COPPE
Instituto Alberto Luiz Coimbra de
Pós-Graduação e Pesquisa de Engenharia
UFRJ

ALGORITMO DE CÁLCULO NODAL PARALELO UTILIZANDO DECOMPOSIÇÃO DE DOMÍNIO NA GPU

Jadna Mara Santos Mendes

Tese de Doutorado 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 Doutor em Engenharia Nuclear.

Orientadores: Fernando Carvalho da Silva
Alan Miranda Monteiro de Lima
Adino A. Heimlich Almeida

Rio de Janeiro
Junho de 2024

ALGORITMO DE CÁLCULO NODAL PARALELO UTILIZANDO DECOMPOSIÇÃO DE DOMÍNIO NA GPU

Jadna Mara Santos Mendes

TESE 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 DOUTOR EM CIÊNCIAS EM ENGENHARIA NUCLEAR.

Orientadores: Fernando Carvalho da Silva
Alan Miranda Monteiro de Lima
Adino A. Heimlich Almeida

Aprovada por: Prof. Fernando Carvalho da Silva
Prof. Alan Miranda Monteiro de Lima
Dr. Adino Américo Heimlich Almeida
Prof. Cláudio Márcio do Nascimento Abreu Pereira
Prof. Adilson Costa da Silva
Prof. Hermes Alves Filho
Prof. Sergio de Oliveira Vellozo

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2024

Mendes, Jadna Mara Santos

Algoritmo de Cálculo Nodal Paralelo Utilizando Decomposição de Domínio na GPU/Jadna Mara Santos Mendes. – Rio de Janeiro: UFRJ/COPPE, 2024.

XI, 63 p.: il.; 29, 7cm.

Orientadores: Fernando Carvalho da Silva
Alan Miranda Monteiro de Lima
Adino A. Heimlich Almeida

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia Nuclear, 2024.

Referências Bibliográficas: p. 49 – 52.

1. Cálculo Neutrônico. 2. Otimização de recarga. 3. GPU. I. da Silva, Fernando Carvalho *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Nuclear. III. Título.

Ao meu pai, José Mendes (in memorian). Sempre me chamava de Doutora Mara.

Agradecimentos

Até aqui me sustentou o Senhor...

Quero deixar meus mais sinceros agradecimentos aos meus orientadores, Fernando Carvalho, Adino Heimlich e Alan Lima, pela paciência, orientação constante e incentivo durante todo esse processo. Sua expertise e dedicação foram fundamentais para a realização deste trabalho. Sou imensamente grata por suas contribuições, críticas construtivas e apoio.

Gostaria também de estender meus agradecimentos ao pessoal do Programa de Engenharia Nuclear da COPPE/UFRJ, que de alguma forma contribuiu durante todo esse tempo. Vocês foram essenciais, oferecendo apoio técnico e moral ao longo dessa jornada. Agradeço ao CNPq pelo apoio financeiro, que tornou possível a realização deste sonho.

Aos membros da banca, meu profundo agradecimento por terem aceitado o convite para avaliar este trabalho. Suas considerações e sugestões são de grande importância e enriqueceram ainda mais esta pesquisa.

Agradeço aos meus familiares e amigos pelo suporte incondicional. À minha mãe e meus irmãos, que sempre acreditaram na possibilidade desta conquista e me apoiaram constantemente. Ao meu companheiro Leonardo e ao nosso amado filho Leo, por serem a força nos dias mais difíceis e por compartilharem comigo cada momento desta caminhada. Aos poucos, mas verdadeiros amigos, que vibram comigo a cada nova conquista, minha eterna gratidão. Quero fazer um agradecimento especial ao meu amigo, Vellozo, que me indicou para este doutorado e sempre esteve me apoiando. Sua confiança e palavras de apoio foram fundamentais nessa jornada.

Por fim, não posso deixar de agradecer a mim mesma por nunca ter pensado em desistir, mesmo diante de muitos desafios. “Acreditar sempre” foi o lema que me guiou e me trouxe até aqui. A todos, o meu muito obrigado.

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

ALGORITMO DE CÁLCULO NODAL PARALELO UTILIZANDO DECOMPOSIÇÃO DE DOMÍNIO NA GPU

Jadna Mara Santos Mendes

Junho/2024

Orientadores: Fernando Carvalho da Silva
Alan Miranda Monteiro de Lima
Adino A. Heimlich Almeida

Programa: Engenharia Nuclear

Esta tese de doutorado propõe o desenvolvimento de um algoritmo de cálculo nodal paralelo na GPU utilizando o Método de Expansão Nodal (NEM) de quarta ordem com fuga transversal quadrática, resultando no desenvolvimento do programa Cuda_NEM. O objetivo é melhorar a eficiência e a precisão dos métodos de cálculo em reatores nucleares. A técnica de Decomposição de Domínio, com o método *Red-Black*, é empregada para paralelizar o cálculo nodal na GPU, visando reduzir o tempo de execução e aumentar o desempenho computacional.

Os resultados obtidos com o programa Cuda_NEM são comparados com os do código CNFR para três benchmarks da literatura. Observa-se que o algoritmo proposto mantém a precisão dos cálculos, com desvios máximos de -2.0 pcm no fator de multiplicação efetivo e -0.98% na densidade de potência normalizada no Elemento Combustível (EC).

Conclui-se que o algoritmo desenvolvido é promissor para otimização de modelos de recarga de combustível nuclear. Como trabalho futuro, pretende-se concluir a paralelização de todos os cálculos neutrônicos na GPU e explorar maneiras de aprimorar ainda mais a eficiência e a precisão do programa de cálculo neutrônico.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

PARALLEL NODAL CALCULATION ALGORITHM USING DOMAIN DECOMPOSITION ON GPU

Jadna Mara Santos Mendes

June/2024

Advisors: Fernando Carvalho da Silva
Alan Miranda Monteiro de Lima
Adino A. Heimlich Almeida

Department: Nuclear Engineering

This doctoral thesis proposes a parallel nodal calculation algorithm on the GPU using the fourth-order Nodal Expansion Method (NEM) with quadratic transverse leakage, resulting in the development of the Cuda_NEM program. The aim is to improve the efficiency and accuracy of calculation methods in nuclear reactors. The Domain Decomposition technique, with the Red-Black method, is employed to parallelize the nodal calculation on the GPU, aiming to reduce execution time and increase computational performance

The results obtained with the Cuda_NEM program are compared with those of the CNFR code for three benchmarks from the literature. It is observed that the proposed algorithm maintains the accuracy of the calculations, with maximum deviations of -2.0 pcm in the effective multiplication factor and -0.98% in the normalized assembly power density.

It is concluded that the developed algorithm is promising for optimizing nuclear fuel reloading models. As future work, it is intended to complete the parallelization of all neutron calculations on the GPU and explore ways to further improve the efficiency and accuracy of the neutronic calculation program.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Considerações gerais

A pesquisa do uso das GPUs da NVIDIA, através da tecnologia CUDA (Compute Unified Device Architecture) [1], e as possíveis abordagens de paralelização de códigos computacionais em linguagens altamente eficientes e massivamente paralelas, tem como principal objetivo reduzir o tempo de execução dos algoritmos paralelos em comparação com os algoritmos sequenciais. A simulação do funcionamento de um reator nuclear envolve a modelagem de uma variedade de processos físicos, incluindo a solução da equação de difusão de nêutrons. Diante dessa complexidade, fica evidente que a otimização e adaptação dos algoritmos sequenciais para operação em paralelo, especialmente no contexto das GPUs, são excelentes opções a serem consideradas para melhorar o desempenho computacional.

Na seção 1.2, é apresentado o Método de Expansão Nodal (NEM) [2], destacando sua relevância na modelagem de reatores nucleares e sua aplicação na representação do comportamento dos nêutrons nesse sistema. Além disso, exploramos a paralelização do cálculo nodal, enfatizando sua importância na simulação de reatores nucleares e como essa abordagem pode significativamente acelerar o processo de cálculo. Citamos também o uso das GPUs para aumentar a eficiência computacional em problemas de alto custo computacional, incluindo simulações na área nuclear. Ao revisar estudos anteriores, examinaremos as técnicas de paralelização adotadas e os benefícios observados ao empregar GPUs nesse contexto.

Dessa forma, identificamos estudos prévios que abordaram questões relacionadas a este tema, analisando suas limitações e lacunas para justificar a relevância desta pesquisa. Por fim, destacamos como este estudo contribuirá para o avanço da Física de Reatores Nucleares, proporcionando uma melhor compreensão e soluções eficazes para desafios computacionais importantes nesta área fundamental da ciência.

Na revisão bibliográfica, é fundamental compreendermos a base teórica que sus-

tenta a aplicação do NEM e a importância da paralelização do cálculo nodal, especialmente no contexto da simulação de reatores nucleares. Ao explorar os estudos anteriores e as abordagens de paralelização empregadas, poderemos identificar lacunas e oportunidades de pesquisa que justificam e direcionam o foco deste trabalho. Assim, este capítulo serve como uma ponte entre a contextualização do problema e a análise da literatura existente, preparando o terreno para a aplicação do NEM no contexto da GPU e discussão dos resultados obtidos.

1.2 Revisão bibliográfica

Existem processos de cálculo na engenharia nuclear que consomem muito tempo computacional, tornando a obtenção da solução do problema envolvido no processo demorada e custosa. Isto ocorre, por exemplo, no processo de cálculo de otimização de modelos de recarga de combustível nuclear [3], onde a avaliação de diferentes configurações de núcleo do reator, geradas no processo, tem que ser feita. Neste caso, para realizar a avaliação de uma configuração de núcleo, um programa de cálculo neutrônico que resolve, como citado na seção 1.1, a equação da difusão de nêutrons, ao longo do período de queima, é executado. E são estes cálculos neutrônicos os responsáveis por tornar o processo demorado e custoso, uma vez que é necessário realizar um número muito grande de avaliações de diferentes configurações de núcleos.

Os métodos nodais de malha grossa, como o Método de Expansão Nodal (NEM) [2] implementado no código CNFR [4] e [5], podem ser usados para diminuir o tempo de cálculo de cada avaliação. O NEM de quarta ordem com fuga transversal quadrática implementado no código CNFR, para solução da equação da difusão de nêutrons tridimensional e a dois grupos de energia, é preciso e veloz.

O NEM é um método de malha grossa consistente usado nos últimos quarenta anos para calcular o fluxo de nêutrons em duas ou três dimensões, por exemplo, em [6], [7], [8], [9] e [10].

Portanto, o método nodal que consta neste código pode ser usado no processo de cálculo de otimização de modelos de recarga de combustível nuclear. Porém, no CNFR as equações constitutivas do NEM são resolvidas usando um algoritmo sequencial de cálculo em CPU, fazendo com que o tempo de cálculo para avaliação de uma configuração de núcleo seja alto. Isto torna inviável, então, que várias possíveis configurações de núcleos venham a ser avaliadas em um tempo razoável, por exempolo, em [11] que trata o problema de otimização de recarga de combustível nuclear em reatores do tipo PWR.

Sendo assim, com a finalidade de se ter a possibilidade de avaliação de um número maior de configurações de núcleos do reator em um tempo razoável, no processo de

cálculo de otimização de modelos de recarga de combustível nuclear, uma proposta de solução das equações constitutivas do NEM de quarta ordem com fuga transversal quadrática [5], de forma paralela em GPU, é apresentada nesta tese.

As GPUs são dispositivos com alto poder de processamento quando comparadas com as CPUs e apresentam baixo consumo de energia por FLOP [12] tornando-as adequadas para realizar cálculos mais rápidos em muitos problemas de engenharia nuclear, por exemplo, o que foi feito por [13], [14] e [15]. Estes dispositivos são representantes do paradigma SIMD (Single Instruction Multiple Data) [16] e são adequados para efetuar todos os tipos de cálculos com ênfase em problemas de álgebra linear, na solução de sistemas lineares e sistemas de inteligência artificial e ainda redes neurais artificiais [17].

Considerando que o CNFR é um código preciso e confiável, foi implementado neste trabalho uma versão em C++ e CUDA [18] que adequa as equações constitutivas do NEM ao modelo de programação em GPU. O programa resultante desta adequação, denominado Cuda_NEM, foi então utilizado na obtenção do fator de multiplicação efetivo e na densidade de potência normalizada no EC em geometria Cartesiana tridimensional cujos resultados encontrados foram confrontados com aqueles obtidos pelo CNFR, para benchmarks disponíveis na literatura.

A consistência dos resultados obtidos com o Cuda_NEM em relação ao cálculo nodal presente no programa CNFR nos incentivou a explorar a paralelização de outros cálculos essenciais do programa, por exemplo, a depleção isotópica e a reconstrução pino a pino. Esses processos são fundamentais para abordar o problema de otimização de recarga. No entanto, devido a restrições de tempo, não foi possível realizar essa extensão durante o período da tese, e deixamos isso como uma sugestão de trabalho futuro. É importante destacar que os algoritmos e resultados relacionados ao cálculo nodal totalmente paralelo na GPU, utilizando a técnica de Decomposição de Domínio implementado com o *Red-Black*, foram publicados na revista *Nuclear Engineering and Design* [19].

A implementação do algoritmo proposto tem o potencial de impactar significativamente diversos aspectos da engenharia e do projeto de reatores nucleares, ao tornar mais eficientes os cálculos que simulam o comportamento do reator. Essa melhoria na eficiência computacional não apenas promove avanços na tecnologia nuclear, mas também contribui para aprimorar a segurança, e escalabilidade desses cálculos.

Em resumo, este estudo introduziu um algoritmo eficaz para cálculos nodais paralelos, aproveitando a capacidade da GPU e a estratégia de decomposição de domínio. Os próximos capítulos envolvem a descrição dos cálculos realizados para validar e avaliar o desempenho desse algoritmo.

1.3 Objetivos da tese

1.3.1 Objetivos Gerais

Os objetivos gerais deste trabalho são contribuir para o desenvolvimento de novos métodos de projeto do núcleo e operação de reatores nucleares que sejam mais eficientes, seguros e econômicos. Com isso, avançar o conhecimento científico e tecnológico relacionado à energia nuclear, com o objetivo final de melhorar a segurança, eficiência e sustentabilidade dos sistemas de energia nuclear.

1.3.2 Objetivos específicos na área de Física de Reatores

- Implementar um algoritmo de cálculo nodal completamente paralelo, utilizando GPUs, para acelerar significativamente os processos de cálculo nodal.
- Paralelizar o programa baseado no código CNFR, que atualmente executa os cálculos de forma sequencial na CPU, empregando a técnica de decomposição de domínio integrada ao método Red-Black na GPU.
- Reduzir significativamente o tempo necessário para realizar os cálculos nodais, ao mesmo tempo em que se busca melhorar o desempenho computacional em comparação com as práticas atuais.

Esses objetivos específicos fornecem uma direção clara para a implementação do algoritmo e destacam a abordagem adotada para alcançar melhorias tangíveis nos processos de cálculo nodal em reatores nucleares.

1.4 A organização da tese

Para uma melhor compreensão, esta tese de doutorado foi organizada ao longo de mais 5 capítulos além desta introdução.

No Capítulo 2 é apresentado o Método de Expansão Nodal (NEM), conforme utilizado no cálculo nodal sequencial. Explora-se a estrutura e funcionamento deste método como base para compreensão do desenvolvimento subsequente do algoritmo paralelo.

O Capítulo 3 introduz a técnica de decomposição de domínio implementada ao método Red-Black, utilizado no cálculo nodal paralelo. Detalha-se a adaptação dessa técnica para o ambiente de computação paralela na GPU, destacando sua importância para o desempenho do algoritmo.

No Capítulo 4 são apresentados os detalhes da implementação paralela na GPU, juntamente com os procedimentos adotados para tratar as equações constitutivas

do NEM em paralelo. Este capítulo oferece uma visão aprofundada da arquitetura do algoritmo paralelo e das técnicas específicas utilizadas para otimização.

Já no Capítulo 5 são exibidos os resultados de cálculo obtidos com o código CNFR e o programa Cuda_NEM para benchmarks encontrados na literatura. Além disso, são apresentadas análises e discussões desses resultados, fornecendo uma compreensão detalhada sobre o desempenho e eficácia do algoritmo proposto.

E por fim, no Capítulo 6 são apresentadas as conclusões do trabalho, resumindo os principais resultados e contribuições. Além disso, são oferecidas sugestões para trabalhos futuros, visando a continuidade do desenvolvimento e aprimoramento do cálculo neutrônico.

Este capítulo introdutório estabeleceu as bases para compreender o contexto e os objetivos desta pesquisa. Exploramos a importância da eficiência computacional na análise do cálculo nodal em sistemas neutrônicos, destacando a necessidade de novas abordagens para tornar os processos mais rápidos e eficazes.

No próximo capítulo, aprofundaremos nosso entendimento ao apresentar o Método de Expansão Nodal (NEM) tradicional. Este método serve como ponto de partida fundamental para o desenvolvimento do algoritmo proposto, proporcionando uma compreensão sobre os princípios subjacentes ao cálculo nodal sequencial. Através da análise detalhada do NEM, estaremos preparados para explorar as inovações e otimizações introduzidas em nossa abordagem paralela na GPU. Assim, avançaremos em direção ao objetivo de aprimorar a eficiência, segurança e sustentabilidade dos sistemas de energia nuclear.

Capítulo 2

Método de Expansão Nodal

Neste capítulo apresentamos o Método de Expansão Nodal (Nodal Expansion Method - NEM) [2] baseado em um algoritmo sequencial. O NEM é um método utilizado em análise neutrônica de reatores nucleares para obter soluções aproximadas para a distribuição espacial de nêutrons. Ele é particularmente útil em cálculos acoplados de neutrônica e termo-hidráulica em reatores nucleares.

O NEM, implementado no código CNFR, destaca-se por sua eficácia no cálculo do fluxo de nêutrons em sistemas complexos. Especificamente, a versão de quarta ordem com fuga transversal quadrática do NEM, presente no CNFR, oferece uma solução precisa e rápida para a equação da difusão de nêutrons em três dimensões e dois grupos de energia. Na próxima seção, examinaremos mais de perto os princípios e a aplicação das equações constitutivas do NEM, preparando o terreno para nossa discussão sobre o desenvolvimento subsequente do algoritmo paralelo na GPU.

2.1 Equações Constitutivas do NEM

Nesta seção, apresentamos as equações constitutivas do NEM e mostramos porque a varredura dos nodos, utilizada na versão tradicional, precisou ser ajustada a fim de ser paralelizável no contexto da GPU.

Neste trabalho, o NEM será usado para resolver numericamente a equação de difusão de nêutrons a dois grupos de energia, em geometria cartesiana tridimensional numa região homogênea chamada de nodo. Os resultados obtidos pelo NEM fornecem quantidades físicas de interesse que são: correntes líquidas médias nas faces do nodo, fluxos médios nas faces do nodo, fluxos médios no volume de nodo e o fator de multiplicação efetivo. No NEM, o termo de fuga transversal que aparece durante o processo de integração da equação da difusão é aproximado por um polinômio quadrático.

O NEM é um método de malha grossa que trabalha com grandes regiões homogeneizadas, chamadas nodos. No núcleo do reator nuclear, os EC's são divididos em

paralelepípedos (nodos) com a dimensão da ordem da área da base do EC e divisões axiais de mesma ordem, implicando, consequentemente, na divisão do núcleo em paralelepípedos contíguos, como pode ser observado na figura 2.1, onde o volume desse nodo é dado por $V_n = a_x^n a_y^n a_z^n$.

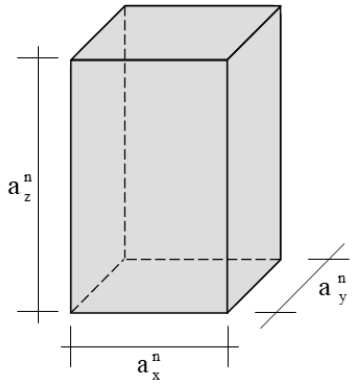

Figura 2.1: Representação esquemática de um nodo e suas faces.

Os métodos nodais trabalham com regiões (ou nodos) nas quais os parâmetros nucleares são uniformes, ou seja, não variam espacialmente dentro dessa região.

Para n representando um nodo qualquer, g o grupo de energia, a_u^n a largura do nodo n na direção u, $u = x, y, z$ e s uma das faces (l e r) do nodo n na direção u, o NEM será detalhado nas equações a seguir.

A equação da continuidade de nêutrons em geometria cartesiana tridimensional a dois grupos de energia ($g = 1, 2$) é dada por:

$$\sum_{u=\{x,y,z\}} \frac{\partial}{\partial u} J_{gu}(x,y,z) + \Sigma_{tg}(x,y,z)\phi_g(x,y,z) = \frac{1}{K_{eff}}\chi_g \sum_{g'=1}^{2} \nu\Sigma_{fg'}(x,y,z)\phi_{g'}(x,y,z) + \\ + \sum_{g'=1}^{2} \Sigma_s^{g'\to g}(x,y,z)\phi_{g'}(x,y,z), \tag{2.1}$$

onde

$$J_{gu}(x,y,z) = J_{gu}^{+}(x,y,z) - J_{gu}^{-}(x,y,z) = -D_g(x,y,z)\frac{\partial}{\partial u}\phi_g(x,y,z), \tag{2.2}$$

A integração da Eq. 2.1 em V_n e posterior divisão por V_n, resulta na seguinte equação de balanço nodal:

$$\sum_{u=x,y,z} \frac{1}{a_u^n} \left\{ \bar{J}_{gur}^n - \bar{J}_{gul}^n \right\} + \Sigma_{tg}^n \bar{\phi}_g^n = \sum_{g'=1}^{2} \left\{ \frac{\chi_g}{K_{eff}} \nu\Sigma_{fg'}^n + \Sigma_{gg'}^n \right\} \bar{\phi}_{g'}^n, \tag{2.3}$$

onde o fluxo nédio no nodo n é assim definido:

$$\bar{\phi}_g^n \equiv \frac{1}{V_n} \int_{V_n} \phi_g(x,y,z) dV \tag{2.4}$$

e $\bar{J}_{gur}^n$ e $\bar{J}_{gul}^n$ são as correntes líquidas médias nas faces do nodo n assim definidas:

$$\bar{J}_{gus}^n \equiv \frac{1}{a_v^n a_w^n} \int_0^{a_v^n} \int_0^{a_w^n} J_{gu}(u_s^n, v, w)\, dv\, dw \tag{2.5}$$

e $\nu\Sigma_{fg}^n$, $\Sigma_{gg'}^n$ and Σ_{tg}^n são as seções de choque macroscópicas ν-fissão do grupo g, espalhamento de g' para g e total do grupo g, respectivamente, no nodo n.

No entanto, para resolver a Eq. (2.3), precisamos encontrar uma relação entre as correntes líquidas médias nas faces e os fluxos médios no nodo. A priori, podemos tentar encontrar uma relação integrando Eq. (2.2) sobre a área $A_u^n = a_v a_w$, transversalmente à direção u, e *a posteriori* divisão por A_u^n resultando em equações dadas por:

$$\bar{J}_{gus}^n = -\bar{D}_g^n \frac{d}{du} \bar{\Psi}_g^n(u) \Big|_{u=u_s^n}, \tag{2.6}$$

e

$$\bar{J}_{gus}^n = \bar{J}_{gus}^{+n} - \bar{J}_{gus}^{-n}, \tag{2.7}$$

onde:

$$\bar{J}_{gus}^n \equiv \frac{1}{a_v^n a_w^n} \int_0^{a_v^n} \int_0^{a_w^n} J_{gu}^{(} u_s^n, v, w)\, dv\, dw, \tag{2.8}$$

$$\bar{\Psi}_{gu}^n(u) \equiv \frac{1}{a_v^n a_w^n} \int_0^{a_v^n} \int_0^{a_w^n} \phi_g(u, v, w)\, dv\, dw \tag{2.9}$$

e

$$\bar{D}_g^n \equiv \frac{1}{3\Sigma_{trg}^n}. \tag{2.10}$$

No NEM, os fluxos unidimensionais são expressos por uma expansão polinomial de quarta ordem, que são representadas pelas funções de base $h_k(\xi)$ [2], onde $\xi \equiv \frac{u}{a_u^n}$. Dessa forma:

$$\bar{\Psi}_{gu}^n(u) = \bar{\phi}_g^n + \sum_{k=1}^{4} C_{kgu}^n \, h_k\left(\frac{u}{a_u^n}\right). \tag{2.11}$$

Usando a aproximação da difusão, podemos relacionar os fluxos médios nas faces com as correntes parciais médias nas faces do nodo, qual seja:

$$\bar{\Psi}_{gus}^n = 2(\bar{J}_{gus}^{+n} + \bar{J}_{gus}^{-n}). \tag{2.12}$$

Aplicando as condições de contorno à expansão na Eq. 2.11, são calculados os coeficientes de primeiro e segundo grau C_{1gu}^n e C_{2gu}^n, dados pelas equações:

$$\begin{cases} C_{1gu}^n = \dfrac{1}{2}\left(\bar{\Psi}_{gur}^n - \bar{\Psi}_{gul}^n\right) \\ C_{2gu}^n = \bar{\phi}_g^n - \dfrac{1}{2}\left(\bar{\Psi}_{gur}^n + \bar{\Psi}_{gul}^n\right) \end{cases} \tag{2.13}$$

Para calcular os chamados coeficientes secundários, C_{3gu}^n e C_{4gu}^n, o NEM usa a equação de difusão de nêutrons integrada sobre a área A_u^n, transversalmente à direção u e *a posteriori* divisão por A_u^n , ou seja,

$$\begin{aligned} &- D_g^n \frac{d^2}{du^2}\bar{\Psi}_{gu}^n(u) + \Sigma_{tg}^n \bar{\Psi}_{gu}^n(u) = \\ &\sum_{g'=1}^{2}\left\{\frac{\chi_g}{K_{eff}}\nu\Sigma_{fg'}^n + \Sigma_{gg'}^n\right\}\bar{\Psi}_{g'u}^n(u) - L_{gu}^n(u)\,, \end{aligned} \tag{2.14}$$

onde $L_{gu}^n(u)$, a fuga transversal, é assim definida:

$$L_{gu}^{n}(u) \equiv \frac{1}{a_v\, a_w} \int_0^{a_v} \int_0^{a_w} \left\{ -\frac{\partial}{\partial v}(D_g(u,v,w)\frac{\partial}{\partial v}\phi_g(u,v,w)) \right\} dv\, dw +$$

$$\frac{1}{a_v\, a_w} \int_0^{a_v} \int_0^{a_w} \left\{ -\frac{\partial}{\partial w}(D_g(u,v,w)\frac{\partial}{\partial w}\phi_g(u,v,w)) \right\} dv\, dw\,. \tag{2.15}$$

O NEM representa a fuga transversal pela seguinte expansão polinomial:

$$L_{gu}^{n}(u) = \bar{L}_{gu}^{n} + \sum_{k=1}^{2} \alpha_{kgu}^{n}\, h_k\left(\frac{u}{a_u^n}\right)\,, \tag{2.16}$$

onde os coeficientes de expansão, α_{kgu}^{n}, são dados por:

$$\begin{cases} \alpha_{1gu}^{n} = \frac{1}{2}\left(L_{gur}^{n} - L_{gul}^{n}\right) \\ \alpha_{2gu}^{n} = \bar{L}_{gu}^{n} - \frac{1}{2}\left(L_{gur}^{n} + L_{gul}^{n}\right) \end{cases} \tag{2.17}$$

com $\bar{L}_{gu}^{n}$, L_{gur}^{n} e L_{gul}^{n} calculados como na referência [2]. Onde a fuga transversal à direção u média no nodo é assim definida:

$$\bar{L}_{gu}^{n} = \sum_{\xi=v,w} \frac{(\bar{J}_{gu\xi}^{+n} - \bar{J}_{gu\xi}^{-n})}{a_\xi^n}. \tag{2.18}$$

Já as fugas transversais à direção u nas faces do nodo são definidas de acordo com as seguintes definições:

i) Para interfaces entre nodos:

$$L_{gus}^{n} = \frac{\left(a_u^{n_v}\bar{L}_{gu}^{n} + a_u^{n}\bar{L}_{gu}^{n_v}\right)}{(a_u^{n_v} + a_u^{n})}, \tag{2.19a}$$

com n_v sendo um nodo vizinho ao nodo n (debaixo, da frente ou da esquerda, se $s = l$ ou, então, na direita, detrás ou de cima se $s = r$).

ii) Para faces voltadas para o contorno e sob a condição de vácuo:

$$L_{gus}^{n} = \left(\frac{\bar{\Psi}_{gus}^{n}}{\bar{\phi}_g^n}\right)\bar{L}_{gu}^{n}. \tag{2.19b}$$

Assumindo que a fuga transversal $L_{gu}^{n}(u)$ seja conhecida, podemos calcular os coeficientes secundários, C_{3gu}^{n} e C_{4gu}^{n}, usando o método de resíduos ponderados [20]. Este método é aplicado à Eq. (2.14) usando as funções de base $h_1(\xi) = 2\xi - 1$ e

$h_2(\xi) = 6\xi(1-\xi) - 1$, que são iguais as usadas na expansão polinomial de quarta ordem de $\bar{\Psi}^n_{gu}(u)$, para finalmente obter C^n_{3gu} e C^n_{4gu}.

Os coeficientes C^n_{kgu}, para $k = 3, 4$, são obtidos resolvendo-se os respectivos sistemas de equações lineares e algébricas que resultam da equação a seguir.

$$\left(\frac{12D^n_g}{(a^n_u)^2} + \theta_k \Sigma^n_{tg}\right) C^n_{kgu} - \theta_k \sum_{g'=1}^{2} \left(\frac{\chi_g}{K_{eff}} \nu\Sigma^n_{fg'} + \Sigma^n_{gg'}\right) C^n_{kg'u} = S^n_{kgu}; \quad \text{para } g = 1, 2, \tag{2.20a}$$

com

$$S^n_{kgu} \equiv \theta_{k-2} \left(\Sigma^n_{tg} C^n_{k-2gu} - \sum_{g'=1}^{2} \left(\frac{\chi_g}{K_{eff}} \nu\Sigma^n_{fg'} + \Sigma^n_{gg'}\right) C^n_{k-2g'u} + \alpha^n_{k-2gu} \right); \quad \text{para } k = 3, 4, \tag{2.20b}$$

onde $\theta_1 = -\frac{1}{3}, \theta_3 = \theta_2 = \frac{1}{5}$ e $\theta_4 = \frac{3}{35}$.

Já os coeficientes para o cálculo das correntes parciais de saída do nodo são dados da seguinte forma:

$$\begin{cases} A^n_{0gu} = \dfrac{6D^n_{gu}}{\left(1 + 12D^n_{gu}\right)} \\ A^n_{1gu} = \dfrac{\left(1 - 48(D^n_{gu})^2\right)}{((1 + 4D^n_{gu})(1 + 12D^n_{gu}))} \\ A^n_{2gu} = \dfrac{-8D^n_{gu}}{((1 + 4D^n_{gu})(1 + 12D^n_{gu}))} \\ A^n_{3gu} = \dfrac{6D^n_{gu}}{1 + 4D^n_{gu}} \end{cases} \tag{2.21}$$

Com $D^n_{gu} \equiv \frac{D^n_g}{a^n_u}$.

Uma vez obtidos os coeficientes secundários, os fluxos médios nodais são obtidos resolvendo-se o sistema de equações lineares e algébricas que resulta da equação a seguir.

$$\left(\Sigma^n_{tg} + 2 \sum_{u=x,y,z} \frac{A^n_{0gu}}{a^n_u}\right) \bar{\phi}^n_g - \sum_{g'=1}^{2} \left(\frac{\chi_g}{K_{eff}} \nu\Sigma^n_{fg'} + \Sigma^n_{gg'}\right) \bar{\phi}^n_{g'} = S^n_g; \quad \text{para } g = 1, 2, \tag{2.22a}$$

com

$$S^n_g \equiv 2 \sum_{u=x,y,z} \frac{A^n_{0gu} \left(2\left(\bar{J}^{+n}_{gul} + \bar{J}^{-n}_{gur}\right) - C^n_{4gu}\right)}{a^n_u}. \tag{2.22b}$$

Por fim, podemos calcular as correntes parciais de saída do nodo da seguinte forma:

$$\begin{cases} \bar{J}^{-n}_{gul} = A^n_{0gu}\left\{\bar{\phi}^n_g + C^n_{4gu}\right\} + A^n_{1gu}\bar{J}^{+n}_{gul} + A^n_{2gu}\bar{J}^{-n}_{gur} - A^n_{3gu}C^n_{3gu} \\ \bar{J}^{+n}_{gur} = A^n_{0gu}\left\{\bar{\phi}^n_g + C^n_{4gu}\right\} + A^n_{2gu}\bar{J}^{+n}_{gul} + A^n_{1gu}\bar{J}^{-n}_{gur} + A^n_{3gu}C^n_{3gu} \end{cases} \tag{2.23}$$

Na seção 2.2, o algoritmo 1 mostra como é feita a varredura dos nodos nesse processo de cálculo sequencial. Compreender este processo é essencial, pois ele forma a base para entender as subsequentes otimizações e adaptações necessárias para a paralelização em GPU.

2.2 Varredura Nodal Sequencial

O Programa de Cálculo Nodal, aqui apresentado, utiliza as equações constitutivas do NEM, conforme resumidas na seção 2.1, que resolve a equação da difusão de nêutrons 3D a dois grupos de energia usando expansão polinomial de quarta ordem com fuga transversal quadrática [5]. O esquema utilizado para fazer a varredura dos nodos é o mesmo adotado em [4].

O esquema de varredura nodal sequencial, com iterações externas, é mostrado no algoritmo 1. Vale lembrar que na primeira etapa do processo iterativo, todas as correntes de entrada e saída em todas as faces e todos os nodos são inicializados como 1. .

Algorithm 1 Varredura Nodal Sequencial

1: Inicializa-se K_{eff}, $\bar{\phi}_g^n$ e $\bar{J}_{gus}^{\mp n}$
2: **while** K_{eff} e $\bar{\phi}_g^n$ não convergirem **do**
3: **for** $n = 1$ até N **do**
4: Atualização das correntes de entrada dos nodos vizinhos ao nodo n
5: Cálculo dos fluxos médios nas faces do nodo usando a Eq. (2.12)
6: Cálculo dos coeficientes primários usando a Eq. (2.13)
7: **for** cada direção u **do**
8: Cálculo da fuga transversal média usando a Eq. (2.18)
9: Cálculo das fugas transversais nas faces do nodo usando a Eq. (2.19a) ou (2.19b)
10: Cálculo dos coeficientes da expansão da fuga transversal usando a Eq. (2.17)
11: **end for**
12: Cálculo dos coeficientes secundários usando as Eqs. (2.20a e 2.20b)
13: Cálculo dos coeficientes das correntes parciais de saída usando a Eq. (2.21)
14: Cálculo do fluxo médio nodal usando as Eqs. (2.22a e 2.22b)
15: Cálculo da correntes parciais de saída usando a Eq. (2.23)
16: **end for**
17: Cálculo do K_{eff}
18: **end while**

O algoritmo 1 mostra claramente que o NEM é, por natureza, sequencial. Além disso, devido a solução ser local em um dado nodo, torna-se necessário impor continuidade de fluxo e corrente para estender essa solução para as demais regiões que compõe o núcleo do reator. Neste algoritmo, isso é feito através do cálculo das correntes de entrada e de saída entre nodos vizinhos, bem como das condições de contorno e simetria que são atualizadas nodo a nodo. Devido a essa dependência sequencial entre os cálculos dos nodos, a paralelização direta deste processo no contexto da GPU não é viável. Para contornar essa limitação, implementamos o método *Red-Black*, conforme será detalhado na subseção 4.1.6. Vale lembrar que o cálculo nodal sequencial original emprega o esquema de varredura detalhado no algoritmo 1, já no cálculo nodal paralelo o esquema de varredura dos nodos é diferente, como apresentado no algoritmo 2.

O acoplamento nodal entre nodos vizinhos é fundamental para garantir a continuidade de fluxo e correntes nas interfaces dos nodos. Na Figura 2.2, apresentamos o esquema de acoplamento nodal entre dois nodos vizinhos $n1$ e $n2$, tanto em uma configuração sequencial quanto em uma configuração paralela.

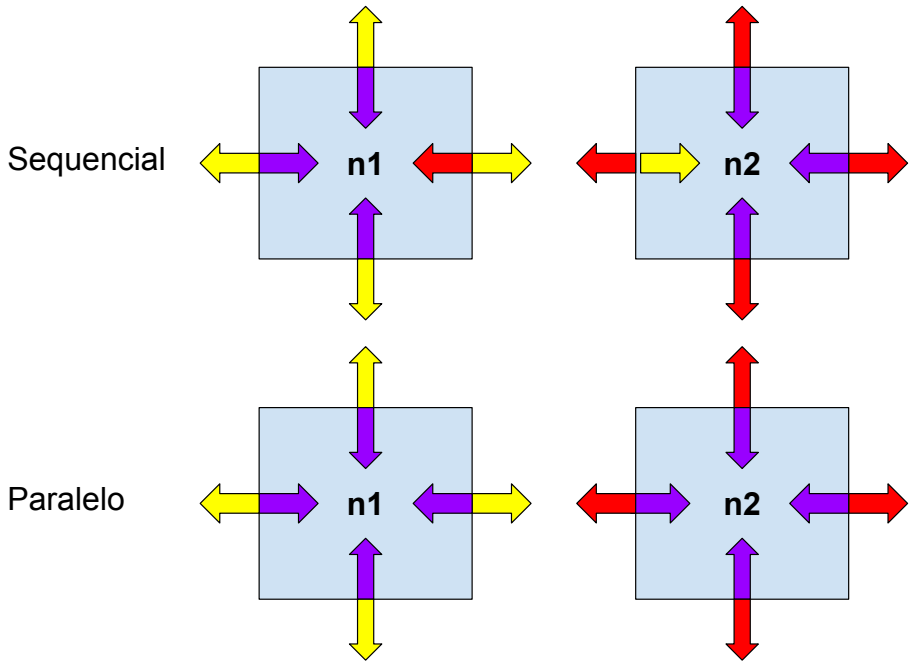

Figura 2.2: Esquema de acoplamento nodal entre dois nodos vizinhos

A figura mostra a continuidade de fluxo e correntes nas interfaces dos nodos em configuração sequencial (acima) e em configuração paralela (abaixo).

Na configuração sequencial, a corrente de saída do nodo $n1$ é utilizada como corrente de entrada do nodo vizinho ($n2$) na mesma iteração, garantindo a dependência direta dos dados entre os nodos vizinhos.

Por outro lado, na configuração paralela, durante o cálculo, os dados não são atualizados imediatamente. Em vez disso, essas atualizações ocorrem na próxima iteração, permitindo a execução paralela dos cálculos sem conflitos de dependência de dados.

Esse esquema de cálculo paralelo foi essencial para maximizar a eficiência computacional ao realizar cálculos em paralelo na GPU, pois permite uma execução mais rápida dos cálculos nodais.

Então, embora a varredura nodal sequencial apresente limitações para paralelização direta na GPU, a implementação do método *Red-Black* oferece uma solução eficaz para contornar essa questão. Nos próximos capítulos, mostraremos como conseguimos contornar essa dependência e realizar os cálculos paralelos de forma assíncrona nos nodos, garantindo, ao mesmo tempo, a simultaneidade necessária para obter ganhos de desempenho reais.

Na próxima seção, abordaremos a definição da distribuição de potência normalizada no EC, fundamental para a compreensão do comportamento do reator. Esta distribuição é definida em termos da variável fluxo de nêutrons, calculada utilizando

o NEM, que fora apresentado na seção 2.1.

2.3 Distribuição de Potência

A distribuição de potência em um reator nuclear refere-se à maneira como a potência gerada pelo reator está distribuída espacialmente ao longo do seu núcleo. Essa distribuição é utilizada como parâmetro crítico para garantir o funcionamento seguro e eficiente do reator. Dessa forma, é essencial o uso de algoritmos capazes de prever as distribuições de fluxos de nêutrons, e com isso, as distribuições de potência e taxas de reações, em cada parte do núcleo, de forma eficiente e precisa, seja em cálculos de projetos de núcleo ou mesmo em operações de reatores nucleares. Sendo esta variável definida por meio das equações apresentadas a seguir.

A distribuição de densidade de potência é assim definida:

$$p(x,y,z) \equiv \sum_{g=1}^{2} \omega\Sigma_{fg}(x,y,z)\phi_g(x,y,z), \tag{2.24}$$

onde $\omega\Sigma_{fg}(x,y,z)$ é o produto da energia revertida por fissão pela seção de choque macroscópica de fissão para o grupo g.

Mas, como parte da análise dos resultados refere-se à densidade de potência normalizada no EC, esta grandeza é definida como se segue.

A densidade de potência média no nodo n é dada por:

$$\bar{p}^n = \sum_{g=1}^{G} w\Sigma_{fg}^n \bar{\phi}_g^n, \tag{2.25}$$

onde $\bar{\phi}_g^n$ é o mesmo definido na Eq. 2.4.
Com isso, a densidade de potência média de um Elemento Combustível (EC) pode ser assim calculada [21]:

$$\bar{P}_{EC} = \frac{1}{V_{EC}} \sum_{n \in EC} \bar{p}^n V_n, \tag{2.26}$$

sendo V_n o volume do nodo n e V_{EC} o volume de um EC.

Já a densidade de potência média do núcleo é dada por:

$$\bar{P} = \frac{1}{V_{ativo}} \sum_{EC} \bar{P}_{EC} V_{EC}, \tag{2.27}$$

sendo V_{ativo} o volume do núcleo ativo.

Sendo assim, a densidade de potência normalizada no EC é dada por:

$$\bar{p}^{EC} = \frac{\bar{P}_{EC}}{\bar{P}}. \tag{2.28}$$

O NEM calcula os fluxos de nêutrons em cada nodo que serão usados para determinar a distribuição de potência média no EC e no núcleo. Com esses resultados, podemos calcular os fatores de pico de potência que são utilizados como critério de seleção de núcleo em cálculo de otimização de recarga. No CNFR, o programa ainda calcula as densidades de potência médias nas faces e nos cantos do nodo.

Na subseção 5.1.2, são apresentados os resultados de cálculos com os benchmarks em termos da densidade de potência normalizada no EC.

Neste capítulo, apresentamos o NEM em detalhes, desde suas equações constitutivas até a implementação prática do algoritmo de varredura dos nodos e o cálculo da densidade de potência normalizada no EC. Ao entender os fundamentos e o funcionamento do NEM, estabelecemos uma base para explorar abordagens avançadas de otimização e paralelização. No próximo capítulo, concentraremos nossa atenção na técnica de Decomposição de Domínio, especificamente na sua aplicação utilizando o método *Red-Black*. Esta abordagem inovadora visa acelerar ainda mais os cálculos nodais, aproveitando o poder da computação paralela na GPU. Ao fazer isso, continuamos caminhando em direção ao desenvolvimento de um algoritmo de cálculo nodal paralelo altamente eficiente e robusto para sistemas de cálculos neutrônicos.

Capítulo 3

Decomposição de domínio

Neste capítulo, exploramos a técnica de Decomposição de Domínio e sua aplicação no contexto do cálculo nodal paralelo na GPU. Em particular, destacamos a implementação dessa técnica através do método *Red-Black* [22] para realizar a varredura dos nodos de forma paralela. Ao compreendermos a metodologia por trás da decomposição de domínio e sua integração com o cálculo nodal paralelo, estaremos preparados para analisar os benefícios e desafios associados a essa abordagem. Este capítulo representa um passo importantíssimo para o desenvolvimento de um algoritmo de cálculo nodal paralelo e eficiente, contribuindo assim para a otimização dos cálculos nodais paralelos e possível aplicação aos demais cálculos neutrônicos.

O algoritmo do NEM é dito sequencial uma vez que os nodos são varridos sequencialmente, e as correntes de saída nas faces do nodo são usadas como correntes parciais de entrada para seus seis nodos vizinhos. Portanto, no contexto da GPU, foi necessário desacoplar esses nodos e os cálculos atuais parciais durante o processo iterativo para obter efetivamente um desempenho paralelo.

Neste capítulo, abordamos a técnica de Decomposição de Domínio no contexto do NEM, com foco na implementação paralela na GPU utilizando o método *Red-Black*. Inicialmente, descrevemos como a decomposição de domínio é aplicada ao cálculo nodal paralelo, destacando sua importância para a otimização do desempenho computacional. Em seguida, discutimos a definição do domínio e como ela influencia a divisão e distribuição dos cálculos entre os processadores da GPU.

Além disso, apresentamos uma análise da implementação do método *Red-Black* para realizar a varredura dos nodos de forma paralela. Como exemplo, utilizamos a rotina de cálculo das correntes parciais de saída na varredura NEM na GPU. Essa rotina compõe, desde a alocação de memória até o cálculo das correntes de saída para cada nodo. Um exemplo dessa rotina está no apêndice A.2. Esta análise detalhada das rotinas implementadas proporciona uma compreensão mais completa do funcionamento do algoritmo paralelo na GPU e dos desafios enfrentados durante sua implementação.

Nesse capítulo deve ficar claro a Decomposição de Domínio e sua integração com o NEM, pois isso é fundamental para a análise dos resultados obtidos com a implementação paralela na GPU apresentados na seção 5.1.2.

3.1 Decomposição de domínio

Neste sentido, a decomposição de domínio baseada no algoritmo de Schwarz ([23]) é usada para tornar o processo de varredura dos nodos paralelizável.

A decomposição de domínio refere-se a um conjunto de técnicas do tipo "dividir para conquistar", muito utilizadas em análise numérica e solução de equações diferenciais parciais. Essas técnicas são baseadas numa decomposição de domínio, ao qual a equação diferencial é formulada, em subdomínios [24].

Nesta tese, a ideia é obter a solução da equação em cada subdomínio e depois utilizá-la para obter a solução no domínio original. Por exemplo, no cálculo do fluxo de nêutrons, a solução direta das equações no domínio original requer a solução do sistema linear formado pelas equações 2.22a e 2.22b para cada nodo. Resolver diretamente este sistema linear para os diversos nodos de, por exemplo, centenas de configurações de núcleo tem um alto custo computacional e/ou leva muito tempo de processamento. Neste caso, uma abordagem de decomposição de domínio requer "somente" resolver esse sistema de equações nos subdomínios várias vezes simultaneamente, em vez de resolver o problema diretamente no domínio original.

3.1.1 Representação do domínio

As técnicas de decomposição de domínio são amplamente utilizadas em diversas áreas da matemática [25] e [26] para dividir um problema de valor limite em problemas menores em subdomínios e iterar para coordenar a resposta entre eles. No nosso problema, o núcleo do reator, de acordo com a nodalização feita inicialmente, é representado pelo domínio $\mho$ que está dividido em subdomínios $\mho_i$ disjuntos, ou seja, $\partial\mho_i \cup \partial\mho_j = \varnothing, i \neq j$. Assim, cada subdomínio representa um nodo ou no caso do *Red-Black*, os nodos do tipo *Red* e os nodos do tipo *Black*, e assim, os cálculos algébricos do NEM são executados em paralelo em cada subdomínio.

Essa técnica alterna entre células vermelhas e pretas, garantindo que células da mesma cor não sejam vizinhas diretas, o que permite atualizações simultâneas sem conflitos de dependência de dados dentro do subdomínio.

Na Figura 3.1, apresentamos a aplicação do método *Red-Black*. Cada nodo é representado como uma célula vermelha ou preta, onde as células pretas não são vizinhas das células vermelhas.

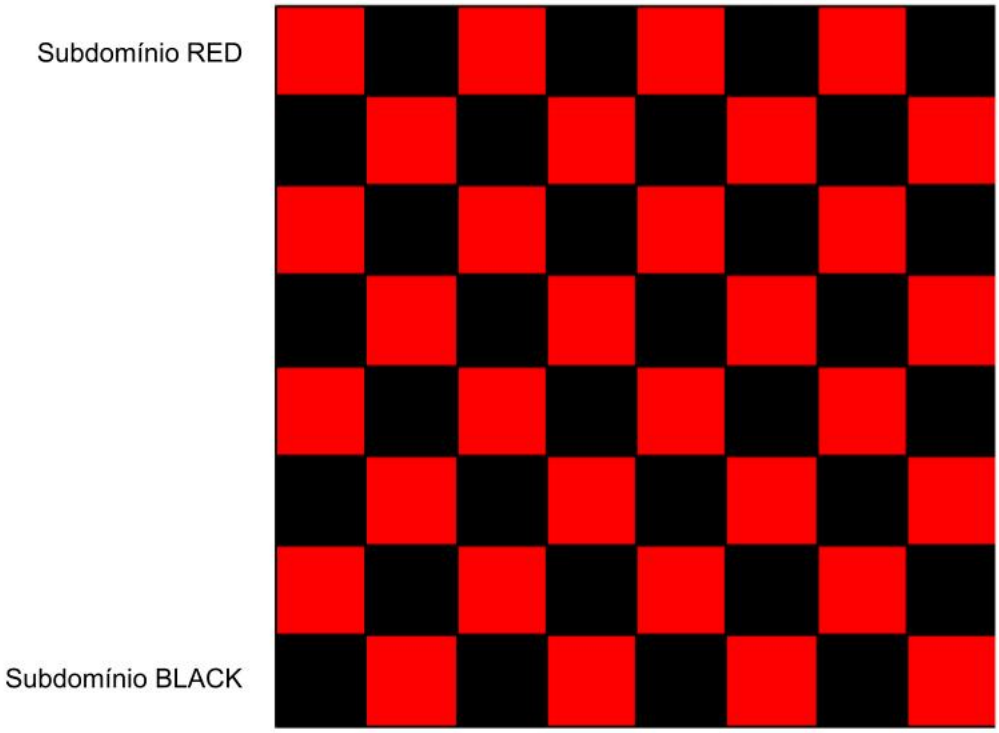

Figura 3.1: Representação do domínio (subdomínios em *Red-Black*)

Essa organização do domínio é essencial para maximizar a eficiência computacional ao realizar cálculos em paralelo na GPU, pois permite uma execução mais rápida e eficiente do cálculo nodal.

3.1.2 *Red-Black*

Nesta tese, a técnica de decomposição de domínio foi implementada ao *Red-Black*, inspirado no método de Gauss-Seidel [27], em dois subdomínios, o subdomínio *Red* e o subdomínio *Black*, onde a principal regra é que esses subdomínios sejam independentes, ou seja, os nodos que compõem o subdomínio *Red* não sejam vizinhos dos nodos que compõem o subdomínio *Black*. Essa independência se justifica no momento da varredura dos nodos, onde os nodos pertecente a cada subdomíno são executados de forma não simultânea para que um não interfira no cálculo do outro, ou seja, primeiramente são executados, por exemplo, todos os nodos pertencente ao subdomínio *Red* e depois são executados todos os nodos pertencente ao subdomínio *Black*.

Nisso, foi feita, por meio de lista encadeada de nodos *Red* e nodos *Black*, o processo de varredura. Primeiramente escolhe-se, por exemplo, os nodos do subdomínio *Red* e vai sendo montado algo parecido com o tabuleiro de xadréz com os nodos do subdomínio *Black*. Essa estrutura de dados é formada por meio do esquema linha x

coluna x camada já predefinido na nodalização.

Para melhor exemplificar este caso, nas equações 2.22a e 2.22b, pode-se observar que, por exemplo, se os nodos $n1$ e $n2$ forem vizinhos na direção u, e o cálculo da corrente parcial de saída de n1 preceder os cálculos que utilizam esta corrente parcial de entrada para o nodo n2 não há problema, entetanto, o caso contrário cria uma condição de corrida [28]. Além disso, a execução em GPU não permite definir a sequência de *threads* que serão executados.

Implementar um método *Red-Black* paralelo em uma GPU é um desafio complexo devido à natureza das operações envolvidas na estrutura do método e às limitações das GPUs em relação à execução simultânea de operações complexas em estruturas de dados arbitrárias. No entanto, é possível criar uma versão *Red-Black* paralela na GPU usando as técnicas de programação paralela disponíveis.

Neste trabalho, o método *Red-Black* foi implementado no cálculo nodal paralelo, especificamente no esquema de varredura dos nodos na GPU. Esse artifício foi implementado para que os nodos sejam varridos de forma simultânea, dentro do subdomínio correspondente, tornando assim, o processo de cálculo paralelo e otimizado. Nesta subseção constam alguns detalhes teóricos dessa técnica, mas sua implementação computacional na GPU está detalhada na subseção 4.1.6 para o contexto dessa pesquisa.

Ao finalizar este capítulo, destacamos a importância da técnica de Decomposição de Domínio e do método Red-Black na otimização do cálculo nodal paralelo na GPU. Entendendo os princípios subjacentes e as implementações práticas dessas técnicas, pode-se explorar os aspectos computacionais da implementação paralela na GPU, que serão abordados no próximo capítulo.

No Capítulo 4, adentraremos no âmbito da implementação computacional paralela utilizando a arquitetura CUDA na GPU. Discutiremos questões fundamentais, como o acesso eficiente à memória e a organização dos *threads*, que desempenham um papel fundamental na maximização do desempenho e na escalabilidade do algoritmo paralelo. Isso direciona ao desenvolvimento de um algoritmo de cálculo nodal paralelo altamente eficiente para o cálculo neutrônico, aproveitando ao máximo o potencial da computação paralela na GPU.

Capítulo 4

Implementação computacional paralela

A implementação computacional paralela é uma técnica fundamental na programação, na qual múltiplas tarefas são executadas simultaneamente para aprimorar o desempenho e a eficiência de um programa. Essa prática é especialmente comum em computação de alto desempenho e em sistemas multi-core, nos quais as tarefas podem ser divididas e executadas em paralelo, aproveitando ao máximo os recursos disponíveis. A escolha da técnica de implementação paralela depende do ambiente de execução e dos requisitos específicos do problema a ser resolvido.

Nesta tese, como mencionado nos capítulos anteriores, adotamos a implementação paralela na GPU como parte integrante de nossa abordagem para cálculos nodais paralelos. Este capítulo é dedicado à apresentação detalhada do método, baseado na decomposição de domínio e na otimização para GPU. Exploraremos os passos do algoritmo e as estratégias de implementação adotadas, com o objetivo de maximizar a eficiência computacional e alcançar resultados significativos no cálculo nodal e possível aplicação aos demais cálculos neutrônicos.

4.1 Implementação em GPU

Nesta seção, é detalhada a implementação do algoritmo de varredural nodal paralelo e a paralelização do cálculo nodal de malha grossa para execução em GPU.

Para o contexto dessa pesquisa, foi criada uma biblioteca de operadores matriciais que permitem alocação e manipulação de matrizes densas de dimensões de 1 a 6 e vetores densos, utilizando orientação a objeto e *Template Metaprogramming* [29] em C++ e CUDA. Além disso, foram construídos, de maneira análoga, operadores para matrizes esparsas em formato COO, CSR e ELL [30]. Estas bibliotecas podem ser utilizadas tanto em CPU quanto em GPU e a alocação dinâmica das matrizes densas

em GPU utiliza uma alocação de memória em duas dimensões.

Ao introduzir a implementação do algoritmo de varredura nodal e a paralelização do cálculo nodal de malha grossa para execução em GPU, é importante destacar que essa seção estabelece a base técnica para o desenvolvimento subsequente. Através da descrição da biblioteca de operadores matriciais e sua capacidade de manipular eficientemente matrizes densas e vetores densos de 1 até 6 dimensões, bem como a implementação de operadores para matrizes esparsas em diferentes formatos, estabelecemos as bases para a análise detalhada que se segue. Na próxima subseção, exploraremos o processamento genérico possibilitado por essa infraestrutura técnica, delineando suas aplicações específicas no contexto deste estudo e discutindo as implicações para a eficiência computacional e a escalabilidade das simulações realizadas.

4.1.1 Processamento Genérico

A linguagem CUDA assumiu um grande protagonismo no campo da computação científica nos últimos anos, principalmente nas aplicações de inteligência artificial, aprendizado profundo e de redes neurais artificiais [17].

De forma geral, o *hardware* da NVIDIA de menor custo é constituído por GPUs utilizadas em *desktops* para jogos. E, neste caso, a NVIDIA utiliza processadores com um número reduzido de unidades de ponto flutuante de precisão dupla e a relação entre o número de núcleos CUDA e as unidades de ponto flutuante de precisão dupla varia entre 1/32 e 1/64. É claro que isto impactará no desempenho computacional, dependendo do tipo de simulação realizada. Além disso, outros fatores que não o número de núcleos CUDA podem impactar no desempenho computacional, por exemplo, o tamanho do barramento de memória, o tipo e a velocidade da memória DRAM utilizada, o *clock* das memórias e do processador. Neste trabalho, algumas dessas limitações serviram para modelar a maneira como o código foi desenhado, com o intuito de minimizar principalmente as perdas de desempenho devido ao desalinhamento do acesso à memória.

Segundo o NVIDIA Guide [31], o desempenho do acesso à memória pelo programa em CUDA depende da distribuição dos endereços de memória, acessados pelos *threads* nos *warps*. Esta distribuição pode afetar o rendimento das instruções e é específica para cada tipo de memória, que pode ser global, shared, constante, local ou textura. E quanto mais dispersos estiverem os endereços de memória acessados pelos *thread*s do *warp*, menor será o desempenho computacional.

Outro fator que pode impactar no desempenho computacional, é o acesso contíguo às posições de memória. Diz-se que as transações de memória são coalescentes [1] quando a GPU acessa a memória em um bloco de 32 bytes e que está armazenada consecutivamente na memória. Neste caso, apenas uma única transação de memó-

ria será utilizada para recuperar todos os 32 bytes uma vez. Caso contrário, diz-se que a memória é não coalescente e, neste caso, a GPU precisará realizar múltiplas transações para recuperar todos os dados. A NVIDIA informa que a utilização de memória bidimensional, alocada pelas funções cudaMallocPitch() e cuMemAllocPitch(), produz um melhor desempenho computacional, pois neste caso o acesso à memória é sempre coalescente.

Em resumo, esta subseção destacou a importância da linguagem CUDA na computação paralela, especialmente em áreas como inteligência artificial e aprendizado profundo. Discutimos as características do *hardware* NVIDIA, desde GPUs de baixo custo até modelos mais avançados, e como essas características afetam o desempenho computacional, incluindo o número de núcleos, a distribuição de endereços de memória e a coalescência das transações de memória. Também exploramos como algumas dessas limitações influenciaram o design do código neste trabalho, com o objetivo de mitigar perdas de desempenho. Na próxima subseção, abordaremos a taxonomia de Flynn, fornecendo um contexto teórico importante para entendermos as características de paralelismo e classificação de arquiteturas de computação.

4.1.2 A taxionomia de Flynn

A taxonomia de Flynn [16] é uma classificação proposta por Michael J. Flynn em 1966 para categorizar arquiteturas de computadores com base na forma como os processadores executam instruções. Ela é amplamente usada na computação para distinguir diferentes tipos de processadores. Esta taxonomia define as seguintes quatro classes principais:

1. SISD (Single Instruction, Single Data):

   "Instrução Única, Dado Único". Nesse tipo de arquitetura, um único processador executa uma única instrução em um único conjunto de dados por vez. É o modelo tradicional de computador uniprocessador, onde as instruções são executadas sequencialmente em um único fluxo de dados.

2. SIMD (Single Instruction, Multiple Data):

   "Instrução Única, Dados Múltiplos". Nesse tipo de arquitetura, um único processador executa a mesma instrução em múltiplos conjuntos de dados simultaneamente. Isso é comum em processadores vetoriais e algumas GPUs, onde a mesma operação é aplicada a vários elementos de dados em paralelo.

3. MISD (Multiple Instruction, Single Data):

   "Instruções Múltiplas, Dado Único". Na arquitetura MISD, vários processadores executam diferentes instruções em um único conjunto de dados. Essa categoria é rara na prática e não é amplamente usada.

4. MIMD (Multiple Instruction, Multiple Data):

   "Instruções Múltiplas, Dados Múltiplos". Nesse tipo de arquitetura, vários processadores independentes executam diferentes instruções em conjuntos de dados distintos. Essa é a categoria mais comum em sistemas multiprocessados, como clusters de computadores, supercomputadores e alguns tipos de servidores. Cada processador opera de forma independente, permitindo o processamento paralelo de tarefas.

A Taxonomia de Flynn ajuda a classificar e entender as arquiteturas de computadores com base em como as instruções e os dados são tratados pelos processadores, o que é importante para projetar sistemas de computação e entender as capacidades de processamento paralelo.

Após definirmos a taxonomia de Flynn, que fornece uma base teórica para compreendermos o paralelismo e a classificação de arquiteturas de computação, agora nos voltamos para examinar o acesso à memória. A próxima subseção abordará como os programas em CUDA lidam com a memória, incluindo as estratégias de acesso e otimização para garantir o máximo desempenho computacional. Ao entendermos tanto a estrutura de processamento quanto a gestão eficiente de memória no contexto das GPUs, estaremos mais bem preparados para otimizar e aprimorar nosso algoritmo para ambientes de computação paralela.

4.1.3 Acesso à memória

Uma consideração extremamente importante no contexto da implementação paralela na GPU é o acesso eficiente à memória. Enquanto a GPU possui memória física organizada principalmente em formato 2D ou 3D, nosso problema requer o acesso a uma estrutura de dados mais complexa, representando as seis faces dos nodos.

Para lidar com essa complexidade, recorremos ao acesso à memória 2D *pitched* da GPU. Nessa abordagem, os dados são organizados em uma estrutura 2D, mas com um espaçamento (*pitch*) entre as linhas, permitindo um acesso eficiente a elementos adjacentes (faces dos nodos vizinhos).

No nosso caso, cada nodo do problema é associado a seis variáveis, uma para cada face. A organização dos dados em uma estrutura *pitched* permite que os cálculos sejam realizados de maneira eficiente para cada uma dessas faces. Assim, garantimos uma utilização otimizada dos recursos da GPU e uma execução rápida e eficaz dos cálculos nodais paralelos. Como o acesso à memória é um aspecto crítico na computação e no design de sistemas de computadores, a eficiência do acesso à memória pode afetar significativamente o desempenho de um sistema [32]. Por isso, existem várias considerações importantes relacionadas ao acesso à memória, incluindo:

Hierarquia de Memória:

Os sistemas de computadores modernos geralmente têm uma hierarquia de memória que consiste em diferentes níveis de memória, como registradores da CPU, memória cache, memória principal (RAM) e armazenamento em disco. A memória é organizada em hierarquias devido à diferença de velocidade e custo entre os diferentes tipos de memória. Aproximar-se de memórias mais rápidas (por exemplo, cache) é mais rápido, mas também mais caro.

Latência e Largura de Banda:

A latência é o atraso entre o momento em que uma solicitação de memória é feita e o momento em que os dados são realmente lidos ou gravados na memória. A largura de banda se refere à quantidade de dados que pode ser transferida entre a CPU e a memória em um determinado período de tempo. O acesso à memória é influenciado tanto pela latência quanto pela largura de banda.

Localidade:

Os princípios de localidade, como a localidade de referência espacial (acesso frequente a dados próximos no espaço) e a localidade de referência temporal (acesso repetido aos mesmos dados ao longo do tempo), são fundamentais para otimizar o acesso à memória. Algoritmos de cache e estruturas de dados são projetados para explorar esses princípios.

Acesso Aleatório X Acesso Sequencial:

O acesso à memória pode ser aleatório, no qual os dados são acessados de maneira imprevisível, ou sequencial, onde os dados são acessados em ordem. O acesso sequencial é geralmente mais eficiente, pois permite que o sistema aproveite a localidade.

Arquitetura da CPU e Memória:

A arquitetura da CPU e a organização da memória variam entre diferentes tipos de sistemas. Por exemplo, sistemas com várias CPUs, como servidores e supercomputadores, têm requisitos de acesso à memória diferentes dos sistemas de dispositivos móveis.

Acesso à Memória Compartilhada em Programação Concorrente:

Em programação concorrente, o acesso à memória compartilhada por várias *thread*s ou processos pode levar a problemas de concorrência, como corridas de dados e condições de corrida. Técnicas como mutexes e semáforos são usadas para garantir

acesso seguro à memória compartilhada.

Memória Virtual:

Em sistemas operacionais, a memória virtual é usada para criar a ilusão de que o sistema tem mais memória física do que realmente possui. Isso envolve a tradução de endereços virtuais para endereços físicos e a gestão da página de troca para alocar eficazmente a memória física. O acesso à memória é um aspecto crucial do design de sistemas de computadores e da otimização de código, e entender os princípios relacionados a ele é fundamental para desenvolvedores de *software* e engenheiros de sistemas. Melhorar o acesso à memória é frequentemente um dos principais desafios no desenvolvimento de *hardware* e *software* para melhorar o desempenho do sistema.

Quando se usa GPUs para tarefas de computação de propósito geral, o acesso à memória continua sendo um aspecto fundamental, mas as considerações específicas variam um pouco em comparação com a programação em CPU.

Aqui estão algumas considerações adicionais relacionadas ao acesso à memória quando se utiliza uma GPU:

Memória Global da GPU:

As GPUs possuem sua própria memória global dedicada, separada da memória principal do sistema (RAM). O acesso à memória global da GPU pode ser mais lento em comparação com a memória de cache na CPU, portanto, é importante minimizar as transferências de dados entre a CPU e a GPU.

Transferência de Dados entre CPU e GPU:

Transferir dados entre a CPU e a GPU pode ser uma operação custosa em termos de tempo. É importante minimizar essas transferências sempre que possível, pois elas podem ser um gargalo no desempenho. O uso eficaz da memória global da GPU para armazenar dados temporários é uma estratégia comum.

Memória Compartilhada:

As GPUs geralmente têm uma memória compartilhada que pode ser usada para armazenar dados temporários que são compartilhados entre as *threads* em um bloco. O acesso à memória compartilhada é significativamente mais rápido do que o acesso à memória global da GPU, tornando-a uma opção atraente para melhorar o desempenho em algumas situações.

Coalescência de Acesso:

Para otimizar o acesso à memória global da GPU, é importante que os *threads* acessem os dados de forma cooperativa, o que é chamado de coalescência. Isso

envolve o acesso a dados em um padrão que minimize a latência da memória e maximize a largura de banda.

Memória Constante e Textura:

As GPUs também possuem memória constante e memória de textura, que são projetadas para otimizar tipos específicos de acesso a dados, como leitura de texturas em processamento de imagens. Usar essas memórias especiais pode melhorar o desempenho em casos específicos.

Memória Unificada:

Alguns modelos de GPU mais recentes e sistemas suportam memória unificada, que permite que a CPU e a GPU compartilhem a mesma memória física. Isso simplifica o gerenciamento de memória, mas ainda requer considerações cuidadosas ao acessar dados na GPU.

Paralelismo Massivo:

A principal vantagem da GPU é seu paralelismo massivo, permitindo que milhares de *threads* executem simultaneamente. No entanto, o acesso à memória deve ser cuidadosamente coordenado para evitar conflitos e garantir que as *threads* funcionem eficientemente em paralelo.

Ao programar em GPU, é essencial otimizar o acesso à memória para aproveitar ao máximo o poder de processamento paralelo da GPU. Isso envolve minimizar as transferências de dados entre a CPU e a GPU, otimizar o acesso à memória global e compartilhada da GPU e considerar o paralelismo massivo oferecido pela GPU ao projetar algoritmos e estruturas de dados.

Agora, na Figura 4.1, apresentaremos os tipos de memória disponíveis na GPU. Existem cinco tipos principais de memória na GPU, cada um com suas características e usos específicos e todas com características de tamanho, acesso e tempos de latência diferentes. Por isso, faz-se necessário examinar esses tipos de memória em detalhes, destacando suas vantagens e limitações e discutindo as melhores práticas para sua utilização eficaz no contexto em que se quer implementar.

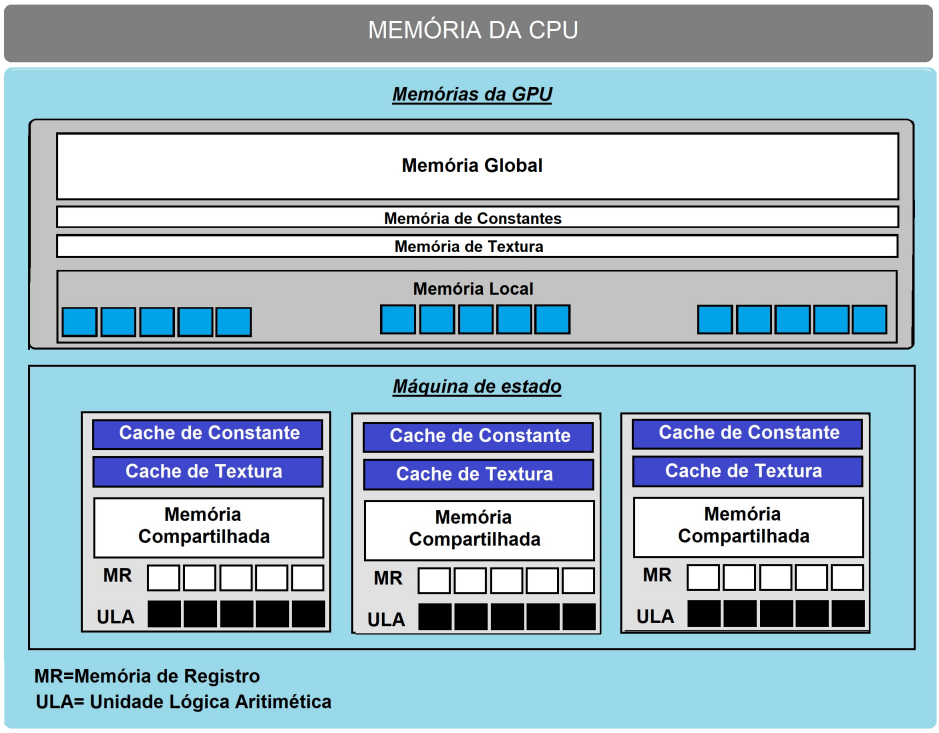

Figura 4.1: Esquema de memórias no contexto da GPU.

A arquitetura da GPU abrange uma variedade de tipos de memória, cada um com funções específicas para otimizar o desempenho e a eficiência de processamento. Conforme ilustrado na Figura 4.1, os principais tipos de memória incluem:

- Memória do Host (CPU): Esta é a memória principal do sistema, controlada pela CPU, e serve como interface entre a CPU e a GPU.
- Memória de Constantes: Destinada ao armazenamento de valores constantes que são acessados por todos os threads em um bloco, essa memória oferece baixa latência e é usada para parâmetros estáticos do kernel.
- Memória Global: É a principal área de armazenamento de dados na GPU, acessível por todos os threads e mantida durante toda a execução do programa.
- Memória Compartilhada: Localizada no nível do bloco, essa memória é compartilhada entre os threads de um mesmo bloco e oferece acesso rápido, sendo útil para compartilhar dados entre os threads.
- Memória de Texturas: Projetada para armazenar dados de texturas para acesso otimizado por shaders de processamento de imagens, essa memória oferece recursos de filtragem e interpolação.

- Memória de Registros: Cada thread possui seu próprio conjunto de registradores para armazenar dados temporários durante a execução do kernel, oferecendo o acesso mais rápido, mas com uma capacidade limitada.

Esses tipos de memória desempenham papéis distintos na arquitetura da GPU e são utilizados de forma estratégica para otimizar o desempenho e a eficiência dos algoritmos paralelos.

Em resumo, o acesso eficiente à memória desempenha um papel fundamental na otimização do desempenho de algoritmos paralelos na GPU. Nesta seção, exploramos as considerações importantes relacionadas ao acesso à memória, destacando a necessidade de uma organização cuidadosa dos dados e o uso de técnicas adequadas, como o acesso à memória 2D "pitched", para garantir um desempenho máximo. A compreensão desses princípios é essencial para o desenvolvimento de algoritmos paralelos eficientes e escaláveis.

Entender e otimizar esse acesso permite minimizar latências e maximizar a utilização dos recursos de *hardware*. No entanto, mesmo com um acesso à memória bem estruturado, é essencial abordar outro desafio significativo em computação paralela: as condições de corrida. Na subseção a seguir, citamos como as condições de corrida podem impactar a execução dos algoritmos e as estratégias utilizadas para evitá-las.

4.1.4 Condições de corrida

Condições de corrida (*race conditions*) na programação em GPU são previstas pela lei de Amdahl [28] e ocorrem quando múltiplos *threads* tentam acessar e/ou modificar simultaneamente o mesmo recurso compartilhado, como variáveis ou áreas de memória. Essa situação pode levar a resultados não determinísticos, inconsistências nos dados ou até mesmo falhas no programa, neste caso, quando resulta num processo eventualmente dependente da sequência ou sincronia de demais eventos .

É importante ter em mente a concorrência e a sincronização de dados ao trabalhar com programação paralela para evitar condições de corrida e outros problemas associados.

Para evitar condições de corrida em computação paralela na GPU, podem ser utilizadas diversas estratégias, incluindo sincronização de *threads* com barreiras de sincronização como *__syncthreads()* e operações atômicas; particionamento de dados através de dados privados por thread e redução paralela; design algorítmico com decomposição de domínio e o método *Red-Black*; gerenciamento de memória eficiente usando memória compartilhada e registros; e modelagem e análise de acesso com análise estática e ferramentas de *profiling* como *cuda-memcheck* e *Nsight*.

Evitar que ocorram condições de corrida é fundamental para garantir a precisão e a estabilidade das operações em ambientes de computação paralela. Superadas essas

questões, podemos explorar a programação em CUDA, uma ferramenta poderosa que nos permite aproveitar o potencial das GPUs. Na próxima subseção, discutiremos os aspectos essenciais da programação em CUDA, destacando como essa tecnologia facilita a implementação eficiente dos cálculos nodais em paralelo.

4.1.5 Programação em CUDA

A programação de uma GPU utilizando o CUDA [31] e [1] é orientada à *thread*s, ou linhas de execução, que é uma forma de um processo dividir a si mesmo em duas ou até mesmo centenas de tarefas, que podem ser executadas simultaneamente. Neste modelo, quando um grid com blocos de *thread*s é criado, este consiste em centenas de processos unitários que podem ser identificados de maneira única pelo *hardware*, dependendo apenas de sua posição no bloco.

É importante ressaltar que o programador não tem influência na sequência sob a qual um *thread* individual ou um bloco de *threads* é processado. O *hardware* é responsável por isso através de gerenciadores internos de despacho e agendamento. Esses gerenciadores são responsáveis por distribuir e coordenar a execução dos *threads* e blocos de *threads* nos múltiplos núcleos da GPU, otimizando a utilização dos recursos disponíveis e garantindo a eficiência na execução das tarefas. Desta forma, a ordem de execução pode variar, o que requer que os programas sejam escritos de maneira a serem robustos e livres de dependências de ordem específicas para garantir resultados corretos e consistentes.

A programação em CUDA é uma abordagem que permite aproveitar o poder de processamento das GPU's para tarefas de computação paralela. É amplamente usada para acelerar o processamento de dados e cálculos em tarefas que podem ser paralelizadas, como processamento de imagens, simulações científicas, aprendizado de máquina e muito mais. Aqui estão os principais conceitos e passos envolvidos na programação em CUDA:

Configuração do Ambiente:

Para começar a programar em CUDA, se faz necessário um ambiente de desenvolvimento que inclua um compilador CUDA, como o NVCC (NVIDIA CUDA Compiler), e uma GPU NVIDIA compatível. Também é possível instalar o kit de desenvolvimento CUDA (CUDA Toolkit) fornecido pela NVIDIA.

Design de *kernel*:

No CUDA, o código que é executado na GPU é chamado de *kernel*. É preciso projetar o *kernel* de forma que ele possa ser executado em paralelo por muitos *threads* na GPU. Isso envolve identificar tarefas que podem ser paralelizadas e dividir o

trabalho entre *threads*.

Alocação de Memória:

Se faz necessário alocar memória na GPU para armazenar dados que serão processados pelo kernel. Isso é feito usando funções CUDA para alocar memória global ou compartilhada.

Lançamento de *kernel*:

Lança-se o *kernel* a partir da CPU para a GPU, especificando o número de blocos e *threads* por bloco que executarão o *kernel*. Cada *thread* executará o mesmo código do *kernel*, mas geralmente com dados diferentes.

Sincronização:

É importante controlar a sincronização entre as *threads* para garantir que o cálculo seja feito corretamente. É possível usar funções de sincronização para coordenar os *threads* quando necessário.

Transferência de Dados:

Para enviar dados entre a CPU e a GPU, deve-se usar funções de cópia de memória para transferir os dados da CPU para a memória da GPU.

Gerenciamento de Erros:

É importante verificar erros ao programar em CUDA, pois eles podem ocorrer durante a alocação de memória, transferência de dados e execução do *kernel*. O CUDA fornece funções para verificar e lidar com erros.

Otimização:

A otimização é uma parte importante da programação em CUDA. Isso envolve ajustar o tamanho dos blocos, *threads* e memória compartilhada, bem como a escolha de algoritmos eficientes para tirar o máximo proveito do hardware da GPU.

Depuração e Perfis:

Para facilitar a depuração e a otimização, é possivel usar ferramentas de depuração e perfilamento fornecidas pelo ambiente de desenvolvimento CUDA, como o NVIDIA Nsight e o Visual Profiler.

A programação em CUDA é uma ferramenta poderosa para acelerar cálculos intensivos e paralelizáveis em GPU. No entanto, é importante ter em mente que nem todas as tarefas podem ser paralelizadas eficazmente em GPU, e a programação em CUDA pode ser complexa em comparação com a programação serial tradicional em

CPU. Portanto, ela é melhor utilizada em cenários que se beneficiam de paralelismo maciço.

A programação em CUDA nos permite explorar o paralelismo massivo oferecido pelas GPUs, tornando possível acelerar significativamente os cálculos envolvidos na simulação de reatores nucleares. Após discutirmos os fundamentos e as técnicas de programação em CUDA, a próxima subseção aborda a solução das equações constitutivas do NEM no contexto da GPU. Essas equações são fundamentais para o cálculo nodal, e sua implementação paralela é um dos principais focos desta tese.

4.1.6 Solução das Equações Constitutivas do NEM

A solução das equações constitutivas do NEM apresenta um esquema onde a varredura dos nodos é baseada na utilização de mapas de vetores indexados. Estes mapas, ou *hash tables*, são determinados previamente, antes do início da execução do processo iterativo de cálculo do fator de multiplicação efetivo e do fluxo de nêutrons. As *hash tables* são necessárias para aplicação das condições de continuidade e de contorno nas faces de cada nodo, pois dependem da geometria, das condições de simetria e do tipo de condição de contorno do problema. Em particular tem-se dois mapas que definem dois conjuntos disjuntos de nodos, que denominados RED e BLACK, onde nenhum nodo pertencente ao conjunto RED é vizinho a outro nodo no mesmo conjunto, o mesmo ocorrendo para os nodos do conjunto BLACK, como apresentado na Figura 3.1. Diz-se que um nodo n_1 é um vizinho do nodo n_2, se existe uma interface entre n_1 e n_2, como mostrado na Figura 2.2.

Diz-se que uma linha de execução, ou *thread*, é uma sequência de instruções dadas ao processador em uma cadeia interligada de processos. Na programação em CUDA diz-se que um warp é um conjunto de 32 *threads* consecutivos em um bloco de *threads* e todos os *threads* de uma *warp* executam a mesma instrução. E uma vez que um bloco de *threads* é lançado em um processador este bloco é dividido em um conjunto de *warps* para execução.

Deve-se lembrar que na execução de uma rotina na GPU é obrigado definir, no instante de sua chamada, quantos *threads* serão utilizados e quantos serão lançados simultaneamente de cada vez em um bloco. Assim, por exemplo, se 1025 nodos são divididos em blocos unidimensionais de 32 *threads*, serão necessárias 33 posições de um grid unidimensional para varrer todos estes nodos.

Tanto o *grid* quanto os *blocks* podem ter dimensão 1, 2 ou 3. Optou-se, neste trabalho, por utilizar *grids* e *blocks* bidimensionais de modo a facilitar a programação, utilizando a memória alocada de forma bidimensional. Esta escolha se justifica, pois utilizamos esta abordagem na simulação de várias configurações de núcleos de reator simultaneamente, como ocorre no processo de otimização de modelos de recarga de

combustível nuclear.

As GPUs são processadores baseados na topologia SIMD (Single Instruction Multiple Data). Isto pode obrigar o processador a executar de forma serial as instruções que apresentem divergência de *threads*. Ou seja, quando dois *threads* do mesmo warp, após uma instrução condicional, devem seguir direções diferentes no programa, a máquina de estados na GPU bloqueia este processo e os serializa. Neste sentido, a indexação dos nodos em função dos mapas se faz necessária para evitar a utilização de instruções condicionais IF-THEN-ELSE no interior das iterações externas do processo iterativo de cálculo do fluxo de nêutrons, pois este tipo de instrução eventualmente força a serialização das instruções, implicando na perda de desempenho computacional. No apêndice A.1, mostramos um exemplo de uma rotina do programa original em Fortran que contém diversos condicionais desse tipo.

No entanto, apesar do esforço empregado no mapeamento, em algumas rotinas houve a necessidade da utilização de segmentos de código condicionado, por exemplo, nos cálculos da fuga transversal e das correntes de entrada no nodo (que por condição de continuidade são iguais as correntes de saída dos nodos vizinhos). Nestes dois casos, utilizou-se um método de programação "*non-standard*" onde a sequência de instruções é independente do resultado do teste de condição. A sequência de comandos, neste caso, independe do *thread*, pois o que muda é o coeficiente do seguinte polinômio de primeiro grau $p(h) = (1-h) \times x + h \times y$, onde x e y são as variáveis que se deseja retornar através do teste enquanto h é uma variável booleana ou inteira. Por exemplo, seja S o valor do resultado sob condição, V o valor de entrada e h o resultado do teste, então $S = (1-h) \times V + h \times W$, onde $h = (X > Y)$. se h é verdadeiro, retorna 1, implicando que $S = W$; caso contrario , $S = V$. De maneira análoga, se fossem duas as condições de teste $h = (X_0 > Y_0) \wedge (X_1 < Y_1)$. Onde $\wedge$ e $\vee$ representam AND e OR lógicos, respectivamente.

O exemplo dado a seguir consiste em substituir a função

$$f(x, y, v, w) = \begin{cases} v; & x = y \\ w; & x \neq y \end{cases}$$

por uma sequência de instruções e o polinômio de primeiro grau acima mencionado. Sejam x e y inteiros e a variável booleana $h \equiv\sim (x \curlywedge y)$, onde $\curlywedge$ é o operador XOR bit a bit em C++ e $\sim$ implica em inverter a variável. Assim, se algum dos bits das variáveis x e y for diferente, então h retorna o valor true $\equiv 1$. E a rotina $f(x, y, v, w)$ retorna o valor $h \times v + (1-h) \times w$.

Todos os pontos abordados nesta subseção foram utilizados no desenvolvimento do programa Cuda_NEM, inclusive o esquema RED e BLACK de varredura dos nodos para calcular a dsitribuição do fluxo de nêutrons. O algoritmo de varreura

nodal paralelo, usado na GPU para cálculo as equações constitutivas do NEM, é mostrado no algoritmo 2.

Algorithm 2 Varredura Nodal Paralela

1: Inicializa-se K_{eff}, $\bar{\phi}_g^n$ e $\bar{J}_{gus}^{\pm n}$
2: **while** K_{eff} e $\bar{\phi}_g^n$ não convergirem, **do**
3: **for** C é RED até BLACK **do**
4: **for** Para todo nodo com cor C **do**
5: Atualização das correntes de entrada dos nodos vizinhos ao nodo n
6: **end for**
7: **for** Para todo nodo com cor C **do**
8: Cálculo dos fluxos médios nas faces do nodo usando a Eq. (2.12)
9: Cálculo dos coeficientes primários usando a Eq. (2.13)
10: **end for**
11: **for** Para todo nodo com cor C **do**
12: **for** cada direção u **do**
13: Cálculo da fuga transversal média usando a Eq. (2.18)
14: Cálculo das fugas transversais nas faces do nodo usando a Eq. (2.19a) ou (2.19b)
15: Cálculo dos coeficientes da expansão da fuga transversal usando a Eq. (2.17)
16: **end for**
17: **end for**
18: **for** Para todo nodo com cor C **do**
19: Cálculo dos coeficientes secundários usando as Eqs. (2.20a and 2.20b)
20: **end for**
21: **for** Para todo nodo com cor C **do**
22: Cálculo dos coeficientes das correntes parciais de saída usando a Eq. (2.21)
23: **end for**
24: **for** Para todo nodo com cor C **do**
25: Cálculo do fluxo médio nodal usando as Eqs. (2.22a and 2.22b)
26: **end for**
27: **for** Para todo nodo com cor C **do**
28: Cálculo da correntes parciais de saída usando a Eq. (2.23)
29: **end for**
30: **end for**
31: Cálculo do K_{eff}
32: **end while**

Neste capítulo, exploramos diversos aspectos fundamentais da implementação computacional paralela do NEM no contexto da GPU. Discutimos desde considerações importantes relacionadas ao acesso à memória até questões fundamentais de programação em CUDA e a solução das equações constitutivas do NEM.

Uma das principais contribuições deste trabalho é a implementação eficiente e escalável de um algoritmo de cálculo nodal paralelo na GPU, fundamentado na técnica de Decomposição de Domínio e paralelismo. Esta abordagem permite realizar

cálculos nodais com maior rapidez e eficiência, aproveitando o poder computacional da GPU.

Além disso, nosso trabalho introduz inovações significativas no tratamento de condições de corrida, garantindo a consistência e a integridade dos resultados mesmo em ambientes paralelos. A abordagem proposta demonstra um avanço significativo em relação aos métodos tradicionais, oferecendo uma solução segura e eficaz.

Ao combinar técnicas avançadas de computação paralela com o NEM, nosso trabalho contribui de forma substancial para o avanço do conhecimento científico e tecnológico no campo da física de reatores nucleares. A implementação bem-sucedida deste algoritmo representa não apenas um marco importante nesta tese, mas também uma contribuição valiosa para a comunidade científica e para o desenvolvimento futuro de, por exemplo, técnicas de otimização de recarga de combustível nuclear mais seguras, eficientes e sustentáveis.

O próximo passo desta tese é a apresentação, discussão e análises dos resultados obtidos com a implementação do algoritmo proposto. No Capítulo 5, analisaremos em detalhes os resultados das simulações realizadas com o Cuda_NEM, avaliando o desempenho do algoritmo em três diferentes cenários e comparando-o com os resultados obtidos pelo CNFR que utiliza o cálculo nodal convencional. Além disso, discutiremos as implicações dos resultados para a área de física de reatores nucleares e exploraremos possíveis direções para trabalhos futuros. Em suma, o Capítulo 5 representa uma etapa fundamental para a validação e compreensão do impacto do nosso trabalho, fornecendo uma base sólida para avanços futuros nesta área de pesquisa.

Capítulo 5

Resultados Numéricos e Discussões

Neste capítulo, são fornecidos detalhes sobre a implementação do sistema, incluindo tanto o aspecto de *software* quanto de *hardware* utilizados no desenvolvimento do algoritmo proposto. Apresentamos os resultados obtidos por meio das implementações realizadas, destacando métricas de desempenho computacional e os resultados de otimização dos cálculos alcançados.

Além disso, interpretamos os resultados dos cálculos realizados com o Cuda_NEM, discutindo suas implicações para o cálculo nodal. Realizamos uma comparação detalhada dos resultados com o CNFR, visando identificar avanços significativos e áreas de melhoria. Também analisamos os desafios enfrentados durante a pesquisa, discutindo possíveis soluções e aprendizados adquiridos ao longo do processo.

5.1 Cálculos CPU X GPU

Nesta seção, abordamos os problemas tratados, os resultados obtidos e as análises resultantes dos cálculos nodais executados pelos programas desenvolvidos tanto para a CPU quanto para a GPU.

Faremos uma comparação detalhada dos resultados alcançados por meio dessas implementações, destacando as diferenças de desempenho, eficiência e escalabilidade entre as duas abordagens. Além disso, analisaremos as implicações desses resultados para a prática da física de reatores nucleares, buscando identificar padrões e tendências que possam orientar futuros desenvolvimentos e melhorias.

Esta seção é essencial para avaliar o desempenho relativo da implementação paralela na GPU em comparação com a abordagem sequencial na CPU.

5.1.1 Apresentação dos Benchmarks

Conforme já foi dito no Capítulo 1, três benchmarks em geometria Cartesiana tridimensional encontrados na literatura, quais sejam, IAEA [6], LRABWR [33] e LMWLWR [34], foram tratados tanto com o código CNFR quanto com o programa Cuda_NEM. A configuração geométrica de cada um destes benchmarks são mostradas nas figuras 5.1, 5.2 e 5.3, respectivamente. Enquanto que nas tabelas 5.1, 5.2 e 5.3 são apresentados os dados nucleares para cada um dos benchmarks, respectivamente.

O primeiro benchmark apresentado é o IAEA, cuja configuração (radial e axial), para 1/4 núcleo, é mostrada na Fig. 5.1 e os dados nucleares mostrados na Tabela

5.1.

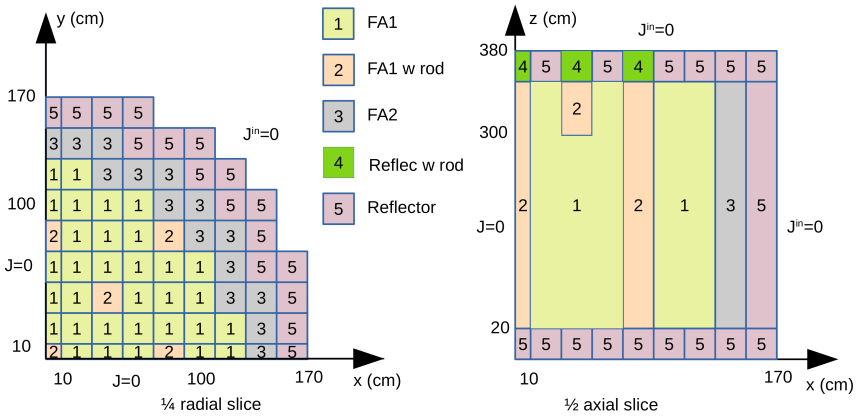

Figura 5.1: Configuração geométrica do benchmark IAEA

Tabela 5.1: Dados Nucleares do benchmark IAEA

Região	g	Σ_{ag} (cm^{-1})	$\nu\Sigma_{fg}$ (cm^{-1})	D_g (cm)	$\Sigma_s^{g\to(3-g)}$ (cm^{-1})	$w\Sigma_{fg}$ (cm^{-1})
1	1	0.010	0.000	1.5	0.02	0.000
	2	0.085	0.135	0.4	0.00	0.135
2	1	0.010	0.000	1.5	0.02	0.000
	2	0.130	0.135	0.4	0.00	0.135
3	1	0.010	0.000	1.5	0.02	0.000
	2	0.080	0.135	0.4	0.00	0.135
4	1	0.000	0.000	2.0	0.04	0.000
	2	0.055	0.000	0.3	0.00	0.000
5	1	0.000	0.000	2.0	0.04	0.000
	2	0.010	0.000	0.3	0.00	0.000

Os valores de referência (densidades de potência normalizada no EC e fator de multiplicação efetivo) para o benchmark IAEA foram gerados pelo programa VENTURE [35].

O segundo benchmark apresentado é o LRABWR, cuja configuração (radial e axial), para 1/4 núcleo, é mostrada na Fig. 5.2 e os dados nucleares mostrados na Tabela 5.2.

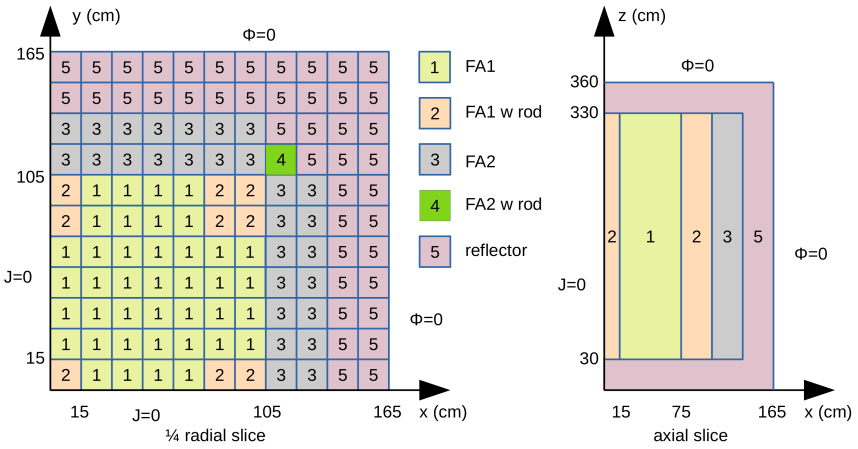

Figura 5.2: Configuração geométrica do benchmark LRABWR

Tabela 5.2: Dados nucleares do benchmark LRABWR

Região	g	Σ_{ag} (cm^{-1})	$\nu\Sigma_{fg}$ (cm^{-1})	D_g (cm)	$\Sigma_s^{g\to(3-g)}$ (cm^{-1})	$w\Sigma_{fg}$ (cm^{-1})
1	1	0.0082520	0.004602	1.255000	0.025330	0.004602
	2	0.1003000	0.109100	0.211000	0.000000	0.109100
2	1	0.0071810	0.004609	1.268000	0.027670	0.004609
	2	0.0704700	0.086750	0.190200	0.000000	0.086750
3	1	0.0080020	0.004663	1.259000	0.026170	0.004663
	2	0.0834400	0.102100	0.209100	0.000000	0.102100
4	1	0.0080020	0.004663	1.259000	0.026170	0.004663
	2	0.0733240	0.102100	0.209100	0.000000	0.102100
5	1	0.0006034	0.000000	1.257000	0.047540	0.000000
	2	0.0191100	0.000000	0.159200	0.000000	0.000000

Os valores de referência para o benchmark LRABWR foram gerados através de um cálculo de malha fina feito pelo código KIDD [33].

O terceiro e último benchmark apresentado é o LMWLWR, cuja configuração (radial e axial), para 1/4 núcleo, é mostrada na Fig. 5.3 e os dados nucleares mostrados na Tabela 5.3.

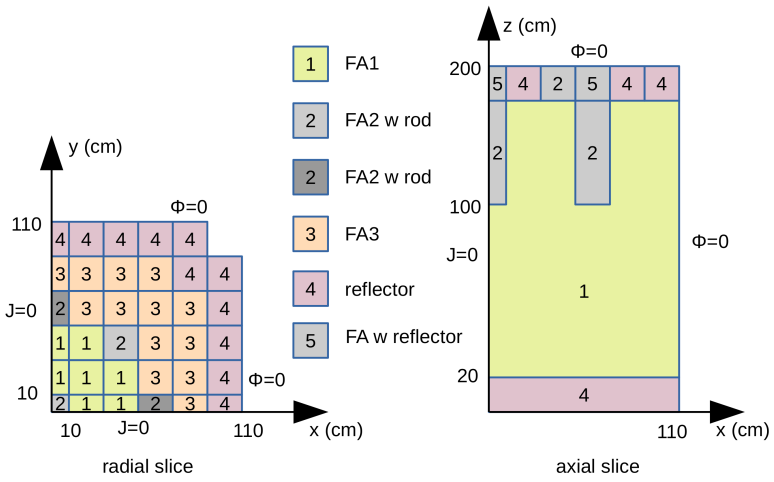

Figura 5.3: Configuração geométrica do benchmark LMWLWR

Tabela 5.3: Dados nucleares do benchmark LMWLWR

Região	g	Σ_{ag} (cm^{-1})	$\nu\Sigma_{fg}$ (cm^{-1})	D_g (cm)	$\Sigma_s^{g\to(3-g)}$ (cm^{-1})	$w\Sigma_{fg}$ (cm^{-1})
1	1	0.01040206	0.006477691	1.423913	0.01755550	8.301808786E-14
	2	0.08766217	0.112732800	0.356306	0.00000000	1.444783565E-12
2	1	0.01095206	0.006477691	1.423913	0.01755550	8.301808786E-14
	2	0.09146217	0.112732800	0.356306	0.00000000	1.444783565E-12
3	1	0.01099263	0.007503284	1.425611	0.01717768	9.616208774E-14
	2	0.09925634	0.137800400	0.350574	0.00000000	1.766049926E-12
4	1	0.002660573	0.000000000	1.634227	0.02759693	0.000000000E+00
	2	0.04936351	0.000000000	0.264002	0.00000000	0.000000000E+00
5	1	0.01095206	0.006477691	1.423913	0.01755550	8.301808786E-14
	2	0.09146217	0.112732800	0.356306	0.00000000	1.444783565E-12

Já os valores de referência para o benchmark LMWLWR foram gerados com cálculo de malha fina feito pelo código POLCA-T [36].

Concluímos a apresentação dos três benchmarks tratados, destacando suas características e relevância para nossa análise comparativa entre as implementações em CPU e GPU. Esses benchmarks fornecem uma base confiável para avaliar o desempenho e a eficácia de nosso algoritmo.

Na próxima subseção, procederemos com a apresentação e análise detalhada dos resultados obtidos por meio das implementações em CPU e GPU. Faremos uma comparação dos resultados dos cálculos realizados em ambos os casos, visando identificar padrões, tendências e diferenças significativas entre as abordagens.

Esta análise aprofundada dos resultados é super importante para validar a eficácia e a viabilidade de nossa implementação paralela na GPU e para fornecer informações importantes que orientarão futuros desenvolvimentos e aprimoramentos em nosso algoritmo.

5.1.2 Apresentação e análises dos resultados

Nesta subseção, destacamos os resultados das simulações realizadas com o código CNFR e o programa Cuda_NEM, utilizando três benchmarks encontrados na literatura. O objetivo principal é avaliar o desempenho e a precisão do algoritmo de varredura nodal paralelo implementado na GPU, em comparação com o algoritmo de varredura nodal sequencial já utilizado na CPU.

Apresentaremos uma análise detalhada dos resultados obtidos em ambos os ambientes de execução, buscando demonstrar que o algoritmo paralelo na GPU não compromete a precisão dos resultados em relação ao algoritmo sequencial na CPU. Além disso, examinaremos outras métricas de desempenho, como tempo de execução e eficiência computacional, para avaliar o impacto da implementação paralela na GPU sobre o tempo de processamento e a escalabilidade do algoritmo. Essa análise comparativa é fundamental para validar a eficácia e a viabilidade do novo algoritmo.

Para realização dos cálculos efetuados neste trabalho, utilizando os benchmarks anteriormente apresentados, foram utilizados os compiladores GCC 8.5 e Cuda 10.2 e o software CMake 3.2 na construção do programa Cuda_NEM e execução em Linux CentOS 8. O computador utilizado nos cálculos foi um Intel dual Xeon E5-2680 v4, com 28 cores e 56 threads, tendo 32 Gb de DDR4/2400. Por outro lado, a unidade de processamento gráfico é uma GT 1030 com 2 Gb de GDDR5, que possui 384 cuda cores e 12 unidades de processamento de dupla precisão.

A nodalização adotada nos cálculos realizados, para obtenção dos resultados aqui apresentados, é mostrada, para cada um dos benchmarks, na tabela 5.4.

Tabela 5.4: Nodalização espacial

	IAEA	LRABWR	LMWLWR
Nodos por EC	1x1	2x2	2x2
Área do Nodo	20x20 cm^2	7.5x7.5 cm^2	10x10 cm^2
Número e Tamanho da Malha das Camadas			
Refletor Inferior	1 x 20 cm	4 x 7.5 cm	2 x 10 cm
Núcleo Ativo	17 x 20 cm	40 x 7.5 cm	8 x 20 cm
Refletor Superior	1 x 20 cm	4 x 7.5 cm	2 x 10 cm

Deve ser observado que em nenhuma das referências (Vondy et al., 1975), (Kim, 1983) e (Kotchoubey, 2015) são apresentadas as tolerâncias usadas nos respectivos processos iterativos para cálculo do fator de multiplicação efetivo e da distribuição de fluxo de nêutrons. Portanto, para os cálculos aqui realizados foram adotadas as tolerâncias mostradas na tabela 5.5, onde $\epsilon_{K_{eff}}$ e ϵ_{fluxo} são, respectivamente, as tolerâncias adotadas nos critérios de convergência do fator de multiplicação efetivo e da distribuição de fluxo de nêutrons.

Tabela 5.5: Tolerâncias

	IAEA	LRABWR	LMWLWR
$\epsilon_{K_{eff}}$	1e-9	1e-8	1e-7
ϵ_{fluxo}	1e-8	1e-7	1e-6

Deve ser observado que a mesma nodalização e as mesmas tolerâncias, apresentadas nas tabelas 5.4 e 5.5, respectivamente, foram utilizadas tanto no código

CNFR quanto no programa Cuda_NEM. Além disso, os resultados obtidos com os dois programas de cálculo (CNFR e Cuda_NEM) são apresentados para todos os benchmarks tratados.

Nas figuras 5.4 e 5.5 são apresentadas as densidades de potência normalizada no EC (valores de referência e valores calculados) e os desvios percentuais relativos (Dev %) entre os valores calculados (Calc) e os valores de referência (Ref), para o IAEA benchmark. Pode-se observar que os resultados obtidos com o programa Cuda_NEM são tão precisos quanto aqueles que foram obtidos com o código CNFR, quando comparados com os valores de referência.

0.729 0.731 0.27	1.283 1.286 0.23	1.423 1.430 0.49	1.195 1.197 0.17	0.610 0.610 0.00	0.953 0.952 -0.10	0.958 0.955 -0.31	0.773 0.770 -0.39
	1.398 1.404 0.43	1.432 1.437 0.35	1.291 1.295 0.31	1.072 1.072 0.00	1.055 1.054 -0.09	0.974 0.970 -0.41	0.753 0.748 -0.66
		1.369 1.371 0.15	1.311 1.315 0.31	1.181 1.184 0.25	1.088 1.087 -0.09	0.997 0.991 -0.60	0.707 0.708 0.14
			1.179 1.182 0.25	0.972 0.972 0.00	0.923 0.920 -0.33	0.864 0.862 -0.23	
				0.475 0.475 0.00	0.699 0.693 -0.86	0.608 0.610 0.33	
Ref Calc Dev %					0.597 0.597 0.00		

Figura 5.4: Densidades de potência normalizada no EC para o IAEA obtidas com o código CNFR

0.729 0.732 0.45	1.283 1.288 0.40	1.423 1.433 0.67	1.195 1.198 0.26	0.610 0.610 0.08	0.953 0.951 -0.23	0.958 0.953 -0.51	0.773 0.768 -0.64
	1.398 1.407 0.62	1.432 1.441 0.62	1.291 1.298 0.50	1.072 1.072 0.04	1.055 1.053 -0.22	0.974 0.968 -0.60	0.753 0.747 -0.86
		1.369 1.380 0.82	1.311 1.318 0.52	1.181 1.184 0.25	1.088 1.086 -0.20	0.997 0.989 -0.83	0.707 0.706 -0.14
			1.179 1.183 0.34	0.972 0.971 -0.07	0.923 0.919 -0.43	0.864 0.861 -0.41	
				0.475 0.474 -0.13	0.699 0.692 -0.96	0.608 0.609 0.21	
Ref Calc Dev %					0.597 0.595 -0.25		

Figura 5.5: Densidades de potência normalizada no EC para o IAEA obtidas com o código Cuda_NEM

Cabe ressaltar que todos os desvios relativos percentuais (Dev %) apresentados foram calculados da seguinte forma:

$$Dev \quad \% = \left(1 - \frac{Calc}{Ref}\right) \times 100\%. \tag{5.1}$$

Nas figuras 5.6 e 5.7 são apresentadas as densidades de potência normalizada no EC (valores de referência e valores calculados) e os desvios percentuais relativos entre os valores calculados e os valores de referência, para o LRABWR benchmark. Também pode-se observar, para este benchmark, que os resultados obtidos com o

programa Cuda_NEM são tão precisos quanto aqueles que foram obtidos com o código CNFR, quando comparados com os valores de referência.

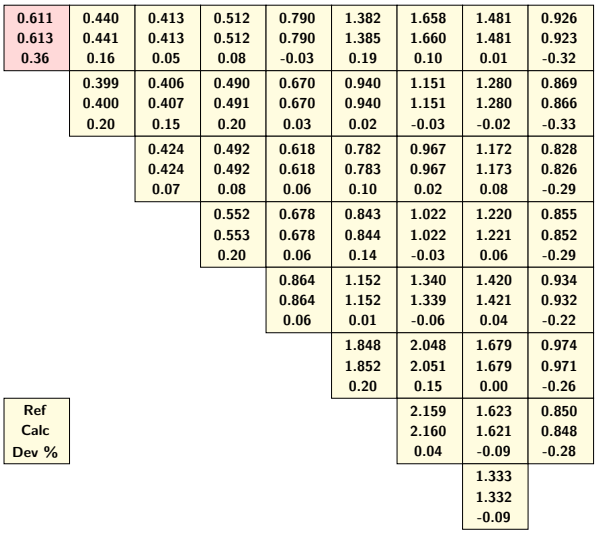

Figura 5.6: Densidades de potência normalizada no EC para o LRABWR obtidas com o código CNFR

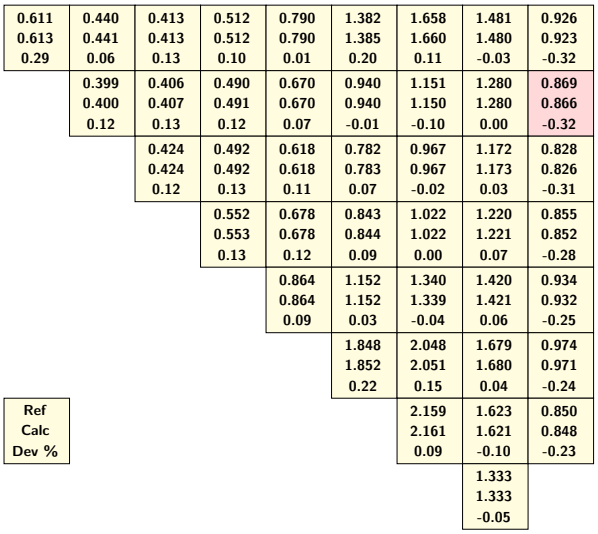

Figura 5.7: densidades de potência normalizada no EC para o LRABWR obtidas com o código Cuda_NEM

Já nas figuras 5.8 e 5.9 são apresentadas as densidades de potência normalizada no EC (valores de referência e valores calculados) e os desvios percentuais relativos entre os valores calculados e os valores de referência, para o LMWLWR benchmark. Aqui também é observado que os resultados obtidos com o programa Cuda_NEM são tão precisos quanto aqueles que foram obtidos com o código CNFR, quando comparados com os valores de referência.

Na tabela 5.6 são apresentados os fatores de multiplicação efetivo (valores de referência e valores calculados) e os desvios relativos em pcm entre os valores calculados e aqueles de referência, para cada um dos três benchmarks tratados.

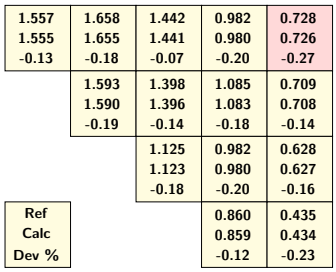

Figura 5.8: Densidades de potência normalizada no EC para o LMWLWR obtidas com o código CNFR

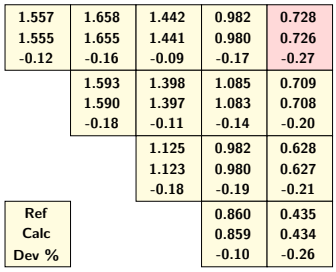

Figura 5.9: Densidades de potência normalizada no EC para o LMWLWR obtidas com o código Cuda_NEM

Tabela 5.6: Desvios relativos (em pcm) no K_{eff}.

Benchmark	Referência	Calculado			
		NEM	Desvio	Cuda_NEM	Desvio
IAEA	1.02903	1.02900	-2.0	1.02901	-1.9
LRABWR	0.99640	0.99638	-1.5	0.99638	-1.5
LMWLWR	0.99964	0.99963	-1.0	0.99961	-2.0

Observa-se dos resultados mostrados na tabela 5.6 que o programa Cuda_NEM tem praticamente a mesma precisão que fora obtida pelo código CNFR, no que diz respeito ao parâmetro integral K_{eff}. Cabe ressaltar que os desvios relativos em pcm apresentados na tabela 5.6 foram calculados da seguinte forma:

$$Desvio = \left(1 - \frac{Calc}{Ref}\right) \times 10^5. \tag{5.2}$$

Na tabela 5.7 são apresentados os desvios relativos percentuais máximos obtidos na comparação dos valores calculados com aqueles da referência, para cada um dos benchmarks tratados, usando tanto o código CNFR quanto o programa Cuda_NEM. Pode ser observado que embora os desvios obtidos com o programa Cuda_NEM não sejam exatamente iguais àqueles obtidos com o código CNFR, eles são praticamente das mesma ordem.

Tabela 5.7: Desvios relativos percentuais máximos da potência normalizada

	IAEA	LRABWR	LMWLWR
NEM	-0.86	+0.36	-0.27
Cuda_NEM	-0.96	-0.32	-0.34

Os resultados apresentados nas tabelas 5.6 e 5.7 demonstram que o algoritmo paralelo em GPU desenvolvido não altera a precisão do Método de Expansão Nodal, nem no cálculo do K_{eff} e nem no cálculo da densidade de potência normalizada no EC.

Na tabela 5.8 são apresentados o número de iterações externas e o tempo de computação gasto nos cálculos de cada um dos benchmarks aqui tratados, tanto com o código CNFR, que foi executado na CPU, quanto com o programa Cuda_NEM, que foi executado na GPU.

Tabela 5.8: Número de iterações e tempo de execução (s).

Benchmark	CPU		GPU	
	Iterações	Tempo	Iterações	Tempo
IAEA	650	1.0	1749	4.8
LRABWR	802	17.0	2178	67.0
LMWLWR	319	0.6	647	1.4

Os resultados apresentados na tabela 5.8 não traduzem a importância do algoritmo paralelo em GPU, desenvolvido para obter a solução das equações constitutivas do NEM e poder ser utilizado no processo de cálculo de otimização de modelos de recarga de combustível nuclear. Onde sabe-se que um número muito grande de avaliações de diferentes configurações de núcleos é feito durante o processo de otimização. Então, com a finalidade de mostrar que este algoritmo reduz o tempo de cálculo quando um certo número de configurações de núcleo são avaliadas simultaneamente, repetiu-se os cálculos de cada um dos três benchmarks aqui utilizados 32 vezes. Na tabela 5.9 são mostrados os tempos de cálculo gastos neste procedimento usando o código CNFR, que é executado de forma sequencial em CPU, e o programa Cuda_NEM, que por sua vez é executado em paralelo na GPU.

Tabela 5.9: Tempo de execução (s).

Benchmark	CPU	GPU
IAEA	32.0	25.9
LRABWR	544.0	373.6
LMWLWR	19.2	5.6

Como pode ser visto pelos resultados apresentados na tabela 5.9, que mesmo com uma GPU que tem apenas 12 processadores de dupla precisão, o algoritmo desenvolvido se mostra promissor quando se aumenta o número de configurações de núcleos a serem avaliadas simultaneamente, mesmo fazendo muito mais iterações, como mostrado na tabela 5.8.

No decorrer da pesquisa de toda essa pesquisa, utilizamos a GPU apresentada anteriormente, pois era GPU disponível na época, que possuía especificações modestas em comparação com a GPU mais avançada utilizada posteriormente.

Para avaliar o desempenho do algoritmo desenvolvido e evidenciar a importância do hardware na eficiência do algoritmo, realizamos testes adicionais utilizando uma GPU mais robusta a NVIDIA GeForce RTX 4080 com 9728 cuda cores, 16 Gb de GDDR6X e $\approx$ 152 unidades de processamento de dupla precisão.

A nova GPU, com capacidades de processamento superiores, demonstrou um desempenho significativamente melhor, conforme ilustrado na Tabela 5.10 e na Figura 5.10. Esses tempos de cálculo na GPU foram obtidos utilizando o Benchmark IAEA (núcleo inteiro com 4579 nodos).

Observa-se que o tempo de cálculo utilizando a nova GPU foi bastante reduzido em comparação com os resultados obtidos com a GPU anterior. Este ganho de desempenho pode ser atribuído ao maior número de núcleos CUDA, à melhor arquitetura de processamento paralelo e à maior largura de banda da memória.

Tabela 5.10: Tempos de cálculo para diferentes números de configurações

N config	**t CPU (s)**	**t GPU (s)**
1	1	1.67603
8	8	1.93212
16	16	2.44632
32	32	3.79497
64	64	9.06578
128	128	13.1823
256	256	30.9841
512	512	62.3585

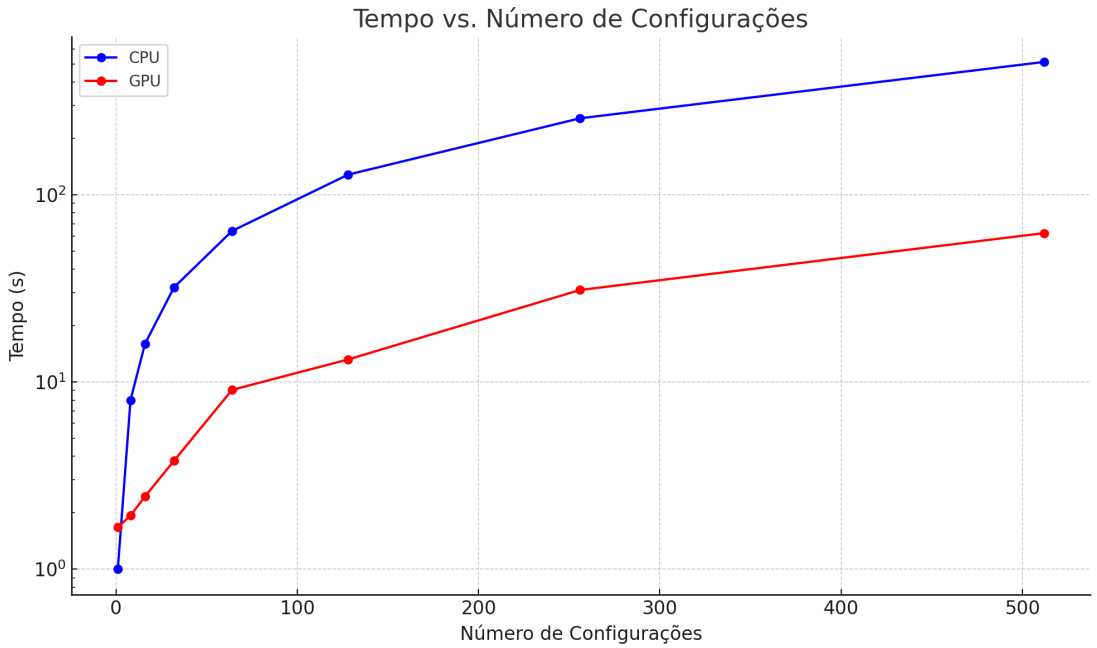

Figura 5.10: Comparação dos tempos de cálculo da CPU e GPU em relação ao número de configurações

A Figura 5.10 mostra a comparação dos tempos de cálculo da CPU e GPU em relação ao número de configurações. O gráfico apresenta os tempos de processa-

mento em segundos, com o eixo y em escala logarítmica, destacando a eficiência do processamento paralelo na GPU em comparação com a CPU.

Os resultados mostram que quanto melhor a GPU utilizada, maior é o desempenho obtido pelo algoritmo desenvolvido. Esta melhoria é particularmente notável em cenários com um grande número de configurações a serem avaliadas, onde a eficiência do processamento paralelo é imprescindível.

Portanto, a comparação entre diferentes GPUs não só valida a eficiência do algoritmo paralelo desenvolvido, mas também sublinha a importância de se utilizar hardware de ponta para obter o máximo desempenho em aplicações de computação intensiva.

A análise dos resultados obtidos com as simulações realizadas tanto com o CNFR quanto com o Cuda_NEM revelou informações significativas sobre o desempenho e a precisão do algoritmo de varredura nodal paralelo implementado na GPU. Nossas investigações detalhadas mostraram que, apesar da implementação paralela na GPU, os resultados obtidos com o Cuda_NEM são tão precisos quanto os obtidos com o CNFR, o algoritmo sequencial utilizado na CPU.

Essa constatação é de suma importância, pois valida a eficácia e a confiabilidade do nosso método, confirmando que a introdução de paralelismo na GPU não compromete a precisão dos resultados. Além disso, os resultados também demonstraram melhorias significativas em termos de tempo de execução e eficiência computacional, destacando os benefícios da implementação paralela na GPU para acelerar os cálculos nodais.

Esses resultados fortalecem a base teórica e prática do nosso trabalho, oferecendo uma sólida evidência de sua eficácia e viabilidade. Eles fornecem uma base consistente para futuras pesquisas e desenvolvimentos na área de física de reatores nucleares, abrindo portas para aplicações mais amplas e sofisticadas no campo da energia nuclear.

No próximo capítulo, apontaremos possíveis direções para trabalhos futuros, visando aprimorar ainda mais nosso método e sua aplicabilidade em outros cenários e contextos de aplicação.

Capítulo 6

Conclusões e propostas de trabalhos futuros

Neste capítulo, consolidamos os principais resultados e conclusões obtidos ao longo desta pesquisa, destacando suas implicações para a Física de Reatores e seu potencial impacto no problema de otimização de recarga de combustível nuclear. Além disso, fornecemos uma visão geral das contribuições significativas deste trabalho para a comunidade científica e apresentamos sugestões para pesquisas futuras, visando a continuidade e o aprimoramento das investigações nesta área de estudo.

Este capítulo representa o culminar de nosso esforço de pesquisa e oferece uma reflexão abrangente sobre os resultados alcançados, bem como sobre suas implicações e aplicações potenciais. Ao final deste capítulo, esperamos fornecer uma visão clara e abrangente do valor e das contribuições desta pesquisa para o avanço do conhecimento científico e tecnológico no campo da energia nuclear.

6.1 Conclusões

Neste trabalho, propomos um algoritmo para resolver as equações constitutivas do Método de Expansão Nodal (NEM) de quarta ordem com fuga transversal quadrática, implementado em paralelo no contexto da GPU. Desenvolvemos o programa de cálculo Cuda_NEM com base nesse algoritmo, utilizando a linguagem CUDA. Os resultados obtidos com o Cuda_NEM foram comparados com os do código CNFR, utilizando benchmarks como IAEA, LRABWR e LMWLWR, encontrados na literatura.

Os resultados mostram que o Cuda_NEM produz resultados comparáveis aos do CNFR, evidenciando que nosso algoritmo paralelo na GPU mantém a precisão dos cálculos. Observamos um desvio máximo de -2.0 pcm no fator de multiplicação efetivo para o benchmark LMWLWR e um desvio máximo de -0.96% na densidade de potência normalizada no EC para o benchmark IAEA. Esses resultados estão resumidos nas Tabelas 5.6 e 5.7. Além disso, observamos uma redução significativa no tempo de execução em comparação com o CNFR, especialmente ao simular várias configurações de núcleos simultaneamente, como mostrado na Tabela 5.9.

Com base nessas conclusões, afirmamos que nosso algoritmo e programa de cálculo nodal são promissores para otimização de modelos de recarga de combustível nuclear. A capacidade de realizar um grande número de avaliações de diferentes configurações de núcleos de forma eficiente e precisa é fundamental nesse contexto,

e nosso trabalho representa um avanço significativo nessa direção.

No entanto, há oportunidades para melhorias e extensões futuras deste trabalho. Sugere-se explorar maneiras de aprimorar ainda mais a eficiência do algoritmo paralelo na GPU, bem como investigar outras possíveis aplicações dessa abordagem em problemas relacionados à física de reatores nucleares.

6.2 Proposta de trabalho futuro

Como trabalho futuro, pretendemos concluir a paralelização de todos os cálculos essenciais para tratar o problema de otimização de recarga de combustível nuclear na GPU. Isso envolverá não apenas o cálculo nodal, mas também outros aspectos fundamentais do processo, visando alcançar uma otimização abrangente e eficiente de todo o sistema de cálculo neutrônico. Além disso, planejamos explorar maneiras de aprimorar ainda mais a eficiência do algoritmo paralelo na GPU, bem como investigar novas técnicas e abordagens para melhorar a precisão e o desempenho do programa de cálculo nodal.

Referências Bibliográficas

[1] NVIDIA. *CUDA C++ Best Practices Guide*, 2023.

[2] FINNEMANN, H., BENNEWITZ, F., WAGNER, M. "Interface current techniques for multidimensional reactor calculations", *Atomkernenergie*, v. 30, n. 2, pp. 123–128, 1977.

[3] DE LIMA, A. M. M., SCHIRRU, R., DA SILVA, F. C., et al. "A nuclear reactor core fuel reload optimization using artificial ant colony connective networks", *Annals of Nuclear Energy*, v. 35, 2008.

[4] SILVA, F. C., ALVIM, A. C. M., MARTINEZ, A. S. "An Alternative Nodal Expansion Method without Inner Iteration", *(PHYSOR 2010): Advances in Reactor Physics to Power the Nuclear Renaissance*, 2010.

[5] ALVIM, A. C. M., SILVA, F. C., MARTINEZ, A. S. "Improved Neutron Leakage Treatment on Nodal Expansion Method for PWR Reactors", *International Journal of Engineering and Physical Sciences*, v. 6, pp. 282–285, 2012.

[6] CHRISTENSEN, B. *Three-dimensional static and dynamic reactor calculations by the nodal expansion method.* Tese de Doutorado, RISO National Laboratory DK-4000, 1985.

[7] MIRÓ, R., GINESTAR, D., VERDÚ, G., et al. "A nodal modal method for the neutron diffusion equation. Application to BWR instabilities analysis", *Annals of Nuclear Energy*, v. 29, n. 10, pp. 1171–1194, 2002. ISSN: 0306-4549. doi: https://doi.org/10.1016/S0306-4549(01)00103-7. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0306454901001037>.

[8] MENDES, J. M. S., PAIXÃO, S. B., VELLOZO, S. D. O. "New 3D Diffusion Code Based on The Nodal Polynomial Expansion", *Brazilian Journal of Radiation Sciences*, v. 8, 2021. doi: 10.15392/bjrs.v8i3A.1489. Disponível em: <https://www.bjrs.org.br/revista/index.php/REVISTA/article/view/1489>.

[9] PAIXÃO, S. B., SILVA, F. C. "New basis functions for reactor core calculations using the nodal expansion method", *Annals of Nuclear Energy*, v. 166, pp. 108714, 2022. ISSN: 0306-4549. doi: https://doi.org/10.1016/j.anucene.2021.108714. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0306454921005909>.

[10] ZHOU, X. "Jacobian-free Newton Krylov coarse mesh finite difference algorithm based on high-order nodal expansion method for three-dimensional nuclear reactor pin-by-pin multiphysics coupled models", *Computer Physics Communications*, v. 282, pp. 108509, 2023. ISSN: 0010-4655. doi: https://doi.org/10.1016/j.cpc.2022.108509. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0010465522002284>.

[11] CHAPOT, J. L. C. *Otimização automática de recargas de reatores a água pressurizada utilizando Algoritmos Genéticos*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2000.

[12] HAJIRASSOULIHA, A., TABERNER, A. J., NASH, M. P., et al. "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms", *Signal Processing: Image Communication*, v. 68, pp. 101–119, 2018.

[13] HEIMLICH, A., MOL, A. C. A., PEREIRA, C. M. N. A. "GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation", *Progress in Nuclear Energy*, v. 53, n. 2, pp. 229–239, 2011.

[14] HEIMLICH, A., SILVA, F. C., MARTINEZ, A. S. "Parallel GPU implementation of PWR reactor burnup", *Annals of Nuclear Energy*, v. 91, pp. 135–141, 2016.

[15] HEIMLICH, A., SILVA, F. C., MARTINEZ, A. S. "Fast and accurate GPU PWR depletion calculation", *Annals of Nuclear Energy*, v. 117, pp. 165–174, 2018.

[16] FLYNN, M. "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, v. 21, n. 9, pp. 948–960, 1972.

[17] RASCHKA, S., PATTERSON, J., NOLET, C. "Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence", *Information*, v. 11, n. 4, 2020. ISSN: 2078-2489. doi: 10.3390/info11040193. Disponível em: <https://www.mdpi.com/2078-2489/11/4/193>.

[18] KIRK, D. "NVIDIA CUDA software and GPU parallel computing architecture". In: *ISMM*, v. 7, pp. 103–104, 2007.

[19] MENDES, J., HEIMLICH, A., DE LIMA, A., et al. "An algorithm for solving the equations of the Nodal Expansion Method in parallel using GPU", *Nuclear Engineering and Design*, v. 416, pp. 112751, 2024. ISSN: 0029-5493. doi: https://doi.org/10.1016/j.nucengdes.2023.112751. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0029549323006003>.

[20] HATAMI, M. *Weighted residual methods: principles, modifications and applications*. Iran, Academic Press, 2017.

[21] DA SILVA, A. C. *Simulação da multiplicação subcrítica de nêutrons através da solução da equação híbrida da difusão com fontes externas de nêutrons*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, 2011.

[22] PINHEIRO, A. L., SHIRRU, R., PEREIRA, C. M. "An optimization of a GPU-based parallel wind field module", *INAC 2017: International Nuclear Atlantic Conference*, 2017.

[23] QUARTERONI, A., VALLI, A. *Domain Decomposition Methods for Partial Differential Equations*. Oxford, Clarendon Press, 1999.

[24] GALVIS, J. "Introdução aos Métodos de Decomposição de Domínio". In: *Publicações Matemáticas*, 2009.

[25] HAGSTROM, J. N. "Computational Complexity of PERT Problems", *John Wiley & Sons*, 1988.

[26] BENAMOU, J.-D., DESPRÈS, B. "A Domain Decomposition Method for the Helmholtz Equation and Related Optimal Control Problems", *Journal of Computational Physics*, v. 136, n. 1, pp. 68–82, 1997. ISSN: 0021-9991. doi: https://doi.org/10.1006/jcph.1997.5742. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0021999197957429>.

[27] ELMAGHRBAY, M., AMMAR, R., RAJASEKARAN, S. "Fast GPU algorithms for implementing the red-black Gauss-Seidel method for Solving Partial Differential Equations". In: *2013 IEEE Symposium on Computers and Communications (ISCC)*, pp. 000269–000274, 2013. doi: 10.1109/ISCC.2013.6754958.

[28] AMDAHL, G. M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, NJ, Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California", *Solid-State Circuits Society Newsletter, IEEE*, v. 12, n. 3, pp. 19–20, 2007.

[29] ABRAHAMS, D., GURTOVOY, A. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Boston, MA, Pearson Education, 2004.

[30] BUNCH, J. R., ROSE, D. J. *Sparse matrix computations*. Michigan, Academic Press, 2014.

[31] NVIDIA. *CUDA C++ Programming Guide*, 2019.

[32] HENNESSY, J. L., PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. USA, Elsevier, 2012.

[33] KIM, C. H. "A modified Borresen's Coarse-Mesh Solution to the LRA-BWR Benchmark Problem", *Nuclear Engineering and Technology*, v. 15, n. 2, pp. 135–141, 1983.

[34] LANGENBUCH, S., MAURER, W., WERNER, W. "Coarse-mesh flux-expansion method for the analysis of space-time effects in large light water reactor cores", *Nuclear Science and Engineering*, v. 63, n. 4, pp. 437–456, 1977.

[35] VONDY, D., FOWLER, T., CUNNINGHAM, G. *VENTURE: A code block for solving multigroup neutronics problems applying the finite-difference diffusion-theory approximation to neutron transport*. Relatório técnico, Oak Ridge National Lab., Tenn.(USA), 1975.

[36] KOTCHOUBEY, J. *POLCA-T Neutron Kinetics Model Benchmarking*. Tese de Doutorado, KTH Royal Institute of Technology, 2015.

Apêndice A

Algumas partes de códigos

A.1 Justificativa para a utilização de Tabelas de Hash na programação em GPU

A rotina que realiza o cálculo da figura transversal original, escrita em Fortran, possui diversos condicionais do tipo IF, THEN, ELSE. Na programação em GPU, esses condicionais podem atrapalhar o paralelismo, tornando essas partes do código sequenciais. Para contornar essa limitação, foram utilizadas tabelas de hash (hash tables).

A necessidade de substituir os condicionais por tabelas de hash se justifica pelo elevado número de condicionais e iterações no processo. Por exemplo, apenas nessa rotina, existem 27 condicionais desse tipo. O processo iterativo, por sua vez, executa aproximadamente 200 iterações. Esse valor, multiplicado pelo número de condicionais, multiplicado pelo número de configurações do núcleo e multiplicado pelo número de gerações, como é realizado no problema de otimização de recarga, torna inviável manter esses condicionais no código.

Na versão adaptada para execução em GPU, os condicionais foram substituídos por tabelas de hash, permitindo um paralelismo mais eficiente. Isso foi essencial para garantir que o código fosse executado de maneira otimizada na GPU, evitando os gargalos que condicionais sequenciais poderiam causar. A abordagem com tabelas de hash permite acessar diretamente os resultados pré-calculados, reduzindo o tempo de execução e melhorando a eficiência do cálculo.

Essa modificação foi fundamental para viabilizar a execução do algoritmo em GPU, especialmente em cenários de alta complexidade, como no problema de otimização de recarga, onde o número de configurações do núcleo e o número de gerações tornam o processamento extremamente intensivo.

A seguir, apresentamos um trecho reduzido do código original em Fortran, destacando as linhas com os condicionais, com as demais linhas de código omitidas.

```
    subroutine FugaTransversal (Linha, Coluna, Divisao_Axial, Nodo)

    use Fluxos

    use Nodos_Vizinhos

    use Coeficientes_Fuga

    use Mapa_Nucleo       , only : Largura_X                 ,
      Largura_Y                    , &
                                    Nodos_Nucleo             ,
                   Espessura_Divisoes_Axiais

    implicit none

    integer                   :: Nodo                        ,
      Linha                      , &
                                 Nodo_M                      ,
                   Coluna                      , &
                                 Direcao                     ,
                   Divisao_Axial

    real*8                    :: Razao                       , Denominador

    real*8 , dimension(2)     :: Fuga_Media                  ,
      Fuga_Media_Vizinho          , &
                                 Fuga_Media_Nodo_M

    real*8 , dimension(2,3)   :: Fuga_Media_Nodo

    real*8 , dimension(2,3,2) :: Fuga_Face


!   ***********************************************************
!   ***                                                     ***
!   ***    Cálculo das fugas transversais médias no nodo    ***
!   ***                                                     ***
!   ***********************************************************
        ...
!   ***********************************************************
!   ***                                                     ***
!   ***    Fuga transversais à direção x, na face direita   ***
!   ***                                                     ***
!   ***********************************************************

    if (Nodo_Vizinho_Direita > 0) then

!      Cálculo da fuga transversal média no nodo vizinho da direita
!      ============================================================
         ...

!      Cálculo da fuga na interface entre nodos
```

```
!      ======================================
        ...

    else

       if (Nodo_Vizinho_Direita == 0) then

!         Para o caso de fluxo nulo contorno
!         ================================
        ...

       else

!         Para o caso de corrente de entrada nula no contorno
!         ================================================
        ...

       end if

    end if

!   ************************************************************
!   ***                                                      ***
!   ***     Fuga transversal à direção y, na face detrás     ***
!   ***                                                      ***
!   ************************************************************

    if (Nodo_Vizinho_Tras > 0) then

!      Cálculo da fuga transversal média no nodo vizinho detrás
!      ========================================================
        ...

!      Cálculo da fuga na interface entre nodos
!      ========================================
        ...

    else
        ...

!      Para o caso de fluxo nulo no contorno
!      =====================================
        ...

!      Para o caso de 1/8 de núcleo (nodos na diagonal)
!      ================================================
        ...

    end if

!   ************************************************************
!   ***                                                      ***
!   ***    Fuga transversal à direção y, na face da frente   ***
```

```
!   ***                                                          ***
!   ************************************************************

    if (Nodo_Vizinho_Frente > 0) then

!       Cálculo da fuga transversal média no nodo vizinho da frente
!       ===========================================================
         ...

!       Cálculo da fuga na interface entre nodos
!       ========================================
         ...

    else
         ...

!       Para o caso de fluxo nulo no contorno
!       =====================================
         ...
!       Para eixo passando na face do nodo
!       ==================================
         ...

!       Para eixo passando no meio do nodo
!       ==================================
         ...

    end if

!   ************************************************************
!   ***                                                      ***
!   ***    Fuga transversal à direção x, na face esquerda    ***
!   ***                                                      ***
!   ************************************************************

    if (Nodo_Vizinho_Esquerda > 0) then

!       Cálculo da fuga transversal média no nodo vizinho da esquerda
!       =============================================================
         ...

!       Cálculo da fuga na interface entre nodos
!       ========================================
         ...

    else
         ...

!       Para o caso de fluxo nulo no contorno
!       =====================================
         ...

!       Para eixo passando na face do nodo
```

```
!      ==================================
        ...

!      Para eixo passando no meio do nodo
!      ==================================
        ...

!      Para o caso de 1/8 de núcleo (nodos na diagonal)
!      ================================================
        ...

    end if

!   ************************************************************
!   ***                                                      ***
!   ***    Fuga transversal à direção z, na face debaixo     ***
!   ***                                                      ***
!   ************************************************************

    if (Nodo_Vizinho_Baixo > 0) then

!      Cálculo da fuga transversal média no nodo vizinho debaixo
!      =========================================================
        ...

!      Cálculo da fuga na interface entre nodos
!      ========================================
        ...

    else

       if (Nodo_Vizinho_Baixo == 0) then

!         Para o caso de fluxo nulo no contorno
!         =====================================
        ...

       else

!         Para o caso de corrente de entrada nula no contorno
!         ===================================================
        ...

       end if

    end if

!   ************************************************************
!   ***                                                      ***
!   ***    Fuga transversal à direção z, na face de cima     ***
!   ***                                                      ***
!   ************************************************************
```

```
    if (Nodo_Vizinho_Cima > 0) then

!      Cálculo da fuga transversal média no nodo vizinho de cima
!      ==========================================================
        ...

!      Cálculo da fuga na interface entre nodos
!      ========================================
        ...

    else

       if (Nodo_Vizinho_Cima == 0) then

!         Para o caso de fluxo nulo no contorno
!         =====================================
        ...
       else
        ...

       end if

    end if

!   ************************************************************
!   ***                                                      ***
!   ***     Coeficientes da expansão da fuga transversal     ***
!   ***                                                      ***
!   ************************************************************
        ...
!   Fim da rotina
!   *************

    end subroutine FugaTransversal
```

A.2 Rotina de Kernels da Varredura dos nodos usando a técnica *Red-Black*

Neste apêndice, apresentamos a rotina de kernels utilizada para a varredura dos nodos, implementando a técnica *Red-Black* na paralelização do algoritmo de cálculo nodal na GPU. Essa rotina serve como um exemplo de como todas as rotinas originalmente escritas em Fortran foram reescritas na linguagem CUDA para serem executadas na GPU.

Como já foi dito no capítulo 3, a técnica *Red-Black* é um método eficiente para a paralelização de cálculos em malhas, onde os nodos são divididos em dois grupos (vermelho e preto) para evitar conflitos de acesso simultâneo a dados compartilhados.

A seguir, mostramos o código da rotina de kernels, incluindo algumas linhas comentadas do código original em Fortran. Nessas linhas comentadas, aparecem os condicionais que precisaram ser modificados para a implementação em CUDA.

```
namespace nodal{

void __global__ kernel_correntes_saida_red_jmais(
  double *IJmenos, double *IJmais, const double *__restrict__ FluxNodo, const double *__restrict__ Coef_Sec,
  const double *__restrict__ Coef_Angus, const int *__restrict__ Indice_Nodo){
  for ...{
    ...
      }
    }
  }
}

void __global__ kernel_correntes_saida_red_jmenos(
  double *IJmenos, double *IJmais, const double *__restrict__ FluxNodo, const double *__restrict__ Coef_Sec,
  const double *__restrict__ Coef_Angus, const int *__restrict__ Indice_Nodo){
  for ... {
    ...
      }
    }
  }
}
void __global__ kernel_correntes_saida_black_jmais(
  double *IJmenos, double *IJmais, const double *__restrict__ FluxNodo, const double *__restrict__ Coef_Sec,
  const double *__restrict__ Coef_Angus, const int *__restrict__ Indice_Nodo){
  for ... {
    ...
      }
    }
  }
}

void __global__ kernel_correntes_saida_black_jmenos(
  double *IJmenos, double *IJmais, const double *__restrict__ FluxNodo, const double *__restrict__ Coef_Sec,
  const double *__restrict__ Coef_Angus, const int *__restrict__ Indice_Nodo){
  for ... {
    ...
      }
    }
  }
}

__device__ __inline__ void device_kernel_calc_ijin_red_black( double *IJmenos, double *IJmais,
  ...

```

```
45 // if (Nodo_Vizinho_Esquerda > 0) => Condição de continuidade
46   ...
47 // if(Nodo_Vizinho_Esquerda ==0) => Fluxo nulo no contorno
48   ...
49 // if(Nodo_Vizinho_Esquerda ==-1) => Corrente de entrada nula no
     contorno
50   ...
51 // if(Nodo_Vizinho_Esquerda ==-2) => Corrente líquida nula na face do
     nodo
52   ...
53 // if(Nodo_Vizinho_Esquerda ==-3) => Corrente líquida nula no eixo de
     simetria
54   ...
55 // if(Nodo_Vizinho_Esquerda ==-4) => Corrente líquida nula na diagonal
     (um oitavo de núcleo)
56   ...
57
58 // Para a face direita do nodo
59 // if (Nodo_Vizinho_Direita > 0)
60   ...
61 // if (Nodo_Vizinho_Direita == 0)
62   ...
63 // if (Nodo_Vizinho_Direita ==-1) => Corrente_Parcial_Jmenos
     (:,1,2,Nodo) = 0.0D+00
64   IJm[indx12] = ijm12 = if_a_eq_b<double>(-1, Nviz, 0.0 , ijm12);
65
66 // Para a face da frente do nodo
67 // if (Nodo_Vizinho_Frente > 0)
68   ...
69 // Fluxo nulo no contorno
70 // if(Nodo_Vizinho_Frente ==  0) => Corrente_Parcial_Jmais(:,2,1,Nodo)
     = - Corrente_Parcial_Jmenos(:,2,1,Nodo)
71   ...
72 // if(Nodo_Vizinho_Frente == -1) => Corrente_Parcial_Jmais(:,2,1,Nodo)
     = 0.0D+00
73   ...
74 // Corrente líquida nula na face do nodo
75 // if(Nodo_Vizinho_Frente == -2)
76   ...
77 // Corrente líquida nula no eixo de simetria
78 // if(Nodo_Vizinho_Frente == -3) => Corrente_Parcial_Jmais(:,2,1,Nodo)
     = Corrente_Parcial_Jmenos(:,2,2,Nodo)
79   ...
80
81 // Para a face detrás do nodo
82 // if (Nodo_Vizinho_Tras > 0)
83   ...
84 // Fluxo nulo no contorno
85 // if(Nodo_Vizinho_Tras == 0) => Corrente_Parcial_Jmenos(:,2,2,Nodo) =
     - Corrente_Parcial_Jmais(:,2,2,Nodo)
86   ...
87 // if(Nodo_Vizinho_Tras == -1) => Corrente_Parcial_Jmenos(:,2,2,Nodo) =
     0.0D+00
```

```
 88   ...
 89 // Corrente líquida nula na diagonal (um oitavo de núcleo)
 90 // if(Nodo_Vizinho_Tras == -4) => Corrente_Parcial_Jmenos(:,2,2,Nodo) =
      Corrente_Parcial_Jmenos(:,1,2,Nodo)
 91   ...
 92
 93 // Para a face debaixo do nodo
 94 // if (Nodo_Vizinho_Baixo > 0)
 95    ...
 96 //   nodov = (Nviz > 0) * (Nviz-1);
 97    ...
 98 // Fluxo nulo no contorno
 99 // if (Nodo_Vizinho_Baixo == 0) => Corrente_Parcial_Jmais(:,3,1,Nodo) =
      - Corrente_Parcial_Jmenos(:,3,1,Nodo)
100   ...
101 // Corrente de entrada nula no contorno
102 // if (Nodo_Vizinho_Baixo ==-1) => Corrente_Parcial_Jmais(:,3,1,Nodo) =
      0.0D+00
103   ...
104 // Para a face de cima do nodo
105 // if (Nodo_Vizinho_Cima > 0)
106   ...
107 // Fluxo nulo no contorno
108 // if (Nodo_Vizinho_Cima  == 0)
109   ...
110 // if (Nodo_Vizinho_Cima  == -1)
111   ...
112
113 __global__ void calcula_correntes_entrada_red(double *IJmenos, double
      *IJmais, const int *__restrict__ Nodo_Vizinho,
114   ...
115
116 __global__ void calcula_correntes_entrada_black(double *IJmenos, double
      *IJmais, const int *__restrict__ Nodo_Vizinho,
117   ...
118
119
120 void __global__ kernel_calc_fluxo_medio_nodal_red_black(
121   ...
122
123       // dir 1
124   ...
125       // dir 2
126   ...
127       // dir 3
128   ...
129
130 };
131
```

A implementação da técnica *Red-Black* foi importantíssima para garantir a eficiência do cálculo paralelo na GPU. A divisão dos nodos em subdomínios permite que cada subdomínio seja processado simultaneamente, evitando conflitos de acesso a dados compartilhados e maximizando o paralelismo.

Além disso, como mencionado no apêndice A.1, a presença de condicionais IF, THEN, ELSE no código original em Fortran representava um desafio para a paralelização eficiente na GPU. Esses condicionais foram substituídos por consultas a tabelas de hash, permitindo que as decisões lógicas fossem pré-calculadas e armazenadas, resultando em acesso direto e rápido durante a execução dos *kernels*.

Essa abordagem não só evitou a perda de desempenho que os condicionais sequenciais poderiam causar, mas também proporcionou uma melhoria significativa no desempenho do algoritmo, especialmente em cenários complexos e iterativos, como no problema de otimização de recarga de combustível nuclear.

Em resumo, a reescrita das rotinas de cálculo nodal na linguagem CUDA, juntamente com a utilização da técnica *Red-Black* e a substituição dos condicionais por tabelas de hash, foram passos essenciais para alcançar a eficiência necessária para a execução do algoritmo em GPUs de alta performance.