigo Arthur pela sua amizade. Agradeço ao meu amigo Sérgio pelos conselhos de vida e companhia nas passeatas em prol do Brasil.

Agradeço as Artes Marciais que moldaram o meu espírito e caráter. Agradeço aos meus professores de judô, karatê e jiu-jitsu que já passaram pela minha vida e contribuíram no meu aprendizado.

Agradeço aos meus professores orientadores Alan Miranda Monteiro de Lima e Adilson Costa da Silva que me ajudaram nessa caminhada, pois sem eles eu não teria conseguido.

Resumo da Dissertação apresentada à COPPE / UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc).

UTILIZAÇÃO DE PLACAS CUDA PARA PARALELIZAÇÃO DE PROBLEMAS NUCLEARES

Davi Ferreira Fernandes

Outubro/2021

Orientadores: Alan Miranda Monteiro de Lima

Adilson Costa da Silva

Programa: Engenharia Nuclear

Nesta dissertação será apresentada uma análise qualitativa e quantitativa para avaliar o desempenho de diferentes linguagens computacionais, tais como: Fortran, Python e C com aceleração via GPU e CPU. Para isso, utilizaremos as equações da cinética pontual de reatores para seis grupos de precursores de nêutrons. Para resolver numericamente este sistema de equações, usaremos o Método de Runge-Kutta de 4ª Ordem com um passo no tempo da ordem do tempo de vida dos nêutrons prontos. Assim, poderemos avaliar os tempos computacionais gastos em cada simulação e verificar a eficiência na aceleração via GPU e CPU em cada linguagem utilizada.

Abstract of Dissertation presented to COPPE / UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc).

USE OF CUDA PLATES FOR PARALLELIZATION OF NUCLEAR PROBLEMS

Davi Ferreira Fernandes

October/2021

Advisors: Alan Miranda Monteiro de Lima

Adilson Costa da Silva

Department: Nuclear Engineering

In this dissertation, a qualitative and quantitative analysis will be presented to evaluate the performance of different computational languages, such as: Fortran, Python and C with acceleration by GPU and CPU. For this, we will use the reactor point kinetics equations for six groups of neutrons precursors. To numerically solve this system of equations, we will use the 4th Order Runge-Kutta Method with a step in time on the order of the prompt neutron lifetime. Thus, we will be able to evaluate the computational elapsed time in each simulation and verify the efficiency of acceleration by GPU and CPU in each language used.

SUMÁRIO

LISTA DE FIGURAS

CAPÍTULO 1

Introdução

Nesta dissertação, serão usadas as equações da cinética pontual de reatores para avaliar o desempenho de diferentes linguagens computacionais, como por exemplo: Fortran, Python e C com aceleração por meio de GPU e CPU e desta forma, medir os tempos de execução e ganhos computacionais em cada simulação. Para esse fim, usaremos o Método de Runge-Kutta de 4ª Ordem para resolver numericamente este sistema de equações.

Na operação de uma usina nuclear é de suma importância monitorar os parâmetros que envolvem o núcleo do reator. A potência é um dos parâmetros de importante relevância. Um modelo simples que se consegue determinar a potência, considerando-se certa variação da reatividade ao longo do tempo, é o modelo da cinética pontual de reatores (NUNES, 2006).

É importante destacar que, para avaliar o desempenho das acelerações usando códigos paralelos via GPU ou CPU, para as diferentes linguagens de programação utilizadas nesta dissertação, tornou-se necessário escrever um código em linguagem Fortran, C e Python em linguagem sequencial. Esse código em linguagem sequencial serviu como base para o desenvolvimento dos códigos paralelos, como por exemplo, a versão do código em linguagem C paralela conhecida como C CUDA (NVIDIA CORPORATION, 2014) (BUCK, 2007).

É importante diferenciar os dois paradigmas de programação utilizados neste trabalho. A programação mais usual e padrão, chamada de programação sequencial e a programação paralela.

A programação sequencial é basicamente uma série de instruções sequenciais que são executadas em um único processador (Figura 1.1). Nesse sentido, utilizam-se algoritmos sequenciais, onde esses algoritmos são executados em todos os passos na

sequência em que eles surgem, desde o primeiro passo até o último, durante um intervalo de tempo.

Já a programação paralela é vista como um conjunto de pedaços ou partes que são executadas concorrentemente (Figura 1.2). Nesse sentido, cada pedaço é igualmente feito por uma série de instruções sequenciais, porém em conjunto e são executados simultaneamente em vários processadores, com o objetivo de diminuir o tempo de processamento. Desse modo, o paralelismo ou simplesmente, computação paralela, tem como principal objetivo a redução do tempo de execução das tarefas de um determinado problema e também solucionar problemas mais complexos e com maior dimensão. Segundo (ALMEIDA, 2009), os algoritmos paralelos, que são baseados em GPU,são soluções viáveis em problemas com alto custo computacional. Assim, a computação paralela é muito importante para computação de modo geral, pois auxilia a resolver problemas que exigem alto poder computacional. Logo, comparando-se com a programação sequencial, a programação paralela aumenta o desempenho, eficiência e velocidade dos códigos de resolução de problemas computacionais e suas aplicações.

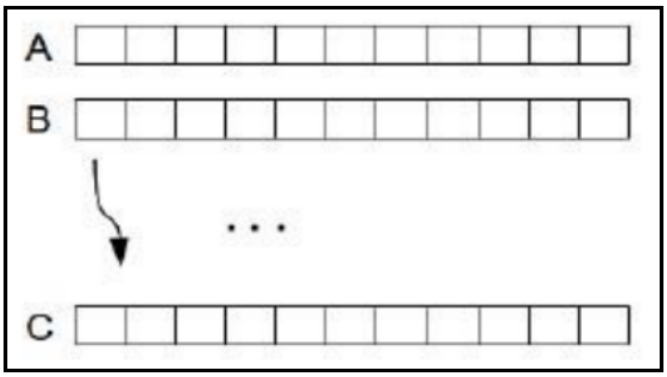

Figura 1.1 - Programação Sequencial (NVIDIA CORPORATION, 2014).

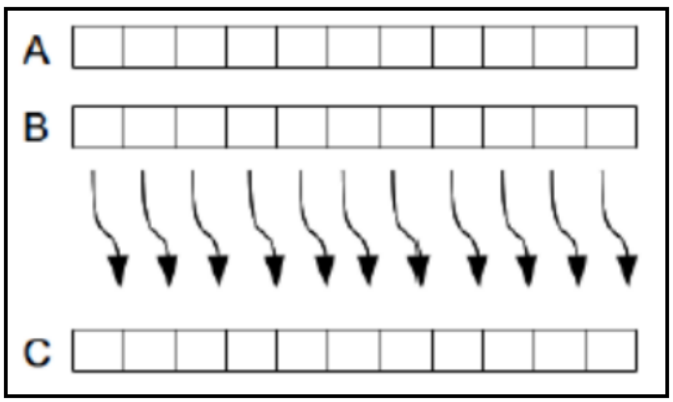

Figura 1.2 - Programação Paralela (NVIDIA CORPORATION, 2014).

No capítulo 2 desta dissertação, serão deduzidas as equações da cinética pontual de reatores, equações essas, que são deduzidas a partir da equação da difusão dependente do tempo, levando em conta a contribuição dos precursores de nêutrons atrasados. Assim como, será apresentado um método numérico de solução para essas equações conhecido como método de Runge-Kutta de 4ª ordem.

No capítulo 3 serão discutidas as diferenças entre CPU e GPU, a taxonomia de Flynn, a definição de CUDA, algumas técnicas de paralelização e a comparação entre CUDA e OpenCL.

No capítulo 4 serão apresentadas implementações computacionais do sistema Runge-Kutta de 4ª ordem nas linguagens: Fortran, C, C CUDA, PYTHON e PYTHON CUDA. Assim como, será mostrado um panorama geral de cada uma dessas linguagens utilizadas nesta dissertação.

No capítulo 5 será feita a validação e a análise de resultados, por meio de tabelas e gráficos, onde será feita a comparação do método escrito em linguagem sequencial e em linguagem paralela.

No capítulo 6, discutiremos os resultados obtidos e apresentaremos as conclusões e sugestões referentes a este trabalho.

CAPÍTULO 2

Fundamentação Teórica

2.1 – Solução Numérica das Equações da Cinética Pontual de Reatores usando o Método de Runge-Kutta

Nesta seção, deduziremos as equações da Cinética Pontual de Reatores para seis grupos de precursores de nêutrons, a partir da equação da difusão de nêutrons dependente do tempo. Por fim, apresentaremos o método de Runge-Kutta de 4ª ordem que será utilizado para resolver numericamente este sistema de equações.

2.1.1 - Equações da Cinética Pontual de Reatores

O modelo de Cinética Pontual de Reatores é deduzido a partir da equação da difusão dependente do tempo, levando em conta a contribuição dos nêutrons atrasados (HETRICK, 1971). A equação da difusão dependente do tempo para seis grupos de precursores de nêutrons sem o termo de fonte externa, é dada por:

$$\frac{\partial N(r,t)}{\partial t} = Dv\nabla^2 N(r,t) - \Sigma_a vN(r,t) + (1-\beta)\Sigma_a k_\infty vN(r,t) + \sum_{i=1}^{6}\lambda_i C_i(r,t), \qquad (2.1)$$

onde os termos da Eq.(2.1) são:

- $N(r,t)dV$= É o número de nêutrons em um elemento de volume dVna posição r no instante de tempo t.
- $Dv\nabla^2 N(r,t)dV$= É a taxa de nêutrons difundidos, isto é, entrando e saindo, em dVna posição r no instante de tempo t.
- $\Sigma_a vN(r,t)dV$= É a taxa de nêutrons absorvidos em dV na posição r no tempo t.

- D = É o coeficiente de difusão de nêutrons.
- v = É a velocidade do nêutron.
- Σ_a = É seção de choque macroscópica de absorção de nêutrons.
- β = É a fração total de nêutrons retardados, que são produzidos por decaimento radioativo de certos produtos de fissão e são chamados de precursores de nêutrons retardados.
- $(1-\beta)\Sigma_a k_\infty v N(r,t)$ = É a contribuição dos nêutrons prontos.
- k_∞ = É o fator de multiplicação para um meio infinito.
- λ_i = É a constante de decaimento radioativo.
- $C_i(r,t)$ = É a concentração do i-ésimo tipo de precursores.
- $\sum_{i=1}^{6} \lambda_i\, C_i(r,t)$ = É a contribuição dos nêutrons retardados.

Notem que estamos assumindo que tanto os parâmetros cinéticos quanto os parâmetros nucleares são independentes da posição e do tempo. A equação da difusão pode ser considerada como uma aproximação para a análise do comportamento dinâmico de vários reatores, em função da densidade de nêutrons $N(r,t)$ no núcleo do reator.

O fator de multiplicação de um meio infinito k_∞ é definido como sendo,

$$k_\infty = \frac{Número\ de\ Nêutrons\ Produzidos}{Número\ de\ Nêutrons\ Absorvidos} \quad , \tag{2.2}$$

onde neste caso, desprezamos o termo de fuga espacial, uma vez que estamos assumindo que o meio é infinito.

Definindo β_i como a fração de nêutrons retardados para o i-ésimo precursor, então temos que a fração total é dada por, $\beta = \sum_{i=1}^{6} \beta_i$. Considerando que os fragmentos de fissão percorrem distâncias muito pequenas, pois são muito pesados. Logo, serão considerados todos os fragmentos de fissão na mesma posição para um determinado instante de tempo e reação de fissão. Com isso,temos a Eq.(2.3):

$$\frac{\partial C_i(r,t)}{\partial t} = \beta_i k_\infty \Sigma_a v N(r,t) - \lambda_i C_i(r,t), \qquad i = 1, \ldots, 6. \tag{2.3}$$

As Eqs.(2.1) e (2.3) constituem um sistema de 7 equações diferenciais parciais dependentes do espaço e do tempo, no qual a densidade de nêutrons e as concentrações de precursores de nêutrons podem ser determinados. Há uma relação direta entre o fluxo de nêutrons e a densidade de nêutrons, dada por,

$$\Phi(r,t) = vN(r,t), \tag{2.4}$$

bem como, com a potência nuclear, tal que

$$P(t) = w\Sigma_f vN(r,t)dV, \tag{2.5}$$

onde$w$ e Σ_f, são respectivamente, a energia liberada na fissão e a seção de choque macroscópica de fissão. Desta forma, as Eqs.(2.1) e (2.3) podem ser utilizadas para análise de transientes locais, pois levam em conta a dependência espacial. Para exemplificar, podemos citar a queda ou retirada de bancos de barras de controle do núcleo.

Existe um número grande de isótopos provenientes dos produtos de fissão que decaem por emissão de nêutrons, logo são membros da família de precursores de nêutrons retardados. Com o objetivo de modelar seu efeito na cinética de nêutrons, é necessário agrupar os nêutrons em grupo de seis ($C_1(r,t)$, $C_2(r,t)$, $C_3(r,t)$, $C_4(r,t)$, $C_5(r,t)$, $C_6(r,t)$) conforme suas $T_{\frac{1}{2}}$ (meia-vida). A tabela 3 apresenta os valores típicos de precursores para U^{235}, onde $\beta \equiv \sum_{i=1}^{6} \beta_i = 0.007$. Logo, os precursores de nêutrons retardados contribuem apenas com 0.7% dos nêutrons.

Tabela 2.1 - Coeficientes Típicos de precursores para $\boldsymbol{U^{235}}$

Grupo	**1**	**2**	**3**	**4**	**5**	**6**
$T_{\frac{1}{2}}[s]$	54.5785	21.8658	6.0274	2.2288	0.4951	0.1791
$\lambda_i[s^{-1}]$	0.0127	0.0317	0.115	0.311	1.4	3.87
$\frac{\beta_i}{\beta}$	0.038	0.2130	0.1880	0.4070	0.1280	0.0260
β_i	0.000266	0.001491	0.001316	0.002849	0.000896	0.000182

Para obtermos as equações da cinética pontual, a partir da equação da difusão de nêutrons dependente do tempo, podemos assumir que $N(r,t)$e $C_i(r,t)$ são separáveis no tempo e espaço. A separabilidade só é válida se o reator estiver muito próximo do estado crítico e se não existir nenhuma perturbação localizada. Assim, teremos as Eqs.(2.6) e (2.7):

$$N(r,t) = f(r)n(t) \tag{2.6}$$

$$C_i(r,t) = g_i(r)c_i(t) \tag{2.7}$$

Substituindo as Eqs.(2.6) e (2.7) na Eq.(2.3), teremos:

$$\frac{dc_i(t)}{dt} = \beta_i k_\infty \Sigma_a v \frac{f(r)}{g_i(r)} n(t) - \lambda_i c_i(t) \tag{2.8}$$

A fração $\frac{f(r)}{g_i(r)}$ é independente do tempo. Para que a Eq.(2.8) seja independente da posição, temos que assumir que $f(r)$ e $g_i(r)$ tenham a mesma forma, isto é, sejam proporcionais. Assumindo $g_i(r) = f(r)$, para todo $i = 1, 2, 3, \ldots, 6$, a Eq.(2.8) fica,

$$\frac{dc_i(t)}{dt} = \beta_i k_\infty \Sigma_a v n(t) - \lambda_i c_i(t), \qquad i = 1,..,6. \tag{2.9}$$

Substituindo as Eqs. (2.6) e (2.7) na Eq.(2.1), vem

$$\frac{dn(t)}{dt} = Dv\frac{\nabla^2 f(r)}{f(r)} n(t) - \Sigma_a v n(t) + (1-\beta)k_\infty \Sigma_a v n(t) + \sum_{i=1}^{6} \lambda_i \frac{g_i(r)}{f(r)} c_i(t) \tag{2.10}$$

A razão $\frac{g_i(r)}{f(r)}$ já foi considerada igual a 1 na Eq.(2.8). Agora, buscaremos eliminar o termo $\frac{\nabla^2 f(r)}{f(r)}$. Levando em consideração que $f(r)$ satisfaz a equação de Helmholtz, temos que

$$\nabla^2 f(r) + B^2 f(r) = 0, \tag{2.11}$$

logo,

$$\frac{\nabla^2 f(r)}{f(r)} = -B^2. \tag{2.12}$$

Substituindo a Eq.(2.12) em (2.10), vem

$$\frac{dn(t)}{dt} = -DvB^2 n(t) - \Sigma_a v n(t) + (1-\beta)k_\infty \Sigma_a v n(t) + \sum_{i=1}^{6} \lambda_i c_i(t), \tag{2.13}$$

onde agrupando os termos, temos que

$$\frac{dn(t)}{dt} = (-DvB^2 - \Sigma_a v + (1-\beta)k_\infty \Sigma_a v) n(t) + \sum_{i=1}^{6} \lambda_i c_i(t). \tag{2.14}$$

Introduzindo novas notações na Eq.(2.14), dadas por,

$$l_\infty \equiv \frac{1}{v\Sigma_a} \tag{2.15}$$

$$L^2 \equiv \frac{D}{\Sigma_a} \tag{2.16}$$

Onde l_∞ é o tempo característico de absorção e L é o comprimento de difusão de nêutrons, que corresponde a distância média percorrida por um nêutron antes de ser absorvido, temos que:

$$\frac{dn(t)}{dt} = \frac{(1-\beta)k_\infty - (1+L^2B^2)}{l_\infty} n(t) + \sum_{i=1}^{6} \lambda_i c_i(t). \tag{2.17}$$

O fator de multiplicação k e o tempo de vida dos nêutrons l é definido como:

$$k \equiv \frac{k_\infty}{1+L^2B^2} \tag{2.18}$$

$$l \equiv \frac{l_\infty}{1+L^2B^2} \tag{2.19}$$

Substituindo as Eqs.(2.18) e (2.19) nas Eq.(2.9) e (2.17), temos:

$$\frac{dn(t)}{dt} = \frac{k-1-\beta k}{l} n(t) + \sum_{i=1}^{6} \lambda_i c_i(t) \tag{2.20}$$

$$\frac{dc_i(t)}{dt} = \frac{\beta_i k}{l} n(t) - \lambda_i c_i(t), \qquad i = 1,..,6. \tag{2.21}$$

Usando as definições de reatividade $\rho(t)$ e do tempo médio de geração de nêutrons Λ em um reator:

$$\rho(t) \equiv \frac{k-1}{k} \tag{2.22}$$

$$\Lambda \equiv \frac{l}{k} \tag{2.23}$$

Substituindo as Eqs.(2.22) e (2.23) nas Eqs.(2.20) e (2.21) obtemos finalmente as equações da cinética pontual de reatores para 6 grupos de precursores de nêutrons atrasados,

$$\frac{dn(t)}{dt} = (\frac{\rho(t)-\beta}{\Lambda})n(t) + \sum_{i=1}^{6} \lambda_i c_i(t) \tag{2.24}$$

$$\frac{dc_i(t)}{dt} = \frac{\beta_i}{\Lambda} n(t) - \lambda_i c_i(t), \qquad i = 1,..,6, \tag{2.25}$$

Onde

- $n(t)$ = É a densidade de nêutrons.
- $c_i(t)$= É o i-ésimo termo da concentração de precursores de nêutrons atrasados.
- $\rho(t)$= É a reatividade.
- β= É a fração total de nêutrons atrasados, que é dada por $\beta \equiv \sum_{i=1}^{6} \beta_i$.
- Λ=É o tempo médio de geração de nêutrons.
- λ_i= É o i-ésimo termo da constante de decaimento.
- β_i= É o i-ésimo termo da fração de nêutrons atrasados.

Essas equações constituem um conjunto de sete equações diferenciais no tempo, onde o único parâmetro cinético dependente do tempo é a reatividade $\rho(t)$. Podemos notar que os parâmetros cinéticos β, Λ e etc, das Eqs.(2.24) e (2.25) estão associados aos parâmetros nucleares da equação da difusão de nêutrons dependente do tempo, tais como seção de choque e o coeficiente de difusão. Logo, os efeitos físicos associados às seções de choque são incorporados nos parâmetros cinéticos. Essas equações, embora simples, permitem investigar a densidade de nêutrons no reator, bem como a potência nuclear em função da mudança de reatividade $\rho(t)$ inserida no núcleo do reator.

Nas Eqs.(2.24) e (2.25) é possível ainda constatar que a densidade de nêutrons cresce quando a concentração de nêutrons precursores cresce e vice-versa. Além do mais, a concentração de precursores de nêutronscresce com a fração de nêutrons atrasados oriundos da fissão nuclear e diminui com o seu decaimento.

As equações da cinética pontual de reatores são importantes, pois constituem um modelo simples e eficiente para o estudo de situações dinâmicas do reator nuclear (NUNES, 2006). Desta forma, é importante conhecer as consequências da introdução da reatividade no núcleo do reator, como por exemplo: durante a partida do reator, retirada ou inserção de bancos de barras de controle, desligamento, acidentes entre outros. Logo, podemos controlar o núcleo do reator com riscos bem menores.

2.1.2 - Método de Runge-Kutta - RK4

O método de Runge-Kutta (FREITAS, 2000) tem como objetivo a resolução de equações diferenciais ordinárias, associada a um problema de valor inicial, tal que

$$\frac{dy(t)}{dt} = f\big(t, y(t)\big), \qquad y(0) = y_0 \tag{2.26}$$

É importante destacar que, o método de Runge-Kutta também pode ser utilizado para solução numérica de sistemas de equações diferenciais ordinárias, associada a problema de valor inicial. A escolha do método de Runge-Kutta de quarta ordem é devido ao seu alto grau de precisão, ou seja, devido ao erro de truncamento ser muito

pequeno, da ordem de $E = O(\Delta t^4)$. Esse método também é conhecido como RK4 e possui médias ponderadas dos valores de uma determinada função *f(t)* em diferentes intervalos $[t_j, t_j + \Delta t]$(SILVA, 2007).

Há outros métodos que são considerados casos particulares de uma família de métodos numéricos para resolução de equações diferenciais ordinárias denominados Métodos de Runge-Kutta, tais como: Os métodos de Euler e Heun. O método de Euler é um método de Runge-Kutta de primeira ordem com erro de truncamento global $E = O(\Delta t)$, enquanto o método de Heun é um método de Runge-Kutta, mas de segunda ordem com erro de truncamento global $E = O(\Delta t^2)$. O método mais eficiente dessa família de métodos de Runge-Kutta é o Método de Runge-Kutta de 4ª Ordem.

Este método possui uma boa precisão para novas funções num intervalo $[t_j, t_{j+1}]$ e usa relações de recorrência, tal que $k_1$entra na equação para k_2 e assim sucessivamente, ou seja,

$$k_1 = f(t_j, y_j) \tag{2.27}$$

$$k_2 = f(t_j + \frac{\Delta t}{2}, y_j + \frac{\Delta t}{2} k_1) \tag{2.28}$$

$$k_3 = f(t_j + \frac{\Delta t}{2}, y_j + \frac{\Delta t}{2} k_2) \tag{2.29}$$

$$k_4 = f(t_j + \Delta t, y_j + \Delta t\, k_3) \tag{2.30}$$

Para o método de Runge-Kutta de 4ª ordem, temos:

$$y_{j+1} = y_j + \frac{\Delta t}{6}[k_1 + 2k_2 + 2k_3 + k_4], \text{com} \quad t_{j+1} = t_j + \Delta t \tag{2.31}$$

Os cálculos dos k_{is} são feitos determinando primeiramente os pontos e posteriormente é calculada a função em cima desses pontos. A figura 2.1 mostra os pontos nos quais as funções são determinadas.

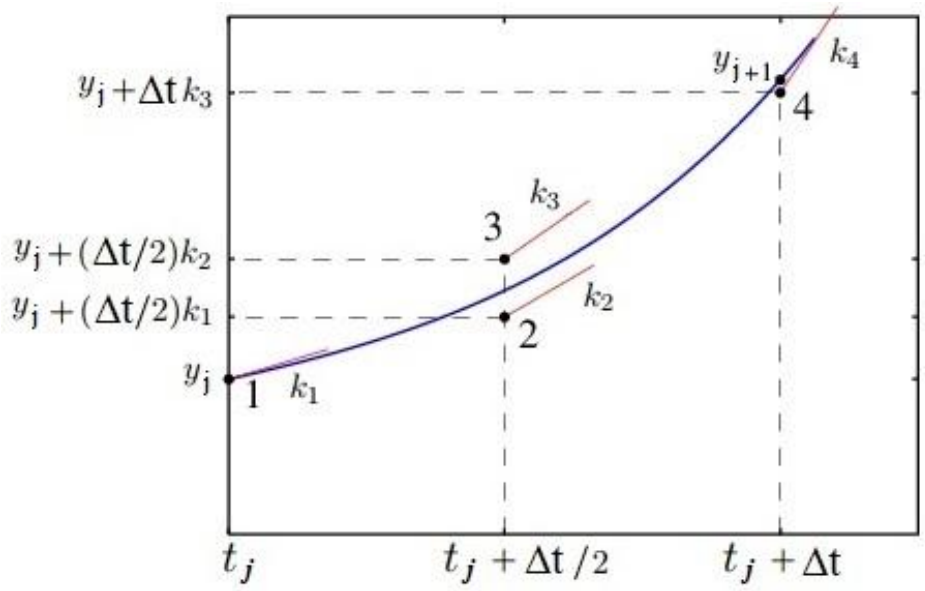

Figura 2.1 - Método de Runge-Kutta de 4ª ordem

Os cálculos dos k_i, $i = 1,..,4$, no método RK4 para obtenção da solução numérica das equações da cinética pontual de reatores, dada pelas Eqs.(2.24) e (2.25), ficam:

Cálculo de k_1:

Utilizaremos os valores previamente conhecidos em t_j tanto para a densidade de nêutrons, quanto para a concentração de precursores, tal que:

$$k_{1D} = \left(\frac{\rho-\beta}{\Lambda}\right) n(t_j) + \sum_{i=1}^{6} \lambda_i c_i(t_j) \tag{2.32}$$

$$k_{1Ci} = \frac{\beta_i}{\Lambda} n(t_j) - \lambda_i c_i(t_j), \qquad i = 1,..,6, \tag{2.33}$$

Cálculo de k_2:

Para o instante de tempo $t_{j+\frac{1}{2}}$, calcularemos a densidade de nêutrons e a concentração de precursores em função de k_1.

$$n\left(t_{j+\frac{1}{2}}\right) = n(t_j) + \frac{\Delta t}{2} k_{1D} \tag{2.34}$$

$$c_i\left(t_{j+\frac{1}{2}}\right) = c_i(t_j) + \frac{\Delta t}{2} k_{1Ci} \tag{2.35}$$

Uma vez conhecido $n\left(t_{j+\frac{1}{2}}\right)$ e $c_i\left(t_{j+\frac{1}{2}}\right)$, podemos calcular k_2, ou seja,

$$k_{2D} = \left(\frac{\rho-\beta}{\Lambda}\right) n\left(t_{j+\frac{1}{2}}\right) + \sum_{i=1}^{6} \lambda_i c_i\left(t_{j+\frac{1}{2}}\right) \tag{2.36}$$

$$k_{2Ci} = \frac{\beta_i}{\Lambda} n\left(t_{j+\frac{1}{2}}\right) - \lambda_i c_i\left(t_{j+\frac{1}{2}}\right), \qquad i = 1,..,6, \tag{2.37}$$

Cálculo de k_3:

Para o instante de tempo $t_{j+\frac{1}{2}}$, calcularemos a densidade de nêutrons e a concentração de precursores em função de k_2, tal que

$$n\left(t_{j+\frac{1}{2}}\right) = n(t_j) + \frac{\Delta t}{2} k_{2D} \tag{2.38}$$

$$c_i\left(t_{j+\frac{1}{2}}\right) = c_i(t_j) + \frac{\Delta t}{2} k_{2Ci} \tag{2.39}$$

Analogamente, uma vez conhecido $n\left(t_{j+\frac{1}{2}}\right)$ e $c_i\left(t_{j+\frac{1}{2}}\right)$, podemos calcular k_3, ou seja,

$$k_{3D} = \left(\frac{\rho-\beta}{\Lambda}\right) n\left(t_{j+\frac{1}{2}}\right) + \sum_{i=1}^{6} \lambda_i c_i\left(t_{j+\frac{1}{2}}\right) \tag{2.40}$$

$$k_{3Ci} = \frac{\beta_i}{\Lambda} n\left(t_{j+\frac{1}{2}}\right) - \lambda_i c_i\left(t_{j+\frac{1}{2}}\right), \qquad i = 1,..,6, \tag{2.41}$$

Cálculo de k4:

Para o instante de tempo t_{j+1}, calcularemos a densidade de nêutrons e a concentração de precursores em função de k_3.

$$n(t_{j+1}) = n(t_j) + \Delta t\, k_{3D} \tag{2.42}$$

$$c_i(t_{j+1}) = c_i(t_j) + \Delta t\, k_{3Ci} \tag{2.43}$$

Uma vez conhecido $n(t_{j+1})$ e$c_i(t_{j+1})$, podemos calcular k_4 da seguinte forma,

$$k_{4D} = \left(\frac{\rho-\beta}{\Lambda}\right) n(t_{j+1}) + \sum_{i=1}^{6} \lambda_i c_i(t_{j+1}) \tag{2.44}$$

$$k_{4Ci} = \frac{\beta_i}{\Lambda} n(t_{j+1}) - \lambda_i c_i(t_{j+1}), \qquad i = 1,..,6, \tag{2.45}$$

Com os $k_i,\ i = 1,..,4$, determinados, podemos usar a Eq.(2.31) e determinar a densidade de nêutrons e a concentração de precursores, usando o método RK4 no instante t_{j+1}, tal que:

$$n(t_{j+1}) = n(t_j) + \frac{\Delta t}{6}[k_{1D} + 2k_{2D} + 2k_{3D} + k_{4D}], \tag{2.46}$$

e

$$c_i(t_{j+1}) = c_i(t_j) + \frac{\Delta t}{6}[k_{1Ci} + 2k_{2Ci} + 2k_{3Ci} + k_{4Ci}]. \tag{2.47}$$

Neste momento, cabe destacar que a escolha de usar essa estrutura de codificação, está associada ao fato de podermos otimizar os laços envolvendo a concentração de precursores e assim, poder otimizar os cálculos. Uma outra, possibilidade de resolver este problema, seria utilizar uma formulação matricial e realizar cálculos envolvendo vetores e matrizes. No entanto, ficaríamos presos às funções intrínsecas da linguagem utilizadas sem poder otimizar esses cálculos.

CAPÍTULO 3

Computação Paralela

A computação paralela é um tipo de computação em que inúmeros cálculos são executados simultaneamente, onde um problema de grande porte é dividido em problemas menores, que podem então, serem resolvidos paralelamente (Figura 3.1). Devido às limitações físicas dos processadores, a computação paralela se tornou predominante nas arquiteturas dos computadores atuais.

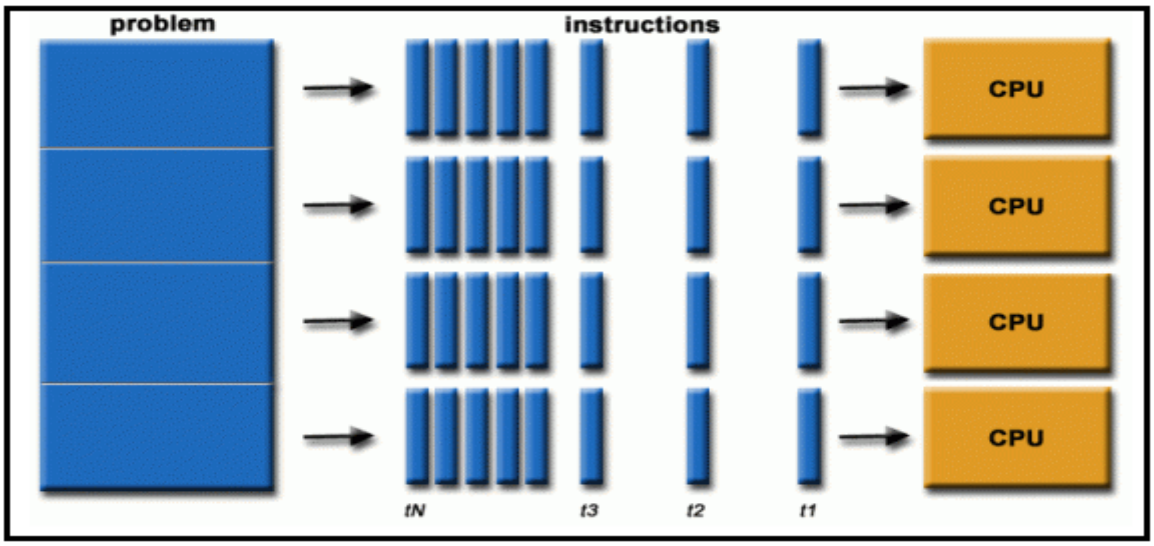

Figura 3.1 - Computação Paralela (MORAIS, 2017).

3.1 - Diferença entre CPU e GPU

Na década de 80, surgem os aceleradores gráficos não programáveis oriundos da indústria de jogos. No início dos anos 2000, surge o primeiro acelerador gráfico programável limitado, porém com uma grande largura de banda da memória dos chips em relação a CPU. E finalmente, com o desenvolvimento dos aceleradores gráficos, surgem as GPUs (GLASKOWSKY, 2009).

Com o avanço da tecnologia, vieram mudanças arquiteturais na computação e consequentemente a evolução do *hardware*. Inicialmente a arquitetura de *hardware* em modelo de CPU, central única de processamento, que é composta basicamente por três componentes: ULA ou Unidade Lógica Aritmética que executa as operações lógicas e aritméticas; A UC ou Unidade de Controle que busca, decodifica e executa as instruções e Registradores que armazenam dados para o processamento.

As CPUs utilizam uma arquitetura orientada ao processamento paralelo, com o objetivo de aumentar seu próprio desempenho. Porém, as CPUs não conseguem alcançar o mesmo desempenho das GPUs, devido às diferenças entre suas arquiteturas.

A GPU é a evolução da CPU, pois utiliza um número maior de transistores para colocar uma quantidade maior de ULAs, possibilitando assim, um número maior de operações lógicas e aritméticas e consequentemente aumentando o poder computacional. A GPU permite um número maior de cálculos ao mesmo tempo e possui um controle de fluxo de forma simples sobre um conjunto de dados. A figura 3.2 mostra a diferença entre a CPU e a GPU, cuja GPU possui um número maior de Unidade de controle, memória cache e Unidades Lógicas Aritméticas (ULAs).

Figura 3.2 - Arquitetura da CPU e GPU (KIRK, D; HWU, W, 2016).

Essa diferença de arquiteturas entre as duas é devido aos seus propósitos distintos. A CPU foi desenhada para ser muito mais flexível, para ser capaz de lidar com

inúmeras tarefas distintas, com o objetivo de manter um equilíbrio entre o poder de processamento e a funcionalidade de propósito geral. A GPU não dá tanta ênfase a flexibilidade, pois prioriza o alto poder de processamento, com a capacidade de executar uma grande quantidade de cálculos paralelos.

O paralelismo tem como métricas de desempenho: o tempo de execução, o espaço de memória requerido, a vazão computacional e a aceleração. Tendo como principais vantagens: a diminuição do tempo de processamento, o melhor desempenho lógico matemático e uma melhor vazão computacional. O paralelismo possui como desvantagens: A concorrência, que identifica as partes da computação a serem executadas em simultâneo; A comunicação e sincronização, que desenha o fluxo de informação de modo que a computação possa ser executada em simultâneo pelos diversos processadores evitando o *deadlocks* e *race conditions*; O balanceamento de carga, que distribui de forma equilibrada e eficiente as diferentes partes da computação pelos diversos processadores, para manter os processadores ocupados durante o tempo de execução.

3.2 - Taxonomia de Flynn

Atualmente, já foram construídos inúmeros computadores paralelos e diante desse cenário, muitos cientistas vêm tentando criar uma classificação em uma taxonomia. A taxonomia mais aceita é a de Michael Flynn (FLYNN, 1972).

A taxonomia de Flynn define as classificações das arquiteturas de computadores em função de suas entradas e saídas, através de dois fatores: sequência de instruções e sequência de dados que são transmitidos para o processador.

A sequência de instruções ou fluxo de instruções é basicamente um conjunto de instruções a serem executadas. Já a sequência de dados ou fluxo de dados, como um conjunto de dados. Com base nesse conhecimento, a figura 3.3 mostra a taxonomia de Flynn que é dividida em quatro categorias (SANTOS, 2018):

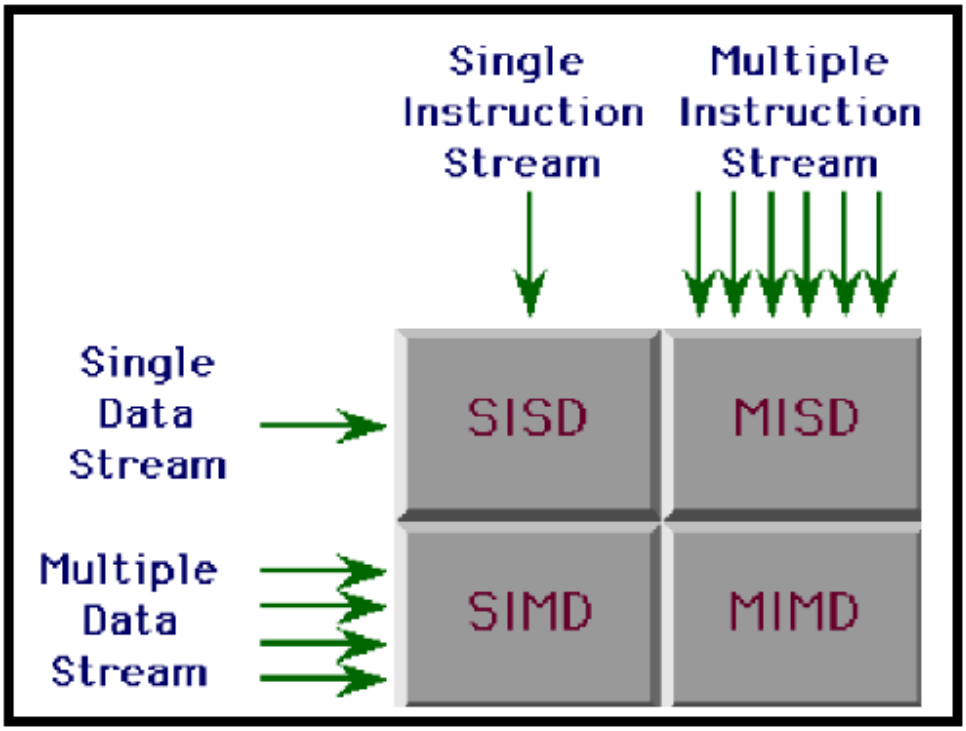

Figura 3.3 - Taxonomia de Flynn.

- SISD - *Single Instruction*, *Single Data* (Figura 3.4) - Essa classificação considera um único fluxo de entrada e um único fluxo de dados de entrada. Essa taxonomia é clássica, pois ela é sequencial e não considera o paralelismo, onde está a maioria dos computadores atualmente, que são baseados na arquitetura de Von Neumann com processadores sequenciais que executam uma instrução completa por vez e sequencialmente.

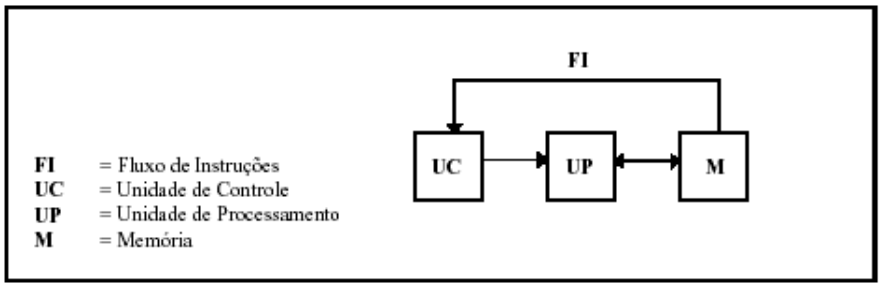

Figura 3.4 - Esquema de um sistema SISD.

- SIMD –*Single Instruction*, *Multiple Data* (Figura 3.5) - Essa taxonomia considera um único fluxo de instrução de entrada e múltiplos fluxos de dados de entrada. Essa configuração processa múltiplos fluxos de dados ao mesmo tempo a cada ciclo e executa operações paralelizáveis. Aqui se enquadram as GPUs, os

FPGAs (*Field Programmable Gate Array*), que são dispositivos projetados para serem configurados pelo consumidor, e os grandes computadores vetoriais.

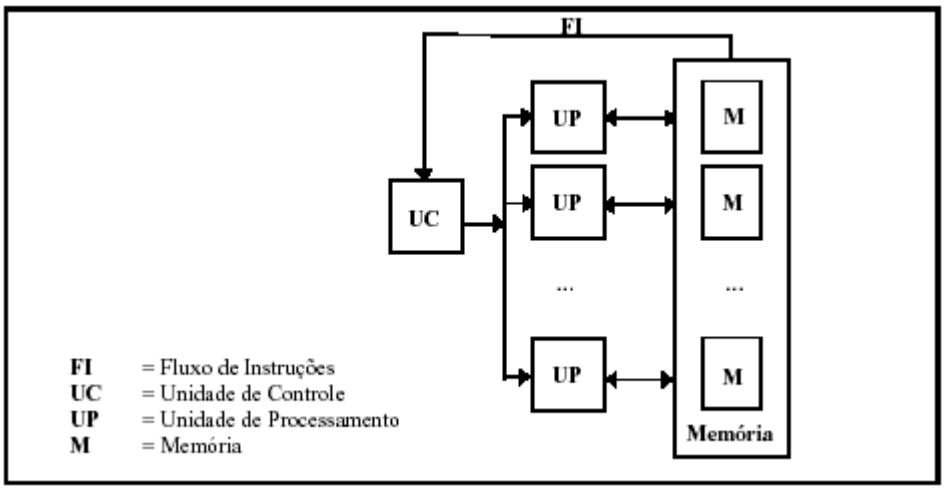

Figura 3.5 - Esquema de um sistema – SIMD.

- MISD –*Multiple Instruction, Single Data* (Figura 3.6) - Essa taxonomia considera múltiplos fluxos de instruções de entrada e apenas um único fluxo de dados de entrada. Essa configuração utiliza múltiplas instruções para manipular um conjunto único de dados, como um vetor. Essa estrutura é utilizada para sistemas que requerem redundância a falhas.

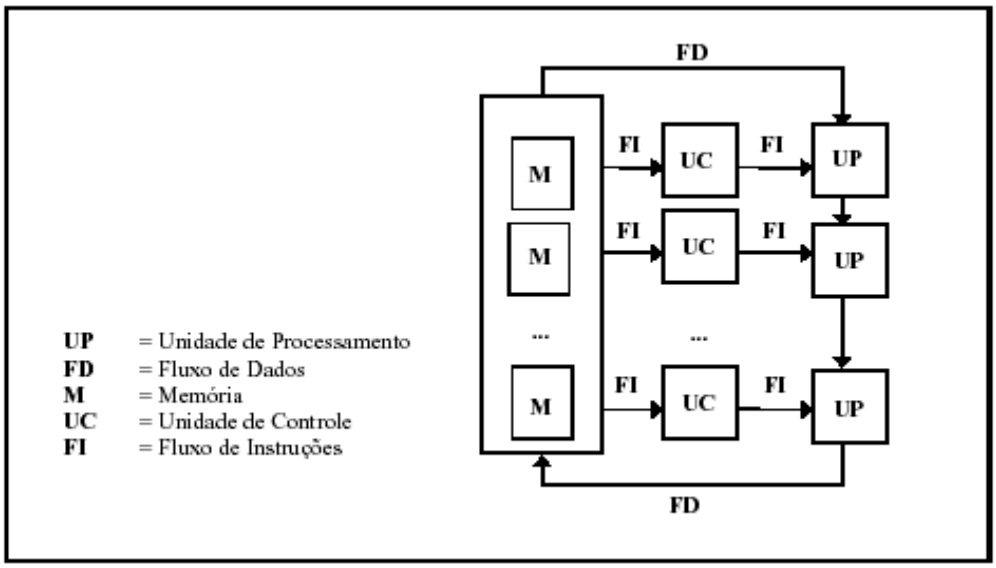

Figura 3.6 - Esquema de um sistema MISD.

- MIMD –*Multiple Instruction, Multiple Data* (Figura 3.7)- Essa taxonomia considera múltiplos fluxos de instruções de entrada e múltiplos fluxos de dados de entrada. Essa configuração é considerada a mais avançada de todas

tecnologicamente, pois produz um elevado desempenho do sistema de computação, onde é encontrado a maioria dos sistemas multiprocessados (multiprocessadores e multicomputadores) que estão dentro dessa taxonomia. Nessa categoria são encontradas as tecnologias de processamento distribuído, como por exemplo: Message Passing Interface (MPI) e Parallel Virtual Machine (PVM).

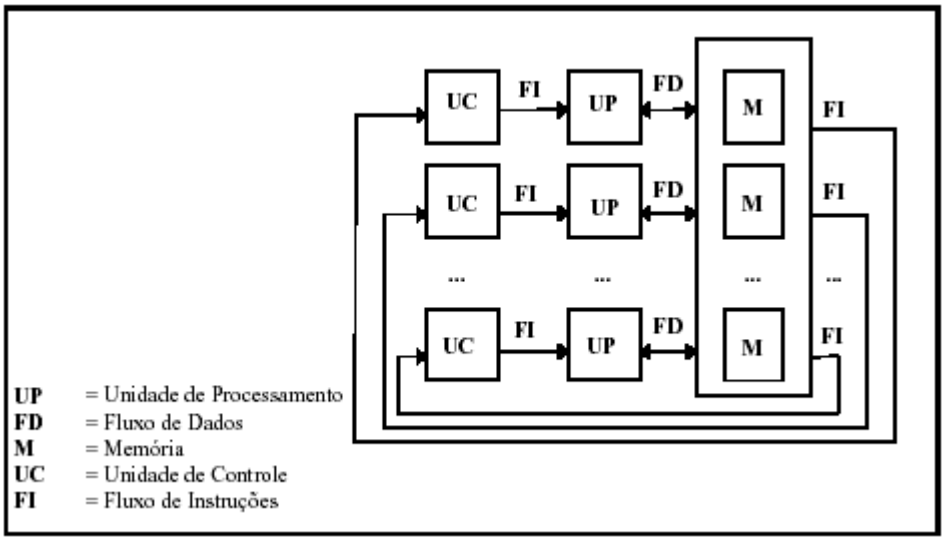

Figura 3.7 - Esquema de um sistema MIMD.

3.3 - GPU

A evolução da computação está diretamente proporcional ao aumento do poder de processamento das máquinas. Pois até o final da década de 90, os microprocessadores eram baseados em CPU, como por exemplo: os microprocessadores da família Intel Pentium e AMD. Porém a partir do início dos anos 2000, a evolução do tipo de arquitetura CPU sofreu uma queda, pois a metodologia da indústria tradicional, que era basicamente aumentar o número de transistores dentro do processador, com o objetivo de aumentar sua velocidade de processamento, se mostrou ineficiente, por causa do alto nível de energia e problemas de dissipação devido ao efeito Joule, tendo como consequência a limitação do aumento da frequência do *clock*, e também o número de atividades que são executadas por período no *clock*, dentro de uma CPU.O *clock* é a frequência com que um processador é capaz de executar as tarefas atribuidas a ele. Isto é, ele mede o número de ciclos por segundo executados pelo processador. Nesse

sentido, quanto maior a frequência, ou seja, o *clock*, menor é o tempo de execução, resultando em um processador mais rápido.

Diante disso, todos os fabricantes de processadores começaram a projetar arquiteturas de microprocessadores com múltiplas unidades de processamento, significando o uso de múltiplos processadores com múltiplas unidades de processamento, tendo como consequência o aumento do poder de processamento (HWU, W; KIRK, D, 2012).

Em 2003, uma nova arquitetura de computadores com muitos núcleos (*manycores*) se destacou, como foram as GPUs, pois são projetadas com núcleos pequenos, porém em grande quantidade, com o objetivo de executar *threads* em paralelo (*multithread*). Uma *thread* significa um fluxo de execução que um determinado programa realiza. Hoje em dia a arquitetura de processadores com múltiplos núcleos já se estabilizou, tendo as GPUs assumido papel principal em relação ao poder de processamento computacional.

As CPUs foram feitas para maximizar a velocidade de execução dos programas sequenciais, tendo poucos núcleos e manipulando poucas *threads*. Porém, as GPUs possuem muitos núcleos. Logo, trabalham com milhares de núcleos e *threads*.

Com essa diferença entre arquiteturas, as GPUs possuem uma performance e desempenho de processamento muito superior às CPUs. Pesquisadores e programadores diante desse conhecimento passaram a usar as GPUs para executar aplicações de propósito geral, até então, que antes só eram praticadas pelas CPUs (PINHEIRO, 2017).

Para exemplificar,podemos citar os processadores da marca NVIDIA, que podem apresentar mais de 5000 núcleos.

Com o aumento da computação de alto desempenho, as GPUs estão evoluindo gradativamente para computadores *manycores* com uma alta capacidade de paralelismo e uma grande potência computacional, e com uma alta taxa de largura de banda de memória. A figura 3.8 mostra uma grande diferença entre as performances das GPU

sem relação às CPUs, sobre a quantidade de operações de ponto flutuante por segundo (Flops).

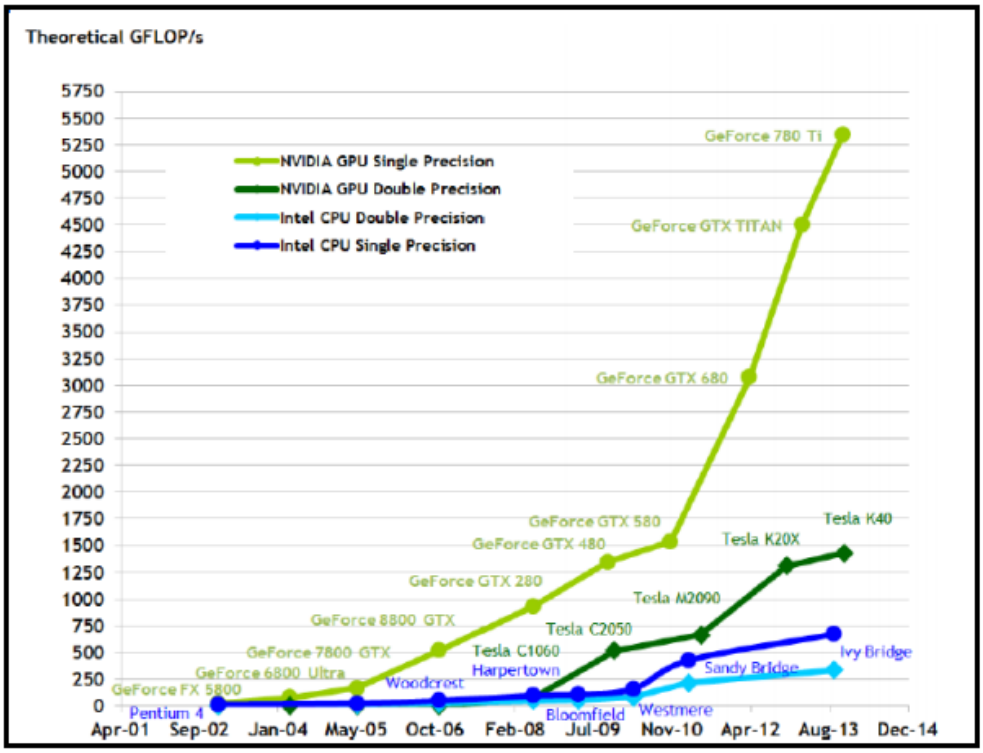

Figura 3.8 - Comparação entre CPU e GPU (Flops) (NVIDIA CORPORATION,2014).

As CPUs Intel trabalham com precisão simples, chegando aos 750 GFlops, enquanto uma GPU NVIDIA, trabalhando com a mesma precisão, consegue superar 5,25 TFlops, isto é, uma performance aproximadamente de 7 vezes maior que a CPU Intel.

A figura 3.9 compara a largura de banda de memória entre a CPU e uma GPU. A CPU da Intel transfere 60 GB/s de dados, porém a GPU NVIDIA transfere aproximadamente 330 GB/s de dados, sendo assim uma performance de aproximadamente 5,5 vezes maior que a CPU. O motivo dessa diferença, é que a GPU é fabricada para computação paralela, sendo construída para que mais transistores sejam dedicados ao processamento de dados, ao invés de cache de dados e controle de fluxo presentes na CPU.

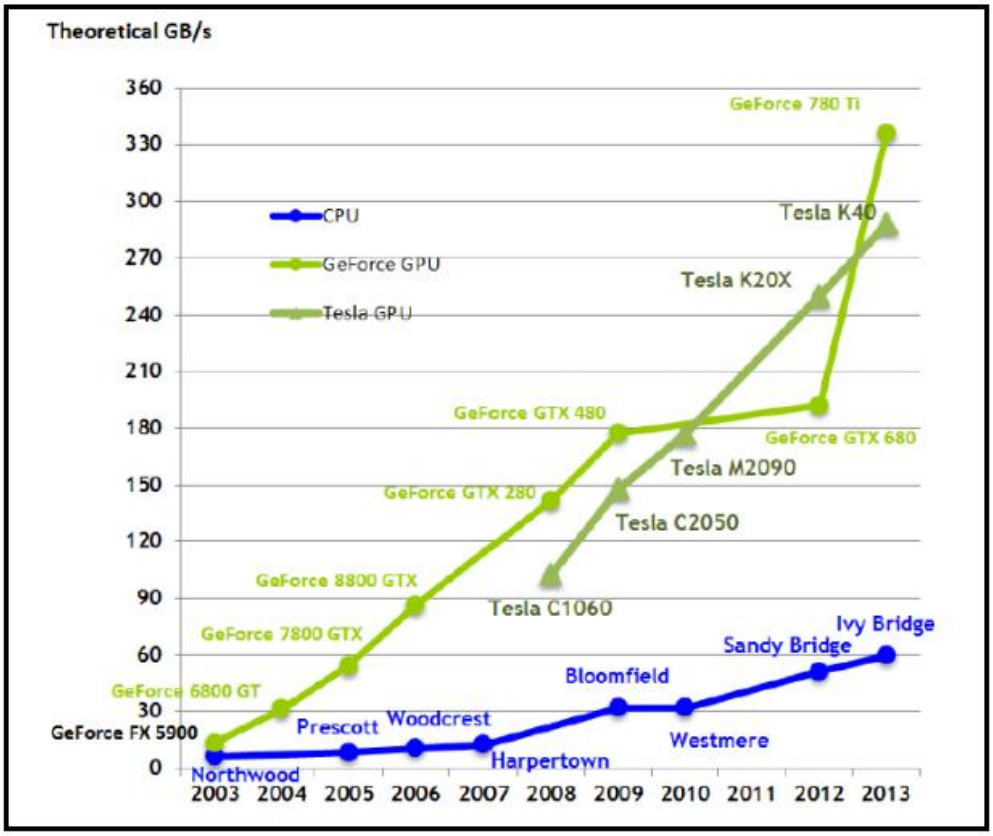

Figura 3.9 - Comparação entre CPU e GPU (Bytes/s)(NVIDIA CORPORATION, 2014).

3.4 - CUDA

CUDA(*Compute Unified Device Architecture*) é basicamente uma plataforma voltada para a computação paralela, gerada para que os desenvolvedores usem o alto potencial do processamento em paralelo da placa de vídeo. Ela foi introduzida em 2006 como plataforma de computação paralela de propósito geral como modelo para GPUs Nvidia.

CUDA tem integração com outros ambientes de programação: C, C++, Fortran, Python, Matlab e Mathematicaepossui diversas aplicações, como por exemplo: imagens médicas, dinâmica de fluido, criptografia e simulações biológicas.

Ele possui alguns problemas, como por exemplo: arenderização de imagens, a transferência de memória que afeta o desempenho e a compatibilidade entre as versões. Uma possível alternativa seria o uso do OpenCL.

O CUDA é uma extensão da linguagem de programação C e possui muitos recursos que ajudam no desenvolvimento da programação paralela. Um dos recursos importantes é a possibilidade de executar as funções *kernel*, que são funções que operam a baixo nível de máquina, como por exemplo, no gerenciamento de memória e processos, rede, subsistemas de arquivos, suporte aos dispositivos e periféricos conectados ao computador.

Quando são invocadas as funções *kernel*, elas são executadas ao mesmo tempo N vezes por N processadores CUDA. Os fluxos de execução independentes quando são disparados na GPU são chamados de *Threads* e essas *Threads* são organizadas em blocos que por sua vez são organizados em grades, como é observado na figura 3.10.

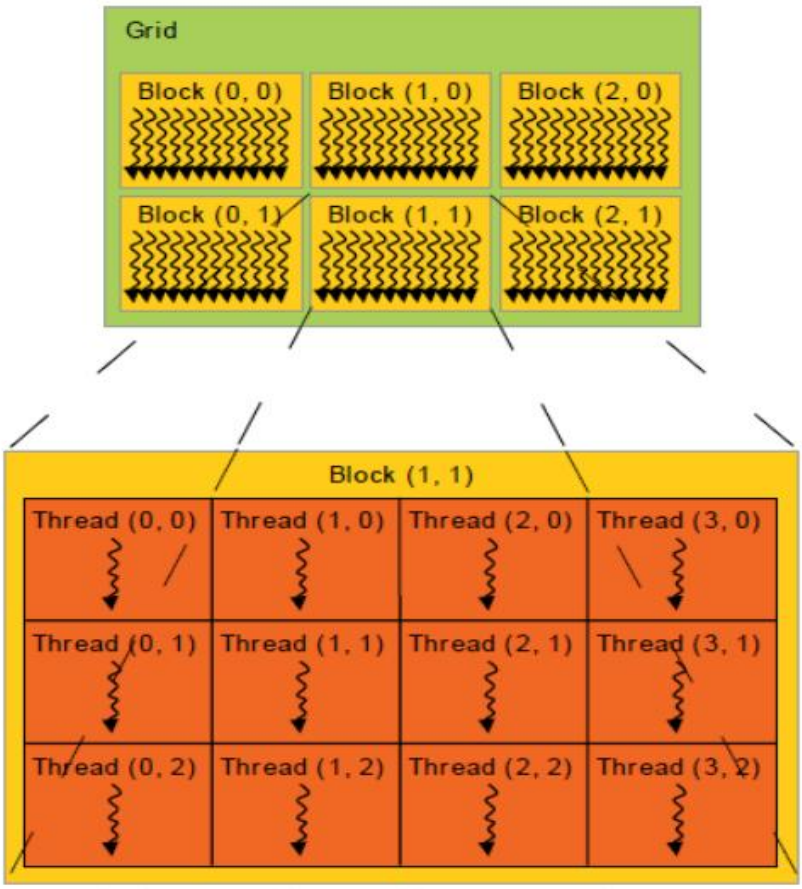

Figura 3.10 - Exemplo de uma grid com 6 blocos contendo 12 threads cada um (CUDA C ProgrammingGuide, 2021).

O processo de indexação em CUDA (Figura 3.11), que basicamente identifica o index global de um *thread*, é dado por:

- gridDim.x = É o número de blocos
- blockDim.x = É o número de *threads* em cada bloco
- blockIdx.x = É o índice do bloco atual dentro da *grid*
- threadIdx.x = É o índice do segmento atual dentro do bloco

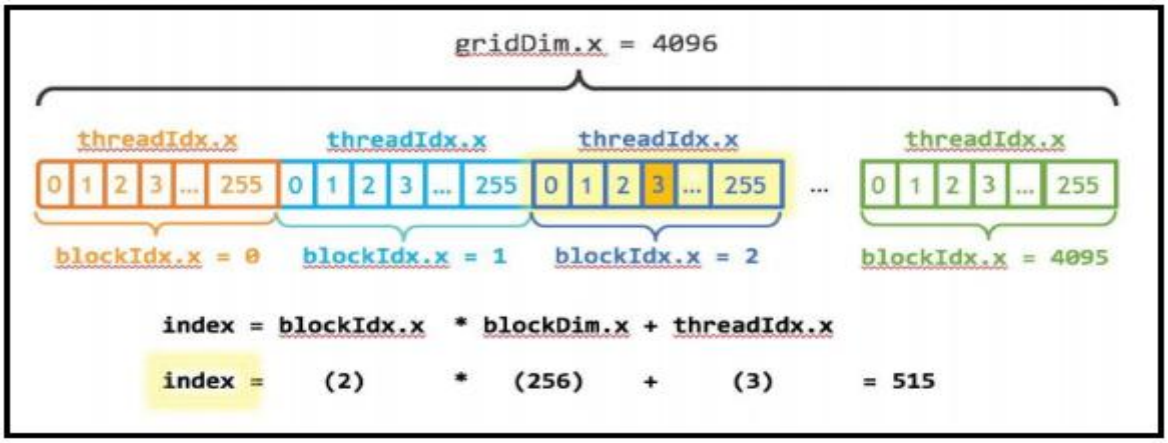

Figura 3.11 - Exemplo do padrão de indexação em CUDA (NVIDIA DEVELOPER, 2017).

A Figura 3.12 mostra as etapas básicas para executar uma função *kernel* em CUDA:

1. Alocação de memória GPU.
2. Dados de entrada transferidos da CPU para GPU.
3. Inicialização do *Kernel.*
4. Dados de saída transferidos para a memória da CPU.

```
// Passo 1: Alocar Espaço do Momória na GPU
cudaMalloc((void**)&d_c, size*sizeof(int));
cudaMalloc((void**)&d_a, size*sizeof(int));
cudaMalloc((void**)&d_b, size*sizeof(int));

// Passo 2: Copiar os Dados de Entrada da Memória do Host
para a Memória do Device
cudaMemcpy(d_a, a, size*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size*sizeof(int), cudaMemcpyHostToDevice);

// Passo 3: Invocar a Função Kernel
soma_Vetores_GPU << <blocks, threadsPerBlock >> >(d_c, d_a,
d_b);

// Passo 4: Copiar os Dados de Saída da Memória do Device
para a Memória do Host
cudaMemcpy(a, d_a, size*sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(b, d_b, size*sizeof(int), cudaMemcpyDeviceToHost);
```

Figura 3.12 - Transferência de dados entre memórias.

O processo de transferência entre memórias pode ser uma desvantagem em relação à aceleração da aplicação. Pois, quanto maior for a quantidade de dados que se deseja transferir, maior será o tempo de transferência. Se o tempo de processamento dessa aplicação for pequeno, comparado ao tempo de transferência de dados, não faz sentido usar paralelismo, pois os ganhos podem ser reduzidos.

As figuras 3.13 e 3.14 mostram respectivamente o cálculo de soma de matrizes de maneira sequencial e como é feito a transição para o código de maneira paralela em CUDA.

```
1  #define N 1000000 //N igual ao número de elementos do vetor
2  //calcula C = A * k + B
3  void somaVetoresSeq(const float a[], const float b[], float c
       [], float k, int n) {
4     int i;
5     for(i=0; i<n; i++)
6        c[i] = a[i] * k  + b[i];
7  }
8  void main() {
9     float a[N], b[N], c[N];
10    //inicializa os vetores a e b
11    ...
12    somaVetoresSeq(a, b, c, 2.0, N);
13 }
```

Figura 3.13 - Código sequencial de soma de matrizes em C.

```
1  //kernel para execução paralela na GPU
2  __global__ void somaVetoresPar(const float *a, const float *b,
       float *c, float k) {
3     int i = threadIdx.x;
4     c[i] = a[i] * k + b[i];
5  }
6  int main() {
7     float a[N], b[N], c[N];
8     //inicializa os vetores a e b
9     ...
10    // invoca o kernel com um bloco de n threads
11    somaVetoresPar<<<1, N>>>(A, B, C, k);
12    ...
13 }
```

Figura 3.14 - Código paralelo de soma de matrizes em CUDA.

Algoritmo de mapeamento do código sequencial para o paralelo:

1. Reservar espaço de memória na CPU para os dados de entrada e saída.
2. Reservar espaço de memória na GPU para os dados de entrada e saída.
3. Transferir osdados de entrada de memória da CPU para a memória da GPU.
4. Disparar o *Kernel.*
5. Transferir os dados processados da memória da GPU para a memória da CPU.
6. Exibir os resultados e executar as finalizações necessárias.

3.5 - Técnicas de Paralelização

Com inúmeros problemas complexos nas diversas áreas da ciência e engenharia, demanda um alto poder de computação a ser resolvido.

Existem inúmeras técnicas de paralelização que tem como objetivo aumentar a performance do programa. As técnicas de paralelização de um algoritmo começam com a divisão do problema principal em problemas menores. Depois, cada problema menor é tratado individualmente e distribuído em vários núcleos da GPU, requisitando do programador diversas ações, como por exemplo: gerenciamento de memória, alocação

de memória na GPU, garantia de independência de dados, o sincronismo entre os dados e a comunicação entre os dados, pois cada um será executado sem o conhecimento do outro (PARHAMI, 2002).

Na grande maioria dos algoritmos, muitos trechos de códigos possuem loops que necessitam um alto custo computacional, logo a paralelização dos loops é uma técnica muito importante e muito difícil (HUANG e HSU, 2000). Basicamente os loops possuem um grande potencial para o paralelismo, porém existem muitas técnicas para realizá-lo, porém o mais difícil é reconhecer a melhor técnica a ser implementada no loop.

Na programação CUDA, paralelizar um loop é uma das tarefas mais comuns utilizando um *kernel*. Existem diversas técnicas de paralelismo de loop:

- Loop Fusion.
- Loop Fission.
- Loop Unrolling.
- Loop Interchanging.
- Loop Inversion.
- Loop Reversal.
- Loop Invariantcodemotion.
- Loop Splitting.
- Loop Unswitching.
- Loop Grid-Stride.

3.5.1 - *Loop Fusion*

É uma técnica que substitui vários loops que interagem no mesmo intervalo em um único loop. Essa técnica não melhora muito a velocidade de execução, pois dependendo da arquitetura que se utilize, cada instrução dentro do loop pode ser executada em *threads* diferentes (Figura 3.15).

```
// Algoritmo original
for (i = 0; i < 100; i++)
    a[i] = 1;
for (i = 0; i < 100; i++)
    b[i] = 2;

// Loop Fusion
for (i = 0; i < 100; i++){
    a[i] = 1;
    b[i] = 2;
}
```

Figura 3.15 - *Loop Fusion.*

3.5.2 - *Loop Fission*

Loop Fission ou também chamado de *loop distribution*, é uma técnica completamente oposta ao *loop Fusion*. Pois nesta técnica, o *loop* é quebrado em inúmeros *loops* que tem a mesma faixa de índice, sendo que cada *loop* com uma parte do corpo original. O propósito dessa técnica é dividir um *loop* grande em *loops* menores, de modo que cada *loop* seja executado em uma*thread* ou processador separado, consequentemente aumentando a performance do algoritmo. Na figura 3.16 temos um exemplo de *loop fission*.

```
// Algoritmo original
for (i = 0; i < 100; i++){
    a[i] = 1;
    b[i] = 2;
}

// Loop Fission
for (i = 0; i < 100; i++)
    a[i] = 1;
for (i = 0; i < 100; i++)
    b[i] = 2;
```

Figura 3.16 - *Loop Fission.*

3.5.3 - *Loop Unrolling*

Loop unrolling ou *loop unwinding* (Figura 3.17), é uma técnica que otimiza a velocidade de execução de um programa em detrimento ao seu tamanho. Seu propósito é acelerar o programa minimizando ou eliminando a aritmética de controle de *loop*, por exemplo: os testes de fim de *loop* e controle de ponteiros (do *loop*). A técnica fundamental, os *loops* são reescritos como uma sequência repetida de instruções, deslocando-se o ponteiro para a próxima instrução sem passar pelo *loop*, e a minimização do número de repetições do *loop*.

```
// Algoritmo original
for (i = 0; i < 100; i++){
    a[i] = 1;
}

// Loop Unrolling
for (i = 0; i < 100; i +=5){
    a[i] = 1;
    a[i + 1] = 1;
    a[i + 2] = 1;
    a[i + 3] = 1;
    a[i + 4] = 1;
}
```

Figura 3.17 - *Loop Unrolling.*

3.5.4 - *Loop Interchanging*

Geralmente é melhor paralelizar o *loop* mais externo e um conjunto de *loops*, uma vez que o *overhead*, que é um processamento em excesso, é menor. Porém, nem sempre é seguro paralelizar os *loops* mais externos, devido a dependências que podem ser carregadas por esses *loops*. Isso é ilustrado na figura 3.18.

```
// Algoritmo original
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

Figura 3.18 - *Loop* aninhado que não pode ser paralelizado.

Neste exemplo, o *loop* com a variável de índice i, não pode ser paralelizado, devido a uma dependência entre duas iterações sucessivas dos *loops* (*loop* i e *loop* j). Os dois *loops* podem ser trocados, assim o *loop* paralelo (o *loop* j) se torna o *loop* externo (Figura 3.19):

```
// Loop Interchanging
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

Figura 3.19 - *Loop Interchanging.*

3.5.5 - *Loop Inversion*

Essa técnica de paralelização é uma otimização de transformação do *loop*, no qual um *loop while* (Figura 3.20) é substituído por um bloco *if* que contém um *loop do-while* (Figura 3.21). Quando usado corretamente, ele pode melhorar o desempenho, devido ao *pipelining* de instruções.

```
// Algoritmo Original
int i, a[100];
i = 0;
while (i < 100){
    a[i] = 0;
    i++;
}
```

Figura 3.20 - *Loop while*.

```
// Loop Inversion
int i, a[100];
i = 0;
if (i < 100){
    do {
        a[i] = 0;
        i++;
    } while (i < 100);

}
```

Figura 3.21 - *Loop do-while*.

Apesar da complexidade aparentemente maior da figura 3.21, ele pode ser executado mais rápido em CPUs modernas, pois utilizam *pipeline* de instruções. Por natureza, qualquer salto do código causa uma interrupção no *pipeline*, sendo um prejuízo para o desempenho. Além disso, o *loop inversion* permite movimento seguro do código invariante em *loop*.

3.5.6 - *Loop Reversal*

Essa técnica reverte a ordem na qual os valores são atribuídos à variável de índice. Isso ajuda a eliminar dependências e assim, habilitar outras otimizações. Certas arquiteturas utilizam construções de *loop* no nível de montagem que contam apenas em uma única direção.

O *loop reversal* (Figura 3.22) é mais rápido que o *loop regular* (Figura 3.23). Pois o *loop reversal* gera menos *byte* do que o *loop regular* (TRAV CAV, 2017).

```
// Loop Reversal
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Figura 3.22 - *Loop Reversal*.

```
// Loop Regular
for (int i = 10; i >= 0; i--) {
    System.out.println(i);
}
```

Figura 3.23 - *Loop* Regular.

3.5.7 - *Loop Invariant code motion*

Essa técnica pode melhorar bastante a eficiência movendo uma computação de dentro do *loop* para fora dele sem afetar a semântica do programa. O movimento de código invariante em *loop*, também chamado de *hoisting* ou *scalar promotion*, é uma otimização do compilador que executa esse movimento automaticamente, calculando um valor apenas uma vez, antes do início do *loop*, se a quantidade resultante do cálculo for a mesma para cada iteração do *loop*, ou seja, uma quantidade invariante do *loop*. Isso é particularmente importante com expressões de cálculo de endereço geradas por *loops* sobre matrizes. Para uma implementação correta, essa técnica deve ser usada com inversão, porque nem todo código pode ser movido para fora do *loop*. Na figura 3.24, duas otimizações podem ser aplicadas:

```
// Loop Invariant
int i = 0;
while (i < n){
    x = y + z;
    a[i] = 6 * i + x * x;
    ++i;
}
```

Figura 3.24 - *Loop Invariant.*

Embora o cálculo de (x = y + z) e (x * x) seja invariável ao *loop* (*loop invariant*), é necessário tomar precauções antes de mover o código para fora do *loop*. É possível que a condição do *loop* seja falsa (por exemplo, se *n*fornegativo). Nesse caso, o corpo do *loop* não deverá ser executado. Uma maneira de garantir o comportamento correto é usar uma ramificação condicional fora do *loop*. A avaliação da condição do *loop* pode ter efeitos colaterais, portanto, uma avaliação adicional pela construção *if*, deve ser compensada, substituindo o *loop while* por um *do-while*. Se o código *do-while* estiver em primeiro lugar, não será necessário todo o processo de proteção, pois é garantido que o corpo do *loop* seja executado pelo menos uma vez, como é mostrado na figura 3.25.

```
// Loop Invariant adaptado
int i = 0;
if(i < n){
    x = y + z;
    int const t1 = x * x;
    do {
        a[i] = 6 * i + t1;
        ++i;
    } while (i < n);
}
```

Figura 3.25 - *Loop Invariant* adaptado.

3.5.8 - *Loop Splitting*

O *loop splitting* é uma técnica de otimização de compilador. Ele tenta simplificar um *loop* ou eliminar dependências, dividindo-o em vários *loops* que possuem os mesmos corpos, mas iteram em diferentes partes contíguas do intervalo de índice.

O *loop peeling* é um caso especial de *loop splitting* que divide qualquer primeira (ou última) problemática em poucas iterações do *loop* e as executa fora do corpo do *loop*.

Suponha que um *loop* tenha sido escrito como mostra a figura 3.26:

```
// Loop comum
int p = 10;
for (int i = 0; i < 10 ; ++i){
    y[i] = x[i] + x[p];
    p = i; //add i to variable p
}
```

Figura 3.26–*Loop* comum.

Observe que, p = 10 apenas para a primeira iteração e para todas as outras iterações, p = i - 1. Um compilador pode tirar proveito disso, por meio do *loop peeling*, desacoplando a primeira iteração do *loop*.

Após descacoplar a primeira iteração, o código ficaria como na figura 3.27:

```
// Loop peeling
y[0] = x[0] + x[10];
for(int i = 1; i < 10; ++i){
    y[i] = x[i] + x[i-1];
}
```

Figura 3.27 - *Loop peeling*.

Essa forma equivalente elimina a necessidade da variável p dentro do corpo do *loop*.

O *loop peeling* foi introduzido no GCC(GNU *CompilerCollection*), que é um compilador da linguagem de programação C desenvolvido pelo projeto GNU,na versão 3.4. Foi adicionada uma divisão de *loop* mais generalizada no GCC na versão 7.

3.5.9 - *Loop Unswitching*

O *loop unswitching* é uma otimização do compilador. Ele move uma expressão condicional, que está dentro de um *loop*, para fora desse mesmo *loop*. Desse modo, duplicando o corpo do *loop* e colocando uma versão dele dentro de cada uma das cláusulas *if* e *else* da condicional. Isso pode melhorar a paralelização do *loop*. Como os processadores modernos podem operar rapidamente em vetores, isso aumenta a velocidade.

A figura 3.28 mostra um exemplo simples. Suponha que desejemos adicionar as duas matrizes *x* e *y*, e também fazer algo dependendo da variável *w*. Temos o seguinte código:

```
// Loop dependente
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++) {
    x[i] += y[i];
    if (w)
        y[i] = 0;
}
```

Figura 3.28 - *Loop* dependente da variável *w*.

A expressão condicional dentro desse *loop* dificulta o paralelismo seguro. Nesse sentido,a figura 3.29 mostra o *loop Unswitching*, que faz a desassociação da expressão condicional *if* desse *loop*.

```
// Loop Unswitching
int i, w, x[1000], y[1000];
if (w) {
    for (i = 0; i < 1000; i++) {
        x[i] += y[i];
        y[i] = 0;
    }
} else{
    for (i = 0; i < 1000; i++) {
        x[i] += y[i];
    }
}
```

Figura 3.29 - Loop Unswitching.

Enquanto o *loop unswitching* pode dobrar a quantidade de código gravado, cada um desses novos *loops* agora pode ser otimizado separadamente.

3.5.10 - *Loop Grid-Stride*

O principal objetivo na programação CUDA é paralelizar o *loop* usando o *kernel*. A figura 3.30 é um exemplo clássico de implementação sequencial que usa um *FOR* para *loop*. Para paralelizar esse *loop* com eficiência, precisamos lançar *threads* suficientes para utilizar totalmente a GPU.

```
// Algoritmo original
void Grid_Stride_Loop(int n, float a, float *x, float *y)
{
   for (int i = 0; i < n; ++i)
      y[i] = a * x[i] + y[i];
}
```

Figura 3.30 - *Grid-Stride Loop* - Algoritmo Sequencial.

A orientação comum a ser seguida pela NVIDIA (NVIDIA CORPORATION, 2017) é que devem ser lançadas *threads* para cada elemento de dado. Supondo que temos *threads* bastantes para alcançar todo o tamanho do vetor, logo podemos reescrever o código em CUDA, como é mostrado na figura 3.31.

```
// Algoritmo CUDA
__global__ void Grid_Stride_Loop (int n, float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a * x[i] + y[i];
}
```

Figura 3.31 - Grid-Stride loop - Algoritmo CUDA.

Esse exemplo de programação também é conhecido como *kernel* monolítico, pois toma para si, uma única e enorme grade de *threads* para processar todo o vetor, bastando alocar a quantidade suficiente de *threads* na chamada da função *kernel*, de maneira que cada *thread* se encarregue de uma posição do vetor. Essa técnica assume que a GPU possui uma quantidade de *threads* satisfatória para cobrir todo o vetor, porém quando o tamanho do vetor é maior que a quantidade de *threads* existentes na GPU, essa técnica não poderá ser mais usada, pois não será possível alocar, na chamada da função *kernel*, todas as *threads* necessárias para cobrir em uma única chamada o vetor. Ao paralelizar o código, ao invés de descartar todo o *loop*, podemos usar a técnica do *grid-stride loop*,conforme mostra a figura 3.32.

```
// Algoritmo Grid-Stride Loop
__global__ void Grid_Stride_Loop (int n, float a, float *x, float *y)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; i += blockDim.x * gridDim.x)
        y[i] = a * x[i] + y[i];
}
```

Figura 3.32 - Grid-Stride Loop - Algoritmo Grid-Stride Loop.

Esta técnica ao invés de assumir que a grade de *threads* é grande bastante para alocar todo o vetor, ela faz *loops* no vetor com um tamanho de *grid* de cada vez. Note que o passo do *loop* é blockDim.x * gridDim.x que é o número total de segmentos do grid. Logo, se houver 2048 *threads* no *grid*, o *thread* 0 irá calcular os elementos 0,

2048, 4096, 8192 e assim por diante. Usando um *loop* com passo igual ao tamanho do *grid*, garante-se que todo o endereço dentro do *wraps* é uma unidade-passo, assim obtemos o uso máximo dos *threads* assim como na versão monolítica. No momento em que o *kernel* é executado com um *grid* suficientemente grande para alcançar todas as iterações do *loop*, o algoritmo *grid-stride loop* tem essencialmente o mesmo custo computacional da instrução *if* do *kernel* monolítico, pois o incremento do *loop* somente será avaliado quando a condição do *loop* for verdadeira.

3.6 - OpenCLvs CUDA

O OpenCL(*Open ComputingLanguage*) é um padrão aberto, mantido pelo Khronos Group, que permite o uso de GPUs para desenvolvimento de aplicações paralelas (NVIDIA CORPORATION, 2012). O OpenCL é uma API de baixo nível, ou seja, ele é uma linguagem mais próxima da máquina. Os desenvolvedores de *software*, ao usar a API OpenCL, podem executar *kernels* escritos em um subconjunto da linguagem de programação C em uma GPU.

Ele também permite que os desenvolvedores possam escrever os códigos de programação heterogêneos, fazendo com que estes programas consigam aproveitar tanto os recursos de processamento das CPUs quanto das GPUs.

Além disso, permite programação paralela usando paralelismo de dados e de tarefas. OpenCL oferece suporte a CUDA e os desenvolvedores podem fazer chamadas a funções *kernels* utilizando um subconjunto limitado da linguagem de programação C em uma GPU.

Enquanto o CUDA é mantido e aprimorado apenas pela NVIDIA, o OpenCL é suportado por fabricantes como, por exemplo: AMD, NVIDIA, APPLE, INTEL e IBM.

No entanto, apesar de se tratar de um modelo heterogêneo, permitindo o gerenciamento para portabilidade em multiplataformas, o OpenCL pode se mostrar um tanto quanto complexo em comparação ao CUDA.

Assim, independentemente de um código em OpenCL ser suportado por uma grande variedade de dispositivos, isso não significa que o código será executado de forma otimizada em todos eles sem qualquer esforço da parte do programador.

CUDA e OpenCL são *frameworks* para programação em GPU. Ambos permitem o uso de GPUs para computação de tarefas de propósito geral que podem ser paralelizadas. CUDA é uma arquitetura proprietária da NVIDIA, podendo ser utilizada unicamente nas placas gráficas produzidas por esta empresa. O OpenCL é um padrão aberto que pode ser executado em *hardwares* de vários fornecedores.

Alguns programadores afirmam que CUDA é mais eficiente e contém APIs de alto nível, as quais são mais convenientes. Uma vantagem do OpenCL é o fato de ele permitir que qualquer fornecedor implemente suporte OpenCL para seus produtos. CUDA, além de ser mais utilizada, atualmente é a primeira programação paralela a surgir e pode ser considerada uma escolha mais viável em comparação com o OpenCL. A escolha do OpenCL pode parecer óbvia por ser possível desenvolver programas que poderiam ser executados em qualquer GPU, ao invés de desenvolver uma versão (em CUDA) para placas da NVIDIA. Porém, na prática essa escolha pode ser mais complicada, pois o OpenCL oferece funções e extensões que são específicas para cada família.

Além disso, por ter um modelo de gerenciamento para a portabilidade em multiplataformas e multifornecedores, o OpenCL pode ser considerado mais complexo. Tanto o OpenCL, quanto o CUDA são linguagens sintaticamente semelhantes à linguagem C. O CUDA é otimizado em GPUs, porém o OpenCL é mais genérico, ou seja, pode ser otimizado em GPUs ou CPUs.

Segundo (NUNES, 2012), o CUDA é 15 vezes mais rápido que o OpenCL, pois além dele utilizar um conjunto de comandos mais avançados e eficientes, ele também

otimiza sua PTX (Paralell Thread Execution), que é uma máquina virtual que está entre a aplicação e as GPUs NVIDIA, de uma maneira bem mais eficiente que o OpenCL.

É importante mencionar que nesta dissertação foi utilizada CUDA ao invés do OpenCL. Além disso, também foi utilizado placas NVIDIA para o paralelismo, queserá mostrado no capítulo 4.

CAPÍTULO 4

Implementação Computacional do Método de Runge-Kutta

Neste capítulo será apresentado o programa desenvolvido usando o método de Runge-Kutta, detalhado na seção 2.1.2, usando as linguagens de programação: Fortran, Python e C. Para avaliar a eficiência dessas linguagens computacionais em cálculos sequências e paralelos, será usado um passo de tempo muito curto, $\Delta t = 10^{-6}$s. Desse modo, será garantido a precisão e a acurácia do método, bem como, assegurar um número significativo de cálculos a serem efetuados. Desta forma, poderemos medir os tempos computacionais gastos nessas simulações.

Nas implementações dos códigos foi usado um tempo de simulação, t = 100s, onde foram necessários 10^8 cálculos. Desse modo, as equações da cinética pontual mostradas nas equações (2.24) e (2.25) serão resolvidas 100 milhões de vezes. Isso exige um esforço computacional considerável para a realização de cálculos num intervalo razoavelmente curto de tempo. Logo, é possível medir a eficiência dessas linguagens computacionais tendo em vista as acelerações via GPU e CPU.

4.1 - Fortran Sequencial

O primeiro algoritmo computacional utilizado para resolver numericamente as equações da cinética pontual de reatores, usando o método de Runge-Kutta foi em linguagem Fortran. Nesse sentido, a partir desse código, foi implementado as versões Python, C e seus respectivos códigos paralelos.

A linguagem Fortran é uma linguagem estrutural de alto nível usada na Análise Numérica, Engenharia e Ciência da Computação. O termo Fortran vem do acrônimo IBM *Mathematical **FOR**mula **TRAN**slation System*. Essa linguagem foi desenvolvida

na década de 50 por uma equipe de programadores, chefiados por John Backuse ainda é fortemente usada atualmente (GAELZER, 2011). O objetivo de Backus era criar uma linguagem que fosse simples de ser compreendida e utilizada, porém que produzisse um código numérico tão eficiente quanto a linguagem Assembler, ou linguagem de máquina. Desde a sua criação, o Fortran era muito simples de ser usado, pois era possível programar fórmulas matemáticas quase que igualmente escritas de forma simbólica, possibilitando assim, que programas fossem escritos mais rápidos e com pouca perda de eficiência de processamento, sendo que todo cuidado era dedicado à construção do compilador, ou seja, o programa que tem como função, a tradução do código-fonte em Fortran para o Código Assembler.

O Fortran foi também um grande passo revolucionário, pois possibilitou aos programadores dispensarem a tarefa de programar em Linguagem Assembler. Nesse sentido, concentrou-se o foco mais na solução do problema em questão. Outra importante vantagem foi que os computadores se tornaram mais acessíveis financeiramente aos cientistas e engenheiros, sendo assim a programação não estava mais restrita a um número de programadores especialistas ou a grandes corporações.

O Fortran tem como característica a implementação de programas que se destacam pela velocidade de execução. Assim sendo, tem seu principal uso em diversas aplicações científicas com forte poder de processamento computacional, como por exemplo: engenharia, física, astronomia, meteorologia e economia.

Portanto, uma aplicação prática foi implementada em Fortran, para a solução das equações da cinética pontual dos reatores utilizando o método de Runge-Kutta. O programa principal mostrado na figura 4.1 possui uma chamada para sub-rotina que utilizao método numérico de Runge-Kutta (figura 4.3), onde resolve 100 milhões de vezes as equações da cinética pontual mostradas nas equações (2.24) e (2.25). Há um módulo dedeclaração de variáveis públicas (figura 4.2) que é utilizado no programa principal, assim como, na sub-rotina do método de Runge-Kutta.

```
!   ------------------------------------------------------
!   Solucao das Equacoes da Cinetica Pontual de Reatores
!   ------------------------------------------------------
    program MetodosNumericos

    use DeclaraDados

!   Leitura dos parâmetros cinéticos
    ...

!   Tempo inicial de simulação
    ...

!   Inicializa o tempo
    ...

!   Calcula as condições iniciais
    ...

!   Solução numérica
    call SistemaRungeKutta

!   Tempo final de simulação
    ...

!   Grava o tempo de simulação
    ...

    write (2,15) Tempo_Final - Tempo_Inicial
15  format ('Tempo de simulação = ', f8.4, ' segundos.')

    close (unit = 2)

    end program
```

Figura 4.1 - Programa Principal da Solução das Equações da Cinética Pontual de Reatores.

```
!   Declaração de Variáveis
!   -----------------------
    module DeclaraDados

      integer          :: N                    , Intervalo_Gravacao
      double precision :: Beta (6)             , Lambda (6)
      double precision :: Delta_T              , Tempo_Simulacao              , Tempo
      double precision :: Beta_Total           , Tempo_Medio_Geracao          , Reatividade
      double precision :: Concentracao (6)     , Concentracao_Anterior (6)
      double precision :: Densidade            , Densidade_Anterior           , Soma_Concentracao

      double precision :: Concentracao_Anterior_Mais_Meio (6)    , Concentracao_Anterior_Mais_Um (6)
      double precision :: Densidade_Anterior_Mais_Meio           , Densidade_Anterior_Mais_Um

    end module
```

Figura 4.2 - Módulo de Declaração de Variáveis.

```
!   ------------------------------------------------------
!   Resolve Sistema de Equações usando Método Runge-Kutta
!   ------------------------------------------------------
    Subroutine SistemaRungeKutta

    use DeclaraDados
    ....

!   Grava os resultadados
    ...

     do while (Tempo <= Tempo_Simulacao)

!      Cálculo de K1 para densidade de nêutrons e concentração de precursores
       ...

!      Cálculo de K2 para densidade de nêutrons e concentração de precursores
       ...

!      Cálculo de K3 para densidade de nêutrons e concentração de precursores
       ...

!      Cálculo de K4 para densidade de nêutrons e concentração de precursores
       ...

!      Solucção para densidade de nêutrons e concentração de precursores
       ...

!      Define um contador para gravação de dados
       ...

!      Atualiza a densidade de nêutrons e concentração de precursores
       ...

     end do

    end subroutine
```

Figura 4.3 - SubrotinaRunge-Kutta.

Ao se analisar mais profundamente a figura 4.3 é possível identificar um *loop* principal do-while, onde são resolvidas 100 milhões de vezes as equações da cinética pontual e expandido para as figuras 4.4 a 4.8, pode-se identificar os *loops* para os cálculos dos Ks e da solução para densidade de nêutrons e concentração de precursores. Nas seções futuras esses *loops* ***(do - while)*** e ***(do - end do)*** serão paralelizados para diminuir o tempo de execução.

```
Cálculo de K1 para densidade de nêutrons e concentração de precursores
Soma_Concentracao = 0.0d+00
do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior (i)
end do

K1D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior + Soma_Concentracao

do i = 1, 6
  K1C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior - lambda (i) * Concentracao_Anterior (i)
end do
```

Figura 4.4 - Cálculo de K1 para densidade de nêutrons e concentração de precursores.

```
Cálculo de K2 para densidade de nêutrons e concentração de precursores
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5d+00 * Delta_T * K1D

do i = 1, 6
  Concentracao_Anterior_Mais_Meio (i) = Concentracao_Anterior (i) + 0.5d+00 * Delta_T * K1C (i)
end do

Soma_Concentracao = 0.0d+00

do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end do

K2D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio + Soma_Concentracao

do i = 1, 6
  K2C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio - lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end do
```

Figura 4.5 - Cálculo de K2 para densidade de nêutrons e concentração de precursores.

```
Cálculo de K3 para densidade de nêutrons e concentração de precursores
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5d+00 * Delta_T * K2D

do i = 1, 6
  Concentracao_Anterior_Mais_Meio (i) = Concentracao_Anterior (i) + 0.5d+00 * Delta_T * K2C (i)
end do

Soma_Concentracao = 0.0d+00

do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end do

K3D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio + Soma_Concentracao

do i = 1, 6
  K3C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio - lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end do
```

Figura 4.6 - Cálculo de K3 para densidade de nêutrons e concentração de precursores.

```
Cálculo de K4 para densidade de nêutrons e concentração de precursores
Densidade_Anterior_Mais_Um = Densidade_Anterior + Delta_T * K3D

do i = 1, 6
  Concentracao_Anterior_Mais_Um (i) = Concentracao_Anterior (i) + Delta_T * K3C (i)
end do

Soma_Concentracao = 0.0d+00

do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior_Mais_Um (i)
end do

K4D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Um + Soma_Concentracao

do i = 1, 6
  K4C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Um - lambda (i) * Concentracao_Anterior_Mais_Um (i)
end do
```

Figura 4.7 - Cálculo de K4 para densidade de nêutrons e concentração de precursores.

```
Solucção para densidade de nêutrons e concentração de precursores
Densidade = Densidade_Anterior + ( Delta_T / 6.0d+00 ) * ( K1D + 2.0d+00 * ( K2D + K3D ) + K4D )

do i = 1, 6
  Concentracao (i) = Concentracao_Anterior (i) + ( Delta_T / 6.0d+00 ) * ( K1C (i) + 2.0d+00 * ( K2C (i) + K3C (i) ) + K4C (i) )
end do
```

Figura 4.8 - Solução para densidade de nêutrons e concentração de precursores.

4.2 - Python Sequencial

O Python é uma linguagem de alto nível, pois sua sintaxe é mais próxima do ser humano e de fácil compreensão. Ela é também multiparadigma, ou seja, suporta vários paradigmas, como por exemplo: Os paradigmas imperativos, funcionale orientado a objeto. É uma linguagem de tipagem dinâmica e forte, com uma de suas características fundamentais a fácil leitura do código, e se comparado a outras linguagens, exige poucas linhas de códigos (PYTHON, 2021).

A linguagem Python foi criada por Guido Van Rossum em 1991, no Instituto de Pesquisa Nacional para Matemática e Ciência da Computação, com base na linguagem ABC e deriva em parte da linguagem C.

As principais vantagens do Python são:

- É uma linguagem com sintaxe muito simples.
- Possui gerenciamento de memória automático.
- É de simples programação e de fácil aprendizado.
- Tem uma sintaxe intuitiva.
- É de código aberto (*Open Source*) a todos.
- Modularizada.
- Multiplataforma.
- Possui muitas bibliotecas disponíveis para consulta.
- Existe uma grande comunidade de usuários.

As principais desvantagens do Python são:

- Linguagem interpretada - Sendo assim, é mais lenta que as linguagens compiladas, pois a linguagem interpretada executa o código linha por linha.
- Fraca em navegadores e computação mobile -Ainda que seja uma ótima linguagem do lado do servidor, é pouco usado no lado do cliente.
- As camadas de acesso ao banco de dados são subdesenvolvidas - Pois se for comparada ao banco de dados da linguagem java, como JDBC (Java DataBaseConnectivity) e ODBC (Open DataBaseConnectivity) são

inferiores e assim, tem como consequência uma menor aplicação em grandes empresas.

- Falta de profissionais capacitados.
- Linguagem em desenvolvimento constante e sem uma padronização forte.
- Pouca documentação.

Atualmente, ela possui inúmeras aplicações, como por exemplo:

- Computação gráfica.
- Servidores de Aplicações.
- Data Science.
- Machine Learning.
- Big Data.
- Desenvolvimento Web.

Diante disso, é importante detalhar as diferenças entre a linguagem compilada e interpretada. A tabela 4.1 mostra esse detalhamento.

Tabela 4.1 - Compiladores vs Interpretadores (CODE MASTERS, 2015).

	VANTAGENS	**DESVANTAGENS**
Compiladores	Execução mais rápida.	Várias etapas de tradução.
	Permite estruturas de programação mais complexas para a sua execução.	Programação final é maior necessitando de mais memória.
	Permite a otimização do código fonte.	Processo de correção de erros e depuração é mais demorado.
Interpretadores	Depuração do programa é mais simples.	Execução do programa é mais lenta.
	Consome menos memória.	Estruturas de dados demasiado simples.
	Resultado imediato do programa ou rotina desenvolvida.	Necessário fornecer o programa fonte ao usuário.

Portanto, a tabela 4.1 acima mostra que a principal vantagem dos compiladores é a rapidez de cada operação, em detrimento da velocidade baixa dos interpretadores. Porém, isso não significa que uma linguagem é mais certa do que a outra, mas sim o que será feito no programa ou como será produzido.

Na figura 4.9 é mostrada a função do método de Runge-Kutta em Python, que anteriormente era tratada como sub-rotina no código em Fortran, onde são feitos 100 milhões de cálculos das equações da cinética pontual por meio do loop *while*. Nas figuras 4.10 até 4.14, são mostrados os cálculos dos Ks e a solução para densidade de nêutrons e concentração de precursores. Os *loops While* e *For* são considerados gargalos computacionais, pois diminui a performance computacional do código. Cada loop será paralelizado em Python CUDA, com o objetivo de diminuir o tempo de execução.

```
#---------------------------------------
# Função Sistema_Runge_Kuta():
#---------------------------------------
def Sistema_Runge_Kuta():

  #-------------------------------------------------------------------------
  # Laço até o tempo de simulação
  #-------------------------------------------------------------------------
  n = 0
  while (Tempo <= Tempo_Simulacao ):
    n += 1
    Tempo = Tempo + Delta_T

    #-------------------------------------------------------------------------
    # Cálculo de K1 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Cálculo de K2 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Cálculo de K3 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Cálculo de K4 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Solução para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...
```

Figura 4.9- Função Runge-Kutta do Python.

```
#-------------------------------------------------------------------------
# Cálculo de K1 para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Soma_Concentracao = 0.0

for i in range(len(Beta)):
  Soma_Concentracao=Soma_Concentracao+Lambda[i]*Concentracao_Anterior[i]

K1D = ((Reatividade-Beta_Total)/Tempo_Medio_Geracao)*Densidade_Anterior+Soma_Concentracao

for i in range(len(Beta)):
  K1C[i] =(Beta[i]/Tempo_Medio_Geracao)*Densidade_Anterior-Lambda[i]*Concentracao_Anterior[i]
```

Figura 4.10 - Cálculo de K1 em Python.

```
#-------------------------------------------------------------------------
# Cálculo de K2 para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Densidade_Anterior_Mais_Meio=Densidade_Anterior+ 0.5*Delta_T*K1D

for i in range(len(Beta)):
  Concentracao_Anterior_Mais_Meio[i]=Concentracao_Anterior[i]+0.5*Delta_T*K1C[i]

Soma_Concentracao = 0.0

for i in range(len(Beta)):
  Soma_Concentracao=Soma_Concentracao+Lambda[i]*Concentracao_Anterior_Mais_Meio[i]

K2D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Meio+Soma_Concentracao

for i in range(len(Beta)):
  K2C[i]=(Beta[i]/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Meio-Lambda[i]*
    Concentracao_Anterior_Mais_Meio[i]
```

Figura 4.11- Cálculo de K2 em Python.

```
#-------------------------------------------------------------------------
# Cálculo de K3 para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5*Delta_T*K2D

for i in range(len(Beta)):
  Concentracao_Anterior_Mais_Meio[i] =Concentracao_Anterior[i]+0.5*Delta_T*K2C[i]

Soma_Concentracao = 0.0

for i in range(len(Beta)):
  Soma_Concentracao=Soma_Concentracao+Lambda[i]*Concentracao_Anterior_Mais_Meio[i]

K3D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Meio+Soma_Concentracao

for i in range(len(Lambda)):
  K3C[i] =(Beta[i]/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Meio-Lambda[i]
  *Concentracao_Anterior_Mais_Meio[i]
```

Figura 4.12- Cálculo de K3 em Python.

```
#-------------------------------------------------------------------------
# Cálculo de K4 para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Densidade_Anterior_Mais_Um=Densidade_Anterior+Delta_T*K3D

for i in range(len(Beta)):
  Concentracao_Anterior_Mais_Um[i]=Concentracao_Anterior[i]+Delta_T*K3C[i]

Soma_Concentracao = 0.0

for i in range(len(Beta)):
  Soma_Concentracao=Soma_Concentracao+Lambda[i]*Concentracao_Anterior_Mais_Um[i]

K4D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Um + Soma_Concentracao

for i in range(len(Beta)):
  K4C[i] =(Beta[i]/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Um-Lambda[i]*
    Concentracao_Anterior_Mais_Um[i]
```

Figura 4.13- Cálculo de K4 em Python.

```
#-------------------------------------------------------------------------
# Solução para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Densidade=Densidade_Anterior+(Delta_T/6.0)*(K1D+2.0*(K2D+K3D)+K4D)

for i in range(len(Beta)):
  Concentracao[i]=Concentracao_Anterior[i]+(Delta_T/6.0)*(K1C[i]+2.0*(K2C[i]+K3C[i])+K4C[i])
```

Figura 4.14- Solução para densidade de nêutrons e concentração de precursores em Python.

4.3 - C Sequencial

A linguagem C é considerada uma linguagem compilada de alto nível, estruturada, imperativa, procedural padronizada pela ISO. Ela foi criada por um cientista da computação chamado Dennis Ritchie no ano de 1972, na empresa AT&T Bell Labs com o objetivo de desenvolver o sistema operacional Unix (escrito em Assembler)(BETRYBE, 2020).

A linguagem C derivou-se de outras duas linguagens: a BCPL e a Algol 68 (BETRYBE, 2020). Ainda que tenha sidopensada no desenvolvimento do Unix, atualmente ela possui inúmeras aplicações, como por exemplo:

- Mercado de jogos eletrônicos.
- Editores de imagem e vídeo.
- Sistema de automação.
- Sistemas Operacionais.

Essa linguagem possui algumas vantagens, tais como:

- Geração de códigos rápidos com baixo tempo de execução.
- Estrutura simples e flexível.
- Portabilidade.
- Geração de código eficiente.
- Simplicidade.
- Confiabilidade.
- Fácil uso.
- Regularidade.

Portanto, a popularidade da linguagem C foi tão grande que influenciou a criação de outras estruturas e sintaxes de linguagens, como C++, Objective C, AWK, BitC, C#, C Shell, D, Euphoria, Java, Java Script, Perl, PHP e Python.

A figura 4.15 mostra a versão em C da função do método deRunge-Kutta que será utilizado para a obtenção da solução numérica das equações da cinética pontual de reatores. Já nas figuras 4.16 até 4.20, são mostrados os cálculos dos Ks e solução para densidade de nêutrons e concentração de precursores. Cada loop será paralelizado em C CUDA, com o mesmo objetivo, diminuir o tempo de execução do programa.

```
//-----------------------------------------------
// IMPLEMENTAÇÃO DA FUNÇÃO SISTEMA RUNGE KUTTA
//-----------------------------------------------
void Sistema_Runge_Kutta(void){


    //-----------------------------------------------
    // LAÇO ATÉ O TEMPO DE SIMULAÇÃO
    //-----------------------------------------------
    int n = 0;
    while (tempo <= tempo_simulacao){
        n += 1;
        tempo = tempo + delta_t;

        //-------------------------------------------------------------------------
        // Cálculo de k1 para densidade de nêutrons  e concentração de precursores
        //-------------------------------------------------------------------------
        //...
        //-------------------------------------------------------------------------
        // Cálculo de k2 para densidade de nêutrons  e concentração de precursores
        //-------------------------------------------------------------------------
        //...

        //-------------------------------------------------------------------------
        // Cálculo de k3 para densidade de nêutrons  e concentração de precursores
        //-------------------------------------------------------------------------
        //...

        //-------------------------------------------------------------------------
        // Cálculo de k4 para densidade de nêutrons  e concentração de precursores
        //-------------------------------------------------------------------------
        //...

        //-------------------------------------------------------------------------
        // Solução para densidade de nêutrons e concentração de precursores
        //-------------------------------------------------------------------------
        //...
```

Figura 4.15- Função Runge-Kutta do C.

```
//-------------------------------------------------------------------------
// Cálculo de k1 para densidade de nêutrons e concentração de precursores
//-------------------------------------------------------------------------
float soma_concentracao = 0.0;

for(i = 0; i < 6 ; i++){
    soma_concentracao = soma_concentracao + lambida[i]*concentracao_anterior[i];
}

K1D = ((reatividade - beta_total)/tempo_medio_geracao) * densidade_anterior + soma_concentracao;

for(i = 0; i < 6 ; i++){
    K1C[i] =(beta[i]/tempo_medio_geracao) * densidade_anterior-lambida[i]
              * concentracao_anterior[i];
}
```

Figura 4.16- Cálculo de K1 em C.

```
//------------------------------------------------------------------------
// Cálculo de k2 para densidade de nêutrons e concentração de precursores
//------------------------------------------------------------------------
densidade_anterior_mais_meio=densidade_anterior + 0.5 * delta_t * K1D;

for(i = 0; i < 6 ; i++){
    concentracao_anterior_mais_meio[i] = concentracao_anterior[i] + 0.5 *delta_t * K1C[i];
}

soma_concentracao = 0.0;

for(i = 0; i < 6 ; i++){
    soma_concentracao = soma_concentracao + lambida[i] * concentracao_anterior_mais_meio[i];
}

K2D =((reatividade-beta_total)/tempo_medio_gera ao) * densidade_anterior_mais_meio
       + soma_concentracao;
for(i = 0; i < 6 ; i++){
    K2C[i]=(beta[i]/tempo_medio_geracao) * densidade_anterior_mais_meio - lambida[i]
            * concentracao_anterior_mais_meio[i];
}
```

Figura 4.17- Cálculo de K2 em C.

```
//------------------------------------------------------------------------
// Cálculo de k3 para densidade de nêutrons e concentração de precursores
//------------------------------------------------------------------------
densidade_anterior_mais_meio = densidade_anterior + 0.5*delta_t*K2D;

for(i = 0; i < 6 ; i++){
    concentracao_anterior_mais_meio[i] = concentracao_anterior[i] + 0.5 * delta_t * K2C[i];
}

soma_concentracao = 0.0;

for(i = 0; i < 6 ; i++){
    soma_concentracao = soma_concentracao + lambida[i] * concentracao_anterior_mais_meio[i];
}

K3D = ((reatividade - beta_total)/tempo_medio_geracao) * densidade_anterior_mais_meio
       + soma_concentracao;
for(i = 0; i < 6 ; i++){
    K3C[i] =(beta[i]/tempo_medio_geracao)*densidade_anterior_mais_meio-lambida[i]
             *concentracao_anterior_mais_meio[i];
}
```

Figura 4.18- Cálculo de K3 em C.

```
//---------------------------------------------------------------------
// Cálculo de k4 para densidade de nêutrons e concentração de precursores
//---------------------------------------------------------------------
densidade_anterior_mais_um = densidade_anterior+delta_t*K3D;

for(i = 0; i < 6 ; i++){
    concentracao_anterior_mais_um[i] = concentracao_anterior[i]+delta_t*K3C[i];
}

soma_concentracao = 0.0;

for(i = 0; i < 6 ; i++){
    soma_concentracao=soma_concentracao+lambida[i]*concentracao_anterior_mais_um[i];
}

K4D =((reatividade-beta_total)/tempo_medio_geracao)*densidade_anterior_mais_um
        + soma_concentracao;
for(i = 0; i < 6 ; i++){
    K4C[i] =(beta[i]/tempo_medio_geracao)*densidade_anterior_mais_um
              -lambida[i]*concentracao_anterior_mais_um[i];
}
```

Figura 4.19- Cálculo de K4 em C.

```
//---------------------------------------------------------------------
// Solução para densidade de nêutrons e concentração de precursores
//---------------------------------------------------------------------
densidade = densidade_anterior + (delta_t/6.0)*(K1D+2.0*(K2D+K3D)+K4D);

for(i = 0; i < 6 ; i++){
    concentracao[i] = concentracao_anterior[i]
                      + (delta_t/6.0)*(K1C[i]+2.0*(K2C[i]+K3C[i])+K4C[i]);
}
```

Figura 4.20- Solução para densidade de nêutrons e concentração de precursores em C.

4.4 – Fortran Paralelo

Com o objetivo de otimização do código em Fortran, foi introduzido o comando *Forall*, em substiuição ao construto *Do*, como forma de paralelismo em uma CPU.

Na execução do comando *Do*, o processador realiza cada iteração sucessiva em ordem, com uma atribuição seguida da outra. Em uma plataforma paralelizada isto representa um grande problema para a otimização do código. Neste sentido, este problema foi enfatizado com a construção do HPF (*High Performance* Fortran), que consiste em uma versão do Fortran 90 para sistemas de computação paralela. Após avanços realizados pelo HPF, em relação ao Fortran 90, foram incorporados ao Fortran 95, entre eles o comando *Forall* (SILVA, 2011).

Objetivo do *Forall* é oferecer ao programador uma estrutura que ofereça os mesmos recursos alcançados com o *loop Do*, mas que sejam paralelizáveis automaticamente, no momento em que o programa estiver executando em uma plataforma paralela (SILVA, 2011).

A instrução *Forall* é implementadano cálculo das condições iniciais no programa principal da solução das equações da Cinética Pontual de Retores, como pode ser visualizado na figura 4.21. O *Forall* também é implementado no cálculo dos Ks e na Solução para densidade de nêutrons e concentração de precursores, como mostra as figuras 4.22 até 4.26.

```
!   Calcula as condições iniciais
    Densidade_Anterior = 1.0d+00
    forall (i = 1: 6)
      Concentracao_Anterior (i) = ( Beta (i) * Densidade_Anterior ) / ( Lambda (i) * Tempo_Medio_Geracao )
    end forall
```

Figura 4.21 – Cálculo de Condições Iniciais para o *forall*.

```
Cálculo de K1 para densidade de nêutrons e concentração de precursores
Soma_Concentracao = 0.0d+00
do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior (i)
end do

K1D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior + Soma_Concentracao

forall (i = 1: 6)
  K1C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior - lambda (i) * Concentracao_Anterior (i)
end forall
```

Figura 4.22 - Cálculo de K1 para densidade de nêutrons e concentração de precursores para o *forall*.

```
Cálculo de K2 para densidade de nêutrons e concentração de precursores
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5d+00 * Delta_T * K1D

forall (i = 1: 6)
  Concentracao_Anterior_Mais_Meio (i) = Concentracao_Anterior (i) + 0.5d+00 * Delta_T * K1C (i)
end forall

Soma_Concentracao = 0.0d+00

do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end do

K2D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio + Soma_Concentracao

forall (i = 1: 6)
  K2C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio - lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end forall
```

Figura 4.23 - Cálculo de K2 para densidade de nêutrons e concentração de precursores para o *forall*.

```
Cálculo de K3 para densidade de nêutrons e concentração de precursores
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5d+00 * Delta_T * K2D

forall (i = 1: 6)
  Concentracao_Anterior_Mais_Meio (i) = Concentracao_Anterior (i) + 0.5d+00 * Delta_T * K2C (i)
end forall

Soma_Concentracao = 0.0d+00

do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end do

K3D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio + Soma_Concentracao

forall (i = 1: 6)
  K3C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Meio - lambda (i) * Concentracao_Anterior_Mais_Meio (i)
end forall
```

Figura 4.24 - Cálculo de K3 para densidade de nêutrons e concentração de precursores para o *forall*.

```
Cálculo de K4 para densidade de nêutrons e concentração de precursores
Densidade_Anterior_Mais_Um = Densidade_Anterior + Delta_T * K3D

forall (i = 1: 6)
  Concentracao_Anterior_Mais_Um (i) = Concentracao_Anterior (i) + Delta_T * K3C (i)
end forall

Soma_Concentracao = 0.0d+00

do i = 1, 6
 Soma_Concentracao = Soma_Concentracao + lambda (i) * Concentracao_Anterior_Mais_Um (i)
end do

K4D = ( ( Reatividade - Beta_Total ) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Um + Soma_Concentracao

forall (i = 1: 6)
  K4C (i) = ( Beta (i) / Tempo_Medio_Geracao ) * Densidade_Anterior_Mais_Um - lambda (i) * Concentracao_Anterior_Mais_Um (i)
end forall
```

Figura 4.25 - Cálculo de K4 para densidade de nêutrons e concentração de precursores para o *forall*.

```
Solucção para densidade de nêutrons e concentração de precursores
Densidade = Densidade_Anterior + ( Delta_T / 6.0d+00 ) * ( K1D + 2.0d+00 * ( K2D + K3D ) + K4D )

forall (i = 1: 6)
  Concentracao (i) = Concentracao_Anterior (i) + ( Delta_T / 6.0d+00 ) * ( K1C (i) + 2.0d+00 * ( K2C (i) + K3C (i) ) + K4C (i) )
end forall
```

Figura 4.26 - Solução para densidade de nêutrons e concentração de precursores para o *forall*.

4.5 - Python CUDA

Como já mencionado anteriormente, para diminuir o tempo de execução, foi feito o paralelismo do *loop* principal *While* e dos *loops* internos *For* de cada K e também da solução para densidade de nêutrons e concentração de precursores.

Na figura 4.27 abaixo, é utilizado o loop Grid-Stride em substituição do loop while do código original que faz 100 milhões de cálculos das equações da cinética de reatores. Desse modo, usa-se o loop Grid-Stride para uma maior eficiência do código em paralelo, como é mencionado na figura 3.32 da subseção 3.5.10.

```
@cuda.jit
def Runge_Kutta_func_cuda():
  #-------------------------------------------------------------------------
  # Laço até o tempo de simulação
  #-------------------------------------------------------------------------
  n = 0
  tx = cuda.threadIdx.x
  ty = cuda.blockIdx.x
  bw = cuda.blockDim.x
  i = tx + ty * bw

  for i in range(Tempo_Simulacao):
    Tempo[0] = Tempo[0] + Delta_T[0]
    n += 1

    #-------------------------------------------------------------------------
    # Cálculo de K1 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...
    #-------------------------------------------------------------------------
    # Cálculo de K2 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Cálculo de K3 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Cálculo de K4 para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...

    #-------------------------------------------------------------------------
    # Solução para densidade de nêutrons e concentração de precursores
    #-------------------------------------------------------------------------
    #...
```

Figura 4.27- Função Runge-Kutta do Python CUDA.

A figura 4.28 mostra o paralelismo do K1, sendo que nesse caso foi criada uma função paralela que além de receber as variáveis de entrada, essa função possui também entradas das *threads* por blocos e blocos por *grid*, ao qual será feito o paralelismo.

```
1 #-------------------------------------------------------------------------
2 # Cálculo de K1 para densidade de nêutrons e concentração de precursores
3 #-------------------------------------------------------------------------
4 Soma_Concentracao_func_jit_cuda[threadsperblock, blockspergrid](device_Beta_Arr,
5                                                                 device_Lambda_Arr,
6                                                                 device_Soma_Concentracao_Arr,
7                                                                 device_Concentracao_Anterior_Arr)
8 cuda.synchronize()
9
10 K1D = ((Reatividade - Beta_Total)/Tempo_Medio_Geracao)
11       * Densidade_Anterior
12       + numpy.sum(device_Soma_Concentracao_Arr.copy_to_host())
13
14 K1C_func_jit_cuda[threadsperblock, blockspergrid](device_K1C_Arr,
15                                                  device_Beta_Arr,
16                                                  Tempo_Medio_Geracao,
17                                                  Densidade_Anterior,
18                                                  device_Lambda_Arr,
19                                                  device_Concentracao_Anterior_Arr)
20 cuda.synchronize()
```

Figura 4.28- Cálculo de K1 para Python CUDA.

Na figura 4.28, as funções criadas para paralelizar os *loops* estão na linha 4 e linha 14 do código, respectivamente: Soma_Concentracao_func_fit_cuda e K1C_func_jit_cuda. É necessário ainda, na programação paralela a sincronização das *threads* utilizando a função pré-definida do python CUDA chamada cuda.python(), que tem como objetivo evitar problemas de concorrência, pois assim as *threads* passam a executar em sincronia umas com as outras. O sincronismo impede que duas ou mais *threads* acessem o mesmo recurso ao mesmo tempo (DEVMEDIA, 2016).

As funções paralelas da figura 4.28 foram expandidas na figura 4.29 e 4.30:

```
#----------------------------
# Soma Concentração
#----------------------------
@cuda.jit
def Soma_Concentracao_func_jit_cuda(Beta, Lambda, Soma_Concentracao, Concentracao_Anterior):
  i = cuda.grid(1)
  Soma_Concentracao[i-1] = 0.0
  if i < Beta.size:
     Soma_Concentracao[i] = Soma_Concentracao[i-1] + Lambda[i] * Concentracao_Anterior[i]
```

Figura 4.29- Função paralela Soma Concentração.

```
#----------------------------
# K1C
#----------------------------
@cuda.jit
def K1C_func_jit_cuda(K1C,
                      Beta,
                      Tempo_Medio_Geracao,
                      Densidade_Anterior,
                      Lambda,
                      Concentracao_Anterior):
  j = cuda.grid(1)
  if j < Lambda.size:
    K1C[j] =(Beta[j]/Tempo_Medio_Geracao)
              *Densidade_Anterior-Lambda[j]*Concentracao_Anterior[j]
```

Figura 4.30- Função paralela K1C.

As funções paralelas das figuras 4.29 e 4.30 usam o *Grid-Stride loop* do algoritmo CUDA mostrado anteriormente na figura 3.31. Nesse sentido, todas as funções paralelas do código também usarão esse mesmo algoritmo, devido a sua eficiência no paralelismo como já mencionado na subseção 3.5.10.

As figuras 4.31 até 4.34 mostram o restante dos cálculos para K2, K3, K4 e a solução para densidade de nêutrons e concentração de precursores, respectivamente.

```
#-------------------------------------------------------------------------
# Cálculo de K2 para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5 * Delta_T * K1D

Concentracao_Anterior_Mais_Meio_func_cuda[threadsperblock, blockspergrid](
    device_Concentracao_Anterior_Mais_Meio_Arr,device_Beta_Arr,
    device_Concentracao_Anterior_Arr,Delta_T,device_K1C_Arr)

cuda.synchronize()

Soma_Concentracao_func_jit_cuda[threadsperblock, blockspergrid](
    device_Beta_Arr, device_Lambda_Arr, device_Soma_Concentracao_Arr,
    device_Concentracao_Anterior_Mais_Meio_Arr)

cuda.synchronize()

K2D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)
      *Densidade_Anterior_Mais_Meio
      + numpy.sum(device_Soma_Concentracao_Arr.copy_to_host())

K2C_func_jit_cuda[threadsperblock, blockspergrid](
    device_K2C_Arr, device_Beta_Arr, Tempo_Medio_Geracao,
    Densidade_Anterior_Mais_Meio,device_Lambda_Arr,
    device_Concentracao_Anterior_Mais_Meio_Arr)

cuda.synchronize()
```

Figura 4.31- Cálculo de K2 para Python CUDA.

```
#-------------------------------------------------------------------------
# Cálculo de K3 para densidade de nêutrons e concentração de precursores
#-------------------------------------------------------------------------
Densidade_Anterior_Mais_Meio = Densidade_Anterior + 0.5 * Delta_T * K2D

Concentracao_Anterior_Mais_Meio_func_cuda[threadsperblock, blockspergrid](
    device_Concentracao_Anterior_Mais_Meio_Arr,device_Beta_Arr,
    device_Concentracao_Anterior_Arr,Delta_T,device_K2C_Arr)
cuda.synchronize()

Soma_Concentracao_func_jit_cuda[threadsperblock, blockspergrid](
    device_Beta_Arr, device_Lambda_Arr, device_Soma_Concentracao_Arr,
    device_Concentracao_Anterior_Mais_Meio_Arr)
cuda.synchronize()

K3D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)
      *Densidade_Anterior_Mais_Meio + numpy.sum(device_Soma_Concentracao_Arr.copy_to_host()

K3C_func_jit_cuda[threadsperblock, blockspergrid](
    device_K3C_Arr, device_Beta_Arr, Tempo_Medio_Geracao,
    Densidade_Anterior_Mais_Meio,device_Lambda_Arr,device_Concentracao_Anterior_Mais_Meio_Arr)
cuda.synchronize()
```

Figura 4.32- Cálculo de K3 para Python CUDA.

```
#-----------------------------------------------------------------------
# Cálculo de K4 para densidade de nêutrons e concentração de precursores
#-----------------------------------------------------------------------
Densidade_Anterior_Mais_Um=Densidade_Anterior+Delta_T*K3D

Concentracao_Anterior_Mais_Um_func_cuda[threadsperblock, blockspergrid](
    device_Concentracao_Anterior_Mais_Um_Arr,device_Beta_Arr,
    device_Concentracao_Anterior_Arr,Delta_T,device_K3C_Arr)
cuda.synchronize()

Soma_Concentracao_func_jit_cuda[threadsperblock, blockspergrid](
    device_Beta_Arr, device_Lambda_Arr, device_Soma_Concentracao_Arr,
    device_Concentracao_Anterior_Mais_Um_Arr)
cuda.synchronize()

K4D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)
      *Densidade_Anterior_Mais_Um + numpy.sum(device_Soma_Concentracao_Arr.copy_to_host())

K4D =((Reatividade-Beta_Total)/Tempo_Medio_Geracao)
      *Densidade_Anterior_Mais_Um + numpy.sum(device_Soma_Concentracao_Arr.copy_to_host())

K4C_func_jit_cuda[threadsperblock, blockspergrid](
    device_K4C_Arr,device_Beta_Arr,Tempo_Medio_Geracao,
    Densidade_Anterior_Mais_Um,device_Lambda_Arr,device_Concentracao_Anterior_Mais_Um_Arr)
cuda.synchronize()
```

Figura 4.33- Cálculo de K4 para Python CUDA.

```
#-----------------------------------------------------------------------
# Solução para densidade de nêutrons e concentração de precursores
#-----------------------------------------------------------------------
Densidade = Densidade_Anterior+(Delta_T/6.0)*(K1D+2.0*(K2D+K3D)+K4D)

Concentracao_func_cuda[threadsperblock, blockspergrid](
    device_Concentracao_Arr,device_Beta_Arr,
    device_Concentracao_Anterior_Arr,Delta_T,device_K1C_Arr,
    device_K2C_Arr,device_K3C_Arr,device_K4C_Arr)
cuda.synchronize()
```

Figura 4.34 - Solução para densidade de nêutrons e concentração de precursores em Python CUDA.

As figuras 4.35 até 4.41 mostram as funções paralelas dos Ks e a solução para densidade de nêutrons e concentração de precursores.

```
#----------------------------
# Concentração Anterior
#----------------------------
@cuda.jit
def Concentracao_Anterior_func_jit_cuda(
    Beta, Lambda, Soma_Concentracao, Concentracao_Anterior):
  i = cuda.grid(1)
  if i < Beta.size:
     Concentracao_Anterior[i]=(Beta[i]*Densidade_Anterior)
      /(Lambda[i] * Tempo_Medio_Geracao);
```

Figura 4.35- Função paralela Concentração Anterior.

```
#---------------------------------
# Concentração Anterior Mais Meio
#---------------------------------
@cuda.jit
def Concentracao_Anterior_Mais_Meio_func_cuda(
    Concentracao_Anterior_Mais_Meio,Beta,Concentracao_Anterior,Delta_T,K1C):
  j = cuda.grid(1)
  if j < Beta.size:
    Concentracao_Anterior_Mais_Meio[j] = Concentracao_Anterior[j] + 0.5
                                        * Delta_T * K1C[j]
```

Figura 4.36- Função paralela Concentração Anterior Mais Meio.

```
#---------------------------------
# K2C
#---------------------------------
@cuda.jit
def K2C_func_jit_cuda(
    K2C,Beta,Tempo_Medio_Geracao,Densidade_Anterior_Mais_Meio,
    Lambda,Concentracao_Anterior_Mais_Meio):
  j = cuda.grid(1)
  if j < Beta.size:
    K2C[j]=(Beta[j]/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Meio
              -Lambda[j]*Concentracao_Anterior_Mais_Meio[j]
```

Figura 4.37- Função paralela K2C.

```
#----------------------------
# K3C
#----------------------------
@cuda.jit
def K3C_func_jit_cuda(
    K3C,Beta,Tempo_Medio_Geracao,
    Densidade_Anterior_Mais_Meio,Lambda,Concentracao_Anterior_Mais_Meio):
  j = cuda.grid(1)
  if j < Beta.size:
    K3C[j]=(Beta[j]/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Meio
            -Lambda[j]*Concentracao_Anterior_Mais_Meio[j]
```

Figura 4.38- Função paralela K3C.

```
#--------------------------------
# Concentração Anterior Mais Um
#--------------------------------
@cuda.jit
def Concentracao_Anterior_Mais_Um_func_cuda(
    Concentracao_Anterior_Mais_Um,Beta,Concentracao_Anterior,Delta_T,K3C):
  j = cuda.grid(1)
  if j < Beta.size:
    Concentracao_Anterior_Mais_Um[j] = Concentracao_Anterior[j] + Delta_T * K3C[j]
```

Figura 4.39- Função Concentração Anterior Mais Um.

```
#----------------------------
# K4C
#----------------------------
@cuda.jit
def K4C_func_jit_cuda(
    K4C,Beta,Tempo_Medio_Geracao,Densidade_Anterior_Mais_Um,
    Lambda,Concentracao_Anterior_Mais_Um):
  j = cuda.grid(1)
  if j < Beta.size:
    K4C[j] =(Beta[j]/Tempo_Medio_Geracao)*Densidade_Anterior_Mais_Um-Lambda[j]
    *Concentracao_Anterior_Mais_Um[j]
```

Figura 4.40- Função paralela K4C.

```
#-------------------------------------------------------------------------
# Concentração
#-------------------------------------------------------------------------
@cuda.jit
def Concentracao_func_cuda(
    Concentracao,Beta,Concentracao_Anterior,Delta_T,K1C,K2C,K3C,K4C):
  j = cuda.grid(1)
  if j < Beta.size:
    Concentracao[j] = Concentracao_Anterior[j] + (Delta_T/6.0)
                      * (K1C[j] + 2.0* (K2C[j] + K3C[j]) + K4C[j])
```

Figura 4.41- Função paralela Concentração.

4.6 - C CUDA

O código sequencial na seção 4.3 foi paralelizado também usando o mesmo princípio, ou seja, implementando o paralelismo sobre o *loop* principal *While* e também em cada *loop For* desse código em C sequencial, usando o *loop* *Grid-Stride*.

A figura 4.42 mostra o loop Grid-Stride do C CUDA em substituição ao loop principal While do código em Fortran. Como já mencionado, esse loop principal irá realiza 100 milhões de cálculos para obtenção da solução numérica das equações da cinética de reatores para um tempo de simulação de 100 s. Sendo assim, torna-se necessário paralelizá-lo para diminuição do tempo de execução do programa.

```
__global__ void Sistema_Runge_Kutta(){

    //-----------------------------------------------
    // LAÇO ATÉ O TEMPO DE SIMULAÇÃO
    //-----------------------------------------------
    int n = 0;
    tempo = 0.0;

    for (int j = blockIdx.y * blockDim.y + threadIdx.y;
         j <= tempo_simulacao ; j += blockDim.y * gridDim.y){

        tempo = tempo + delta_t;
        n+=1;

        //-------------------------------------------------------------------------
        // Cálculo de k1 para densidade de nêutrons e concentração de precursores
        //-------------------------------------------------------------------------
        //...
        //-------------------------------------------------------------------------
        // Cálculo de k2 para densidade de nêutrons e concentração de precursores
        //-------------------------------------------------------------------------
        //...

        //-------------------------------------------------------------------------
        // Cálculo de k3 para densidade de nêutrons e concentração de precursores
        //-------------------------------------------------------------------------
        //...

        //-------------------------------------------------------------------------
        // Cálculo de k4 para densidade de nêutrons e concentração de precursores
        //-------------------------------------------------------------------------
        //...

        //-------------------------------------------------------------------
        // Solução para densidade de nêutrons e concentração de precursores
        //-------------------------------------------------------------------
        //...
}
```

Figura 4.42- Função Runge-Kutta do C CUDA.

As figuras 4.43 até 4.47 mostram os cálculos paralelizados, também usando loops Grid-Stride sobre os Ks e a solução para densidade de nêutrons e concentração de precursores.

```
//-------------------------------------------------------------------------
// Cálculo de k1 para densidade de nêutrons e concentração de precursores
//-------------------------------------------------------------------------
for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         soma_concentracao[i] = soma_concentracao[i-1]
                              + lambida[i]*concentracao_anterior[i];
}

K1D = ((reatividade - beta_total)/tempo_medio_geracao) * densidade_anterior
        + soma_concentracao[5];

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x){
         K1C[i] =(beta[i]/tempo_medio_geracao) * densidade_anterior
                  - lambida[i] * concentracao_anterior[i];
}
```

Figura 4.43- Cálculo de K1 para C CUDA.

```
//-------------------------------------------------------------------------
// Cálculo de k2 para densidade de nêutrons e concentração de precursores
//-------------------------------------------------------------------------
densidade_anterior_mais_meio = densidade_anterior + 0.5 * delta_t * K1D;

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         concentracao_anterior_mais_meio[i] = concentracao_anterior[i]
                                            + 0.5 *delta_t * K1C[i];
}

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         soma_concentracao[i] = soma_concentracao[i-1] + lambida[i]
                              * concentracao_anterior_mais_meio[i];
}

K2D = ((reatividade-beta_total)/tempo_medio_geracao)
       * densidade_anterior_mais_meio + soma_concentracao[5];

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         K2C[i]=(beta[i]/tempo_medio_geracao) * densidade_anterior_mais_meio
                  - lambida[i] * concentracao_anterior_mais_meio[i];
}
```

Figura 4.44- Cálculo de K2 para C CUDA.

```
//--------------------------------------------------------------------------
// Cálculo de k3 para densidade de nêutrons e concentração de precursores
//--------------------------------------------------------------------------
densidade_anterior_mais_meio = densidade_anterior + 0.5*delta_t*K2D;

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         concentracao_anterior_mais_meio[i] = concentracao_anterior[i]
                                              + 0.5 * delta_t * K2C[i];
}

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         soma_concentracao[i] = soma_concentracao[i-1] + lambida[i]
                                * concentracao_anterior_mais_meio[i];
}

K3D = ((reatividade - beta_total)/tempo_medio_geracao) *
        densidade_anterior_mais_meio + soma_concentracao[5];

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         K3C[i] =(beta[i]/tempo_medio_geracao) * densidade_anterior_mais_meio
                  - lambida[i] * concentracao_anterior_mais_meio[i];
}
```

Figura 4.45- Cálculo de K3 para C CUDA.

```
//--------------------------------------------------------------------------
// Cálculo de k4 para densidade de nêutrons e concentração de precursores
//--------------------------------------------------------------------------
densidade_anterior_mais_um = densidade_anterior + delta_t * K3D;

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         concentracao_anterior_mais_um[i] = concentracao_anterior[i]
                                            +delta_t*K3C[i];
}

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         soma_concentracao[i] = soma_concentracao[i-1] + lambida[i]
                                * concentracao_anterior_mais_um[i];
}

K4D = ((reatividade - beta_total)/tempo_medio_geracao)
        * densidade_anterior_mais_um + soma_concentracao[5];

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x) {
         K4C[i] =(beta[i] / tempo_medio_geracao)
                  * densidade_anterior_mais_um - lambida[i]
                  * concentracao_anterior_mais_um[i];
}
```

Figura 4.46- Cálculo de K4 para C CUDA.

```
//------------------------------------------------------------------------
// Solução para densidade de nêutrons e concentração de precursores
//------------------------------------------------------------------------
densidade = densidade_anterior + (delta_t/6.0)*(K1D + 2.0*(K2D + K3D) + K4D);

for (int i = blockIdx.x * blockDim.x + threadIdx.x;
     i < SIZE; i += blockDim.x * gridDim.x){
         concentracao[i] = concentracao_anterior[i]
                           + (delta_t/6.0)*(K1C[i]+2.0*(K2C[i]+K3C[i])+K4C[i]);
}
```

Figura 4.47- Solução para densidade de nêutrons e concentração de precursores em C CUDA.

CAPÍTULO 5

Análise de Resultados

5.1 - Validação do Método de Runge-Kutta

Para a validação do método numérico, comparamos os resultados obtidos pelas equações da cinética pontual de reatores, dadas pelas Eqs. (2.24) e (2.25), usando o método de Runge-Kutta de 4ª Ordem, com os resultados de referência obtidos pelo Método de Diferenças Finitas (MDF) (SILVA, 2007).

Utilizamos os seguintes parâmetros cinéticos nas simulações: $\Lambda = 0.00002s$, β_i = (0.000266, 0.001491,0.001316, 0.002849, 0.000896, 0.000182), $\beta = 0.007$, $\lambda_i(s^{-1})$= (0.0127, 0.0317, 0.115,0.311, 1.4, 3.87). O intervalo de tempo considerado nos cálculos foi de $\Delta t = 10^{-6}s$, com densidade inicial de nêutrons $n_o = 1$.

Estes parâmetros foram introduzidos nos algoritmos computacionais (Fortran, Fortran Paralelo, C, C Paralelo, Python, Pyton Paralelo) para avaliar o comportamento da densidade de nêutrons e da concentração de precursores no reator.

Nas tabelas 5.1 e 5.2 são apresentados os valores da densidade de nêutrons utilizando o método de Runge-Kutta de 4ª Ordem e o Método de Diferenças Finitas (SILVA, 2007) para alguns valores de reatividade e instantes de tempo.

Tabela 5.1 – Densidade de nêutrons $n(t)$ para step de reatividade $\rho = 0.003$

Método	Linguagem	$t = 1s$	$t = 10s$	$t = 20s$
RK4	Fortran	2.2098	8.0191	28.297
	Fortran Paralelo	2.2098	8.0191	28.297
	C	2.2098	8.0191	28.297
	C Paralelo	2.2098	8.0191	28.297
	Python	2.2098	8.0191	28.297
	Python Paralelo	2.2098	8.0191	28.297
MDF	Fortran	2.2098	8.0192	28.297

Tabela 5.2 – Densidade de nêutrons $n(t)$ para step de reatividade $\rho = 0.007$

Método	Linguagem	$t = 0.01s$	$t = 0.5s$	$t = 2s$
RK4	Fortran	4.5088	5.3458×10^{3}	2.0591×10^{11}
	Fortran Paralelo	4.5088	5.3458×10^{3}	2.0591×10^{11}
	C	4.5088	5.3458×10^{3}	2.0591×10^{11}
	C Paralelo	4.5088	5.3458×10^{3}	2.0591×10^{11}
	Python	4.5088	5.3458×10^{3}	2.0591×10^{11}
	Python Paralelo	4.5088	5.3458×10^{3}	2.0591×10^{11}
MDF	Fortran	4.5088	5.3457×10^{3}	2.0589×10^{11}

Analisando os resultados obtidos pelas tabelas 5.1 e 5.2, podemos verificar que o Método de Runge-Kutta de 4ª ordem aplicado às equações da cinética pontual de reatores é bastante satisfatório, independente da linguagem de programação utilizada. Sendo assim, iremos analisar o comportamento da densidade de nêutrons para uma dada inserção de reatividade no núcleo do reator, de modo, a poder avaliar o desempenho computacional de cada linguagem computacional utilizada nessa dissertação.

5.2 – Desempenho das Linguagens Computacionais

Nesta seção, será avaliado o desempenho das linguagens computacionais: Fortran, Python, Python CUDA, C, C CUDA e plataforma Colab Google, com aceleração via CPU e GPU. A plataforma Colab Google, ou Google *Colaboritory*, é um serviço de nuvem gratuito hospedado pelo Google, onde são disponiblizadas placas CUDAs para testes. Desse modo, será medido os tempos computacionais gastos em cada simulação realizada tanto na versão sequencial quanto na versão paralela. Vale ainda destacar, que foi usado nesta dissertação quatro configurações de máquinas para os testes computacionais, como é mostrado na tabela 5.3:

Tabela 5.3 - Configurações das máquinas utilizadas

	CPU 1	**CPU 2**	**CPU 3**	**CPU 4**	**GPU 1**	**GPU 2**	**GPU3**	**GPU4**
Processador	Ryzen 3600 XT	Ryzen 3700 X	CORE I5 7500	INTEL Xeon	RTX 3070 Ti	RTX 2060	GTX 1070	Tesla K80
Núcleos	8	8	4	2	6144	1920	1920	2496
Threads	16	16	4	2	-	-	-	-
Base Clock	3.6 GHz	3.6 GHz	3.40 GHz	2.20 GHz	1575 MHz	1365 MHz	1506 MHz	562 MHz
BoostClock	4.5 GHz	4.4 GHz	3.8 GHz	-	1770 MHz	1680 MHz	1683 MHz	824 MHz
MemoryClock	3200 Mhz	3200 Mhz	3200 Mhz	2199 Mhz	1188 Mhz	1750 MHz	2002 MHz	1253 MHz
MemorySize	32GB	32GB	32GB	12GB	8 GB	6 GB	8 GB	12 GB
Cache L3	32 MB	32 MB	6 MB	5 MB	-	-	-	-

A tabela 5.4 mostra os resultados comparativos entre os tempos de execução dos algoritmos sequencial e paralelo, usando os processadores: CPU 1 (Ryzen 3600 XT), CPU 2 (Ryzen 3700 X), CPU 3 (Intel Core-I5 7500), CPU 4 (Intel XEON Colab), GPU 1 (RTX 3070 Ti), GPU 2 (RTX 2060), GPU 3 (GTX 1070) e GPU 4 (NVIDIA Tesla K80 Colab).

Ao se analisar a tabela 5.4, é possível observar que o tempo de execução do algoritmo em Fortran é um pouco mais rápido do que o algoritmo em: Fortran paralelo,

sequencial em C; sequencial em Python. Porém, verifica-se que o tempo de execução do algoritmo em Fortran é mais lento do que os tempos de execuções dos algoritmos paralelos em C e Python, ou seja, comprovando-se assim, a eficácia dos algoritmos paralelos sobre os algoritmos sequenciais.

A partir dos tempos de execução da tabela 5.4, é possível constatar ainda, que as linguagens compiladas Fortran e C são mais rápidas que a linguagem interpretada Python.

É possível também verificar na tabela 5.4, que os tempos de execução da plataforma Colab dos algoritmos paralelos são mais rápidos que os algoritmos sequenciais executados nessa mesma plataforma. Porém, ao se comparar os tempos de execução dos algoritmos da plataforma Colab com os tempos das demais máquinas usadas neste trabalho, verificou-se que o Colab é mais lento, devido a alguns fatores: Primeiramente, o tempo de latência da rede, cujo é o tempo que a informação leva do computador original para chegar ao computador de origem, é lento. Em segundo lugar, os usuários da plataforma Colab que usam a versão gratuita, possuem uma configuração mais básica, onde são disponibilizados processadores, memória e GPU menos potentes do que para usuários assinantes de pacotes mais completos, que possuem acesso a máquinas de última geração. Por fim, os processadores usados pela plataforma Colab (CPU 4 e GPU 4) são menos potentes que os demais processadores testados nesta dissertação (CPU 1, CPU 2, CPU 3, GPU 1, GPU 2 e GPU 3).

A partir dos dados da tabela 5.4, para uma visualização melhor de desempenho, podemos fazer um gráfico comparativo entre os tempos de execução das CPUs e GPUs (Figura 5.1). Analisando sob o ponto de vista das características físicas dos processadores, é possível averiguar que a CPU 1 foi a unidade de processamento que teve um melhor desempenho computacional, seguido da CPU 2, CPU 3 e CPU 4. Isso é devido ao fato da CPU 1 possuir um maior número de núcleos, maior número de threads, maior frequência de *clock*, maior tamanho de memória, maior cache L3 do que as demais, como foi mostrado na tabela 5.1. Seguindo esse mesmo raciocínio, é possível constatar que a CPU 2 é mais rápida que a CPU 3, que por sua vez é mais rápida que a CPU 4. Estendendo essa análise às GPUs, verifica-se que elas possuem características bem semelhantes, porém é possível constatar que a GPU 1 tem melhor desempenho do

que as demais GPUs, devido ao fato de ter um maior número de núcleos. Por fim, as GPUs possuem um desempenho computacional melhor que as CPUs, devido ao fato de terem também uma maior quantidade de núcleos.

Tabela 5.4 - Tempo de execução dos processadores e placas de vídeo

Tempos (s)	**CPU 1** Ryzen 3600XT	**CPU 2** Ryzen 3700 X	**CPU 3** Core I5 7500	**CPU 4** Intel Xeon	**GPU 1** RTX 3070Ti	**GPU 2** RTX 2060	**GPU 3** GTX 1070	**GPU 4** Tesla K80
C	3,08s	3,19 s	11,66s	23,30s	-	-	-	-
Fortran	2,85s	2,82 s	3,87s	-	-	-	-	-
Fortran Paralelo	3,00s	3,00 s	4,10s	-	-	-	-	-
C Paralelo	-	-	-	-	0,33s	0,37s	0,40s	1,79s
Python	1102,2s	1359,6s	1359,6s	2537,4s	-	-	-	-
Python Paralelo	-	-	-	-	2,69s	2,93s	4,32s	4,45s

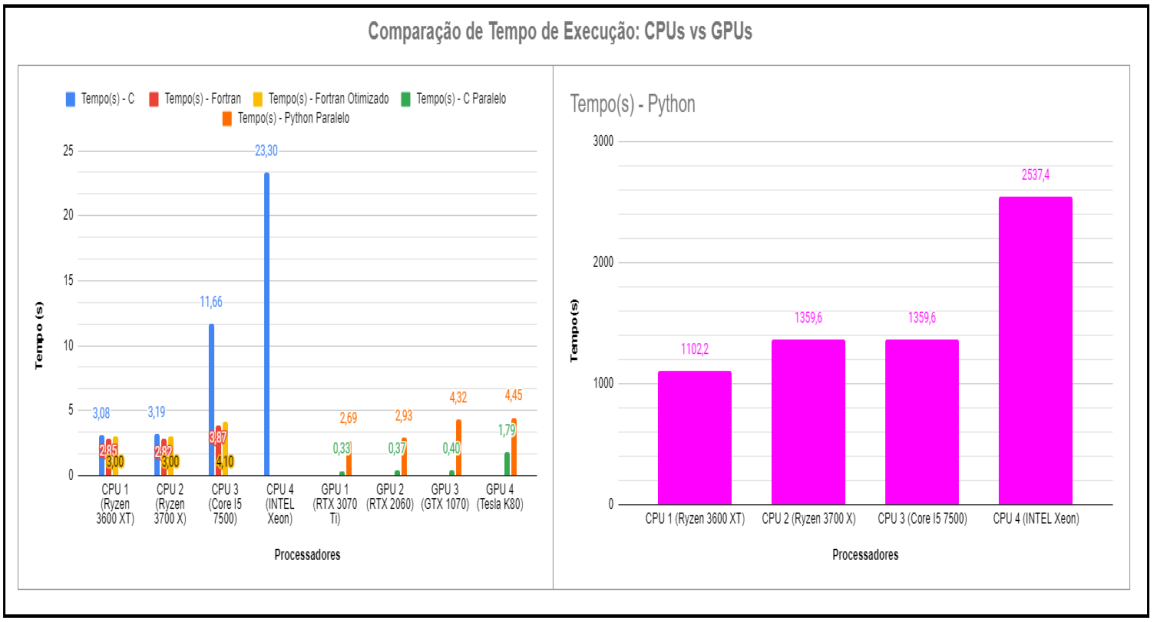

Figura 5.1- Tempos de execução

Na tabela 5.4, como já mencionado anteriormente, nota-se que o tempo de execução do Fortran é menor que o Fortran paralelo, logo isso é um erro. Como o tempo de execução do Fortran não pode ser menor que o Fortran paralelo, foi decidido então, usar dois processadores adicionais para verificar qual foi a falha. A tabela 5.5 mostra as configurações dos processadores extras utlizados neste trabalho, que foram chamados de CPU 5 e CPU 6. A tabela 5.6 mostra os resultados dos tempos de execuções do Fortran e Fortran paralelo, constatando assim que o tempo de execução do Fortran paralelo é menor que o tempo de execução do Fortran. Nesse sentido, achamos que possívelmente o erro vem da instalação ou configuração do software para os processadores: CPU 1, CPU 2, CPU 3, GPU 1, GPU 2 e GPU 3. Dessa forma, constatou-se que o Fortran paralelo é mais rápido que o Fortran, como pode ser visualizado na figura 5.2.

Tabela 5.5 - Configurações das CPU 5 e CPU 6.

	CPU 5	**CPU 6**
Processador	Core I7 4770	Core I7 4790K
Núcleos	4	4
Threads	8	8
Base Clock	3.40 GHz	4.00 GHz
BoostClock	3.90 GHz	4.40 GHz
MemoryClock	3200 Mhz	3200 Mhz
MemorySize	32 GB	16 GB
Cache L3	8 MB	8 MB

Tabela 5.6 - Tempos de execução das CPU 5 e CPU 6

Tempos (s)	**CPU 5** Core I7 4770	**CPU 6** Core I7 4790K
Fortran	32,05 s	27,80 s
Fortran Paralelo	4,21 s	3,14 s

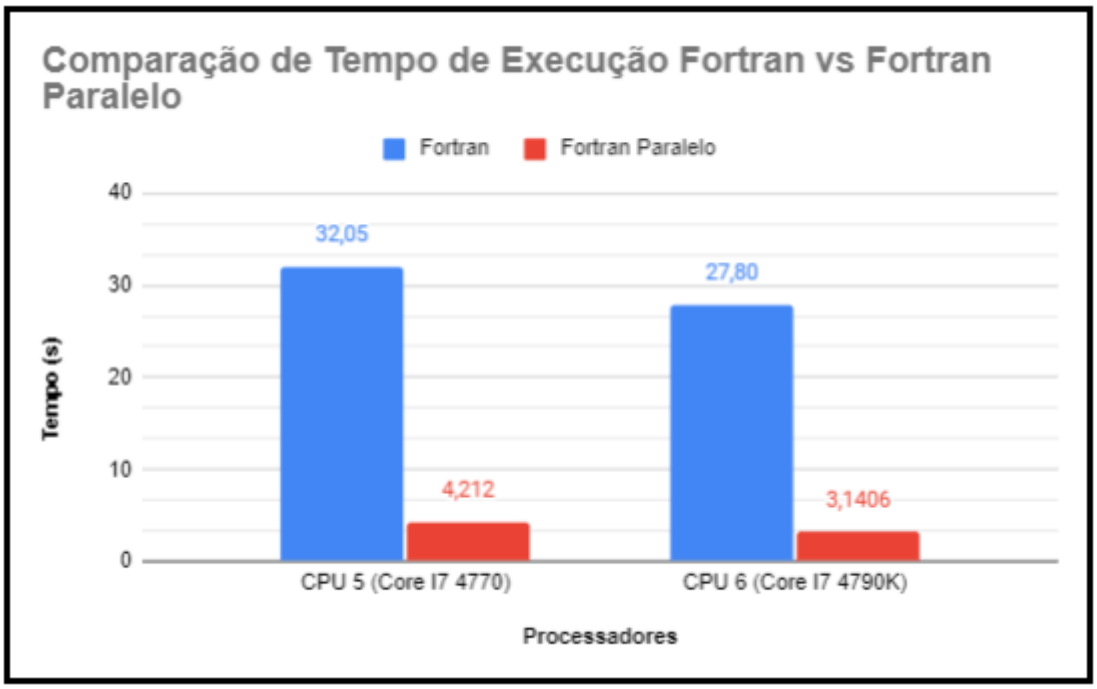

Figura 5.2 - Tempos de execução Fortran vs Fortran Paralelo

Com os dados da tabela 5.4, para efeito comparativo, serão calculados os valores de *speedup*, que é a razão entre o tempo de execução sem melhoria (tempo da CPU) e o tempo de execução com melhoria (tempo da GPU). Nesse sentido, é importante destacar que os valores de *speedup*, quantificam os ganhos da versão paralela em relação à versão sequencial, como podem ser visualizados nas tabelas 5.7 a 5.10. Assim, é possível dizer que quanto maior for o *speedup*, melhor é o ganho da computação paralela, pois mostra que a GPU conseguiu acelerar mais a execução do código.

Porém, é possível verificar que os valores de *speedup* crescem significativamente para algoritmos com linguagem interpretada como python (tabela 5.10) em relação a algoritmos com linguagem compilada como C e Fortran. Pois nas linguagens C e Fortran os tempos de execuções são menores que o Python, tendo assim uma menor aceleração, ou seja, um menor ganho computacional.

Além disso, verifica-se também nas tabelas 5.7 à 5.10, que os valores de *speedup* são menores para as máquinas que utilizam os processadores: CPU 1, CPU 2, CPU 3, GPU 1, GPU 2 e GPU 3 pois são mais potentes do que os processadores utilizados pela plataforma Colab (CPU 4 e GPU 4), como já explicado anteriormente.

Por fim, como não foi feito uma versão em Fortran paralelo via GPU neste trabalho, isto é, foi implementado uma versão em Fortran paralelo via CPU, versão essa que utilizou otimizações do compilador como: Max Speed, Qparallel, OpenMP, Generate Parallel Code, Multithreaded entre outros, além de instruções FORALL no código, calculou-se então, o *speedup* a partir do tempo de execução código sequencial em Fortran usando as CPU 1 à 3, sobre os tempos de execução do algoritmo C paralelo das GPUs 1 à 3, como é visualizado na tabela 5.8. Esse procedimento também foi feito para a obtenção do *speedup* do Fortran paralelo via CPU, como pode ser observado na tabela 5.9. Desse modo, verificou-se que os *speedups* do algoritmo Fortran sequencial são menores que os *speedups* do algoritmo Fortran paralelo, devido ao fato que o Fortran sequencial ter o tempo de execução menor que o Fortran paralelo, como é mostrado na tabela 5.8.

Tabela 5.7 - *Speedups* em C

	CPU 1 Ryzen 3600 XT	**CPU 2** Ryzen 3700 X	**CPU 3** Core I5 7500	**CPU 4** INTELXeon Colab
GPU 4(Tesla K80Colab)	1,72	1,78	6,51	13
GPU 3(GTX 1070)	6,41	6,64	24,29	48,54
GPU 2(RTX 2060)	8,32	8,62	31,51	62,97
GPU 1(RTX 3070 Ti)	9,33	9,66	35,33	70,6

Tabela 5.8 - *SpeedUps* do Fortran

	CPU 1 Ryzen 3600 XT	**CPU 2** Ryzen 3700 X	**CPU 3** Core I5 7500
GPU 3(GTX 1070)	5,93	5,87	8,06
GPU 2(RTX 2060)	7,7	7,62	10,45
GPU 1(RTX 3070 Ti)	8,63	8,54	11,72

Tabela 5.9 - *Speedups* do Fortran Paralelo

	CPU 1 Ryzen 3600XT	**CPU 2** Ryzen 3700 X	**CPU 3** Core I5 7500
GPU 3(GTX 1070)	6,25	6,25	8,54
GPU 2(RTX 2060)	8,1	8,1	11,08
GPU 1(RTX 3070 Ti)	9,09	9,09	12,42

Tabela 5.10 - *Speedups* do Python

	CPU 1 Ryzen 3600 XT	**CPU 2** Ryzen 3700 X	**CPU 3** Core I5 7500	**CPU 4** INTEL Xeon Colab
GPU 4(Tesla K80 - Colab)	247,6	305,5	305,5	570,2
GPU 3(GTX 1070)	255,1	314,7	314,7	587,3
GPU 2(RTX 2060)	376,1	464	464	866
GPU 1(RTX 3070 Ti)	409,7	505,4	505,4	943,2

Para uma melhor visualização dos speedups das tabelas 5.7 até 5.10, foram feitos os gráficos dos *speedups* dos códigos: C, Fortran, Fortran Paralelo e Python, como é mostrado nas figuras 5.3 até 5.6.

As figuras 5.3 à 5.6 tratam dos valores de *speedup,* respectivamente, do algoritmo em C, Fortran, Fortran Paralelo e Python. Ao se fazer uma análise entre as CPUs, é possível verificar que a CPU 4 possui os maiores *speedups*, ou seja, esse processador possui um ganho maior porque tem um tempo de execução sequencial maior e então acaba tendo uma maior aceleração devido a GPU possuir um menor tempo de execução paralelo. Vale ainda destacar que a CPU 4 (unidade processadora da plataforma Colab) é mais lenta que todas as outras CPUs, porque só tem 2 núcleos. Já a CPU 3 tem 4 núcleos, a CPU 1 tem 6 núcleos e a CPU 2 tem 8 núcleos. Embora a CPU1 tenha menos núcleos que a CPU 2, a CPU 1 tem um clock maior que a CPU 2, e por isso ambaspossuem performances semelhantes. Porém, ainda assim a CPU 1 é um pouco mais rápida que a CPU 2. Em resumo, a CPU 1 é a mais rápida de todas, a CPU 2 é a segunda mais rápida, a CPU 3 é a terceira mais rápida e a CPU 4 é a mais lenta utilizada nesta dissertação.

Porém, ao se analisar cada CPU isoladamente, como por exemplo: Observando-se a CPU 1, verifica-se que o valor de *speedup* em relação a GPU 1 foi maior, devido ao fato dessa GPU ser mais potente que as demais. A GPU 1 é mais potente porque possui 6144 núcleos. Logo, possui uma aceleração maior que todas as outras GPUs. Nesse sentido, verifica-se que a GPU 2 é a segunda GPU mais rápida, pois apesar de possuir o mesmo número de núcleos que a GPU 3, ela tem um clock maior. Neste sentido, a GPU 3 é a terceira GPU mais rápida e por último a GPU 4 (GPU da Plataforma Colab) é a mais lenta de todas, pelo fato de ter 2496 núcleos e o menor clock que as demais. Por fim, as GPUs conseguem uma performance superior às CPUs, pelo fato de terem milhares de núcleos (Tabela 5.3).

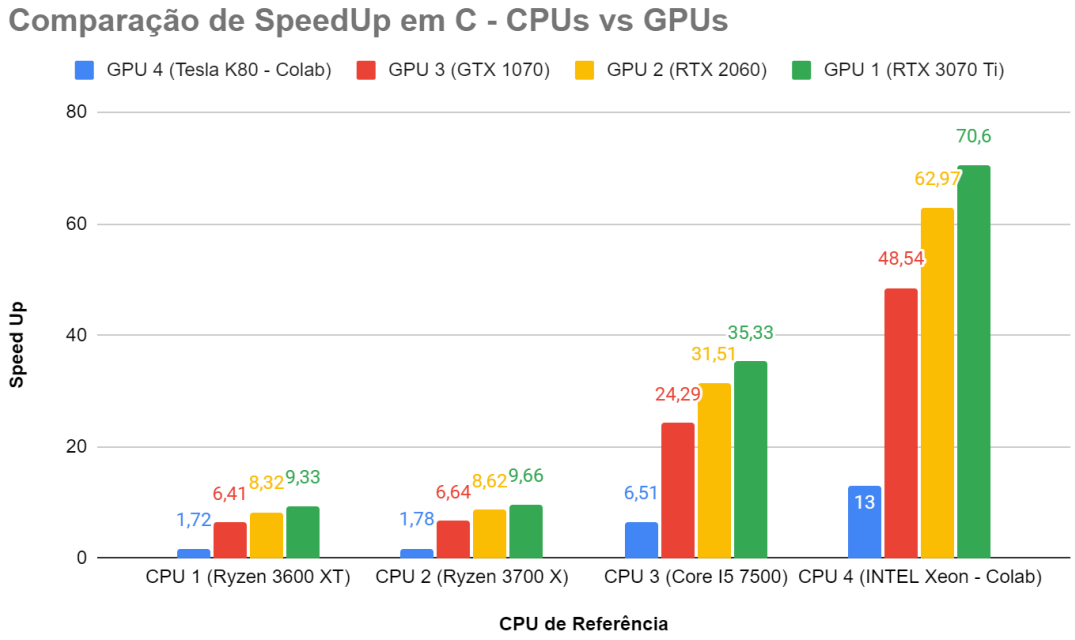

Figura 5.3 - *Speedups* em C

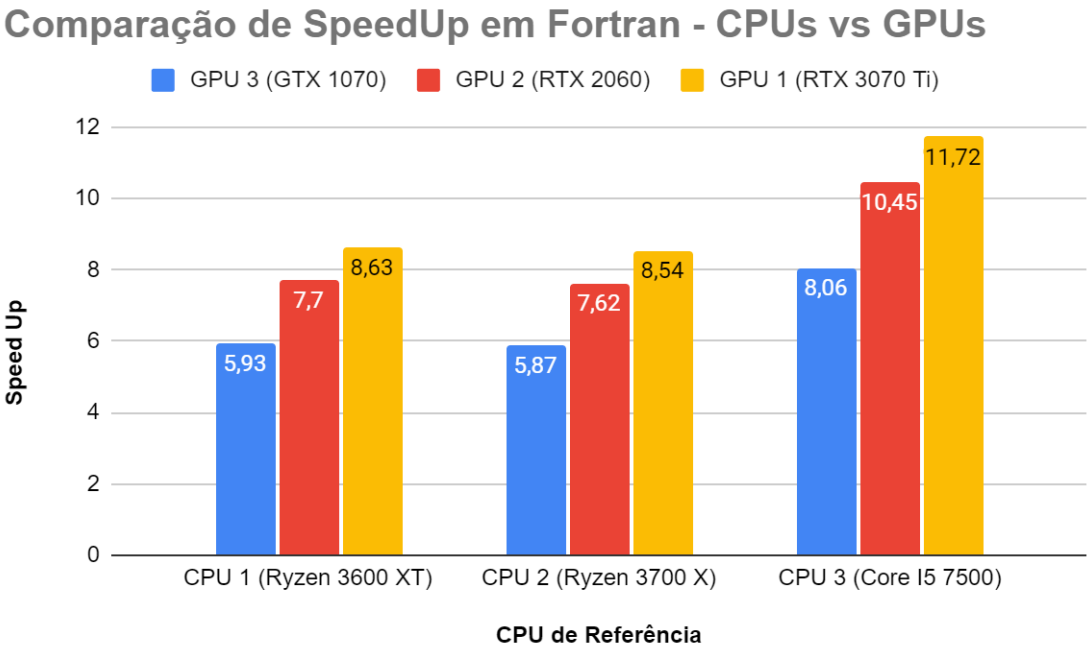

Figura 5.4 - *Speedups* do Fortran

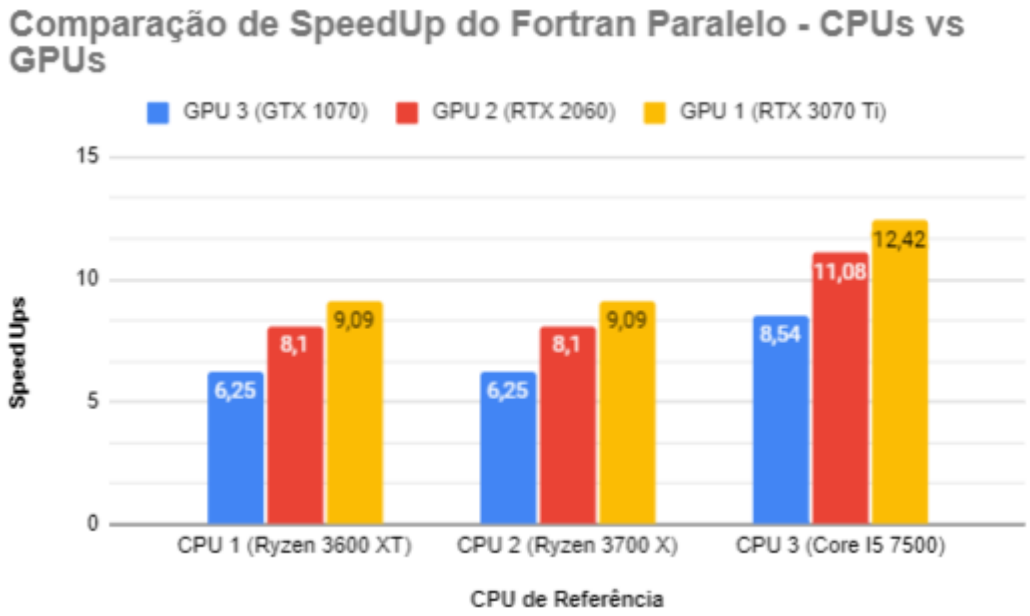

Figura 5.5 - *Speedups* do Fortran Paralelo

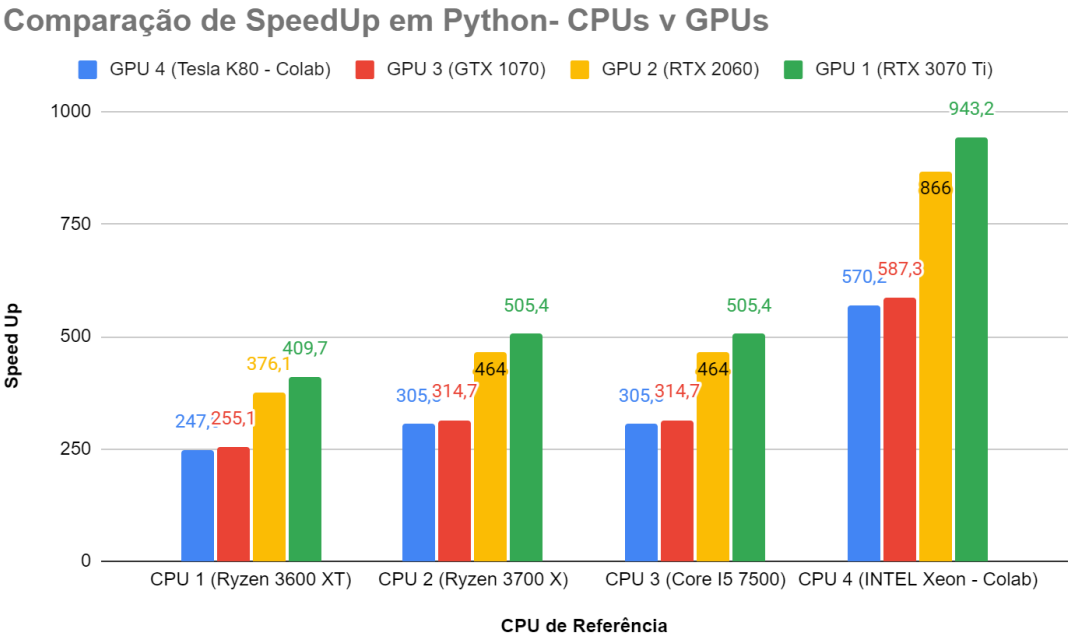

Figura 5.6 - *Speedups* do Python

CAPÍTULO 6

Conclusões

Nesta dissertação, foi utilizada computação paralela baseada em CPU e GPU, para a implementação computacional da solução das equações da cinética pontual de reatoresusando o método de Runge-Kutta de 4ª ordem. Nesse sentido, essas equações foram resolvidas 100 milhões de vezes, permitindo sim, avaliar o desempenho computacional das diferentes linguagens computacionais utilizadas nesta dissertação.

Antes de iniciar o trabalho de paralelização, foi elaborada uma versão sequencial em linguagem C e Python, a partir do código em Fortran. A versão paralela do código, foi implementada utilizando C CUDA em conjunto com a técnica de paralelização *grid-stride loop*. Tanto a versão sequencial C e Python, quanto a versão C e Python CUDA desenvolvidas nesta dissertação, geraram resultados de saída que foram utilizados para a validação desses códigos.

Após a verificação e teste da consistência dos algoritmos implementados, foram realizadas simulações computacionais utilizando 8 máquinas, 4 CPUs e 4 GPUs, como é mostrado na tabela 5.1 do capítulo 5. Apartir das tabelas e gráficos de tempo e *speedups* do capítulo 5, avaliou-se os resultados das simulações, sendo possível detectar, como esperado, que o tempo de execução da computação paralela, baseada em GPU, foi menor que o tempo de execução da computação sequencial. Assim, como foi possível verificar que o tempo de execução das linguagens compiladas são menores que as linguagens interpretadas. Verificou-se também, que os valores de *speedup* entre as CPUs, mostraram que as CPUs mais potentes são aquelas que possuem um número maior de núcleos e clocks, gerando valores de *speedup* menores. Por fim, foi feita uma análise local de valores de *speedup* para cada CPU, e se constatou que as GPUs mais potentes são aquelas que têm um maior número de núcleos e clocks, resultando assim *speedups* maiores para GPUs mais potentes.

Com este trabalho, espera-se contribuir para melhoria no desenvolvimento de códigos computacionais que exigem um número significativo de cálculos, bem como, alto custo computacional. Dentre eles, podemos destacar a solução numérica das equações da cinética pontual de reatores utilizando o método numérico de Runge-Kuta de 4ª ordem. Nesse sentido, espera-se que os resultados encontrados neste trabalho auxiliem em outras investigações futuras, permitindo aperfeiçoamentos e comparações de resultados.

Com o objetivo de estender os resultados encontrados nesta dissertação e melhorar os tempos computacionais obtidos, ficam como sugestões de trabalhos futuros os seguintes itens abaixo:

- A utilização de técnicas de otimização de acesso à memória da GPU para diminuir a sobrecarga de transferências de dados.
- A utilização de múltiplas GPUs em um cluster.

REFERÊNCIAS

ALMEIDA, A. A. H. **Desenvolvimento de Algoritmos Baseados em GPU para solução de Problemas na Área Nuclear**. Rio de Janeiro: PPGIEN/CNEN, 2009.

BETRYBE, **Linguagem C: o que é e quais os principais fundamentos**, 2020. Disponível em: <https://blog.betrybe.com/linguagem-de-programacao/linguagem-c/ >. Acesso em: 10 out. 2021, 10:58.

BUCK, I. **GPU computingwith NVIDIA CUDA**.In: International. Conference on Computer Graphics and Interactive Techniques.ACM New York, NY, USA: [s.n]. 2007.

CODE MASTERS. **Compiladores vs Interpretadores: vantagens e desvantagens**, 2015. Disponível em: <http://codemastersufs.blogspot.com/2015/10/compiladores-versus-interpretadores.html>. Acesso em: 10 out. 2021, 10:43.

CUDA C Programming Guide. **CUDA Toolkit Documentation**, 2021. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Acesso em: 8 out. 2021, 15:41.

DEVMEDIA. **Threads: paralelizando tarefas com os diferentes recursos do Java**, 2016. Disponível em: <https://www.devmedia.com.br/threads-paralelizando-tarefas-com-os-diferentes-recursos-do-java/34309>. Acesso em: 10 out. 2021, 11:23.

FLYNN, M. J. **Some computer organizations and their effectiveness**.[S.1.]: IEEE TRANSACTIONS onComputers, v. 24, n. 9, 1972.

FREITAS, S. R., **Métodos Numéricos, Departamento de Computação e Estatística**, UFMS, 2000.

GAELZER, Rudi, **Introdução ao Fortan 90/95**. Universidade Federal de Pelotas, 2011.

GLASKOWSKY, P. N. **White Paper NVIDIA's Fermi: the first complete GPU Computing Architeture**.[S.1]. 2009.

HETRICK, David L., **Dynamics of Nuclear Reactor**. 1ª edição. Chicago e Londres, The University of Chicago Press Ltda, 1971.

HUANG, T.C., HSU, P.H. "A practical run-time technique for exploiting loop-level parallelism", **Journal of Systems and Software** v.54**,** p. 259-271, 2000.

HWU, W.M.W., KIRK, D.B., **Programming Massively Parallel Processors: A Hands-on Approach**. 2.ed. Elsevier, 2012.

KIRK, D.B., HWU, W.M.W, **Programming Massively Parallel Processors: A Hands-on Approach**. 2ed. Pg4. 2016.

MORAIS, JOSÉ. **Multiprocessamento – ESP32**, 2017. Disponível em: <https://portal.vidadesilicio.com.br/multiprocessamento-esp32/>. Acesso em: 8 out. 2021, 15:45.

NUNES, Anderson Lupo. **Aperfeiçoamento do método de confinamento da rigidez para a solução das equações da cinética pontual**, COPPE/UFRJ, 2006

NUNES, T. G., **Comparação de Desempenho entre OpenCL e CUDA**. IME/USP, 2012.

NVIDIA CORPORATION. **CUDA C ProgrammingGuide**. [S.l]. 2017.

NVIDIA CORPORATION. **CUDA C ProgrammingGuide**. [S.1]. 2014.

NVIDIA CORPORATION. **OpenCL**.[S.1]. 2012.

NVIDIA DEVELOPER. **An Even Easier Introduction to CUDA**, 2017. Disponível em: <https://devblogs.nvidia.com/even-easier-introduction-cuda/>. Acesso em: 8 out. 2021, 15:45.

PARHAMI, B. **Introduction to Parallel Processing: Algorithms and Architectures**. New York, Boston, Dordrecht, London, Moscow: Kluwer Academic Publishers, v. Series in Computers Science, 2002.

PINHEIRO, A. L. S., 2017, **Modelo Computacional Paralelo Baseado em GPU para Cálculo do Campo de vento de um Sistema de Dispersão Atmosférico de Radionuclídeos**. Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

PYTHON. **What is Python? Executive Sumary**, 2021**.** Disponível em: <https://www.python.org/doc/essays/blurb/>. Acesso em: 10 out. 2021, 10:33.

SANTOS, M.C.D., 2018, **Modelo Computacional Paralelo Baseado em GPU para Cálculo em Tempo Real da Dispersão Atmosférica de Radionuclídeos nas vizinhanças de uma Central Nuclear**. Dissertação de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

SILVA, A. C., **Introdução ao Fortran 90**. Apostila.,COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2011.

SILVA, A. C., 2007, **Monitoração da Subcriticalidade em Reatores Nucleares com Fontes Externas de Nêutrons**. Dissertação de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

TRAVCAV. **A look at reverse loops**, 2017. Disponível em: <https://medium.com/@TravCav/why-reverse-loops-are-faster-a09d65473006>.Acesso em: 8 out. 2021, 15:48.