SOLUÇÃO DAS EQUAÇÕES DE DEPLEÇÃO ISOTÓPICA DO COMBUSTÍVEL NUCLEAR POR COMPUTAÇÃO DE ALTO DESEMPENHO

Adino Américo Heimlich Almeida

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: Aquilino Senra Martinez
Fernando Carvalho da Silva

Rio de Janeiro
Março de 2016

SOLUÇÃO DAS EQUAÇÕES DE DEPLEÇÃO ISOTÓPICA DO COMBUSTÍVEL NUCLEAR POR COMPUTAÇÃO DE ALTO DESEMPENHO

Adino Américo Heimlich Almeida

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

Examinada por:

____
Prof. Aquilino Senra Martinez, Ph.D.

____
Prof. Fernando Carvalho da Silva, Ph.D.

____
Prof. Bardo Ernst Josef Bodmann, Ph.D.

____
Prof. Cláudio Márcio do Nascimento Abreu Pereira, D.Sc.

____
Prof. Hermes Alves Filho, D.Sc.

____
Prof. José Antônio Carlos Canedo Medeiros, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2016

Almeida, Adino Américo Heimlich

Solução das equações de depleção isotópica do combustível nuclear por computação de alto desempenho/Adino Américo Heimlich Almeida. – Rio de Janeiro: UFRJ/COPPE, 2016.

XXII, 144 p.: il.; 29, 7cm.

Orientadores: Aquilino Senra Martinez

Fernando Carvalho da Silva

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia Nuclear, 2016.

Referências Bibliográficas: p. 106 – 115.

1. GPU. 2. OPENMP. 3. Queima e Inventário do Combustível. 4. Método de Colocação de Jacobi. 5. Série de Padé. 6. Método de Runge-Kutta-Fehlberg. I. Martinez, Aquilino Senra *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Nuclear. III. Título.

Aos meu amores, minha esposa
Luciana e meus filhos Gabriel e
Guilherme.

Agradecimentos

Agradeço a minha esposa Luciana Chiarini pela compreensão e apoio durante todos estes anos e aos meus filhos Gabriel e Guilherme.

Agradeço em especial ao meu amigos e mestres Adilson Gonçalves e Astréa Barreto pelos sábios conselhos e apoio constante.

Agradeço também aos meus orientadores e grandes mestres, os professores Aquilino Senra Martinez e Fernando Carvalho da Silva.

Agradeço aos amigos Celso Marcelo Lapa, Marcel Waintraub, Leonardo Falcão, Eugenio Marins, Maurício Aghina por todo o apoio, suporte e paciência durante todo o tempo e ao saudoso Luiz Osório de Brito Aghina pela sapiência inspiradora e pelos bons conselhos.

Agradeço ao diretor do Instituto de Engenharia Nuclear, Dr. Paulo Augusto Berquó de Sampaio, pelo apoio e liberação para condução desta tese.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico da Republica Federativa do Brasil (CNPq), ao INCT de Reatores Inovadores e a Fundação de Amparo a Pesquisa de Estado do Rio de Janeiro (FAPERJ) pelo suporte financeiro na aquisição de equipamento utilizado na tese.

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

SOLUÇÃO DAS EQUAÇÕES DE DEPLEÇÃO ISOTÓPICA DO COMBUSTÍVEL NUCLEAR POR COMPUTAÇÃO DE ALTO DESEMPENHO

Adino Américo Heimlich Almeida

Março/2016

Orientadores: Aquilino Senra Martinez
Fernando Carvalho da Silva

Programa: Engenharia Nuclear

A análise do inventário isotópico e da queima do combustível é parte fundamental no processo de simulação computacional de um reator nuclear. Esta tese apresenta três métodos computacionais, implementados em processadores gráficos GPU (*Graphic Processor Unit*), que avaliam a queima do combustível em reatores nucleares de água leve através da solução de um sistema de equações diferenciais acopladas. O objetivo desse trabalho é incorporar ao Código Nacional de Física de Reatores (CNFR) a potência computacional da GPU, empregada na solução destas equações. O cálculo da queima do combustível de um reator PWR representa a maior parte do tempo de execução do programa de simulação. Os programas de otimização do núcleo do reator requerem um grande número de simulações em sua busca. O incremento de desempenho ao se utilizar o processamento em GPU reduz significativamente o tempo de cada simulação permitindo encontrar melhores candidatos para o projeto do núcleo. Os resultados deste estudo mostram que, na avaliação de um ciclo de queima e inventário em um reator PWR típico, um dos três métodos é duzentas vezes mais rápido do que o método empregado anteriormente no CNFR, dentro de 1% de exatidão.

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

ISOTOPIC NUCLEAR FUEL DEPLETION SOLUTION USING HIGH PERFORMANCE COMPUTACIONAL ENVIRONMENT

Adino Américo Heimlich Almeida

March/2016

Advisors: Aquilino Senra Martinez
Fernando Carvalho da Silva

Department: Nuclear Engineering

Isotopic inventory and fuel burnup analysis is a fundamental part in a nuclear reactor simulation. This thesis presents three computational methods, implemented in graphics processors (GPU), which evaluate the fuel burnup in light water nuclear reactors solving a system of coupled differential equations. The objective of this work is to incorporate to the National Code of Reactor Physics (CNFR) the computational power of the GPU used in the solution of these equations. The calculation of fuel burnup in a PWR reactor uses most of the simulation runtime. Reactor core optimization programs require a large number of simulations in the search for a solution. Performance enhancement using the GPU processing significantly reduces the time of each simulation making it possible to find the better candidates for the core design. The results of this study show that in the evaluation of a burnup and inventory cycle, in a typical PWR reactor, one of these three methods is two hundred times faster than the method previously used in CNFR, within 1 % accuracy.

Sumário

Lista de Figuras **xii**

Lista de Tabelas **xviii**

Lista de Abreviaturas **xix**

1 Introdução **1**

2 Revisão da literatura **9**

2.1 Equações de Bateman 11

2.2 As cadeias de actinídeos 13

2.2.1 Cadeias do U 13

2.2.2 Cadeias do Pu 15

2.2.3 Cadeias do Np, Am e Cm 16

2.3 Os produtos de fissão 17

2.3.1 Cadeia do Mo^{95} 17

2.3.2 Cadeia do Gd^{155} 18

2.3.3 Cadeia do Pr^{143} 19

2.3.4 Cadeia do Rh^{103} 19

2.3.5 Cadeia do Rh^{105} 20

2.3.6 Cadeia do Xe^{131} 21

2.3.7 Cadeia do Xe^{135} 21

2.3.8 Cadeia do Sm 22

3 Conceitos matemáticos **24**

3.3 Erro numérico da aritmética em ponto flutuante 26

4 Métodos numéricos **29**

4.1 Matriz exponencial e série de Taylor 29

4.2 Métodos de Runge-Kutta 31

4.2.1 Runge-Kutta-Fehlberg 35

4.3 Métodos de colocação 38

4.3.1 Polinômios ortonormais 39

4.4.1 Interpolação de Lagrange e quadratura de Gauss 42

4.4.2 Método de colocação de Gauss-Jacobi 44

4.5 Método de aproximação por série de Padé 48

5 Computação de alto desempenho **52**

5.1 A taxionomia de Flynn 55

5.2 A lei de Amdahl 58

5.3 Matrizes esparsas e densas 59

5.3.1 Formato BLOCK-DENSE 60

5.3.2 Formato COO (*Coordinated List Format*) 61

5.3.3 Formato CSR (*Compressed Sparse Row*) 62

5.3.4 Multiplicação matriz Vetor 64

5.3.5 Multiplicação matriz Vetor - CSR 64

5.4 Outros formatos 65

5.5 Construção do operador de depleção por soma direta 67

5.6 OpenMP 70

5.6.1 Exemplos 72

5.7 Computação de alto desempenho em GPU 74

5.7.1 *Processing units* 75

5.7.2 A arquitetura SIMT 76

5.7.3 As *Stream Machine eXtended* 76

5.7.4 Estrutura de memória 77
5.7.5 Sincronização e barreiras 80
5.7.6 Extensões ao C/C++ do CUDA 81
5.7.7 *Grids*, blocos, *threads* e indexadores 82
5.7.8 Warps 85
5.7.9 Coalescência e Conflitos no Acesso a memória Global 86
5.7.10 Operações com Vetores 87
5.7.11 Operações com matrizes 91
5.7.12 Computação heterogênea 93
5.7.13 Polimorfismo, herança e funções virtuais 94

6 Resultados, conclusão e proposta de trabalhos futuros **97**
6.1 Analise do desvio na avaliação da sequência de queima 98
6.2 Análise de performance 101
6.3 Conclusões 103
6.4 Proposta de trabalhos futuros 104

Referências Bibliográficas **106**

A Resultados **116**

B Implementação e códigos **132**
B.1 Ambiente de compilação CMake 132
B.2 Programa de teste em FORTRAN 134
B.3 Programa de interface com o FORTRAN 138
B.4 FORTRAN e C *Bindings* 138
B.5 Construção da matriz de depleção 140

C Construção dos operadores de queima **142**
C.1 Operadores de queima 142
C.2 RKF 143
C.3 Método de colocação de Gauss-Jacobi 143

C.4 Método diagonal Padé 144

Lista de Figuras

2.1 Cadeia do Samário . . . 10

2.2 Cadeia de decaimento e transmutação dos actinídeos . . . 14

2.3 Fissão térmica do U^{235} . . . 15

2.4 Fissão térmica do Pu^{238} . . . 16

2.5 Cadeia do Mo^{95} . . . 17

2.6 Cadeia do Gd^{155} . . . 18

2.7 Cadeia do Pr^{143} . . . 19

2.8 Cadeia do Rh^{103} . . . 20

2.9 Cadeia do Rh^{105} . . . 20

2.10 Cadeia do Xe^{131} . . . 21

2.11 Cadeia do Xe^{135} . . . 22

2.12 Cadeia de decaimento do Samário. . . . 23

3.1 Número em ponto flutuante - 32 bits. . . . 27

3.2 Número em ponto flutuante - 64 bits. . . . 28

4.1 *Butcher Tableau* . . . 34

4.2 *Butcher Tableau* do método explícito . . . 35

4.3 Runge-Kutta-Fehlberg *Tableau*. . . . 35

4.4 Runge-Kutta-Fehlberg Algorithm . . . 37

4.5 Encontrar a matriz de transição para a expansão polinomial de Jacobi . . 47

4.6 Operação de escala . . . 49

4.7 Encontra a aproximação diagonal de Padé . . . 50

4.8 Aproximação de Padé. 50

5.1 Taxionomia SISD 56
5.2 Taxionomia SIMD 57
5.3 Taxionomia MISD 57
5.4 Taxionomia MIMD 58
5.5 Matriz exemplo. 60
5.6 Exemplo de matriz densa em blocos. 61
5.7 figure 61
5.8 Multiplicação de matriz por vetor no formato COO. 64
5.9 Multiplicação de matriz por vetor no formato COO em paralelo. 64
5.10 Multiplicação de matriz por vetor no formato CSR 65
5.11 Multiplicação de matriz por vetor no formato CSR em paralelo 65
5.12 Matriz do CNFR - 1 nodo 68
5.13 Matriz do CNFR - 8 nodos 69
5.14 Distribuição de tarefas em OpenMP 71
5.15 OPENMP - Exemplo 1 72
5.16 OPENMP - Exemplo 2 73
5.17 OpenMP, Exemplo 03 73
5.18 *GPU Kepler* 74
5.19 *Stream Processor* 75
5.20 *SMX* 77
5.21 Declaração de variável do tipo *SHARED* 78
5.22 Acesso a memória de texturas 79
5.23 Falha de indexação na compilação. 79
5.24 Alocação de memória da GPU 80
5.25 Alocação de memória *pinned* 80
5.26 Declaração de grid 82
5.27 Grade e blocos de *threads*. 83
5.28 Array uni-dimensional de blocos 1D. 83

5.29 Array uni-dimensional de blocos 2D. 84
5.30 Array uni-dimensional de blocos 3D. 84
5.31 Array bi-dimensional de blocos 1D. 84
5.32 Array bi-dimensional de blocos 2D. 84
5.33 Array bi-dimensional de blocos 3D. 85
5.34 Declaração de função global no CUDA. 85
5.35 Diagrama de Coalescência 86
5.36 Soma de vetores em **R** usando precisão simples. 87
5.37 Soma de vetores em **Z** 88
5.38 Soma de vetores com *Stride* 88
5.39 Soma de vetores usando a memória **SHARED** 88
5.40 Soma de vetores usando a memória **SHARED** e **Double Buffering** . . . 89
5.41 Soma de vetores genérica usando a memória *SHARED* e *Double Buffering* 90
5.42 Soma de vetores Genérica em C++ STL e CUDA 90
5.43 Produto matriz Vetor 92
5.44 Produto matriz-vetor com vetorização 92
5.45 Produto matriz Vetor Vetorizado 93
5.46 Definição da classe Base no arquivo *base.hpp* em C++ 94
5.47 Programa de teste do polimorfismo. 94
5.48 Definição da classe Base com função virtual. 95
5.49 Diagrama de bloco 95

6.1 Desvio *RMSE* (%) *vs* Queima. 99
6.2 Desvio RMSE (%) *vs* método. 101

A.1 Concentrações dos Isótopos de Urânio *vs* Tempo de Queima. 116
A.2 Desvio na avaliação do U^{234}. 117
A.3 Desvio na avaliação do U^{235}. 117
A.4 Desvio na avaliação do U^{236}. 117
A.5 Desvio na avaliação do U^{238}. 118

A.6 Concentrações dos Isotopos de Netúnio *vs* Tempo de Queima. 118
A.7 Desvio na avaliação do Np^{237}. 118
A.8 Desvio na avaliação do Np^{239}. 119
A.9 Concentrações dos Isótopos de Plutônio *vs* Tempo de Queima. 119
A.10 Desvio na avaliação do Pu^{238}. 119
A.11 Desvio na avaliação do Pu^{239}. 119
A.12 Desvio na avaliação do Pu^{240}. 120
A.13 Desvio na avaliação do Pu^{241}. 120
A.14 Desvio na avaliação do Pu^{242}. 120
A.15 Concentrações dos Isótopos de Amerício *vs* Tempo de Queima. 121
A.16 Desvio na avaliação do Am^{241}. 121
A.17 Desvio na avaliação do Am^{242}. 121
A.18 Desvio na avaliação do Am^{242m}. 122
A.19 Desvio na avaliação do Am^{243}. 122
A.20 Concentrações dos Isotopos Cúrio *vs* Tempo de Queima. 122
A.21 Desvio na avaliação do Cm^{242}. 122
A.22 Desvio na avaliação do Cm^{244}. 123
A.23 Concentrações dos Isotopos Zr^{95}, Mo^{95}, Nb^{95} e Ru^{103} *vs* Tempo de Queima. 123
A.24 Desvio na avaliação do Zr^{95}. 123
A.25 Desvio na avaliação do Mo^{95}. 124
A.26 Desvio na avaliação do Nb^{95}. 124
A.27 Desvio na avaliação do Ru^{103}. 124
A.28 Concentrações dos Isotopos de Ródio *vs* Tempo de Queima. 124
A.29 Desvio na avaliação do Rh^{103}. 125
A.30 Desvio na avaliação do Rh^{105}. 125
A.31 Concentrações dos Isotopos de Iodo e Xenônio *vs* Tempo de Queima. . . 125
A.32 Desvio na avaliação do I^{131}. 126
A.33 Desvio na avaliação do Xe^{131}. 126
A.34 Desvio na avaliação do I^{135}. 126

A.35 Desvio na avaliação do Xe^{135}. 126

A.36 Concentrações dos Isotopos Pr^{143} e Nd^{143} *vs* Tempo de Queima. 127

A.37 Desvio na avaliação do Pr^{143}. 127

A.38 Desvio na avaliação do Nd^{143}. 127

A.39 Concentrações dos Isotopos de Promécio *vs* Tempo de Queima. 128

A.40 Desvio na avaliação do Pm^{147}. 128

A.41 Desvio na avaliação do Pm^{148m}. 129

A.42 Desvio na avaliação do Pm^{148}. 129

A.43 Concentrações dos Isotopos de Samário *vs* Tempo de Queima. 129

A.44 Desvio na avaliação do Sm^{147}. 130

A.45 Desvio na avaliação do Pm^{149}. 130

A.46 Desvio na avaliação do Sm^{149}. 130

A.47 Concentrações dos Isotopos Eu^{155} e Gd^{155} *vs* Tempo de Queima. 130

A.48 Desvio na avaliação do Eu^{155}. 131

A.49 Desvio na avaliação do Gd^{155}. 131

B.1 Arquivo CMakeLists.txt - Principal 133

B.2 Arquivo CMakeLists.txt - Configuração do MKL 133

B.3 Arquivo CMakeLists.txt - Configuração do CUDA 134

B.4 Fluxograma do programa de avaliação da queima. 135

B.5 Inicio do programa *deplet* 135

B.6 Alocação de Variáveis de programa e memória 136

B.7 Inicialização do programa 136

B.8 Laço de cálculo dos passos de queima 137

B.9 Interface entre o FORTRAN e o C++ 138

B.10 Interface com o FORTRAN via C *Bindings* 139

B.11 Interface C de acesso ao C++ 139

B.12 Potências da matriz de depleção 141

C.1 Construção da matriz de depleção 142

C.2 Construção do operador RKF 143
C.3 Construção do operador de colocação de Gauss-Jacobi 144
C.4 Construção do operador diagonal de Padé 144

Lista de Tabelas

5.1 Vetores da matriz COO 62

5.2 Vetores da matriz CSR 63

6.1 Erro percentual *RMSE* dos isótopos Am, Cm, U, Pu e Np. 99

6.2 Erro percentual *RMSE* das cadeias Mo, Rh, Pm e Sm. 100

6.3 Erro percentual *RMSE* das cadeias Nd, Gd, Xe. 100

6.4 Tempo dispendido em segundos *vs* Método *vs* Passo de queima. 102

6.5 Tempo total dispendido (segundos) para um ciclo de queima de 498 dias. . 102

Lista de Abreviaturas

AMD8350	Processador do fabricante AMD, p. 27
AMD	Fabricante de Microprocessadores, p. 53
BCSR	Formato de matriz esparsa em blocos CSR, p. 59
C++	Linguagem de programação orientada a objeto, p. 7
CAD	*Computer Aided Design*, p. 105
CASCADE	*Siemens 3D code system for PWR nuclear core design and safety analysis*, p. 2
CMAKE	Sistema *open-source* de construção de programas, p. 7
CNEN	Comissão Nacional de Energia Nuclear, p. 2
CNFR	Código Nacional de Física de Reatores, p. 2
COO	Formato de matriz esparsa, p. 59
COPPE	Instituto Alberto Luiz Coimbra de Pós-graduação e Pesquisa de Engenharia, p. 2
CPU	*Computer Processor Unit*, p. 53
CRAM	Chebychev Rational Approximation Method, p. 104
CSG	*Constructive Solid Geometry*, p. 105
CSR	Formato de matriz esparsa, p. 59

CUDA	*Compute Unified Device Architecture*, p. 6
DIA	Formato de matriz esparsa, p. 59
EDO	Equações Diferenciais Ordinárias, p. 97
EPRI	*Electric Power Research Institute*, p. 107
FMA	*Fused Multiply-Add*, p. 76
FORTRAN	***FOR**mula **TRAN**slation*, p. 7
FPGA	*Field Processors Gate Array*, p. 55
GEDAM	Sistema de Geração de Dados Nucleares, p. 2
GEDAR	Sistema de Geração de Dados do Reator, p. 2
GFLOP	*Giga Floatint Point Operations per Second*, p. 106
GPU	*Graphic Processor Unit*, p. 4
GPU	*Graphics Processor Unit*, p. 53
IAEA	*International Atomic Energy Agency*, p. 16
IEEE-754	Norma de ponto flutuante IEEE, p. 26
IEEE	*Institute of Electrical and Electronics Engineers*, p. 26
INTEL-I7	Processador do fabricante INTEL, p. 27
INTEL	Fabricante de Microprocessadores, p. 53
JAGUAR	Supercomputador construído pela Cray em ORNL, p. 52
LD/ST	*Load/Store Units*, p. 76
MIC	*Many Integrated Core Architecture*, p. 53
MIMD	*Multiple Instruction Multiple Data*, p. 53

MIMD	*Multiple Instruction, Multiple Data streams*, p. 56
MISD	*Multiple Instruction, Single Data stream*, p. 56
MKL	Intel math kernel library, p. 62
MOC	*Method of Characteristics*, p. 55
MPI	*Message Passing Interface*, p. 54
MPact	*Michigan parallel characteristics transport code*, p. 106
NUMA	*Non Uniform Memory Access*, p. 80
NVIDIA	Fabricante de Processadores Gráficos, p. 53
OPENMP	*Open MultiProcessing*, p. 6
ORIGEN2	*Oak Ridge Isotope Generator 2*, p. 31
ORNL	Oak Ridge National Laboratory, p. 52
PWR	*Pressurized Water Reactor*, p. 1
SAV	*Siemens code system for PWR nuclear core design*, p. 2
SFU	*Special Function Units*, p. 76
SIMD / SIMT	*Single Instruction, Multiple Data Streams*, p. 56
SIMD	*Single Instruction Multiple Data*, p. 53
SIMT	*Single Instruction Multiple Thread*, p. 76
SIMULATE	Studsvik's next generation nodal code, p. 2
SISD	*Single Instruction, Single Data Stream*, p. 55
SMX	*Stream Machine eXtended*, p. 76
SM	*Streaming Multiprocessors*, p. 75

SP *Streaming Processors*, p. 75

SRME *Square Root Mean Error*, p. 28

STL *Standard Template Library*, p. 93

SpMv *Sparse Matrix-Vector Multiplication*, p. 37

TITAN Supercomputador construído pela Cray em ORNL, p. 52

UMA *Unified Memory Access*, p. 80

cuBLAS *NVIDIA CUDA Basic Linear Algebra library*, p. 53

cuFFT *NVIDIA CUDA Fast Fourier Transform library*, p. 53

cuSparse *NVIDIA CUDA Sparse Matrix library*, p. 53

Capítulo 1

Introdução

Os reatores nucleares térmicos de água leve pressurizada (PWR) tipicamente devem substituir um terço do combustível gasto por combustível novo entre 12 e 18 meses de operação, e a duração deste ciclo é dependente de vários fatores, tais como: o nível do enriquecimento do U^{235}, o percentual de veneno queimável no núcleo e as concentrações isotópicas do elementos presentes nas pastilhas de combustível.

O *burnup* (WAGNER (2001)), ou queima, do combustível é definido como Mega watt dia por quilograma de Urânio, como algo em torno de 45 $\frac{MWd}{kg}U$ no caso de um enriquecimento do combustível próximo de 4% para os reatores PWR. Maiores tempos de queima requerem maior enriquecimento do combustível (LANG (1991)) e obtê-lo representa um fator relevante em termos econômicos e tecnológicos. O *burnup* e o acompanhamento do inventário compreendem as alterações de composição isotópica provocadas pela fissão do U^{235} e também por outros processos tais como a absorção de nêutrons e emissão de radiações ou partículas por ativação.

A previsão confiável do comportamento operacional do reator ao longo do ciclo de queima do combustível é o fator preponderante na confiabilidade da simulação de possíveis projetos de núcleo. Estes projetos são utilizados por avançados sistemas de busca visando a otimização do custo operacional, do custo do combustível e das estratégias de recarga (TURINSKY *et al.* (2005)).

A confiabilidade do cálculo destes projetos de núcleo depende fundamentalmente da qualidade dos parâmetros neutrônicos e termohidráulicos, parâmetros estes, que depen-

dem substancialmente da exatidão proporcionada por uma adequada geração dos dados nucleares dos materiais que compõe o núcleo do reator, pelo cálculo tridimensional da densidade de potência e pela obtenção de coeficientes integrais medidos *in sito* durante a operação do reator.

A simulação do comportamento físico e nuclear do reator é realizada utilizando simuladores: como o CNFR (ALVIM *et al.* (2006)), o SAV (BURTAK *et al.* (1996)), o CASCADE (GRUMMER *et al.* (2000)) e o SIMULATE (BAHADIR e LINDAHL (2009)).

O código CNFR é um programa de simulação da física de nêutrons em reatores PWR e seu nome é um sigla para *Código Nacional de Física de Reatores*. Esse código foi totalmente desenvolvido no Brasil ao longo dos últimos anos pelo Programa de Engenharia Nuclear da COPPE/ UFRJ. O CNFR é composto por dois sistemas: o GEDAN (Sistema de Geração de Dados Nucleares) e GEDAR (Sistema de Geração de Dados do Reator). No sistema GEDAN efetuam-se a geração:

- Das seções de choque microscópicas e macroscópicas homogeneizadas para o elemento combustível.
- Dos fatores de descontinuidade e funções forma, para cada tipo de elemento combustível.

O sistema GEDAR é a ferramenta computacional que simula:

- Os fluxos de nêutrons térmicos e rápidos nos pinos do combustível.
- A distribuição de densidade de potência nos pinos do combustível.
- A variação espacial da queima nos elementos combustíveis.
- As concentrações isotópicas nos elementos combustíveis.
- A realimentação termo hidráulica dependente da densidade e temperatura do moderador e do combustível.

- A pesquisa de criticalidade, em função da concentração de Boro, em função da potência, e para a posição dos bancos de barras de controle.

Além disso, o CNFR efetua o histórico das simulações de projeto do núcleo, o reinício da simulação do núcleo do reator e o decaimento radioativo na piscina de combustível usado.

O sistema GEDAR é constituído pelos seguintes módulos:

- Aquisição dos dados nucleares.
- Gerência.
- Interpolação na queima.
- Interpolação das variáveis de estado.
- Fluxo de nêutrons.
- Pesquisa de criticalidade.
- Distribuição de potência.
- Realimentação termo hidráulica.
- Reconstrução pino a pino.
- Concentrações isotópicas.
- Queima.
- Histórico dos elementos combustíveis.
- Edição de resultados.

A sua modularidade torna possível acoplar novas rotinas e métodos capazes de efetuar cada tarefa envolvida na simulação do reator. Como mencionado anteriormente, a confiabilidade da simulação depende substancialmente do cálculo da queima e das concentrações isotópicas do combustível ao longo do ciclo.

No processo de simulação, os coeficientes das seções de choque microscópicas e macroscópicas homogeneizadas para cada tipo de elemento combustível são fornecidas pelo sistema GEDAN ao sistema GEDAR, no qual é feita a nodalização. Este procedimento efetua a partição geométrica do núcleo em blocos tridimensionais denominados *nodos* de maneira interativa com o usuário através de uma interface gráfica do módulo de **Gerência**.

O módulo **Interpolação na queima** efetua a interpolação dos fatores de forma e descontinuidade e das seções de choque para os valores específicos médios da queima nas faces e cantos dos nodos do reator. O próximo passo interpola as seções de choque macroscópicas e microscópicas para os valores específicos associados às variáveis de estado: temperatura do combustível, temperatura do moderador, densidade do moderador, concentração de Boro e Xenônio e fluência rápida.

A partir dos dados geométricos dos nodos e das seções de choque macroscópicas e microscópicas produzidos anteriormente, calcula-se o fluxo de nêutrons resolvendo a equação de difusão de nêutrons em dois grupos de energia utilizando o método de expansão nodal NEM (FINNEMANN *et al.* (1977)). Além disso o sistema GEDAR efetua a pesquisa de criticalidade, calcula a distribuição de potência e a realimentação termohidráulica.

Tem-se portanto que a qualidade da avaliação das concentrações isotópicas no núcleo tem impacto fundamental em toda a simulação efetuada pelo sistema GEDAR. As concentrações isotópicas dos nuclídeos no combustível, actinídeos e produtos de fissão, são calculadas no módulo **Concentrações isotópicas** do sistema GEDAR, onde são resolvidas as equações de decaimento e transmutação para cada nodo do reator e para cada passo de queima e realimentam o procedimento, uma vez que as concentrações calculadas são utilizadas no módulo **Interpolação na queima**. Isto significa que mesmo pequenas variações no cálculo das concentrações isotópicas podem ter grande influência na simulação como um todo.

Em um trabalho anterior (PRATA *et al.* (2013)) foi descrito que o cálculo da queima do combustível é o que mais consome recursos computacionais no sistema GEDAR, podendo representar até 70 % do tempo de simulação em cada passo de queima.

Nesse contexto, a contribuição original desta tese é o desenvolvimento de uma solução computacional inovadora baseada em processamento massivamente paralelo utilizando processadores denominados GPU (*Graphic Processor Unit*) para o cálculo das concentrações isotópicas no combustível. O método de Runge-Kutta-Fehlberg (CHENEY e KINCAID (2012)) implementado neste trabalho apresentou um desempenho duzentas vezes superior ao método original utilizado no sistema GEDAR com exatidão próxima de 1 %.

As novas tecnologias de processamento paralelo, baseadas na arquitetura da GPU e aliadas à modularidade e flexibilidade do código CNFR, ampliam substancialmente o desempenho computacional do código. Esse fato em conjunto com a qualidade dos resultados proporcionada pelo CNFR o torna uma poderosa ferramenta à ser empregada no cálculo da recarga do combustível nuclear.

A recarga do núcleo do reator nuclear constitui um problema de otimização combinatória e exige extensos recursos computacionais e tempo para ser resolvido. Neste caso a otimização do núcleo é realizada através de algorítimos baseados em heurística populacional (PEREIRA e LAPA (2003)), social (DE OLIVEIRA e SCHIRRU (2011)), evolução diferencial (DE MOURA MENESES e SCHIRRU (2015)) e outros.

Os programas de simulação da recarga e projeto do núcleo do reator baseados em heurística (KROPACZEK e TURINSKY (1991)), requerem grandes populações de indivíduos (DO NASCIMENTO ABREU PEREIRA *et al.* (1999)), onde cada individuo desta população corresponde a um projeto de núcleo.

Os objetivos básicos da otimização de um projeto de núcleo são:

- Maximizar a densidade de potência;
- Maximizar o tempo de queima do combustível;
- Minimizar o custo da eletricidade.

A otimização está sujeita a várias condições de contorno:

- Manter o reator crítico;

- Manter os coeficientes de criticalidade negativos;
- Manter a reatividade dentro de condições para um desligamento seguro;
- Não exceder a temperatura máxima no combustível e da vareta;
- Manter a potência estável.

Com tantas restrições e com a necessidade de otimizar o reator mantendo as dimensões do núcleo e as características do projeto, o que é feito é rearranjar o posicionamento de cada elemento combustível do reator e testar cada projeto avaliando o desempenho de cada um.

A busca do projeto ótimo pode ser orientada através de seleção natural, depreciação do menos apto, ou outro aspecto de avaliação da qualidade do projeto, em função da potência ou nível de Boro. Isto significa que dezenas ou centenas de cópias de um programa de cálculo neutrônico teriam de ser executados a cada geração da avaliação o que torna a busca trivialmente paralelizável. Contudo pode-se encontrar ótimos locais que não necessariamente sejam o ótimo global, com implicações econômicas importantes. No caso de uma busca intensiva, mais tempo de computação seria necessário para varrer uma parcela maior do espaço de busca e deve-se ressaltar que o tempo de execução desta tarefa também é uma restrição, pois cada dia de interrupção na produção de energia da usina de Angra II representa um custo de aproximadamente R$ 12 milhões.

Com o ganho computacional obtido através das propostas nesta tese pode-se reduzir o tempo de execução da função objetivo, e ampliar o tempo de busca por ótimos locais, possibilitando assim melhores escolhas de projeto de núcleo.

Como mencionado anteriormente, esta é a principal motivação e contribuição deste trabalho, ou seja, o desenvolvimento e implementação de um módulo computacional para o sistema GEDAR que simule a depleção do combustível em um reator do tipo PWR e possua alto desempenho baseado em computação paralela executada em GPU, apresentando a exatidão necessária ao cálculo.

Com este intuito, neste trabalho foram desenvolvidos três métodos numéricos que simulam o comportamento da depleção do combustível no núcleo de um reator de água

leve do tipo PWR, escritos em C++ e utilizando paralelismo em CPU, via OPENMP (CHANDRA (2001)), e GPU utilizando a tecnologia CUDA (NVIDIA (2008b)). Estes métodos são encapsulados em uma biblioteca e serão acoplados como um novo módulo de queima do combustível no sistema GEDAR. Os três métodos foram avaliados em relação à seu desempenho computacional e exatidão na obtenção das concentrações isotópicas e seus resultados confrontados com o método de cálculo de queima padrão do sistema GEDAR.

Devido a rigidez (*stiffness*) (GARFINKEL *et al.* (1977)) do sistema de equações diferenciais e aos requisitos de desempenho adotou-se o método de Runge-Kutta-Fehlberg (CHENEY e KINCAID (2012)). Além deste, foram desenvolvidos outros dois métodos baseados na solução deste sistema de equações diferenciais utilizando exponencial de matrizes: O primeiro, denominado método de colocação de Gauss Jacobi (CHIHARA (2011)) e o outro uma expansão em série de Padé (HIGHAM (2005)), diagonal e com mecanismo de *scaling square*.

Os métodos são encapsulados em uma biblioteca através do ambiente CMAKE (MARTIN e HOFFMAN (2010)) e podem ser acessados por outras linguagens, FORTRAN por exemplo, via C++ *bindings* (METCALF *et al.* (2004)).

O sistema de quações diferenciais sob análise é configurável para qualquer número de nodos, passos de tempo ou cadeias, mas para fins de comparação com as concentrações obtidas através do código CNFR, utilizou-se uma cadeia composta por 17 actinídeos sofrendo realimentação por reações $(n,2n)$ e transmutação por decaimento α e β_{-} e 20 produtos de fissão.

O capítulo 2 aborda a revisão bibliográfica do ponto de vista físico e matemático através da construção do sistema de equações diferenciais acopladas de primeira ordem correspondentes alterações das concentrações dos actinídeos e produtos de fissão. Além disso, a construção dedutiva através das equações de balanço entre produção e desaparecimento do nuclídeo sob análise e uma discussão sobre as aproximações utilizadas no sistema de equações de forma a tratar a dependência mútua entre fluxo de nêutrons e concentração isotópica.

No capítulo são abordados alguns conceitos matemáticos utilizados nesta tese.

No capítulo 4 são discutidos o modelo de armazenamento em memória da matriz de depleção, sua construção e os métodos numéricos propostos para a solução do sistema de equações diferenciais.

No capítulo 5 são descritos os métodos de computação de alto desempenho e programação paralela empregados nesta tese. Pois, como visto por Moler e VanLoan (MOLER e VAN LOAN (1978)) podemos abstrair a solução do problema em pelo menos dezenove maneiras, cada qual com suas próprias restrições de desempenho e precisão.

No capítulo 6.3 são apresentados os resultados das simulações dos três métodos de cálculo desenvolvidos nesta tese e empregados na solução do sistema de equações diferenciais acopladas de primeira ordem que caracterizam a queima do combustível no reator nuclear. Além da conclusão e as propostas de trabalhos futuros.

No apêndice A pode-se observar os gráficos das concentrações dos nuclídeos analisados neste trabalho, bem como o desvio das concentrações em função do método de calculo empregado.

No apêndice B são descritos a implementação do programa de teste dos algoritmos, o método de compilação, a interligação entre as linguagens C++ e FORTRAN.

No apêndice C são descritos os operadores de queima desenvolvidos em linguagem C++.

Capítulo 2

Revisão da literatura

Os reatores PWR do tipo Areva AP utilizam um combustível com enriquecimento em torno de 4% para o U^{235} e seu período de recarga é de dezoito meses. O combustível novo contém uma pequena parcela de U^{234} que é fértil e absorve nêutrons e os outros quase 96% de U^{238} também absorve nêutrons e transmuta em elementos denominados transurânicos ou actinídeos menores. Estes possuem radio-toxicidade e meia-vida tão altos que devem ser confinados por milhares de anos em depósitos de alto custo ou reciclados. Os novos reatores de IV geração (ABRAM e ION (2008)) poderão utilizar estes elementos como combustível em um ciclo fechado (SERP *et al.* (2014)).

Podemos destacar a produção de Pu^{239} que é físsil na faixa térmica e pode aumentar a criticalidade do reator e o Np^{237} produzido após duas capturas sem fissão do U^{235}.

Por outro lado, a fissão do U^{235} e de outros actinídeos, produz algo em torno de 300 fragmentos de fissão, e complexas cadeias de decaimento e captura de nêutrons que possuem mais de 3000 nuclídeos. Estes elementos radio-tóxicos e voláteis como o I^{131}, ou ainda forte absorvedores de nêutrons como o Samário ou Xenônio são contidos na vareta do combustível e devem ser contabilizados no inventário (BURSTALL (1979)). Podemos observar por exemplo na figura 2.1 a cadeia radioativa do *Samário*.

Alguns destes elementos possuem uma grande seção de choque de absorção na faixa térmica do fluxo de nêutrons e podem afetar significativamente o fluxo espacial e o fator de multiplicação do núcleo, tornando-o subcrítico. Como o incremento de sua concentração é proporcional ao fluxo de nêutrons vemos que estes atuam como uma realimentação

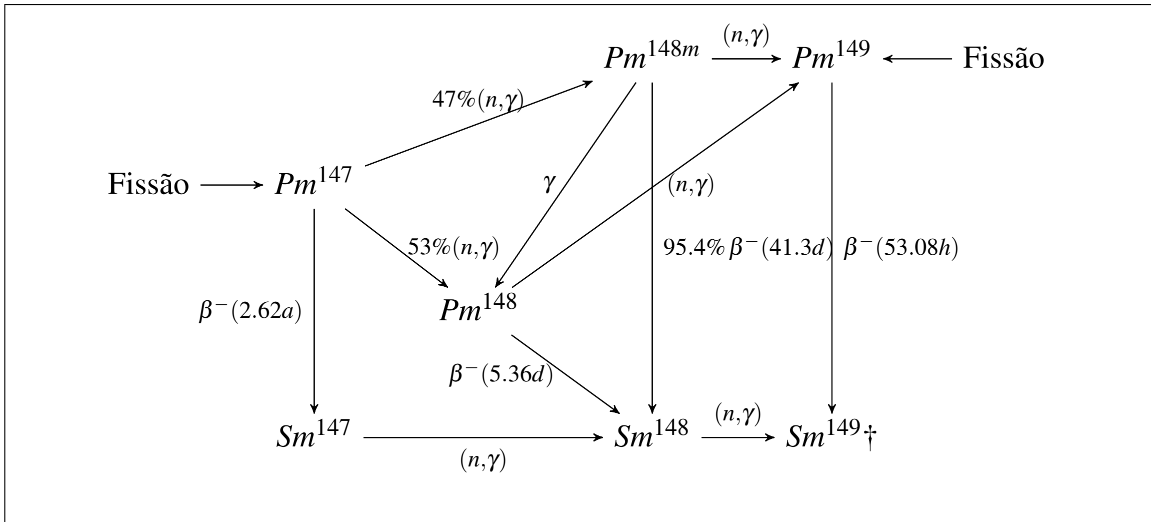

Figura 2.1: Cadeia do Samário

negativa na região do reator onde o fluxo de nêutrons é maior. O fato fundamental na análise e simulação da depleção do combustível do reator, é que o comportamento das concentrações isotópicas do combustível afeta não apenas o inventário e o calor residual, mas também a distribuição de potência em cada elemento combustível, e consequentemente o projeto da recarga do reator.

A queima do combustível em um reator nuclear, ou *burnup* pode ser calculada através da equação (2.1)

$$N_{\Delta T} = \frac{P\Delta T}{N_0}, \tag{2.1}$$

onde $N_{\Delta T}$, com unidade MWd/Kg representa o *burnup* de combustível, P é a potência térmica do reator nuclear, ΔT representa o passo de tempo da operação e N_0 representa a massa inicial de combustível na carga do núcleo. As características e o comportamento temporal dos nuclídeos presentes no combustível nuclear são descritas por equações de balanço entre os fatores de produção e de decréscimos dos nuclídeos, produzidos por decaimento, captura radioativa, fissão e outras reações. A partir deste balanço obtemos equações diferenciais que descrevem o comportamento das concentrações dos nuclídeos presentes no combustível em função do fluxo de nêutrons, temperatura do combustível, temperatura do moderador, concentração do Xenônio e Boro.

As concentrações de determinados isótopos, como por exemplo o Xenônio, influenciam diretamente o comportamento do fluxo de nêutrons no combustível (CHEN (1990)). Este balanço depende das cadeias de decaimento radioativo, das seções de choque microscópicas características de cada reação e das constantes de decaimento, no caso dos elementos instáveis.

Observa-se que o projeto do núcleo de um reator nuclear deve se ater aos seguintes objetivos:

- Maximizar a densidade de potência.
- Maximizar o tempo de queima do combustível.
- Minimizar o custo da eletricidade.

Além disso, o projeto deve se submeter a várias condições de segurança, tais como:

- Manter o reator crítico.
- O projeto deve possuir coeficientes negativos de reatividade.
- A reatividade deve ser suficiente para uma parada segura do reator.
- A temperatura do núcleo não pode exceder a temperatura máxima do *cladding*.
- A potência deve se manter estável e não pode oscilar (HUSARCEK (2004)).

2.1 Equações de Bateman

Os fenômenos de decaimento e transmutação que ocorrem no combustível do reator são descritos por uma equação de balanço para cada nuclídeo envolvido nesta análise, onde são levados em consideração sua produção por fissão, captura e decaimento radioativo em função do grupo de energia e da posição espacial. Chamam-se de equações de Bateman, ao sistema de equações diferenciais ordinárias acopladas, que descreve o comportamento das concentrações ao longo do tempo dos nuclídeos, em cadeias de decaimento ou de evolução linear.

No caso dos elementos denominados actinídeos ou transurânicos o balanço de perda e ganhos de cada um dependem basicamente das seções de choque de captura radiativa.

$$
\begin{aligned}
\frac{dN_k^x}{dt}(t) = & \sum_{i=1,i\neq k}^{\text{Actinídeos}} N_i^x(t) \sum_{g=1}^{G} \left(\sigma_{\gamma,i}^{g,x}(t) - \sigma_{f,i}^{g,x}(t) + \sigma_{(n,2n),i}^{g,x}(t) \right) \phi_x^g(t) \\
& - \sum_{y=1}^{\text{Frações de decaimento}} \lambda_k N_k^x(t) + \sum_{z\neq i}^{\text{Frações de Produção}} \lambda_z N_z^x(t) \qquad (2.2)
\end{aligned}
$$

onde

$\phi_x^g(t)$ representa o fluxo de nêutrons, na posição espacial x e no grupo de energia g.

$N_k^x(t)$ representa a concentração do actinídeo k, na posição espacial x e no tempo t.

$\sigma_{f,i}^{g,x}(t)$ é a seção de choque de fissão do elemento i para o grupo de energia g, na posição espacial x e no tempo t.

$\sigma_{\gamma,i}^{g,x}(t)$ é a seção de choque de captura do actinídeo i para o grupo de energia g, na posição espacial x e no tempo t.

$\sigma_{(n,2n),i}^{g,x}(t)$ é a seção de choque para a reação $(n,2n)$ no actinídeo i para o grupo de energia g, na posição espacial x e no tempo t.

λ_k é a constante de decaimento do nuclídeo k.

λ_z é a constante de decaimento do nuclídeo z.

Além destes temos os produtos de fissão,

$$
\begin{aligned}
\frac{dN_l^x}{dt}(t) = & \sum_{i=1,i\neq l}^{\text{Produtos de Fissão}} \left(N_i^x(t) \sum_{g=1}^{G} \left(\Gamma_{i,l}^{g} \sigma_{f,i}^{g,x}(t) - \sigma_{\gamma,i}^{g,x}(t) \right) \phi_x^g(t) \right) \\
& - \lambda_l N_l^x(t) \qquad (2.3)
\end{aligned}
$$

onde

$N_l^x(t)$ representa a concentração do nuclídeo l, na posição espacial x e no tempo t.

$\sigma_{f,i}^{g,x}(t)$ é a seção de choque de fissão do elemento i para o grupo de energia g, na posição espacial x e no tempo t.

$\Gamma^{g}_{i,l}$ é o rendimento, ou *fission yield*, do nuclídeo l pela fissão do nuclídeo i para o grupo de energia g.

λ_l é a constante de decaimento do nuclídeo l.

$\sigma^{g,x}_{\gamma,i}(t)$ é a seção de choque de captura do actinídeo i para o grupo de energia g, na posição espacial x e no tempo t.

Ao unirmos os sistemas de equações (2.2) e (2.3) obtemos um sistema de equações diferenciais acopladas de primeira ordem descritas por Bateman em um trabalho clássico (BATEMAN (1910)).

Assim, é fundamental para a simulação do núcleo do reator que estas concentrações sejam determinadas com exatidão. E para tanto, será feita uma breve descrição das cadeias dos actinídeos e produtos de fissão utilizadas neste trabalho.

2.2 As cadeias de actinídeos

Nesta seção será descrita a cadeia de actinídeos analisada nesta tese, além é claro de uma breve interpretação do significado das relações entre as seções de choque total e de captura ou absorção que tem um papel fundamental. Podemos observar na figura 2.2 as cadeias de decaimento e transmutação abordadas neste trabalho.

As seções de choque de captura e fissão homogêneas no nodos, que são utilizadas na simulação, são produzidas pelo CNFR após cada passo de queima e são dependentes de diversos fatores, tais como a temperatura do combustível, a temperatura do moderador, a concentração de Boro e a concentração de Xenônio.

2.2.1 Cadeias do U

Nos reatores térmicos de água leve o principal elemento físsil é o U^{235}, que na natureza é encontrado em uma concentração próxima de 0.7% em relação ao U^{238}. Nos reatores PWR torna-se necessário um processo de enriquecimento para elevar a concentração percentual de 0.7% a 3.5%, e da mesma forma o isótopo U^{234} também tem sua concentração aumentada. O U^{234} é fértil, captura um nêutron e transmuta em U^{235}, daí

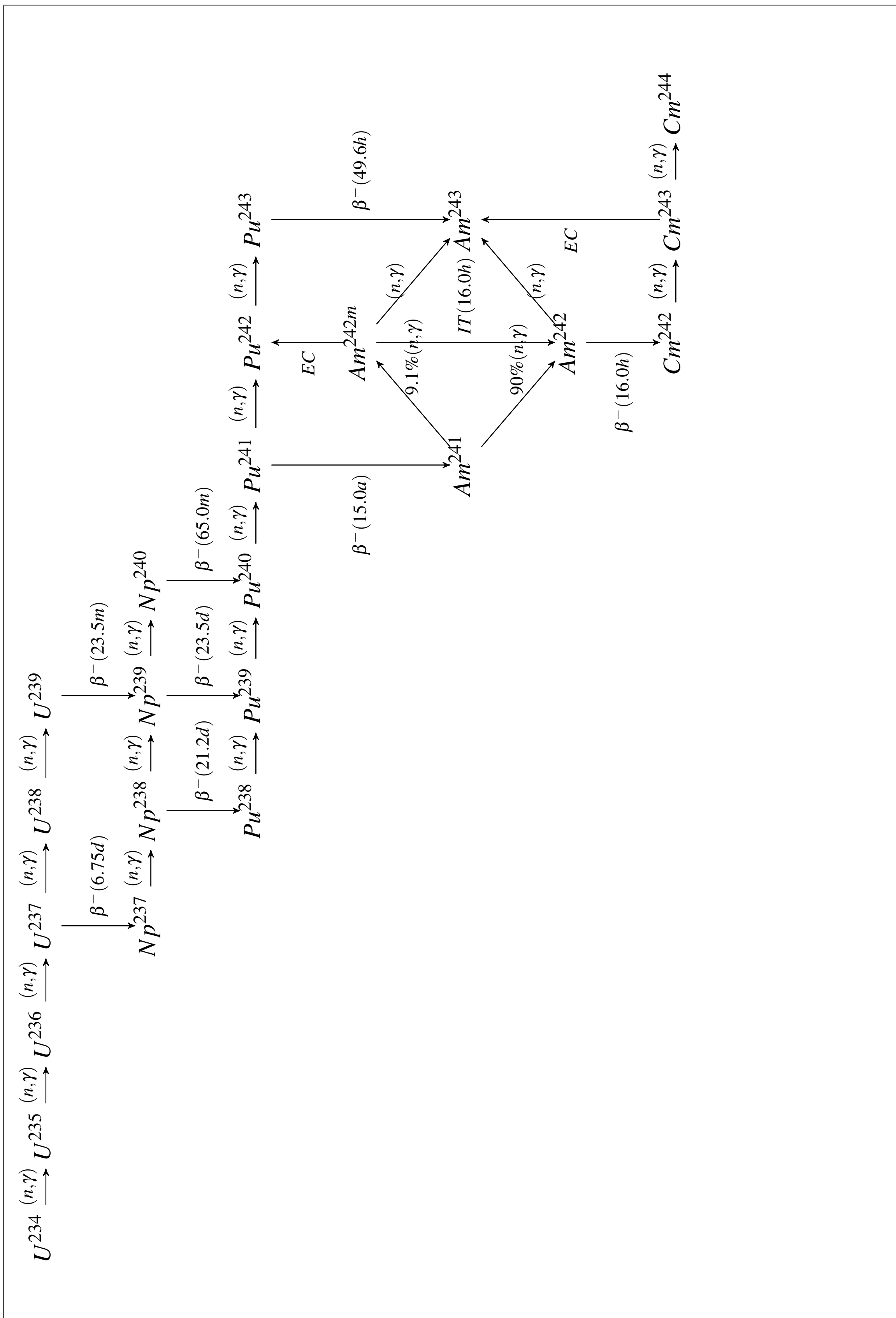

Figura 2.2: Cadeia de decaimento e transmutação dos actinídeos

a importância em contabiliza-lo. Na figura 2.3 podemos observar a distribuição dos produtos de fissão do U^{235} relativa ao número atômico e foi publicada do documento *Los Alamos Laboratory report LA-UR-94-3106* (ENGLAND e RIDER (1994)).

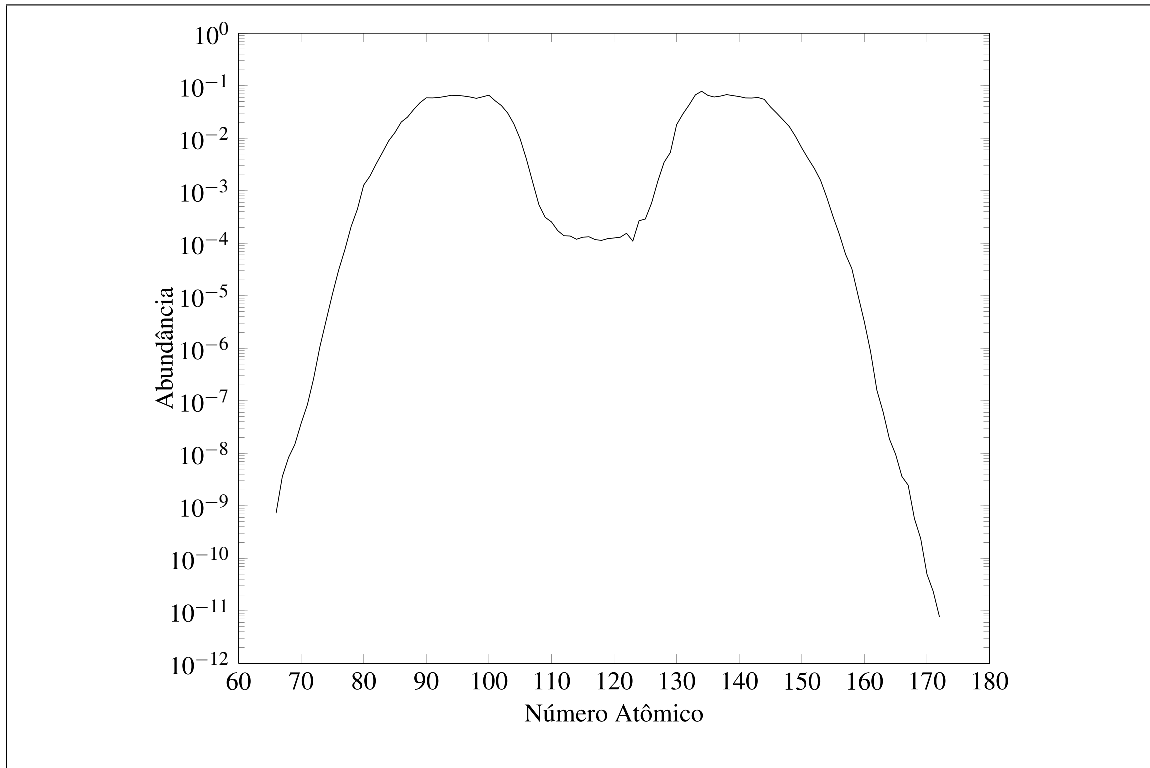

Figura 2.3: Fissão térmica do U^{235}

2.2.2 Cadeias do *Pu*

Observa-se que todos os isótopos do plutônio são físseis ou férteis, mas por outro lado, o Pu^{242} precisa absorver 3 nêutrons antes de transmutar em Cm^{245} que é físsil. Nos reatores térmicos de água leve, algo em torno de 1% do combustível queimado é constituído por plutônio, apresentando uma distribuição isotópica de 52% para o Pu^{239}, 24% para o Pu^{240}, 15% Pu^{241}, 6% para o Pu^{242} e 2% para o Pu^{238} para o combustível substituído na primeira recarga (WORRALL (2009)).

É interessante ressaltar que em combustíveis MOX (INSULANDER BJOERK e FHAGER (2009)) temos um grande percentual de Plutônio e este tem uma curva de distribuição dos produtos de fissão ligeiramente diferente do U^{235} como podemos observar na figura 2.4.

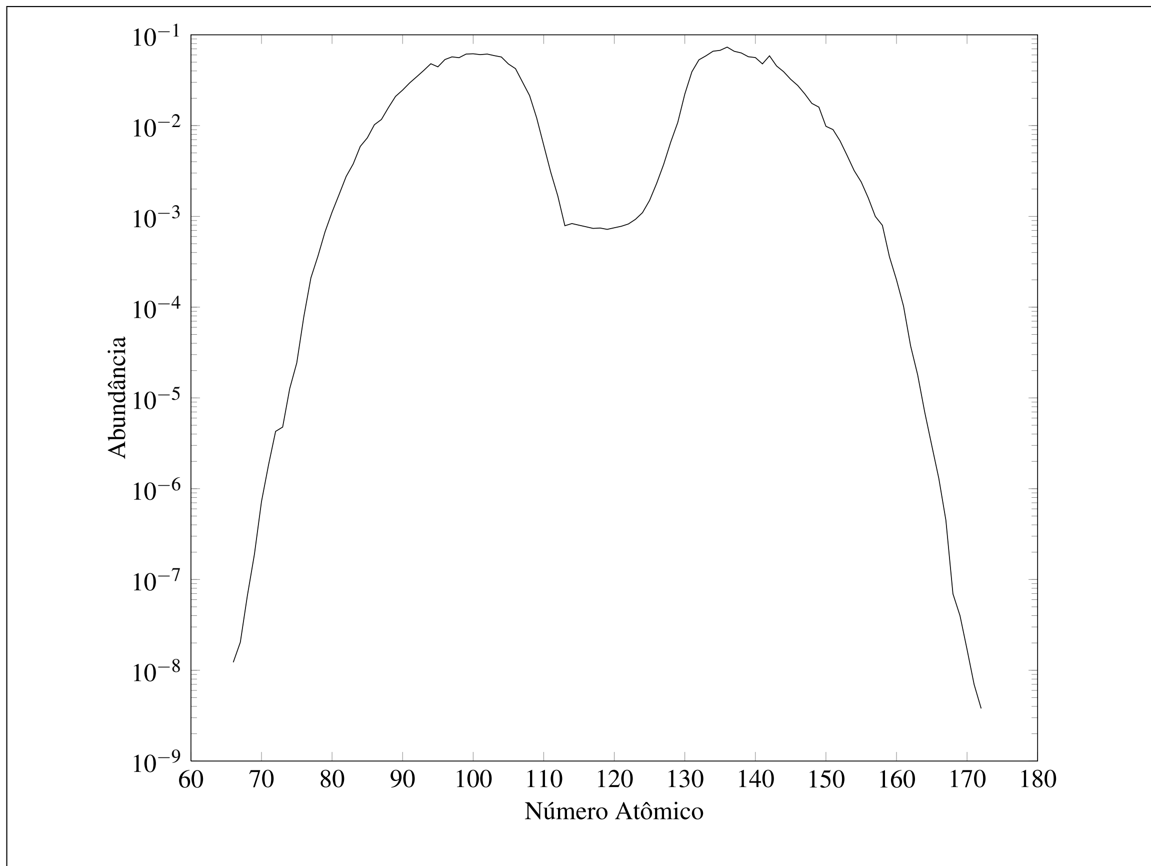

Figura 2.4: Fissão térmica do Pu^{238}

2.2.3 Cadeias do Np, Am e Cm

Os isótopos do elemento Netúnio, cujo simbolo é Np, são criados no combustível do reator por captura de nêutrons pelo Urânio, sem que no entanto ocorra fissão. A necessidade do seu inventário é importante, pois se relaciona com a produção dos elementos Plutônio Pu, Americio Am e Cúrio Cm. Este elementos apresentam alta radio-toxicidade com meia-vida de até centenas de milhares de anos. Seu inventário é requisito dos protocolos de não proliferação de armas e salvaguardas segundo a agência internacional de energia atômica IAEA (NICHOLS *et al.* (2008)).

O inventário estes elementos também é importante na avaliação do projeto do núcleo do reator em virtude de recarga (BAYS *et al.* (2009)) (BAYS *et al.* (2010)), e suas utilização como combustível em futuros reatores de 4ª geração (SALVATORES (2005)).

2.3 Os produtos de fissão

Os produtos de fissão, ou *fission yields*, em um reator nuclear são obtidos pela fissão dos átomos dos elementos físseis presentes no combustível. Podemos dividir os produtos de fissão em em duas classes, os saturados e não-saturados. Chamam-se produtos de fissão saturados aqueles que possuem grandes seções de choque de absorção, de tal forma que após um período de tempo as suas respectivas taxas de produção e decréscimo estão equilibradas e é atingida então uma concentração de equilíbrio, como por exemplo a cadeia do Xe^{135}. Por outro lado, os produtos de fissão não-saturados, possuem seções de choque de absorção relativamente pequenas de modo que podem alcançar a saturação em um intervalo de tempo maior ou nunca.

Segue então uma pequena introdução das características dos produtos de fissão empregados neste trabalho. O símbolo † indica que o nuclídeo é considerado estável. Observa-se que temos um total de 17 actinídeos sob avaliação e o rendimento da fissão de cada um deles tem de ser contabilizado em função de suas seções de choque de fissão e do fluxo de nêutrons nos dois grupos de energia.

2.3.1 Cadeia do Mo^{95}

A cadeia do Molibdênio é avaliada a partir do decaimento do Zircônio Zr^{95}, este decai por β_{-} e sua meia vida é de 64,032 dias. Na figura 2.5 podemos observar que a partir da produção do Zr^{95} pela fissão do U^{235} através de uma série de decaimentos obtemos o elemento Mo^{95}.

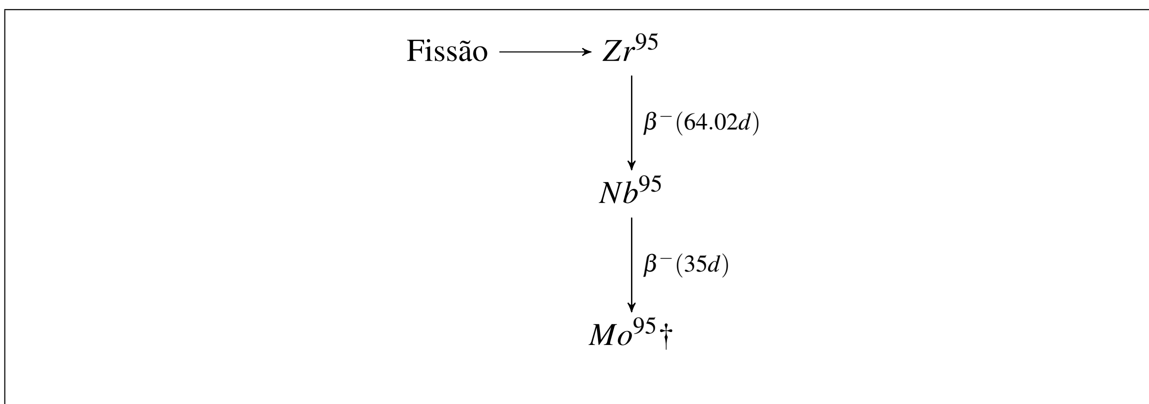

Figura 2.5: Cadeia de decaimento do Mo^{95}.

A cadeia da figura 2.5 é expressa pelo sistema de equações (2.4), onde $\Gamma_i = \sum_{g=1}^{2} \left(\sigma_{f,i}^{g} \phi^{g}\right) N^{i}(t)$ e $\gamma_i^{Zr^{95}}$ é a contribuição por fissão do actinídeo i para a produção do nuclídeo Zr^{95} em dois grupos de energia.

$$\begin{aligned}
\frac{d}{dt} N^{Zr^{95}}(t) &= \sum_{i=1}^{17} \gamma_i^{Zr^{95}} \Gamma_i - \lambda^{Zr^{95}} N^{Zr^{95}} \\
\frac{d}{dt} N^{Nb^{95}}(t) &= \lambda^{Zr^{95}} N^{Zr^{95}} - \lambda^{Nb^{95}} N^{Nb^{95}} \\
\frac{d}{dt} N^{Mo^{95}}(t) &= \lambda^{Nb^{95}} N^{Nb^{95}} - \sum_{g=1}^{2} \left(\sigma_{\gamma,g}^{Mo^{95}} N^{Mo^{95}}(t)\right) \phi^{g}
\end{aligned} \tag{2.4}$$

2.3.2 Cadeia do Gd^{155}

O nuclídeo Eu^{155}, Európio, é um elemento instável, que decai para o Gadolínio por β_{-} com meia vida de 4.7611 anos. Na figura 2.6 observa-se sua cadeia de decaimento. Em alguns projetos, o uso do Gadolínio como veneno queimável absorvendo nêutrons, permite a compensação da reatividade e o ajuste da distribuição de potência no núcleo do reator, e assim possibilita uma alternativa a utilização do Boro, podendo inclusive aumentar o tempo de queima do combustível. Neste caso a pastilha do combustível é constituída por uma mistura de óxido de urânio e óxido de gadolínio $(U\ Gd)O^{2}$ onde o Gadolínio representa até 10 % da massa total da pastilha.

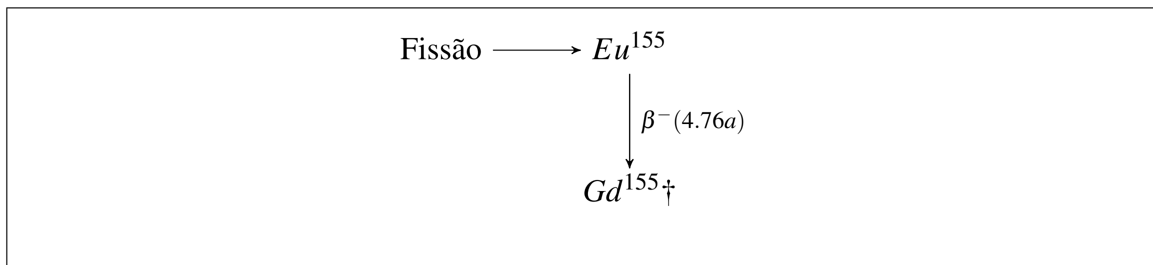

Figura 2.6: Cadeia de decaimento do Gd^{155}.

A cadeia da figura 2.6 é expressa pelo sistema de equações (2.5), onde $\Gamma_i = \sum_{g=1}^{2} \left(\sigma_{f,i}^{g} \phi^{g}\right) N^{i}(t)$ e $\gamma_i^{Eu^{155}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo Eu^{155} em dois grupos de energia.

$$\frac{d}{dt}N^{Eu^{155}}(t) = \sum_{i=1}^{17}\gamma_i^{Eu^{155}}\Gamma_i - \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{Eu^{155}}N^{Eu^{155}}(t)\right)\phi^g - \lambda_{Eu^{155}}N^{Eu^{155}}(t)$$
$$\frac{d}{dt}N^{Gd^{155}}(t) = \lambda^{Eu^{155}}N^{Eu^{155}} - \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{Gd^{155}}N^{Gd^{155}}(t)\right)\phi^g \tag{2.5}$$

2.3.3 Cadeia do Pr^{143}

O nuclídeo Pr^{143}, Praseodímio, é um elemento instável, que decai em β_- com meia vida de 13.57 dias para o elemento Neodímio Nd^{143}. Na figura 2.7 observa-se sua cadeia de decaimento.

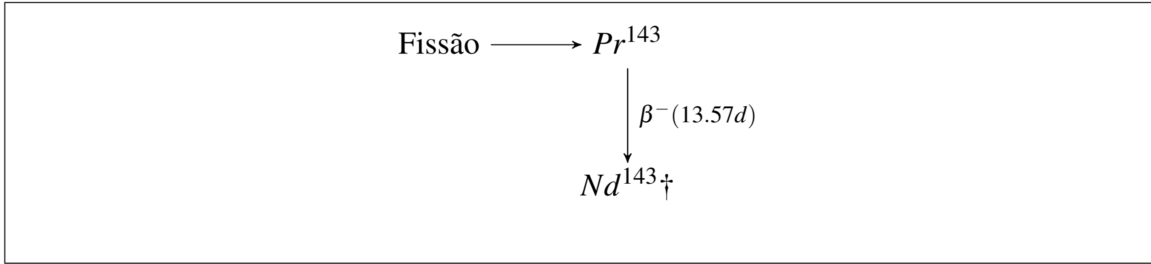

Figura 2.7: Cadeia de decaimento do Pr^{143}.

A figura 2.7 leva ao sistema representado pelo sistema de equações (2.6), onde $\Gamma_i = \sum_{g=1}^{2}\left(\sigma_{f,i}^{g}\phi^g\right)N^i(t)$ e $\gamma_i^{Pr^{143}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo Pr^{143} em dois grupos de energia.

$$\frac{d}{dt}N^{Pr^{143}}(t) = \sum_{i=1}^{17}\gamma_i^{Pr^{143}}\Gamma_i - \lambda_{Pr^{143}}N^{Pr^{143}}(t)$$
$$\frac{d}{dt}N^{Nd^{143}}(t) = \lambda_{Pr^{143}}N^{Pr^{143}}(t) - \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{Nd^{143}}\phi^g\right)N^{Nd^{143}}(t) \tag{2.6}$$

2.3.4 Cadeia do Rh^{103}

O nuclídeo Ru^{103}, Rutênio, é um elemento instável, que decai em β_- com meia vida de 39.26 dias para o Ródio Rh^{103}. Na figura 2.8 observa-se sua cadeia de decaimento, esta figura leva ao sistema representado pelo sistema de equações (2.7), onde $\Gamma_i = \sum_{g=1}^{2}\left(\sigma_{f,i}^{g}\phi^g\right)N^i(t)$ e $\gamma_i^{Ru^{103}}$ é a contribuição por fissão do nuclídeo i para a produção

do nuclídeo Ru^{103} em dois grupos de energia.

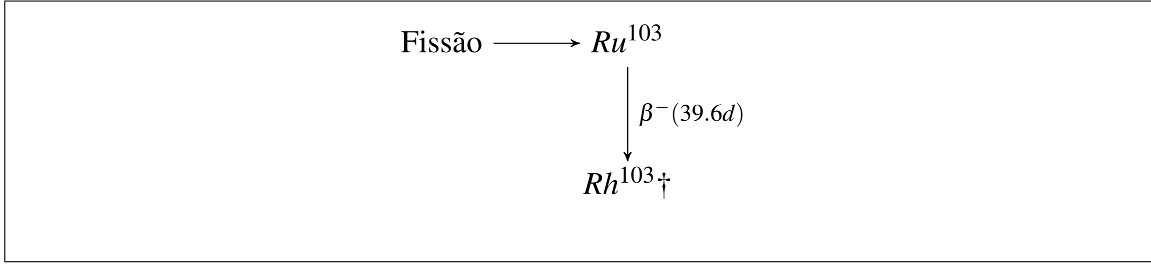

Figura 2.8: Cadeia de decaimento do Rh^{103}.

$$\begin{aligned}
\frac{d}{dt}N^{Ru^{103}}(t) &= \sum_{i=1}^{17} \gamma_i^{Ru^{103}} \Gamma_i - \lambda_{Ru^{103}} N^{Ru^{103}}(t) \\
\frac{d}{dt}N^{Rh^{103}}(t) &= \lambda_{Ru^{103}} N^{Ru^{103}}(t) - \sum_{g=1}^{2} \left(\sigma_{\gamma,g}^{Rh^{103}} \phi^g\right) N^{Rh^{103}}(t)
\end{aligned} \tag{2.7}$$

2.3.5 Cadeia do Rh^{105}

O nuclídeo Rh^{105}, Ródio, é um elemento instável, que decai em β_- com meia vida de 35.36 horas para o Paládio Pd^{105}. Na figura 2.9 observa-se sua cadeia de decaimento e esta figura leva ao sistema representado pela equação (2.8), onde $\Gamma_i = \sum_{g=1}^{2} \left(\sigma_{f,i}^{g} \phi^g\right) N^i(t)$ e $\gamma_i^{Ru^{105}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo Ru^{105} em dois grupos de energia.

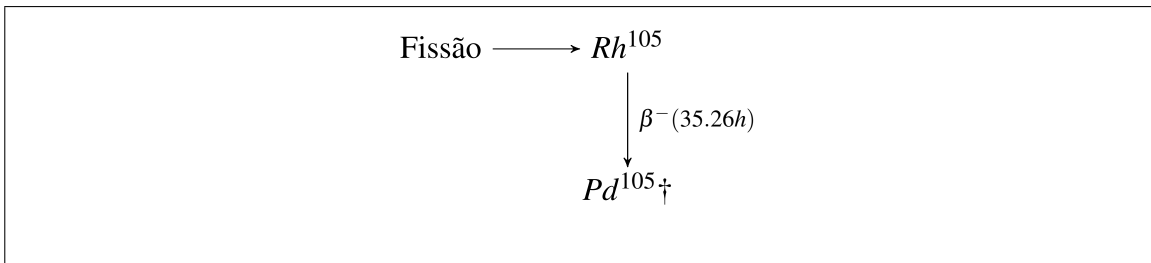

Figura 2.9: Cadeia de decaimento do Rh^{105}.

$$\frac{d}{dt}N^{Ru^{105}}(t) = \sum_{i=1}^{17} \gamma_i^{Ru^{105}} \Gamma_i - \sum_{g=1}^{2} \sigma_{\gamma,g}^{Rh^{105}} \phi^g N^{Rh^{105}}(t) \tag{2.8}$$

2.3.6 Cadeia do Xe^{131}

O nuclídeo I^{131}, Iodo, é um elemento instável, que decai em β_{-} com meia vida de 8.0252 dias para o Xenônio Xe^{131}. Na figura 2.10 observa-se sua cadeia de decaimento, esta figura leva ao sistema representado pelo sistema de equações (2.9), onde $\Gamma_i = \sum_{g=1}^{2}\left(\sigma_{f,i}^{g}\phi^{g}\right)N^{i}(t)$ e $\gamma_i^{I^{131}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo I^{131} em dois grupos de energia.

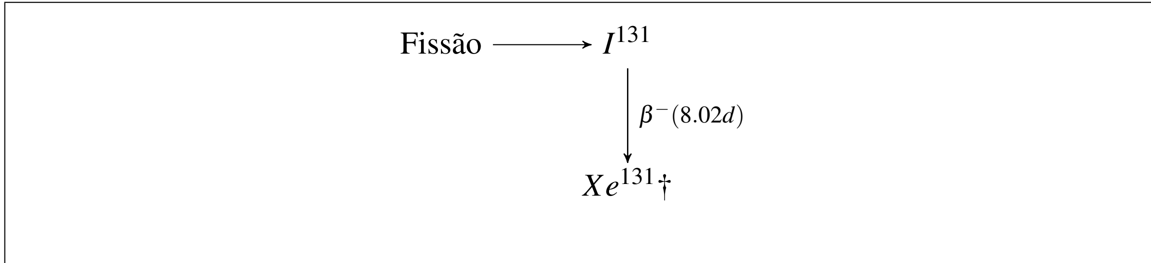

Figura 2.10: Cadeia de decaimento do Xe^{131}.

$$
\begin{aligned}
\frac{d}{dt}N^{Xe^{131}}(t) &= \sum_{i=1}^{17}\gamma_i^{Xe^{131}}\Gamma_i - \lambda_{I^{131}}N^{Xe^{131}}(t) \\
\frac{d}{dt}N^{I^{131}}(t) &= \lambda_{I^{131}}N^{Xe^{131}}(t) - \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{I^{131}}\phi^{g}\right)N^{I^{131}}(t)
\end{aligned} \tag{2.9}
$$

2.3.7 Cadeia do Xe^{135}

O nuclídeo I^{135} é um elemento instável, que decai em β_{-} com meia vida de 6.58 horas e decai em Xenônio Xe^{135}, este por sua vez decai em Cs^{135}, Césio, por β_{-} e sua meia vida é de 9.14 horas. Na figura 2.11 observa-se sua cadeia de decaimento.

As cadeias são apresentadas nas figuras 2.10 e 2.11 e originam o sistema de equações (2.10), onde $\Gamma_i = \sum_{g=1}^{2}\left(\sigma_{f,i}^{g}\phi^{g}\right)N^{i}(t)$ e $\gamma_i^{Xe^{135}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo Xe^{135} e $\gamma_i^{I^{135}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo I^{135} em dois grupos de energia.

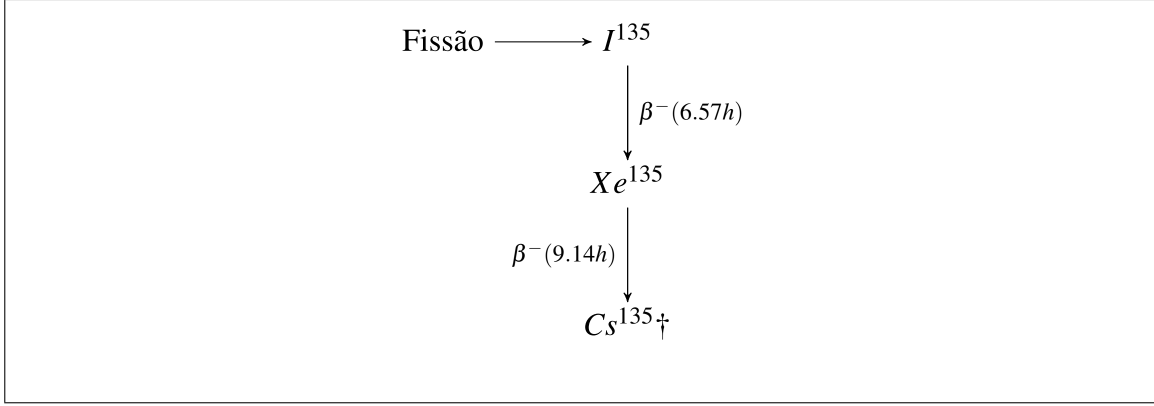

Figura 2.11: Cadeia de decaimento do Xe^{135}.

$$\begin{aligned}
\frac{d}{dt}N^{Xe^{135}}(t) &= \sum_{i=1}^{17}\gamma_i^{Xe^{135}}\Gamma_i - \lambda_{I^{135}}N^{Xe^{135}}(t)\\
\frac{d}{dt}N^{I^{135}}(t) &= \sum_{i=1}^{17}\gamma_j^{I^{135}}\Gamma_i + \lambda_{I^{135}}N^{Xe^{135}}(t) - \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{I^{135}}\,\phi^g\right)N^{I^{135}}(t)
\end{aligned} \tag{2.10}$$

2.3.8 Cadeia do *Sm*

A cadeia do Promécio e Samário envolve vários nuclídeos como o Pm^{147}, Pm^{148}, Pm^{148m}, Pm^{149}, Sm^{147}, Sm^{148}, e Sm^{149} e pode ser observada na figura 2.12. Esta cadeia nos fornece o sistema de equações diferenciais através das equações (2.12) e (2.11).

$$\begin{aligned}
\frac{d}{dt}N^{Pm^{147}}(t) &= \sum_{i=1}^{17}\gamma_i^{Pm^{147}}\Gamma_i - N^{Pm^{147}}(t)\sum_{g=1}^{2}\sigma_{\gamma,g}^{Pm^{147}}\,\phi^g - \lambda^{Pm^{147}}N^{Pm^{147}}\\
\frac{d}{dt}N^{Pm^{148m}}(t) &= \sum_{g=1}^{2}\left(0.47\,\sigma_{\gamma,g}^{Pm^{147}}N^{Pm^{147}}(t) - \sigma_{\gamma,g}^{Pm^{148m}}N^{Pm^{148m}}(t)\right)\phi^g - \lambda^{Pm^{148m}}N^{Pm^{148}}\\
\frac{d}{dt}N^{Pm^{148}}(t) &= \sum_{g=1}^{2}\left(0.53\,\sigma_{\gamma,g}^{Pm^{147}}N^{Pm^{148m}}(t) - \sigma_{\gamma,g}^{Pm^{148}}N^{Pm^{148}}(t)\right)\phi^g\\
&\quad + 0.046N^{Pm^{148m}}\lambda^{Pm^{148m}} - \lambda^{Pm^{148}}N^{Pm^{148}}\\
\frac{d}{dt}N^{Sm^{147}}(t) &= \lambda^{Pm^{147}}N^{Pm^{147}} - \sum_{g=1}^{2}\sigma_{\gamma,g}^{Sm^{147}}\,\phi^g N^{Sm^{147}}
\end{aligned} \tag{2.11}$$

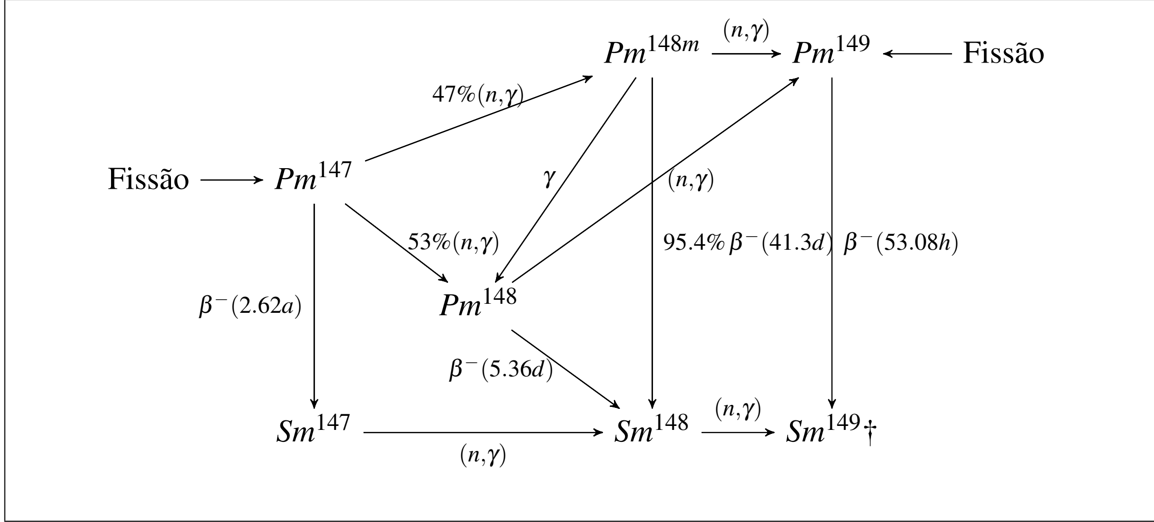

Figura 2.12: Cadeia de decaimento do Samário.

$$
\begin{aligned}
\frac{d}{dt}N^{Pm^{149}}(t) &= \sum_{i=1}^{17}\gamma_i^{Pm^{149}}\Gamma_i + \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{Pm^{148m}}N^{Pm^{148m}}(t) + \sigma_{\gamma,g}^{Pm^{148}}N^{Pm^{148}}(t)\right)\phi^g \\
&\quad -\lambda^{Pm^{149}}N^{Pm^{149}} \\
\frac{d}{dt}N^{Sm^{149}}(t) &= \sum_{g=1}^{2}\left(\sigma_{\gamma,g}^{Sm^{147}}N^{Sm^{147}}(t) - \sigma_{\gamma,g}^{Sm^{149}}N^{Sm^{149}}(t)\right)\phi^g \\
&\quad +0.954\,\lambda^{Pm^{148m}}N^{Pm^{148m}}(t) + \lambda^{Pm^{148}}N^{Pm^{148}}(t)
\end{aligned}
\tag{2.12}
$$

onde $\Gamma_i = \sum_{g=1}^{2}\left(\sigma_{f,i}^{g}\phi^g\right)N^i(t)$ é a contribuição por fissão do nuclídeo i em dois grupos de energia, onde $\Gamma_i = \sum_{g=1}^{2}\left(\sigma_{f,i}^{g}\phi^g\right)N^i(t)$ e $\gamma_i^{Pm^{149}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo Pm^{149}, $\gamma_i^{Pm^{147}}$ é a contribuição por fissão do nuclídeo i para a produção do nuclídeo Pm^{147} em dois grupos de energia.

Capítulo 3

Conceitos matemáticos

Neste capítulo serão descritas algumas definições e proposições utilizadas nesta tese, como o conceito de norma de vetores e matrizes que podem ser observadas com rigor no livro de análise matemática (LIMA (1981)).

Definição 3.1 *Dizemos que uma aplicação linear* $||.|| : \mathbb{R}^n \rightarrow \mathbb{R}$ *é uma norma em* $\mathbb{R}^n$ *se e somente se,* $\forall x, y \in \mathbb{R}^n$ *temos:*

- *O operador* $||.||$ *é positivo definido, ou seja,* $||x|| \geq 0$
- *O operador* $||.||$ *é homogêneo, ou seja, Se* $a \in \mathbb{R} \quad ||a \cdot x|| = |a| \cdot ||x||$
- *Vale a desigualdade triangular,* $||x+y|| \leq ||x|| + ||y||$

Definição 3.2 *Sejam* $x \in \mathbb{R}$ *e* $\|.\|$ *uma norma e* $p \in \mathbb{N} \,|\, 1 \leq p \leq \infty$*. Dizemos que o operador* $\|.\|_p$ *é a norma p em* $\mathbb{R}^n$ *sendo assim definida:*

$$\|x\|_p = \left| |x_1|^p + |x_2|^p + |x_3|^p + |x_4|^p + \cdots + |x_n|^p \right|^{\frac{1}{p}} \tag{3.1}$$

Proposição 3.1 *A partir da definição* (3.2) *temos que a norma euclideana ou norma*[2] *possui a expressão dada pela equação* (3.2)*:*

$$\|x\|_2 = \left| |x_1|^2 + |x_2|^2 + |x_3|^2 + |x_4|^2 +, \cdots + |x_n|^2 \right|^{\frac{1}{2}} \tag{3.2}$$

Definição 3.3 (Norma da matriz) *Seja A uma matrix $m \times n$ sob o corpo $\mathbb{R}$ a aplicação linear $f : A \longrightarrow \mathbb{R}$ é chamada norma de uma matriz se atende as seguintes propriedades,*

- $f(A) \geq 0$
- $f(A) = 0$ *se* $A = 0$
- $f(A+B) \leq f(A) + f(B)$
- $f(\alpha A) = |\alpha| f(A)$, $\forall \alpha \in \mathbb{R}$

Definição 3.4 (Norma de Frobenius) *Somando o módulo de cada a_{ij} da matriz, temos a norma de Frobenius,*

$$\|A\|_f = \left(\sum_{i,j} A_{ij}^2 \right)^{\frac{1}{2}} = tr(AA^T)^{\frac{1}{2}} \tag{3.3}$$

O desenvolvimento da teoria sobre a série de Taylor na reta e no espaço, e das funções de matrizes podem ser complementados através da literatura de Análise Matemática (LIMA (2004)) (LIMA (1976)) e funções de matrizes (HIGHAM (2008)) respectivamente.

Definição 3.5 *Considere $\mathscr{C}$ uma curva fechada de raio R e centro em z_0. Dizemos que f é uma função analítica em um ponto $z_0 \in \mathbb{C}$ se existe a série de potências, que converge para f, dada pela expressão:*

$$f(z) = \sum_{n=0}^{\infty} a_n (z - z_0)^n \tag{3.4}$$

Proposição 3.2 *Teorema da série de Taylor em $\mathbb{R}$. Seja $k \in \mathbb{N}$,$k \geq 1$ e seja $f : \mathbb{R} \to \mathbb{R}$ uma função k vezes diferenciável no ponto $x_0 \in \mathbb{R}$. Então existe uma função $\varphi_k : \mathbb{R} \to \mathbb{R}$ tal que :*

$$\begin{aligned} f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \\ + \varphi_k(x)(x - x_0)^k \end{aligned} \tag{3.5}$$

e fazendo

$$\lim_{x \to x_0} \varphi_k(x) = 0 \tag{3.6}$$

E assim, da proposição 3.2, obtemos o polinômio de Taylor de ordem k, da função f na vizinhança do ponto $x_0 \in \mathbb{R}$, como podemos ver na equação (3.7)

$$P_k(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \tag{3.7}$$

3.3 Erro numérico da aritmética em ponto flutuante

Os processadores atuais utilizam em suas unidades de matemática uma base binária nas quais as normas de ponto flutuante foram definida pela IEEE (*Institute for Electrical and Eletronic Engineers*) (OVERTON (2001)). A IEEE-754 define uma norma de precisão simples com 32-*bits* e de precisão dupla com 64-*bits*.

Definição 3.6 (Base Numérica) *Uma base de uma sistema aritmético é um conjunto de símbolos ou algarismos, com os quais podemos representar um número.*

Por exemplo, na base decimal o conjunto dos algarismos $B = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, um número $N > 0$ em base decimal é representado pela seguinte formulação. Tome duas sequências $x_i, y_j \in B$ onde $i = 0, \cdots, t$ e $j = 1, \cdots, s$ onde t e s representam o número de casas decimais após e antes da virgula com pode ser visto na equação (3.8).

$$N = \overbrace{x_t 10^t + x_{t-1} 10^{t-1} + \cdots + x_1 10^1 x_0 10^0}^{>0}, \underbrace{y_1 10^{-1} + y_2 10^{-2} + \cdots + y_{j-1} 10^{1-s} + y_j 10^{-s}}_{<0} \tag{3.8}$$

Definição 3.7 *O operador de ponto flutuante associa os números reais a um subconjunto $F \subset \mathbf{R}$ tal que se*

$$y \in F \Rightarrow y = \pm \left(\sum_{i=1}^{n} \mathbf{d}_i \beta^{-i} \right) \beta^{\mathbf{e}} \tag{3.9}$$

Para $0 \leq d_j < \beta$ $j = 1, \cdots t$, onde β é a base do sistema de numeração, **e** *é o expoente e* **d** *é a mantissa.*

A aritmética de ponto flutuante F é caracterizada pela base β, que pode ser binária, decimal ou hexadecimal ou qualquer outra; a precisão é dada pelo parâmetro t; e ainda pelos limites do expoente $e_{min} \leq e \leq e_{max}$.

Ao utilizar variáveis declaradas em precisão simples o compilador representa cada uma por uma sequência de 32 *bits* em base 2. O bit mais a esquerda é usado para indicar o sinal da mantissa (zero para positivo e 1 para negativo). A seguir o expoente **e** é representado pelos próximos 8 *bits* e os 23 *bits* finais representam a mantissa **d**, como podemos ver na figura 3.1

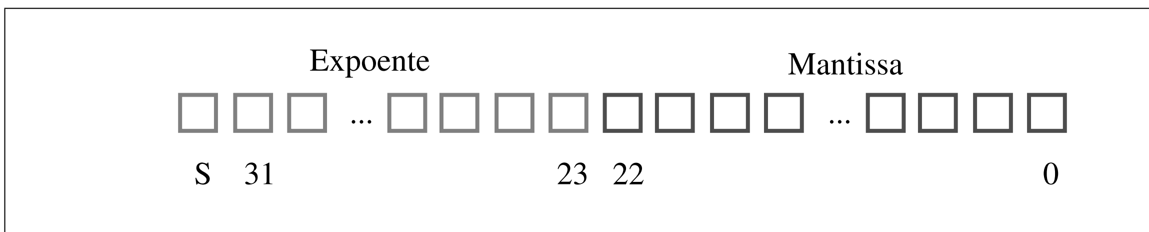

Figura 3.1: Número em ponto flutuante - 32 *bits*.

Ao utilizar variáveis declaradas em precisão dupla o compilador representa cada uma por uma sequência de 64 *bits* em base 2. O bit mais a esquerda é usado para indicar o sinal da mantissa (zero para positivo e 1 para negativo). A seguir o expoente **e** é representado pelos próximos 11 *bits* e os 52 *bits* finais representam a mantissa **d** , como podemos ver na figura 3.2

Os processadores de ponto flutuante atualmente presentes nas GPUs da NVIDIA obedecem a norma IEEE-754. Os processadores padrão x86 presentes em todos os computadores de mesa atuais, tais como os INTEL-I7 ou AMD8350, também são compatíveis com essa norma, contudo algumas opções de compilação produzem aritmética de 80 *bits*

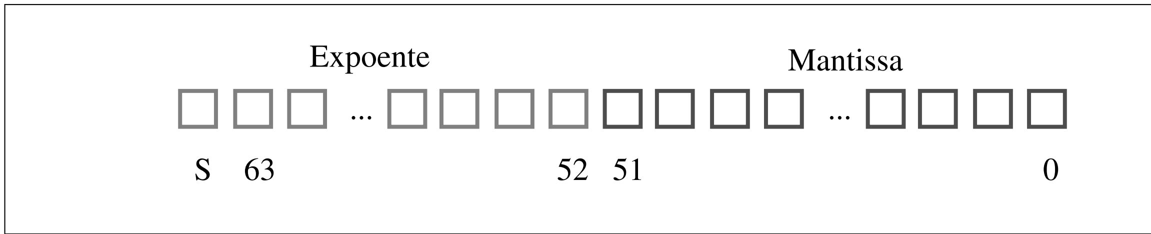

Figura 3.2: Número em ponto flutuante - 64 *bits*.

internamente na unidade de ponto flutuante, sendo depois essa variável truncada para 64 *bits*.

Definição 3.8 (Desvio relativo percentual em $\mathbf{R}^n$) *Sejam $x, y \in \mathbf{R}^n$ a equação que fornece o desvio absoluto é dada por: $err = 100 \cdot \sum_{i=1}^{n} \frac{|x_i - y_i|_1}{max\{x_i, y_i\}}$.*

Definição 3.9 (Distância quadrática média) *O mecanismo de medida utilizado no método da distancia quadrática média ou SRME busca efetuar a comparação entre duas sequências de valores. Este método fornece uma noção da distância euclidiana entre duas sequências de valores, avaliando o desvio percentual entre as duas. Podemos observar o procedimento na equação* (3.10)*. Tome x_i, y_i duas sequências de valores em* $\mathbf{R}$ *então:*

$$\mathbf{SRME} = 100 \times \frac{\sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - y_i)^2}}{(Max\{y_i\} - Min\{y_i\})} \tag{3.10}$$

Capítulo 4

Métodos numéricos

As funções de matrizes tiveram origem dos trabalhos de Cayley (CAYLEY (1858)), e Sylvester (PARSHALL (2006)) no século XIX e envolvem problemas clássicos em diversas áreas da ciência, que vão desde cinética química, problemas de controle (BELLMAN *et al.* (1970)), degradação de poluentes, processos biológicos ou ainda separação de terras raras.

No artigo *Nineteen dubious ways to compute the exponential of a matrix* (MOLER e VAN LOAN (2003)), os pesquisadores expõe diversos métodos e as dificuldades enfrentadas por cada um ao tentar resolver o problema da exponencial de uma matriz. Por outro lado, esta é uma das maneiras de representar um sistema de equações diferenciais de primeira ordem.

Neste capítulo serão descritos os métodos numéricos empregados nesta tese, de tal modo que o problema será investigado sob a ótica de implementação do paralelismo e as dificuldades para tal.

4.1 Matriz exponencial e série de Taylor

Como visto em 2.1 o sistema de equações diferenciais acopladas de primeira ordem descreve o comportamento temporal das concentrações dos nuclídeos presentes no combustível do reator. Estas concentrações são dependentes das seções de choque e do fluxo de nêutrons multi-grupo, e obedecem a seguinte representação obtida de 2.2 e 2.3 :

$$\begin{aligned}\frac{d}{dt}N_k^x(t) = & \sum_{i=1,i\neq k}^{\text{Actinídeos}} N_i^x(t) \sum_{g=1}^{G} \left(\sigma_{\gamma,g}^{i,x}(t) + \sigma_{(n,2n),i}^{g,x}(t) - \sigma_{f,i}^{g,x}(t)\right) \phi_x^g(t) \\ & - \sum_{y=1}^{\text{Frações de decaimento}} \lambda_{i,y} N_i^x(t) + \sum_{z\neq i}^{\text{Frações de Produção}} \lambda_z N_z^x(t) \\ \frac{d}{dt}N_l^x(t) = & \sum_{i=1,i\neq l}^{\text{Produtos de Fissão}} \left(N_i^x(t) \sum_{g=1}^{G} \left(\Gamma_{i,l}^g \sigma_{f,i}^{g,x}(t) - \sigma_{\gamma,g}^{i,x}(t)\right) \phi_x^g(t)\right) \\ & - \lambda_l N_l^x(t) \end{aligned} \tag{4.1}$$

O sistema de equações diferenciais observado na equação (4.1) é não linear pois o fluxo de nêutrons $\phi_x^g(t)$ e as seções de choque utilizadas sofrem alterações em função da composição do combustível. Por outro lado estas variações são muito pequenas ao longo do tempo e se forem consideradas constantes em pequenos intervalos de tempo o sistema da equação (4.1) pode ser escrito como um sistema de equações diferenciais de primeira ordem acopladas com coeficientes constantes. A equação (4.2) descreve o sistema na forma matricial.

$$\frac{d}{dt}\vec{N}(t) = \mathbf{B}.\vec{N}(t), \quad \vec{N}(0) = \vec{N}_0. \tag{4.2}$$

A solução geral da equação (4.2) é dada pela equação (4.3).

$$\frac{d}{dt}\vec{N}(t) = exp(\mathbf{B}t)\vec{N}(0). \tag{4.3}$$

Onde $\vec{N}(0)$ representa a concentração inicial e **B** representa a matriz de transição formada pelos coeficientes de decaimento, captura radioativa, espalhamento $n, 2n$ e etc.

A função $exp(Bt)$ representa a exponencial da matriz de transição e uma solução possível seria dada pela exponencial da matrix $(\mathbf{B}t)$. Fazendo $\mathbf{A} = \mathbf{B}t$ podemos expandir **A** em uma série de Taylor,

$$e^{\mathbf{A}} = (I + \mathbf{A} + \frac{\mathbf{A}^2}{2!} + \frac{\mathbf{A}^3}{3!} + \cdots + \frac{\mathbf{A}^n}{n!}) \quad (4.4)$$

Multiplicando em ambos os lados pelo vetor de concentrações $\vec{N}_0$ obtemos a equação (4.5).

$$e^{\mathbf{A}} \cdot \vec{N}_0 = (I + \mathbf{A} + \frac{\mathbf{A}^2}{2!} + \frac{\mathbf{A}^3}{3!} + \cdots + \frac{\mathbf{A}^n}{n!}) \cdot \vec{N}_0 \quad (4.5)$$

O cálculo da série a partir da matriz de transição pode ser neste caso otimizado utilizando uma decomposição nos produtos matrix-vetor $\{\vec{v}_1, \vec{v}_2, \vec{v}_3, \cdots, \vec{v}_n\}$ como pode ser visto na equação (4.6). Deste modo pode-se economizar processamento e o armazenamento de n matrizes.

$$e^{\mathbf{A}} \cdot \vec{N}_0 = (\vec{N}_0 + \underbrace{\mathbf{A} \cdot \vec{N}_0}_{\vec{v}_1} + \underbrace{\frac{\mathbf{A}}{2}(\underbrace{\mathbf{A} \cdot \vec{N}_0}_{\vec{v}_1})}_{\vec{v}_2} + \underbrace{\frac{\mathbf{A}}{3}(\underbrace{\frac{\mathbf{A}}{2}(\underbrace{\mathbf{A} \cdot \vec{N}_0}_{\vec{v}_1})}_{\vec{v}_2})}_{\vec{v}_3} + \cdots) \quad (4.6)$$

Este método de avaliar as concentrações é adotado pelo sistema ORIGEN2 (CROFF (1983)) e é muito utilizado na área nuclear, tendo mais de 400 citações em artigos até o presente momento. Por outro lado, sofre com o grande número de avaliações necessárias do produto matriz vetor $\mathbf{A}\vec{N}$, o espaço de memória utilizado em sistemas muito grandes e o erro de truncamento relacionado a precisão numérica adotada.

4.2 Métodos de Runge-Kutta

Os métodos de Runge-Kutta (HAIRER *et al.* (1989)) são largamente utilizados para estimar a solução do Problema de Valor Inicial visto na equação (4.7).

$$\frac{d}{dt}y(x) = f(x, y(x)), \quad y(x_0) = y_0 \tag{4.7}$$

Através do método de Euler obtemos uma aproximação da solução do problema de valor inicial proposto em 4.7. Tome uma sequência de pontos $x_1, x_2, x_3. \cdots, x_n$ pertencente ao intervalo onde $f(x, y(x))$ está definida, e um valor h para o intervalo tal que $x_n = x_0 + n \cdot h$ e então $x_{n+1} = x_n + h$.

$$x_{n+1} = x_n + h \Rightarrow h = x_{n+1} - x_n \Rightarrow$$
$$y_n = y_{n-1} + (x_n - x_{n-1}) f(x_{n-1}, y_{n-1}), \quad n = 1, 2, 3, \cdots \tag{4.8}$$

Assim temos que pelo sistema de equações (4.8), observa-se que $y_{n+1} = y_n + h \cdot f(x_n, y_n)$ é uma aproximação da solução da equação diferencial ordinária no ponto x_n e $y_n \approx y(x_n)$.

O método Euler é considerado ineficiente por que um grande número de passos de tempo é necessário para obter uma boa aproximação, e ainda, é possível que o método convirja para algum ponto de acumulação incorreto; e então dispender muito tempo de processamento. A causa da precisão reduzida deste método é utilizar a seguinte aproximação de quadratura:

$$\int_{x_{n-1}}^{x_n} \frac{d}{dx}y(x)dx \approx (x_n - x_{n-1}) \frac{d}{dx}y(x_{n-1}) \tag{4.9}$$

Neste caso a aproximação tem precisão apenas quando $y(x)$ é um polinômio de grau 1. Por outro lado, podemos construir um método com aproximação de segunda ordem tomando a expansão em série de Taylor do problema de valor inicial visto na equação (4.7),

$$y(x+h) = y(x) + h\frac{dy}{dx}(x) + \frac{h^2}{2}\frac{d^2y}{dx^2}(x) + \mathcal{O}(h^3) \tag{4.10}$$

Aplicando a da regra da cadeia em $y''(x)$, como visto na equação (4.11),

$$\begin{aligned} \frac{d^2y}{dx^2}(x) &= f_x(x,y) + f_y(x,y) + \cdots + \frac{d}{dx}y(x) \\ &= f_x(x,y) + f_y(x,y) + \cdots + f(x,y). \end{aligned} \tag{4.11}$$

Supondo que na vizinhança de x, $y(x)$ é constante, temos a equação (4.12).

$$\begin{aligned} y(x+h) &= y(x) + hf(x,y) + \frac{h^2}{2}\big(f_x(x,y) + f_y(x,y) + \cdots + f(x,y)\big) + \mathcal{O}(h^3) \\ &= y(x) + \frac{h}{2}f(x,y) + \frac{h}{2}\big(f(x,y) + f_x(x,y) + f_y(x,y) + \cdots + f(x,y)\big) + \mathcal{O}(h^3). \end{aligned} \tag{4.12}$$

Runge propôs então que se tomasse uma aproximação utilizando passos intermediários como na equação (4.13)

$$y_n = y_{n-1} + h \cdot \sum_{i=1}^{s} b_i k_i + O(h^{s+1}) \tag{4.13}$$

onde

$$k_i = hf\left(t_n + c_i h, h \cdot \sum_{j=1}^{s} a_{ij} k_j\right) \tag{4.14}$$

Onde os a_{ij} são determinados por algum tipo de regra de quadratura, como por exemplo a quadratura de Gauss. Os b_i e c_i são apresentados em uma tabela denominada *Butcher Tableau* (BUTCHER (1963)) como podemos observar na figura 4.1.

De modo análogo, podemos resolver o sistema de equações diferenciais ordinárias

$$
\begin{array}{c|c}
\mathbf{c} & \mathbf{A} \\
\hline
 & \mathbf{b}^{\top}
\end{array}
\quad = \quad
\begin{array}{c|ccccc}
c_1 & a_{11} & a_{12} & a_{13} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & a_{23} & \cdots & a_{2s} \\
c_3 & a_{31} & a_{32} & a_{33} & \cdots & a_{3s} \\
\vdots & \vdots & & \ddots & & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss-1} & a_{ss} \\
\hline
(1) & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}
$$

Figura 4.1: *Butcher Tableau*

através do operador matricial visto na equação (4.2). Deste modo, a forma vetorial do método pode ser observada na equação (4.15)

$$
\begin{aligned}
\vec{y}_n &= \vec{y}_{n-1} + h \cdot \sum_{i=1}^{s} b_i \vec{k}_i + O(h^{s+1}) \\
\vec{k}_i &= hf\left(t_n + c_i h, h \cdot \sum_{j=1}^{s} a_{ij} \vec{k}_j\right)
\end{aligned}
\tag{4.15}
$$

Podemos dividir os métodos de Runge-Kutta em dois tipos básicos: Os métodos podem ser explícitos, neste caso a soma sobre j segue até apenas $i-1$. E portanto, isto significa que os coeficientes acima da diagonal na *Butcher Tableau* são zero. Observe o *Tableau* na figura 4.2, neste caso a matriz de coeficientes do método explícito é triangular inferior, logo podemos determinar cada vetor k_i facilmente por substituição direta. Por outro lado, nos métodos implícitos o somatório dos j vai até s e portanto determinar k_i implica em resolver um sistemas de equações algébricas a cada passo de tempo, implicando em um grande custo computacional.

Em alguns sistemas de equações diferenciais a não linearidade ou ainda a grande discrepância, de várias ordens de grandeza, entre os termos do operador, podem ocasionar dificuldade na convergência devido ao fenômeno da rigidez ou *stiffness*. Neste caso, com abordado em (HAIRER e WANNER (1996)), se faz necessária a implementação de algorítimos de solução implícita, que não são abordados neste trabalho.

No método de Euler o valor fixo do número de passos fixa também o passo de tempo h, mas do ponto de vista computacional é mais interessante que uma mudança no passo de tempo diminua o erro e possivelmente reduza inclusive o número de passos de tempo

$$
\begin{array}{c|c}
\mathbf{c} & \mathbf{A} \\
\hline
 & \mathbf{b}^\top
\end{array}
\quad = \quad
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
(1) & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}
$$

Figura 4.2: *Butcher Tableau* do método explícito

necessários. Este método é denominado Runge-Kutta com passo de tempo adaptativo.

4.2.1 Runge-Kutta-Fehlberg

Nesta versão do método, Fehlberg (SHAMPINE (1977)) utiliza um passo de tempo adaptativo cuja correção é baseada na diferença entre uma aproximação de ordem 5 e outra de ordem 4. Na figura 4.3 pode-se observar o *Tableau* empregado neste método.

$$
\begin{array}{c|c}
\mathbf{c} & \mathbf{A} \\
\hline
 & \mathbf{b}^\top
\end{array}
\quad = \quad
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & b_1^* & b_2^* & \cdots & b_{s-1}^* & b_s^*
\end{array}
$$

Figura 4.3: Runge-Kutta-Fehlberg *Tableau*.

De forma análoga a observada na equação (4.15) o algorítimo foi desenvolvido de maneira vetorial e portanto a função **f** é um operador matricial e os termos $\vec{y}_n$ e $\vec{k}_i$ são vetores de dimensão igual ao número de linhas da matrix associada à **f**. Os elementos $\vec{y}_n$ e $\vec{k}_i$ são obtido por um processo iterativo, fazendo $y_0 = N_0$ e incrementando o passo de tempo assim que o critério de parada é alcançado. Os termos $\vec{k}_i$ são mostrados na equação (4.16)

$$
\begin{aligned}
\vec{k}_1 &= h \cdot \mathbf{f}(t_n, \vec{y}_n) \\
\vec{k}_2 &= h \cdot \mathbf{f}(t_n + \frac{1}{4}h, \vec{y}_n + \frac{1}{4}\vec{k}_1) \\
\vec{k}_3 &= h \cdot \mathbf{f}(t_n + \frac{1}{8}h, \vec{y}_n + \frac{3}{32}\vec{k}_1 + \frac{9}{32}\vec{k}_2) \\
\vec{k}_4 &= h \cdot \mathbf{f}(t_n + \frac{12}{13}h, \vec{y}_n + \frac{1932}{2197}\vec{k}_1 + \frac{7200}{2197}\vec{k}_2 + \frac{7296}{2197}\vec{k}_3) \\
\vec{k}_5 &= h \cdot \mathbf{f}(t_n + h, \vec{y}_n + \frac{439}{216}\vec{k}_1 - 8\vec{k}_2 + \frac{3680}{513}\vec{k}_3 - \frac{845}{4104}\vec{k}_4) \\
\vec{k}_6 &= h \cdot \mathbf{f}(t_n + \frac{1}{2}h, \vec{y}_n - \frac{8}{27}\vec{k}_1 + 2\vec{k}_2 - \frac{3544}{2565}\vec{k}_3 - \frac{1859}{4104}\vec{k}_4 - \frac{11}{40}\vec{k}_5)
\end{aligned} \tag{4.16}
$$

O algoritmo proposto efetua duas aproximações, a primeira de ordem 4 pode ser vista na equação (4.17).

$$
\vec{y}_{n+1} = \vec{y}_n + \frac{25}{216}\vec{k}_1 + \frac{1408}{2565}\vec{k}_3 + \frac{2197}{4104}\vec{k}_4 - \frac{1}{5}\vec{k}_5 \tag{4.17}
$$

e a segunda aproximação é de ordem 5 e pode ser vista na equação (4.18).

$$
\vec{z}_{n+1} = \vec{y}_n + \frac{16}{135}\vec{k}_1 + \frac{6656}{12825}\vec{k}_3 + \frac{28561}{56430}\vec{k}_4 - \frac{9}{50}\vec{k}_5 + \frac{2}{55}\vec{k}_6 \tag{4.18}
$$

O erro absoluto é determinado fazendo-se o produto interno das duas aproximações e pode ser visto na equação (4.20).

$$
\vec{\delta} = |\vec{z}_{n+1} - \vec{y}_{n+1}| \tag{4.19}
$$

$$
err =< \delta, \delta > \tag{4.20}
$$

O passo de tempo adaptativo é escolhido aplicando a equação (4.21) o erro obtido da equação (4.20), o passo de tempo anterior h e o critério de parada ε.

$$h = 0.84 \cdot h \cdot \left(\frac{\varepsilon h}{err}\right)^{\frac{1}{4}} \tag{4.21}$$

O algorítimo pode variar o passo de tempo h entre dois limites (*hmax* e *hmin*). Na figura 4.2.1 pode-se observar que os procedimento necessários de cálculo dos vetores $\vec{k_1}$ a $\vec{k_6}$ não é paralelizável, pois o vetor $\vec{k_2}$ depende da existência do vetor $\vec{k_1}$, o vetor $\vec{k_3}$ depende da existência do vetor $\vec{k_3}$, e assim sucessivamente até o vetor $\vec{k_6}$. As variáveis $error, tolerance, min_h e max_h \in \mathbb{R}$ são entradas da rotina de cálculo.

```
1: x⃗ ← x⃗0
2: h = (max_h − min_h) / 10
3: while error > tolerance do                                  ▷ Where tolerance must be near to 1e^−9
4:     k⃗1 = h·f(t_n, x⃗),
5:     k⃗2 = h·f(t_n + h/4, x⃗ + 1/4 k⃗1)
6:     k⃗3 = h·f(t_n + 3h/8, x⃗ + 3/32 k⃗1 + 9/32 k⃗2)
7:     k⃗4 = h·f(t_n + 12h/13, x⃗ + 1932/2197 k⃗1 − 7200/2197 k⃗2 + 7296/2197 k⃗3)
8:     k⃗5 = h·f(t_n + h, x⃗ + 439/216 k⃗1 − 8k⃗2 + 3680/513 k⃗3 − 845/4104 k⃗4)
9:     k⃗6 = h·f(t_n + h/2, x⃗ − 8/27 k⃗1 + 2k⃗2 − 3544/2565 k⃗3 + 1859/4104 k⃗4 − 11/40 k⃗5)
10:    y⃗_{n+1} = x⃗ + 25/216 k⃗1 + 1408/2565 k⃗3 + 2197/4104 k⃗4 − 1/5 k⃗5)
11:    z⃗_{n+1} = x⃗ + 16/135 k⃗1 + 6656/12825 k⃗3 + 28561/56430 k⃗4 − 9/50 k⃗5 + 2/55 k⃗6)
12:    err = ||z⃗_{n+1} − y⃗_{n+1}||_2                          ▷ Euclidean norm error evaluation
13:    if err > tolerance) then                                ▷ Choice a new time-step
14:        h = h·0.84·(tol/err)^(1/4)
15:    else
16:        t = t + h
17:        x⃗ ← z⃗_{n+1}
18:        if err < (tol/2) then
19:            if max_h < 1.05·h then
20:                h = max_h
21:            else
22:                h = h·1.05
```

Figura 4.4: Runge-Kutta-Fehlberg Algorithm

Por outro lado, pode-se observar que os vetores que representam as concentrações dos nuclídeos e os operadores matriciais apresentam grande tamanho. O objetivo é implementar o algorítimo de maneira a tirar vantagem da tecnologia SIMD, como descrita por 5.1, no capítulo 5.7. No capítulo 5.5 faz-se referência ao operador *soma direta* $\oplus$, sendo este utilizado de forma a contornar o problema da recursividade. No capítulo B.5 apresenta-se a construção do operador de depleção de forma a obter uma função matricial que opera grandes matrizes esparsas e representa a aplicação 4.3. Além disso o ganho de performance veio através das rotinas de multiplicação matriz-vetor (*SpMv*) implemen-

tadas utilizando técnicas de vetorização, no capítulo 5.7.10, e amplo uso das memórias de textura e compartilhada (*TEX MEMORY* and *SHARED MEMORY*) da GPU que são apresentadas no capítulo 5.7.4.

4.3 Métodos de colocação

Um dos métodos de solução do problema 4.7 implementado neste trabalho é baseado na ideia de efetuar a quadratura de Gauss sobre operadores matriciais (SINAP e VAN ASSCHE (1994)), com interpolação nos zeros de polinômios de Jacobi (DATTA *et al.* (1995)). Este é chamado por método de Colocação Polinomial de Jacobi(VILLADSEN e STEWART (1967)) ou método de Gauss-Jacobi, e está implementado em sua forma sequencial no CNFR.

Seja f uma função contínua definida no intervalo $(-1,1)$, a integral $\int\limits_{-1}^{1} f(z)\,dz$ pode ser determinada por uma regra de quadratura definida na equação (4.22). Os pontos $Z = \{z_i\}_{i=1,\ldots,n}$ são chamados de nodos e os coeficientes w_i são chamados de pesos.

$$\int_{-1}^{1} f(z)\,dz = \sum_{i=1}^{N} w_i f(z_i) \tag{4.22}$$

Supondo f como o produto entre uma função peso $w(z)$ e um polinômio ortonormal $P_n(z)$:

$$f(z) = P_n(z) \cdot w(z). \tag{4.23}$$

Uma aproximação da integral na equação (4.22) é dada pela equação (4.24).

$$\int_{-1}^{1} f(z)\,dz = \int_{-1}^{1} P_n(z)w(z)\,dz \approx \sum_{i=1}^{N} \hat{w}_i P_n(\gamma_i) \tag{4.24}$$

Neste caso, os pontos γ_i são as raízes do polinômio de Jacobi que é ortogonais no

intervalo $(-1,1)$ com relação a função peso $w(x) = (1-x)^{\alpha} + (1-x)^{\beta}$, $\alpha, \beta \in \mathbb{R}\, \alpha, \beta > -1$ e seu cálculo é descrito na seção 4.4.2. Serão abordados a seguir alguns aspectos dos polinômios ortonormais.

4.3.1 Polinômios ortonormais

Nesta seção uma breve introdução sobre polinômios ortonormais se faz necessária, e assim vem:

Definição 4.1 *Uma sequência de polinômios $\{P_n(x)\}_{n=0}^{\infty}$ de grau $[P_n(x)] = n$ para cada n, é chamada de ortogonal com respeito a função peso $w(x)$ no intervalo $(a,b) \in \mathbb{R}$ se,* $\int_a^b w(x) p_m(x) p_n(x) dx = h_n \delta_{mn}$ *com* $\delta_{mn} := \begin{cases} 0, & m \neq n \\ 1, & m = n. \end{cases}$

Definição 4.2 *Sejam $f, g \in P(x)$, uma sequência de polinômios em $\mathbb{R}$. Então a integral $\langle f,\ g \rangle := \int_a^b w(x) f(x) g(x) dx$ denota o produto interno dos polinômios f e g. E o intervalo $(a,b) \in \mathbb{R}$ é chamado de intervalo de ortogonalidade.*

Definição 4.3 *Dizemos que um polinômio $k_n(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_0 \in P(x)$ é mônico quando $a_n = 1$.*

Definição 4.4 *Dizemos que uma sequência de polinômios mônicos ortogonais $h_n \in P(x)$ é ortonormal quando $\forall f, g \in h_n$ então $\langle f,\ g \rangle := 1$.*

Observa-se que é possível gerar uma sequência de polinômios ortonormais a partir de uma sequência dada através do método de Gram-Schmidt.

Definição 4.5 *O operador projeção é definido como $proj_u \vec{v} := \dfrac{\langle \vec{v}, \vec{u} \rangle}{\langle \vec{u}, \vec{u} \rangle} \vec{v}$ e projeta o vetor $\vec{v}$ ortogonalmente em $\vec{u}$.*

O processo de Gram-Schmidt consistem e construir uma sequência da seguinte forma

$$\begin{aligned}
\vec{\omega}_1 &= \vec{v}_1 & \vec{q}_1 &= \frac{\vec{\omega}_1}{\|\vec{\omega}_1\|} \\
\vec{\omega}_2 &= \vec{v}_2 - proj_{\omega_1}\vec{v}_2 & \vec{q}_2 &= \frac{\vec{\omega}_2}{\|\vec{\omega}_2\|} \\
\vec{\omega}_3 &= \vec{v}_3 - proj_{\omega_1}\vec{v}_3 - proj_{\omega_2}\vec{v}_3 & \vec{q}_3 &= \frac{\vec{\omega}_3}{\|\vec{\omega}_3\|} \\
& & &\vdots \\
\vec{\omega}_k &= \vec{v}_k - \sum_{i=4}^{k-1} proj_{\omega_i}\vec{v}_k & \vec{q}_k &= \frac{\vec{\omega}_k}{\|\vec{\omega}_k\|}
\end{aligned} \tag{4.25}$$

Onde o conjunto de vetores $\{\vec{q}_1, \vec{q}_2, \cdots, \vec{q}_k\}$ são ortonormais.

Exemplo Considere o seguinte problema, tomando como intervalo de ortogonalidade $(a,b) = (0,1)$, a função peso como $w(x) = 1$ e a sequência $\{1,\ x,\ x^2,\ \ldots\}$ podemos escolher o primeiro polinômio como $p_0(x) = 1$. E assim, utilizando o processo de Gram-Schmidt vem,

$$p_1(x) = x - \frac{\langle x, p_0(x)\rangle}{\langle p_0(x), p_0(x)\rangle} p_0(x) = x - \frac{\langle x, 1\rangle}{\langle 1, 1\rangle} = x - \frac{1}{2}, \tag{4.26}$$

pois,

$$\langle 1,\ 1\rangle = \int_0^1 dx = 1 \text{ and } \langle x,\ 1\rangle = \int_0^1 x dx = \frac{1}{2}.$$

Continuando o processo vem,

$$p_2(x) = x^2 - \frac{\langle x^2, p_0(x)\rangle}{\langle p_0(x), p_0(x)\rangle} p_0(x) - \frac{\langle x^2, p_1(x)\rangle}{\langle p_1(x), p_1(x)\rangle} p_1(x) \tag{4.27}$$

desde que

$$\langle x^2,1\rangle = \int_0^1 x^2 dx = \frac{1}{3},$$

$$\langle x^2,\ x-\frac{1}{2}\rangle = \int_0^1 x^2(x-\frac{1}{2})dx = \frac{1}{4}-\frac{1}{6} = \frac{1}{12}$$

$$\langle x-\frac{1}{2},\ x-\frac{1}{2}\rangle = \int_0^1 (x-\frac{1}{2})^2 dx = \int_0^1 (x^2-x+\frac{1}{4})dx = \frac{1}{3}-\frac{1}{2}+\frac{1}{4} = \frac{1}{12}$$

Repedindo o procedimento temos:

$$\begin{aligned} p_3(x) &= x^3-\frac{3}{2}x^2+\frac{3}{5}x-\frac{1}{20} \\ p_4(x) &= x^4-2x^3+\frac{9}{7}x^2-\frac{2}{7}x+\frac{1}{70} \\ p_5(x) &= x^5-\frac{5}{2}x^4+\frac{20}{9}x^3-\frac{5}{6}x^2+\frac{5}{42}x-\frac{1}{252} \end{aligned} \tag{4.28}$$

De forma que os polinômios ortonormais são dados por:

$$\begin{aligned} q_0(x) &= p_0(x)/\sqrt{h_0} = 1 \\ q_1(x) &= p_1(x)/\sqrt{h_1} = 2\sqrt{3}(x-1/2) \\ q_2(x) &= \frac{p_2(x)}{\sqrt{h_2}} = 6\sqrt{5}(x^2-x+\frac{1}{6}) \\ q_3(x) &= \frac{p_3(x)}{\sqrt{h_3}} = 20\sqrt{7}(x^3-\frac{3}{2}x^2+\frac{3}{5}x-\frac{1}{20}) \end{aligned} \tag{4.29}$$

Observando a sequência polinômios podemos propor e encontrar uma relação de recorrência entre cada um deles.

Proposição 4.4 *Toda sequência de polinômios ortogonais satisfaz uma relação de recorrência de três termos.*

4.4.1 Interpolação de Lagrange e quadratura de Gauss

Seja f uma função contínua no intervalo $(a,\ b)$ e além disso toma-se

$$x_1 < x_2 < \cdots < x_n$$

n pontos distintos neste intervalo. Então existe exatamente um polinômio P com grau $\leq n-1$ tal que:

$$P(x_i) = f(x_i)\ \forall\, i = 1, 2,\ \ldots, n.$$

Pode-se encontra-lo utilizando a interpolação de Lagrange na equação 4.30

$$p(x) = (x-x_1)(x-x_2)\cdots(x-x_n) = \prod_{j=1}^{n}(x-x_j). \tag{4.30}$$

Assim, considere a equação 4.31

$$\begin{aligned} P(x) &= \sum_{i=1}^{n} \frac{p(x)}{(x-x_i)p(x_i)} \\ &= \sum_{i=1}^{n} f(x_i) \frac{(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdot\cdots\ldots(x-x_n)}{(x_i-x_1)(x_i-x_{i-1})(x_i-x_{i+1})(x_i-x_n)} \\ &= \sum_{i=1}^{n} f(x_i) \prod_{j=1, j\neq i}^{n} \frac{(x-x_j)}{x_i - x_j} \end{aligned} \tag{4.31}$$

Seja $\{P_n(x)\}_{n=0}^{\infty}$, uma sequência de polinômios ortogonais no intervalo $(a,\ b)$ com respeito a uma função peso $w(x)$. Então para $x_1 < x_2 < \cdots < x_n$ temos n distintos zeros do polinômio $p_n(x)$. Se f é um polinômio de grau $\leq 2n-1$, então $f(x) - P(x)$ é um polinômio de grau $\leq 2n-1$ com ao menos os zeros $x_1 < x_2 < \cdots < x_n$.

Seja $f(x) = P(x) + r(x)p_n(x)$, onde $r(x)$ é um polinômio de grau $\leq n-1$. Assim

escreve-se,

$$f(x) = \sum_{i=1}^{n} f(x_i) \frac{p_n(x)}{(x - x_i) p_n'(x_i)} + r(x) p_n(x). \tag{4.32}$$

E portanto,

$$\int_a^b w(x) f(x) dx = \sum_{i=1}^{n} f(x_i) \int_a^b \frac{w(x) p_n(x)}{(x - x_i) p_n(x_i)} dx + \int_a^b w(x) r(x) p_n(x) dx. \tag{4.33}$$

Desde que o grau de $r(x) \leq n-1$ pela propriedade da ortogonalidade temos que a última integral é igual a zero. Logo,

$$\int_a^b w(x) f(x) dx = \sum_{i=1}^{n} \lambda_{n,i} f(x_i) \tag{4.34}$$

com

$$\lambda_{n,i} := \int_a^b \frac{w(x) p_n(x)}{(x - x_i) p_n(x_i)} dx, \; i = 1, 2, \; \ldots, \; n \tag{4.35}$$

A equação (4.35) representa a quadratura de Gauss e fornece o valor da integral para o caso em que f é um polinômio de grau $\leq 2n-1$. Se o valor de $f(x_i)$ é conhecido para n zeros $x_1 < x_2 < \cdots < x_n$ do polinômio $p_n(x)$.

Se f não é um polinômio de grau $\leq 2n-1$, isto leva a uma aproximação da integral na equação (4.36).

$$\int_a^b w(x)f(x)dx \approx \sum_{i=1}^{n} \int_a^b \frac{w(x)p_n(x)}{(x-x_i)p_n(x_i)}dx,\ i=1,2,\ \dots,\ n \tag{4.36}$$

4.4.2 Método de colocação de Gauss-Jacobi

Como vimos anteriormente no capítulo 4.1, o sistema de equações diferenciais observado na equação (4.1) é não linear já que fluxo de nêutrons e as seções de choque sofrem alterações em função da composição do combustível. Consideram-se estas variações muito pequenas e portanto constantes no intervalo de tempo do passo de queima. Neste caso, o sistema de equações que governa as concentrações dos nuclídeos no combustível do reator pode ser escrito em forma matricial como visto na equação (4.37).

$$\frac{d\vec{N}}{dt}(t) = \mathbb{A}.\vec{N}(t), \quad t_{l-1} < t < t_l \tag{4.37}$$

Onde os índices n se referem ao nodo do combustível do reator, os índices t ao passo de tempo e a matriz $\mathbb{A}^n = [a_{ij}]_n$ é a matriz que representa a depleção do combustível. Seja então $N^n(t_0)$ a concentração inicial. O problema 4.37 possui, com já vimos, a seguinte solução,

$$\vec{N}(t_l) = T(\Delta t_l)\vec{N}(t_{l-1}) \tag{4.38}$$

Uma aproximação integral de funções de matrizes utiliza a interpolação de Lagrange, a quadratura de Gauss com polinômios de Jacobi (SINAP e VAN ASSCHE (1994)). Os polinômios de Jacobi são ortogonais no intervalo $(-1,1)$ com respeito a função peso $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$. Podendo ser definidos pela formula de Rodrigues (SIMON (2009)).

Sejam α e β $\in \mathbb{R}$ então temos a sua formulação na equação (4.39).

$$P_n^{(\alpha,\beta)}(x) = \frac{(-1)^n}{2^n n!}\frac{1}{w(x)}D^n[w(x)(1-x^2)^n] \tag{4.39}$$
$$= \frac{(-1)^n}{2^n n!}(1-x)^{-\alpha}(1+x)^{-\beta}D^n[(1-x)^{n+\alpha}(1+x)^{n+\beta}]\text{Para} n = 0,1,2, \ldots.$$

O algoritmo empregado é uma solução simplificada, seguindo o procedimento de Ramirez (RAMIREZ (1997)). Este método é implementado em sua formulação sequencial no CNFR (ALVIM *et al.* (2010)) e efetua o cálculo da matriz de transição $T(\Delta t_n)$ associada com a solução do sistema de equações diferenciais 4.2 encontrando a solução de um sistema matricial generalizado dado pela equação (4.40).

$$C \cdot T(\Delta t_n) = D. \tag{4.40}$$

Os operadores matriciais C e D são obtidos através do operador matricial $\mathbf{A}\Delta t_l)$. Neste caso a matriz C é obtida da equação (4.41)

$$C = \sum_{j=0}^{m+2}\left((-1)^{m-j}(m-j+3)!(\gamma_{m-j}+2\gamma_{m-j+1}+\gamma_{m-j+2})(\mathbf{A}\Delta t_l)^j\right) \tag{4.41}$$

e a matriz D é dada pela equação (4.42)

$$D = \sum_{j=0}^{m+3}\left((d_j + q_{j-1} + dj - 1)((\mathbf{A}\Delta t_l)^j)\right) \tag{4.42}$$

Os coeficientes q_k são obtidos através da equação (4.43)

$$q_k = \sum_{i=1}^{k}\left(\frac{d_{i-1}}{(i+1)!}\right), \quad i = 1,2,\cdots,m+2. \tag{4.43}$$

Os coeficientes d_k são obtidos através da equação (4.44)

$$d_{m+2-k} = (k+1)!(-1)^k(\gamma_{k-2} + 2\gamma_{k-1} + \gamma_k), \quad i = 1, 2, \cdots, m+2 \tag{4.44}$$

Observe-se que os coeficientes γ_k são os coeficientes de Jacobi de ordem k, obtidos recursivamente através da relação 4.45,

$$\gamma_{i+1}^m = \frac{(m+1-i)(m+i+\alpha+\beta)}{i(\gamma_i+\beta)}, \quad i = 1, 2, \cdots, m \tag{4.45}$$

com $\gamma_0^m = 1, \quad \forall n \geq 0$, e neste trabalho foram utilizados $\alpha = 2$ e $\beta = 1$. Ao sistema algébrico assim desenvolvido efetua-se em ambos os lados o produto pelo vetor das concentrações $N^n(t_{l-1})$,

$$\begin{aligned} C \cdot T(\Delta t_l) N^n(t_{l-1}) &= DN^n(t_{l-1}) \\ C \cdot N^n(t_l) &= b \end{aligned} \tag{4.46}$$

Na figura 4.5 podemos observar o algorítimo utilizado na obtenção dos operadores matriciais D and C.

O vetor de concentrações é dado pela equação (4.47),

$$\vec{N}(t_n) = T(\Delta t_n) \cdot \vec{N}(t_{n-1}). \tag{4.47}$$

O vetor de concentrações $\vec{N}(t_n)$ é obtido resolvendo o sistema matricial observado na equação (4.40) de forma a encontrar a matriz $T(\Delta t_n)$. Por outro lado, efetuando o produto do vetor concentração $\vec{N}(t_{n-1})$ em ambos os lados da equação (4.47) resulta na equação (4.48), uma maneira mais simples de calcular o vetor $\vec{N}(t_n)$.

```
1: r ← |A|·ΔT                                                    ▷ r é a norma de Frobenius de |A|
2: while r > 2^i do                                              ▷ i = 0
3:     i = i+1
                                                                 ▷ Normalização de |A|
4: |A|_0 = |A| · 1/2^i                                           ▷ i = 1,   Set γ
5: while i <= m do
6:     γ_{i+1} = (i(m−i+1))/((i+1)) · γ_i · (m+i+3)              ▷ i = i+1
7: γ_1 = 1, γ_{m+2} = 0, γ_{m+3} = 0                             ▷ i = 2 factor = 2
8: while i <= m+2 do
9:     factor = factor·(i+1) and d_{m+3−i} = (−1)^i factor(γ_{i−1} + 2γ_i + γ_{i+1})   ▷ i = i+1
10: d_{(m+3)} = 1, d_{(m+2)} = −2(2+γ_2)
11: while i <= m+3 do                                            ▷ i = 2 q_1 = 0
12:     factor = 1 q_i = 0
13:     while j <= i+1 do                                        ▷ j = 1
14:         factor = factor·(j+1) and q_i = q_{i−1} + d_{i−j}/factor   ▷ j = j+1
                                                                 ▷ i = i+1
15: q_{(m+2)} = −2(2+γ_2)
16: while i <= m+2 do                                            ▷ Agora, calcule D
17:     A^{i+1} = A^i · A_0                                      ▷ i = i+1
18: while i <= m+3 do
19:     t = t·Δt;
20:     D = D + D·(d_i + q_{i−1} + d_{i−1})·t                    ▷ i = i+1
21: while i <= m+3 do                                            ▷ Agora, calcule C
22:     t = t·Δt;
23:     factor = factor/(m+4−1)
24:     C = C + C·(−1)^{m−i}·(γ_{m+1−i} + 2γ_{m−i+2}γ_{m−i+3})·factor·t   ▷ i = i+1
                                                                 ▷ END
```

Figura 4.5: Encontrar a matriz de transição para a expansão polinomial de Jacobi

$$C \cdot T(\Delta t_n) \cdot \vec{N}(t_{n-1}) = D \cdot \vec{N}(t_{n-1}) \tag{4.48}$$

Fazendo o vetor $\vec{y} = T(\Delta t_n) \cdot \vec{N}(t_{n-1})$ e $\vec{x} = D \cdot \vec{N}(t_{n-1})$ para obter o sistema linear dado pela equação (4.49).

$$C \cdot \vec{y} = \vec{x} \tag{4.49}$$

O operador C é não simétrico, e o sistema linear na equação (4.49) é solucionado através do algoritmo do gradiente bi-conjugado (SAAD (2003)) e obtém-se assim o vetor de concentrações $\vec{N}(t_n) = \vec{y}$.

4.5 Método de aproximação por série de Padé

Uma das maneiras descritas por Moler (MOLER e VAN LOAN (2003)) de obter a exponencial da matriz é utilizar uma aproximação racional denominada série de Padé. Neste trabalho efetuou-se sua implementação em GPU de forma a obter um objeto de comparação entre os métodos de Runge-Kutta-Fehlberg e o de Colocação de Gauss-Jacobi. Em trabalhos recentes, Higham (HIGHAM (2005)) propõe algorítimos baseados em aproximações racionais e mostra que a aproximação por série de Padé é um método confiável de avaliação da exponencial da matriz.

A condição necessária a convergência da série é que a norma do operador matricial seja pequena, ou seja, tomando a matriz $B = A \cdot \Delta t$, supomos que a norma $|B|$, no sentido de Frobenius (3.3), é suficientemente pequena, então $e^{\mathbf{B}}$ pode ser aproximada por $\left(e^{\mathbf{B}} \approx R(\mathbf{B})\right)$.

A propriedade da matriz exponencial $\left(e^{A} = (e^{\frac{A}{2^n}})^{2^n}\right)$ é utilizada pelo método de escalar e quadrar a matriz, uma vez que a a série de Padé apresenta melhores resultados se a norma $|\mathbf{B}\Delta t|$ é suficientemente pequena (TOROKHTI e HOWLETT (1971)). As variáveis t_n são os passos de queima e o vetor das concentrações é dado pela equação (4.50).

$$\vec{N}(t_n) = e^{\mathbf{B}\Delta t} \cdot \vec{N}(t_{n-1}), \quad n = 1 \cdots N. \tag{4.50}$$

Fazendo $A = (\mathbf{B}\Delta t)$. Nosso objetivo é aproximar a matriz exponencial $e^{\mathbf{A}}$ por um operador racional $R_{pq}(\mathbf{A}) \approx e^{\mathbf{A}}$ obtido através da equação (4.51)

$$R_{pq}(\mathbf{A}) = (D_{pq}(\mathbf{A}))^{-1} N_{pq}(\mathbf{A}). \tag{4.51}$$

A matriz $D_{pq}(\mathbf{A})$ é obtida da equação

$$D_{pq}(\mathbf{A}) = \sum_{j=1}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!}\mathbf{A}^j. \quad (4.52)$$

e assim obtemos através da equação

$$N_{pq}(\mathbf{A}) = \sum_{j=1}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!}\mathbf{A}^j. \quad (4.53)$$

Em sua expressão diagonal, *i.e.*, fazemos $p = q$ e então obtemos

$$R_{pp}(\mathbf{A}) = (\sum_{j=1}^{p} \frac{(2p-j)!p!}{(2p)!j!(p-j)!} - \mathbf{A}^j)^{-1}(\sum_{j=1}^{p} \frac{(2p-j)!p!}{(2p)!j!(p-j)!}\mathbf{A}^j). \quad (4.54)$$

A figura 4.6 mostra o algorítimo de *scaling* implementado para fazer a norma $|A| = |\mathbf{B}\Delta t|$ muito pequena e tornar a série de Padé convergente.

1: A e r são as entradas
2: **while** $\frac{r}{\theta} > s$ **do** ▷ Faça $s = 1$ e $i = 0$
3: $s = s \cdot 2$
4: $i = i + 1$
5: $A = \frac{A}{s}$ ▷ Escale A
6: Retorna A

Figura 4.6: Operação de escala

A rotina principal do algorítimo pode ser observada na figura 4.7.

A figure 4.8 mostra o algorítimo *PadeApproximant(m)*, que efetua a aproximação de $e^A \approx R(A)$ utilizando um modelo simplificado de ordem m. Este algorítimo necessita avaliar um grande número de potências da matriz A de forma a obter o operador F. Além disso é preciso inverter a matriz $(U+V)$ para obter o operador $F = (-U+V)\cdot(U+V)^{-1}$. Este é um procedimento computacionalmente custoso e impacta substancialmente a performance do método.

Foi implementado na GPU o método de inversão da matriz por Jacobi (WESTLAKE (1968)) para encontrar a inversa da matriz $(U+V)$. Let $W = (U+V)$, este método pode

```
1:  r ← |A|
2:  if r ≤ θ_k then
3:      for i = 1 to k do
4:          if r ≤ θ_i then
5:              F ← PadeApproximant(r)          ▷ Calcula a matriz exponencial
6:              Pare
7:  else
8:      A = Scaling(A, r)                       ▷ Operador escala A por 2^r
9:      j ← M(i)                                ▷ A aproximação vem da tabela
10:     F ← PadeApproximant(j)                  ▷ Calcula a exponencial da matrix
11:     while i > 0 do                          ▷ Quadra a matriz
12:         F = F · F
13:         i = i − 1
```

Figura 4.7: Encontra a aproximação diagonal de Padé

```
1:  m é a ordem
2:  if m ∈ {3, 5, 7, 9} then
3:      Calcula A^0, A^1, A^2, ··· A^((m+1)/2)
4:      for j = m + 1 a 2 com passo −2 do
5:          Calcula U = U + c_j · A^(j/2)
6:      U = A · U
7:      for j = m a 1 com passo −2 do
8:          Calcula V = V + c_j · A^((j+1)/2)
9:  if m = 13 then
10:     Calculate A^2, A^4, A^6
11:     U = A · (A^6(c_14 A^6 + c_12 A^4 + c_10 A^2) + c_8 A^6 + c_6 A^4 + c_4 A^2 + c_2 A^0)
12:     V = A^6(c_13 A^6 + c_11 A^4 + c_9 A^2 + c_7 A^6 + c_5 A^4 + c_3 A^2 c_1 A^0)
13: F = (−U + V) · (U + V)^(−1)                 ▷ Return F
```

Figura 4.8: Aproximação de Padé.

ser visto na equação (4.55).

$$
\begin{aligned}
W \cdot W^{-1} &= I \\
(L+D+U) \cdot W^{-1} &= I \\
(L+U) \cdot W^{-1} \cdot W + D \cdot W^{-1} &= I \cdot W \\
(L+U) - I \cdot W &= D \cdot W^{-1} \\
-D^{-1}(L+U) + D^{-1}W^{-1} &= W^{-1}
\end{aligned} \tag{4.55}
$$

Dividindo a matriz W em matrizes D, U e L, respectivamente : diagonal, superior e inferior. Observa-se na equação (4.55) a sugestão de recorrência mostrada na equação (4.56) para obter a inversa da matriz W^{-1}. Foi utilizada uma condição de relaxação $\alpha \approx 1$, escolhida para incrementar a convergência.

$$
-D^{-1}(L+U) + \alpha \cdot D^{-1}W_n^{-1} = W_{n+1}^{-1}, \quad W_0 = D \tag{4.56}
$$

Sejam $a_{ij} \in W_n^{-1}$ e $b_{ij} \in W_{n+1}^{-1}$. A norma de Frobenius é dada pela equação (4.57) e é utilizada como condição de convergência.

$$
\begin{aligned}
N_F &= |W_n^{-1} - W_{n+1}^{-1}|_F < \varepsilon \\
N_F &= \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij} - b_{ij}| < \varepsilon
\end{aligned} \tag{4.57}
$$

Após o cálculo de $F = (-U+V) \cdot (U+V)^{-1}$, a sequência do algorítimo efetua o cálculo de $F = \underbrace{F \cdot F \cdot F \cdot \quad F}_{2^s \quad \text{vezes}}$ para encontrar D_{pp}. Assim, como podemos ver na equação (4.51), a aproximação racional $R(A) \approx e^A$ implica em um novo vetor de concentrações $\vec{N}(t_n) = R(A) \cdot \vec{N}(t_{n-1})$.

Capítulo 5

Computação de alto desempenho

O objetivo principal da computação paralela de alto desempenho neste trabalho é desenvolver os algorítimos apropriados ao problema, identificando as estratégias computacionais adaptadas as novas tecnologias baseadas em GPU e processamento multi núcleo em CPU.

Em um sentido mais amplo, a pesquisa na área nuclear caminha na direção dos reatores de IV geração (KIM (2013)) (RENAULT *et al.* (2009)) que requerem grandes recursos computacionais para sua simulação. Os novos projetos de reatores são avaliados em dispendiosos supercomputadores como o JAGUAR (BLAND *et al.* (2009)) e o TITAN (BLAND (2012)) construído no laboratório ORNL *Oak Ridge National Laboratory* . Estes reatores oferecem novas perspectivas de segurança, geração de eletricidade e tempo de queima do combustível. Por outro lado, a demanda por energia é um problema mundial hoje e no futuro, e grandes *clusters* de computadores são vorazes consumidores de energia, tanto para o processamento quanto para a refrigeração, necessitando também de grandes recursos humanos e materiais.

Por outro lado, novas tecnologias e arquiteturas de computadores surgiram nos últimos anos e mais do que dobraram a expectativa da lei de Moore (BONDYOPADHYAY (1998)), ela foi formulada em um momento da história de computação onde a performance de um circuito semicondutor poderia dobrar a cada 18 meses. Estas novas tecnologias fizeram com que o poder de computação desse um salto e tornasse economicamente viável a computação científica em super computadores de baixo custo na mesa do usuário. Des-

tas podemos destacar duas como principais: Os processadores com múltiplos núcleos e os processadores gráficos.

A tecnologia dos processadores atuais (CPU) evoluiu para uma topologia de múltiplos núcleos com memória compartilhada e acesso único ao barramento de entrada/saída, que de acordo com a taxionomia de Flynn (FLYNN (1972)) são classificados como MIMD (*Multiple Instruction Multiple Data*). Isto torna os programas escritos de forma sequencial ineficientes, no sentido de que o código executa em apenas um núcleo do processador; sendo que em 2013 os fabricantes INTEL (HAMMARLUND *et al.* (2014)) e AMD (YOU *et al.* (2013))já possuíam processadores com até 16 núcleos no substrato. Temos ainda a arquitetura MIC (*Many Integrated Core Architecture*) (SATISH *et al.* (2010)) da INTEL cuja proposta é colocar centenas de processadores MIMD em uma única unidade de processamento.

Por outro lado, a partir de 2008 os fabricantes NVIDIA e AMD, se aproveitaram do grande poder computacional de suas placas de vídeo, anteriormente utilizadas apenas em jogos, de lançaram pacotes de computação genérica baseados em GPU. A NVIDIA disponibilizou um compilador e bibliotecas de álgebra linear cuBLAS (NVIDIA (2008a)) e cuFFT (NVIDIA (2010)) para seus processadores e uma extensão da linguagem C chamada CUDA (NVIDIA (2008b)).

Ao longo dos últimos anos os fabricantes de microprocessadores, neste caso: INTEL e AMD, se dedicaram a desenvolver processadores com múltiplos núcleos de processamento no mesmo encapsulamento, de forma a incrementar a capacidade computacional de seus produtos sem a necessidade de aumentar o *clock* do processador. De forma que os processadores atuais executam com maior performance programas que distribuam os procedimentos entre estes núcleos de processamento.

Os processadores desenvolvidos pela NVIDIA denominados de GPU, ou *Graphics Processor Unit* tem seu grande desempenho baseado em uma topologia sugerida por Flynn (FLYNN (1972)), denominada SIMD (*Single Instruction Multiple Data*). Nas GPU temos centenas ou até milhares de processadores de ponto flutuante (LEVESQUE (2012)) executando a mesma instrução mas com dados diferentes em cada unidade de processa-

mento. Esta peculiaridade torna os programas escritos de maneira sequencial especialmente difíceis de implementar. Por exemplo, a chamada recursiva de uma função tem complexidade algorítmica sequencial. Além disso, a performance em dupla precisão é bem inferior a da precisão simples, o que estimulou alguns pesquisadores a desenvolver algorítimos de precisão mista (CLARK *et al.* (2010)).

Existem diversas maneiras de fazer esta distribuição em tempo de compilação, ou seja, durante a construção do programa, podemos fornecer instruções ao compilador de forma que o programa executável faça esta distribuição. De maneira geral os códigos paralelizáveis utilizam três tecnologias de distribuição de tarefas: *threads* (RANGARAJAN (1991)) (REINDERS (2007)), OPENMP (CHANDRA (2001)) ou MPI (GROPP *et al.* (1999)).

Em 2007 o fabricante NVIDIA lançou no mercado um produto revolucionário, placas de vídeo nas quais é possível executar código genérico denominadas de GPU *Graphic Processor Unit*. Além disso, distribuiu gratuitamente as ferramentas de desenvolvimento de *software* para estes co-processadores denominada CUDA (KIRK (2007)) *Compute Unified Device Architecture* e é baseada em um extremo paralelismo, utilizando centenas ou até milhares de processadores em uma GPU, para obter sua grande performance numérica (ALMEIDA (2009)) (WHITEHEAD e FIT-FLOREA (2011)).

Nos trabalhos (ALMEIDA (2009)), (HEIMLICH *et al.* (2011)) e (WAINTRAUB *et al.* (2011)) foram abordados alguns conceitos de computação genérica em GPU para área nuclear (LINDHOLM *et al.* (2008)), (NICKOLLS e DALLY (2010)). Desde então a computação genérica em GPU se expandiu a praticamente todas as áreas do conhecimento e pode oferecer um grande desempenho computacional aliado a um custo e consumo de energia relativamente baixos (PRICE *et al.* (2014)).

O sistema GEDAR utiliza o método nodal NEM (FINNEMANN *et al.* (1977)) para o cálculo do fluxo neutrônico. Neste caso, o reator é espacialmente dividido em não menos que quatro mil nodos, avaliando uma solução tridimensional, cartesiana e estacionária, em dois grupos de energia da equação de difusão de nêutrons. A avaliação das concentrações isotópicas do combustível é fundamental para a exatidão deste cálculo.

Em nossos testes utilizou-se o sistema GEDAR configurado para efetuar a simulação

do núcleo do reator de Angra II com 32 passos de queima não uniformes. Esta simulação divide o núcleo do reator em uma geometria tridimensional constituída por blocos de 20 cm de aresta e 30 cm de altura, denominados nodos, com seções de choque homogêneas em cada nodo.

Na simulação do *burnup* executada neste estudo utilizou-se uma discretização constituída por 4368 nodos, dos quais 2988 são nodos com combustível. Este número expressivo de nodos impõe que o cálculo do *burnup* e transmutação do combustível durante a simulação do GEDAR impacte fortemente no tempo total. Por outro lado, uma simulação em malha fina utilizando diferenças finitas ou outro modelo, como o método das características (MOC) pode representar a avaliação do *burnup* de mais de 3 milhões de nodos.

As recentes tecnologias de computação baseada em processadores gráficos (GPUs), *Field Processors Gate Array* (FPGA) (PELLERIN e THIBAULT (2005)) e Intel *Many Integrated Core Architecture* (MIC) (DURAN e KLEMM (2012)) se propõe a levar a computação de alto-desempenho a um novo patamar, com baixo custo e potência por Watt. Nesta tese implementaram-se algorítimos em OpenMP e GPU para a solução da equação de depleção do combustível sendo possível também implementa-los na tecnologia MIC, uma vez que esta também se utiliza do OpenMP. Seguem então algumas considerações a respeito das topologias dos processadores e o modo de implementação baseadas na taxionomia de Flynn.

5.1 A taxionomia de Flynn

Flynn definiu quatro classificações ou tipos, de uma arquitetura de processamento computacional (FLYNN (1972)), a partir da concorrência entre processos ou do fluxo de dados na memória.

***Single Instruction, Single Data Stream* (SISD)**

É a taxionomia sequencial, esta não explora o paralelismo, seja por instruções ou fluxo de dados; Processos recursivos ou códigos executando um processadores de

núcleo simples, os quais possuem paralelismo de tarefas apenas por fatiamento de tempo processador.

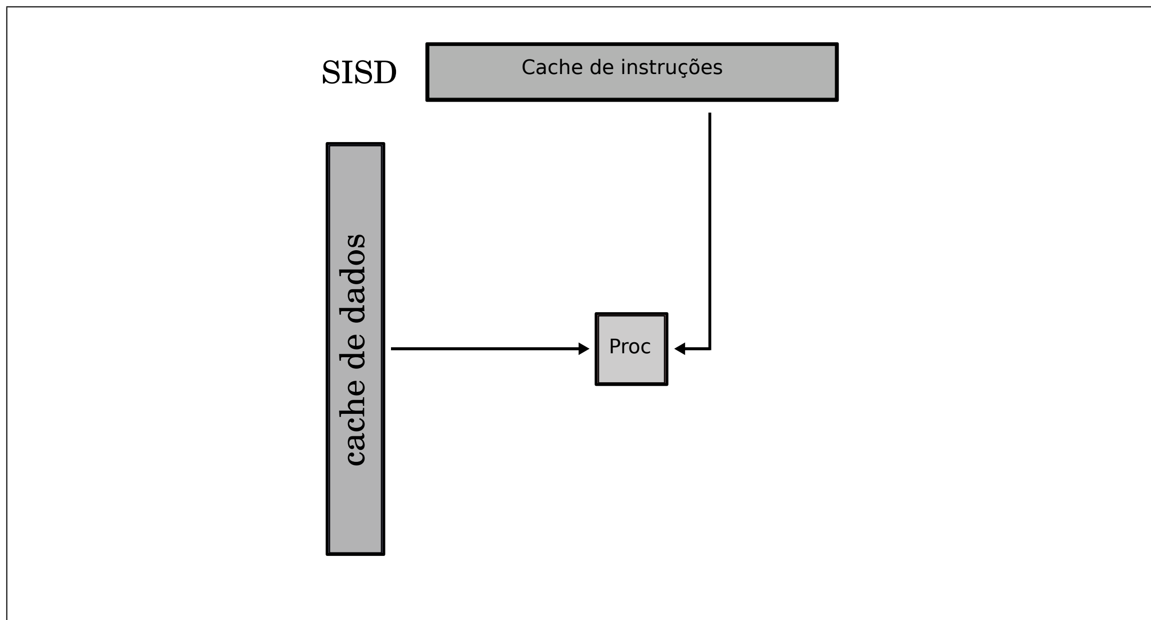

Figura 5.1: Taxionomia SISD

***Single Instruction, Multiple Data Streams* (SIMD / SIMT)**

Descreve um sistema computacional que processa múltiplos fluxos de dados simultaneamente. No sistema SIMD em cada ciclo manipulam-se múltiplas áreas de memória executando a mesma instrução. No sistema SIMT em cada ciclo múltiplos *threads* manipulam a área de dados simultaneamente.

***Multiple Instruction, Single Data stream* (MISD)**

Múltiplas instruções operam em um único fluxo de dados. É geralmente utilizada por sistemas tolerantes a falha, garantindo alta disponibilidade.

***Multiple Instruction, Multiple Data streams* (MIMD)**

Múltiplos processadores autônomos processam simultaneamente instruções diferentes e em espaços de memória diferentes. Podemos associa-los a tecnologias de processamento distribuído tais como o OpenMP e o MPI.

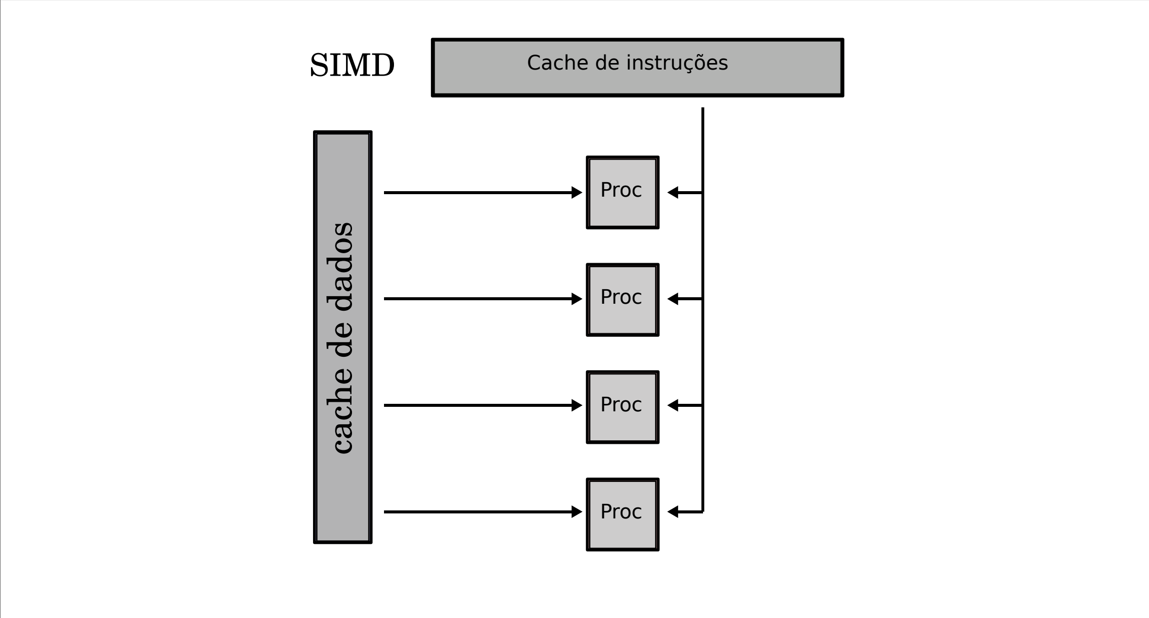

Figura 5.2: Taxionomia SIMD

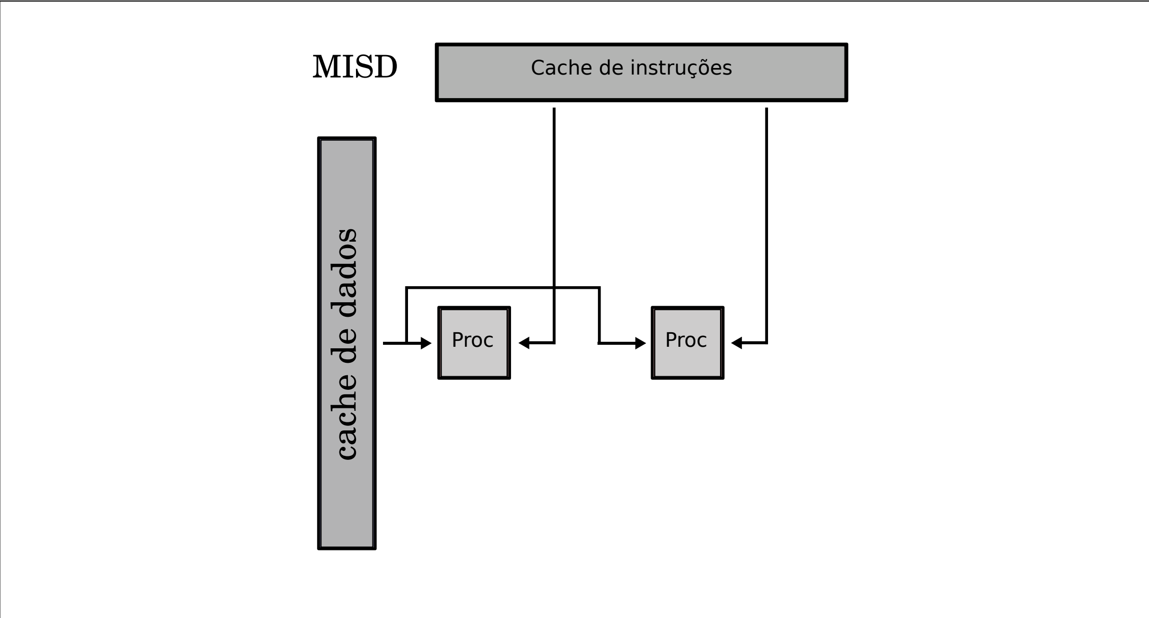

Figura 5.3: Taxionomia MISD

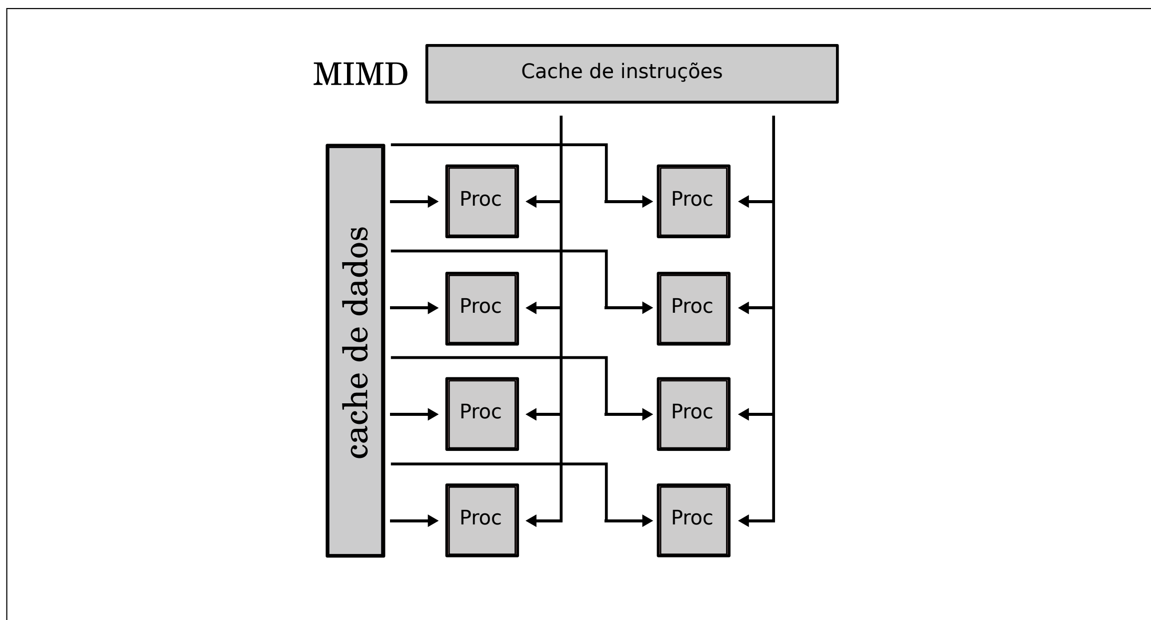

Figura 5.4: Taxionomia MIMD

5.2 A lei de Amdahl

Os processos sequenciais, de comunicação e sincronização entre diferentes sub-tarefas de um problema, são os maiores entraves no processamento de programas que buscam o paralelismo. O desempenho de um sistema que mescla processos paralelos e sequenciais é dado pela lei de Amdhal (AMDAHL (2007)).

A lei de Amdahl (5.1) é um modelo que relaciona o acréscimo de velocidade esperado em um algorítimo que possui parcelas sequenciais e paralelas, ou seja, tomando um sistema computacional onde temos N processadores e se S é o tempo dispendido na parte serial do código e P é o tempo utilizado pelo código pra executar as partes paralelas, então a lei de Amdahl nos diz que:

Algoritmos paralelos em alguns casos são mais difíceis de programar que os sequenciais pois a concorrência pela memória introduz diversos novos problemas, tal como a condição de corrida (*race conditions*); esta é uma falha produzida pelo fato de o resultado do processo ser inesperadamente dependente da sequência ou sincronia de outros eventos.

$$Speedup = \left(\frac{S+P}{S+\frac{P}{N}} \right) \qquad (5.1)$$

5.3 Matrizes esparsas e densas

As necessidades da computação científica impõe aos pesquisadores diversas limitações, sendo talvez a principal delas a utilização de memória dos computadores. A partir do final da década de 60 (CUTHILL e MCKEE (1969)), os programas que solucionavam problemas por elementos finitos (GEORGE (1971)) forçavam uma melhor utilização do escasso espaço de memória dos computadores da época. Nesse contexto surgiram os métodos de alocação de matrizes esparsas e os algorítimos a elas associados como (SAAD (1994)) e (DONGARRA *et al.* (1990)).

Podemos utilizar diversos formatos de manipulação de memória ao manipular este tipo de matriz, matrizes em forma densa (ELMROTH *et al.* (2004)) ou formatos que buscam otimizar a relação entre a esparsidade da matriz e o acesso a memória. Em (PISSANETZKY (1984)) podemos observar a descrição de alguns tipos tais como: DIA, CSR, COO e outros baseados em blocos como BCSR. Cada um deles pode apresentar um melhor desempenho computacional, que dependerá quase que exclusivamente do arranjo dos elementos não zero da matrix.

Neste trabalho foram implementados três tipos de alocação em memória, na construção da matriz de depleção foi utilizado o formato **COO**. De forma a compatibilizar a matriz de depleção aos algorítimos empregados foi realizada uma conversão para o formato **CSR**. Para fins de testes comparativos foram implementados o método de eliminação de Gauss em blocos com pivotamento (BATHE e WILSON (1976)) e sua correspondente matriz densa em blocos **Block-Dense**.

Com o intuito de ilustrar a manipulação e reserva de memória nos diversos tipos de matrizes esparsas empregadas nesta tese, apresenta-se nesta seção uma descrição baseada em exemplos como o a seguir.

$$A = \begin{pmatrix} * & * & 2 & * & 2 & * \\ * & * & * & * & * & * \\ * & 2 & -3 & 2 & * & 5 \\ * & * & * & * & -3 & * \\ 2 & 3 & * & * & * & * \\ * & 3 & 4 & 2 & * & * \\ * & * & * & 2 & 6 & -1 \end{pmatrix}.$$

Figura 5.5: Matriz exemplo.

Observe a matriz A na figura 5.5

Esta matriz possui 7 linhas , 6 colunas e 15 elementos diferentes de zeros. Ao propor um arranjo de memória para esta estrutura de dados as principais requisitos são:

- A reserva de memória para a matriz esparsa deve ser eficiente.
- O desempenho computacional do produtos matrix-matrix e matrix-vetor e da soma de matrizes deve ser adequado, uma vez que estes são os problemas clássicos em álgebra linear aplicada.
- Os métodos devem ser paralelizáveis.
- É desejavel que seja possível reduzir o esforço computacional com alguma espécie de rearranjo.

Estes requisitos são fundamentais, tanto para a construção da matrix de depleção quanto para a solução do sistema de equações diferenciais que ela representa. Na seções seguintes são brevemente descritos os modelos de alocação de memória que utilizamos nesta tese.

5.3.1 Formato BLOCK-DENSE

Neste formato é reservado espaço para todos os blocos dos elementos de uma matriz incluindo os zeros.

Como podemos observar na matriz A da figura 5.6, ela é formada por blocos de matrizes 3×3. Apesar de não ser uma forma otimizada de manipulação, pode ser útil no caso em que o algorítimo que a utiliza não é adequado para utilizar matrizes esparsas. Como

$$A = \begin{pmatrix} 1 & 2 & 0 & * & * & * & * & * & * \\ 0 & -1 & 1 & * & * & * & * & * & * \\ 0 & 2 & -3 & * & * & * & * & * & * \\ * & * & * & 1 & -3 & 1 & * & * & * \\ * & * & * & -1 & 2 & 2 & * & * & * \\ * & * & * & 2 & 0 & 0 & * & * & * \\ * & * & * & * & * & * & -1 & 0 & 4 \\ * & * & * & * & * & * & 1 & 2 & 1 \\ * & * & * & * & * & * & -1 & 0 & 2 \end{pmatrix}$$

Figura 5.6: Exemplo de matriz densa em blocos.

$$A = \begin{pmatrix} * & * & 2 & * & 3 & * \\ * & * & 1 & * & * & * \\ * & 2 & -3 & 2 & * & 5 \\ * & * & * & * & \text{-3} & * \\ 2 & 3 & * & * & * & * \\ * & 3 & 4 & 2 & * & * \\ * & * & * & 2 & 6 & -1 \end{pmatrix} \tag{5.2}$$

Exemplo de matriz esparsa.

Figura 5.7: figure

por exemplo o método de eliminação de Gauss, um método de solução direta (DUFF *et al.* (1986)) ou não iterativo que resolve o sistema de equações lineares $Ax = b$; podendo ser com ou sem pivotamento, provoca o fenômeno descrito como *fill-in* (YANNAKAKIS (1981)) e neste caso sua minimização é um problema NP completo (KUMAR *et al.* (1994)).

5.3.2 Formato COO (*Coordinated List Format*)

No formato COO a matriz esparsa é constituída por três vetores: no primeiro vetor são armazenados os valores, que podem ser números inteiros ou em ponto flutuante. Nos outros dois vetores registram-se a posição relativa ao índice dos três às informações de linha e coluna. Observe a matriz A na figura 5.7.

É uma matriz esparsa de tamanho $A_{7\times6}$, com 14 elementos diferentes de zero. Ora, em formato *denso* esta matriz ocuparia 42 posições de memória, ou seja, 336 *bytes* se lhe fosse reservada uma área em ponto flutuante de dupla precisão. Na linguagem FORTRAN (FOX *et al.* (1990)) uma tal *array* seria do tipo REAL*8, ou ainda em linguagem C

(KERNIGHAN *et al.* (1988)) um *array* do tipo *double*. Então é muito claro que a partir de um determinado nível de esparsidade o formato denso, a princípio, deve ocupar muito mais espaço em memória do que o esparso.

Pode-se determinar a partir de quanto de esparsidade deve-se utilizar o formato COO. Através do seguinte cálculo mostra-se que ao tomar uma matriz quadrada de tamanho $N \times N$ em formato de dupla precisão, necessita-se de mais memória no formato *Denso* do que no formato COO.

- Denso $(N \times N \times 8)$ *bytes*
- COO: $(N \times 8) + 2 \times (N \times 4) = (N+1) \times 8$

É trivial que $N^2 > N+1 \quad \forall N > 1$. Portanto matrizes em formato COO sempre são mais econômicas no uso de memória do que matrizes densas.

Valores: a_{ij}	2	3	1	2	-3	2	5	-3	2	3	3	4	2	2	6	-1
Row: i	1	1	2	3	3	3	3	4	5	5	6	6	6	7	7	7
Column: j	3	5	3	2	4	5	7	5	1	2	2	3	4	4	5	6
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Tabela 5.1: Vetores da matriz COO

Pode se observar na tabela 5.1 a alocação de memória da matriz A no formato COO, que basicamente corresponde a escrever a matriz original como 3 vetores, onde um deles contém os valores propriamente ditos, enquanto os outros dois são apontadores de linha e coluna. E portanto no caso da matriz A,como cada posição no vetor de linha ou coluna ocupa 4 *bytes*, o espaço de memória ocupado seria de $M = 14 \times 8\,bytes + 2 \times 14 \times 4\,bytes = 224$ *bytes*. Neste caso de exemplo a economia parece pequena, mas ao podermos trabalhar com gigantescas matrizes esparsas (DAVIS e HU (2011)) em computadores com limitações de memória, vemos os benefícios de sua utilização.

5.3.3 Formato CSR (*Compressed Sparse Row*)

O formato CSR utiliza três vetores para guardar a informação da matriz, de maneira análoga a matriz no formato COO. Contudo, no primeiro vetor são armazenados os va-

lores, que podem ser números inteiros ou em ponto flutuante. Nos outros dois vetores registram-se na posição relativa ao índice dos três as informações do índice da linha linha e coluna. Observe a matriz A na figura 5.7.

O formato CSR é um dos mais utilizados em álgebra de matrizes esparsas pelo seu poder de compressão e facilidade de uso, sendo adotada nos primórdios pelo SPARKIT (SAAD (1994)). Nos dias atuais, é um padrão em bibliotecas de álgebra linear como: BLAS (WHALEY e DONGARRA (1998)), MKL (INTEL (2007)) e cuBLAS (NVIDIA (2008a)).

Utilizando o mesmo exemplo da matriz A na figura 5.7 para o caso CSR, pode-se observar na tabela 5.2 a forma da matriz. Esta é composta por 3 vetores, um de números reais e os outros dois inteiros.

Valores: a_{ij}	2	3	1	2	-3	2	5	-3	2	3	3	4	2	2	6	-1
Column:	2	5	2	1	2	3	5	4	0	1	1	2	3	3	4	5
RowIndex:	0	2	3	7	8	10	13	16								
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Tabela 5.2: Vetores da matriz CSR

onde ,

- Os valores preenchem um vetor com os elementos diferentes de zero da matriz, podendo ter precisão simples (4 *bytes*) ou dupla (8 *bytes*). E seu indexador é descrito por dois vetores de números inteiros *RowIndex* e *Offset*. Em alguns casos podemos inclusive utilizar valores no conjunto dos números complexos, porém estes não estão inclusos no escopo desta tese.

- O vetor *Column* indica a coluna a qual pertence o valor apontado pelo indexador, ou seja, supondo que o indexador é a variável k, se $Val[k] = a_{ij}$ então $Column[k] = j$

- O vetor *RowIndex* é um vetor de inteiros que contém o inicio da linha, ou seja, se $Val[k] = a_{ij}$ então $RowIndex[i] \leq k \leq RowIndex[i+1]$. Por convenção $RowIndex[n+1] = NNZ + 1$ onde NNZ é o numero de elementos não zero de A

5.3.4 Multiplicação matriz Vetor

O problema *SpMv* (*Sparse Matrix Vector Multiplication*) consiste em encontrar o vetor $\vec{u}$ tal que $\vec{u} = A \cdot \vec{v}$ onde A é uma matriz esparsa de dimensão $n \times m$ e $\vec{v}$ é um vetor de dimensão m com o menor custo computacional possível. Este problema é de grande aplicação na solução de grandes sistemas de equações lineares e as bibliotecas de álgebra linear como a BLAS (DONGARRA *et al.* (1990)) e a LAPACK (ANDERSON *et al.* (1990)).

Multiplicação matriz-Vetor - COO

Suponha que a matriz A possua NNZ posições diferentes de zero com N linhas e M colunas. O exemplo da figura 5.8 mostra como efetuar o produto $y = Ax$ de maneira sequencial, onde x é um vetor de M posições.

```
for (int i=0; i < NNZ; ++i)
  y[Row[i]] += Val[i] * x[Column[i]];
```

Figura 5.8: Multiplicação de matriz por vetor no formato COO.

Pode-se paralelizar esta operação utilizando OPENMP colocando apenas uma indicação ao preprocessador do compilador na figura 5.9.

```
#pragma omp parallel for
  for (int i=0; i < NNZ; ++i)
    y[Row[i]] += Val[i] * x[Column[i]];
```

Figura 5.9: Multiplicação de matriz por vetor no formato COO em paralelo.

5.3.5 Multiplicação matriz Vetor - CSR

Podemos tomar como exemplo que nossa matriz A possua N linhas e M colunas em formato CSR. O código na figura 5.10 mostra como efetuar o produto $y = Ax$, onde x é um vetor de M posições de maneira sequencial.

```
  for (int i = 0; i < L; i++)
    for (int j = RowIndex[i]; j < RowIndex[i+1]; j++)
      y[i] += Val[j] * x[Column[j]];
```

Figura 5.10: Multiplicação de matriz por vetor no formato CSR

Pode-se paralelizar esta operação utilizando OPENMP colocando apenas uma indicação ao preprocessador do compilador, na figura 5.11.

```
#pragma omp parallel for
  for (int i = 0; i < L; i++)
    for (int j = RowIndex[i]; j < RowIndex[i+1]; j++)
      y[i] += Val[j] * x[Column[j]];
```

Figura 5.11: Multiplicação de matriz por vetor no formato CSR em paralelo

5.4 Outros formatos

Outros formatos de armazenamento podem ser utilizados a depender da maneira como a matriz é preenchida e dos precondicionadores do problema. Ao efetuarem-se operações de soma e multiplicação entre vetores e matrizes esparsas encontra-se o fenômeno denominado *fill-in*, que consiste no aparecimento de variáveis espúrias durante as operações em processos interativos. Como por exemplo na eliminação de Gauss do exemplo 5.4.

Exemplo

$$M \cdot \vec{e} = \begin{pmatrix} a & -1 & 1 \\ -1 & a & 0 \\ 1 & 0 & a \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \tag{5.3}$$

Escrevendo na matriz do exemplo 5.4 apenas as variáveis diferentes de zero encontra-se então,

$$M = \begin{pmatrix} * & * & * \\ * & * & 0 \\ * & 0 & * \end{pmatrix}$$

onde os elementos diferentes de zero são marcados com $(*)$ e os que apresentam *fill-in* por $\boxed{*}$. Pode-se observar os passos da eliminação de Gauss abaixo :

$$\begin{pmatrix} * & * & * \\ * & * & 0 \\ * & 0 & * \end{pmatrix} \xrightarrow{1^{a}} \begin{pmatrix} * & * & * \\ 0 & * & \boxed{*} \\ * & \boxed{*} & * \end{pmatrix} \xrightarrow{2^{a}} \begin{pmatrix} * & * & * \\ 0 & * & \boxed{*} \\ 0 & 0 & * \end{pmatrix} \xrightarrow{3^{a}} \begin{pmatrix} * & * & * \\ 0 & * & 0 \\ 0 & 0 & * \end{pmatrix} \xrightarrow{4^{a}} \begin{pmatrix} * & 0 & 0 \\ 0 & * & 0 \\ 0 & 0 & * \end{pmatrix}$$

Vários algoritmos podem ser utilizados no reordenamento da matriz de maneira a minimizar o *fill-in* (CUTHILL (1972)). Além disso podemos adotar modelos de arranjo da memória de maneira a reduzir o *fill-in* e o esforço computacional (SAAD (1994)), seguem como exemplo abaixo alguns:

- DIA : *Diagonal format*.
- ELL : *Ellpack-Itpack generalized diagonal format*.
- JAD : *Jagged Diagonal format*.
- HYB : *COO mixed DIA format*.
- VBR : *Variable Block Row format*.
- SSS : *Symmetric Sparse Skyline format*.

Este trabalho limitou-se a implementar as classes e rotinas relativos aos formatos CSR, COO e DENSE. Na seção seguinte será abordada de maneira teórica a construção do operador de depleção.

5.5 Construção do operador de depleção por soma direta

Uma definição geral do operador em termos de sub-espaços vetoriais pode ser observada na definição (5.1).

Definição 5.1 *Sejam $\Omega_1, \Omega_2, ..., \Omega_m$ elementos de um sub-espaço vetorial $\hat{\Omega}$ sobre um corpo $\mathbb{K}$. Define-se o operador soma direta destes sub-espaços da seguinte forma: $\oplus_{i=1}^{m} = \Omega_1 \oplus \Omega_2 \oplus ... \oplus \Omega_m = \hat{\Omega}$. Este representa a soma dos sub-espaços $\Omega_1, \Omega_2, ..., \Omega_m$ na qual cada elemento em $\hat{\Omega}$ é escrito de forma única como uma combinação linear $\omega_1 + \omega_2 + ... + \omega_m$ onde $\omega_i \in \Omega_i$ onde $i = 1, 2, \cdots, m$, i.e., os sub-espaços Ω_i formam uma base do espaço $\hat{\Omega}$.*

Da definição podemos escrever o seguinte teorema:

Teorema 5.2 *Se Ω_1 e Ω_2 são sub-espaços de um espaço vetorial $\hat{\Omega}$ sobre um corpo $\mathbb{K}$, então $\Omega_1 + \Omega_2 = \Omega_1 \oplus \Omega_2 = \hat{\Omega}$ se e somente se $\Omega_1 \cap \Omega_2 = \varnothing$.*

Assim, se Ω_1 e Ω_2 são sub-espaços de um espaço vetorial $\hat{\Omega}$ a soma direta direta dos dois existe apenas quando o conjunto interseção entre eles é vazio. Utilizando a hipótese de indução chega-se ao corolário 5.3.

Corolário 5.3 *Se $\Omega_1, \Omega_2, ..., \Omega_m$ são sub-espaços de um espaço vetorial $\hat{\Omega}$ sobre um corpo $\mathbb{K}$, então $\sum_{i=1}^{m} \Omega_i = \oplus_{i=1}^{m} \Omega_i = \hat{\Omega}$ se e somente se $\Omega_i \cap \left(\sum_{j \neq i} \Omega_j = \varnothing\right)$ onde $i = 1, 2, \cdots, m$.*

Ora, em nossa proposta o operador *soma direta* é utilizado em um espaço não comutativo. No espaço das matrizes, no qual qualquer par de matrizes A de ordem $m \times n$ e B de ordem $p \times q$ é transformado em uma matriz de ordem $m + p \times n + q$ definida como:

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} = \begin{pmatrix} A_{11} & \cdots & A_{1n} & 0 & \cdots & 0 \\ \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ A_{m1} & \cdots & A_{mn} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & B_{11} & \cdots & B_{1q} \\ \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ 0 & \cdots & 0 & B_{p1} & \cdots & B_{pq} \end{pmatrix} \tag{5.4}$$

Neste caso o operador $\oplus$ produz um tipo especial de matriz por blocos, em nosso caso, cada sub-matriz da soma direta representa o problema de cálculo da depleção em cada nodo do reator . Esta é uma matriz quadrada e diagonal por blocos, onde cada bloco da matriz tem o aspecto na figura 5.12 e os pontos $*$ representam os elementos não zero.

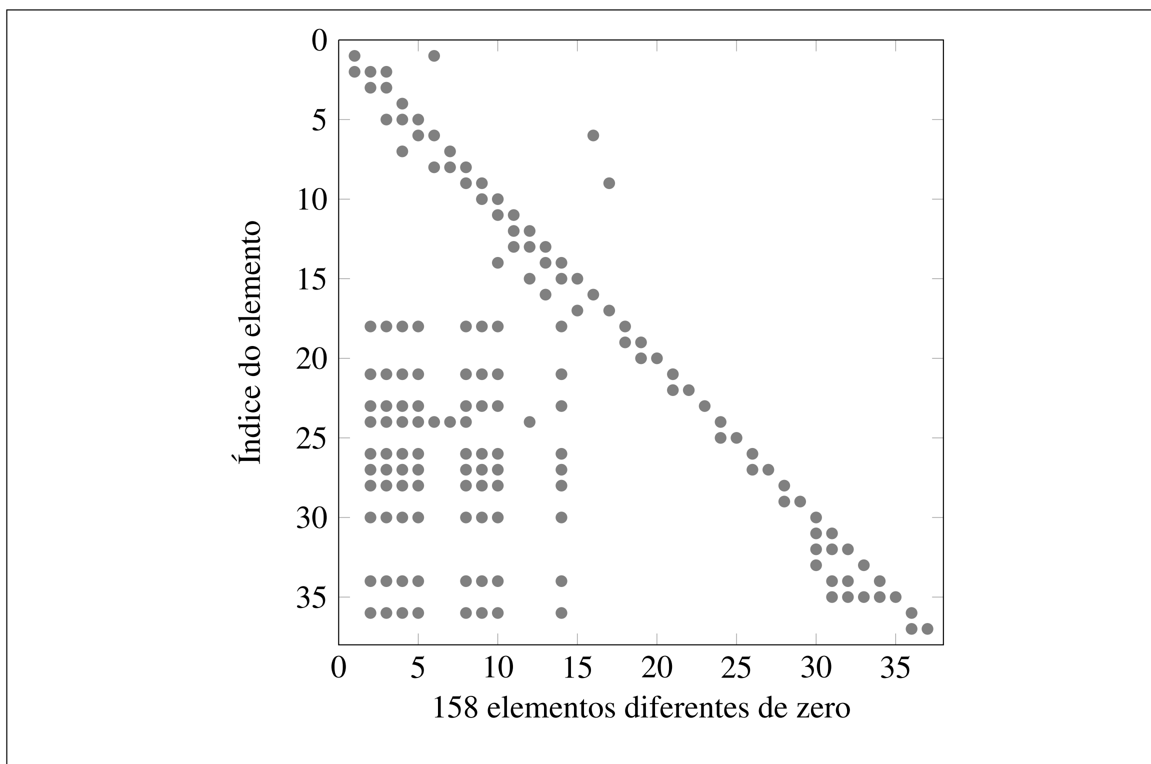

Figura 5.12: Matriz do CNFR - 1 nodo

De modo análogo utiliza-se o operador $\oplus$ para construir uma matrizes de blocos que representa os nodos do núcleo do reator. Na figura 5.13 podemos observar a matriz composta por 8 nodos onde os pontos representam os elementos não zero.

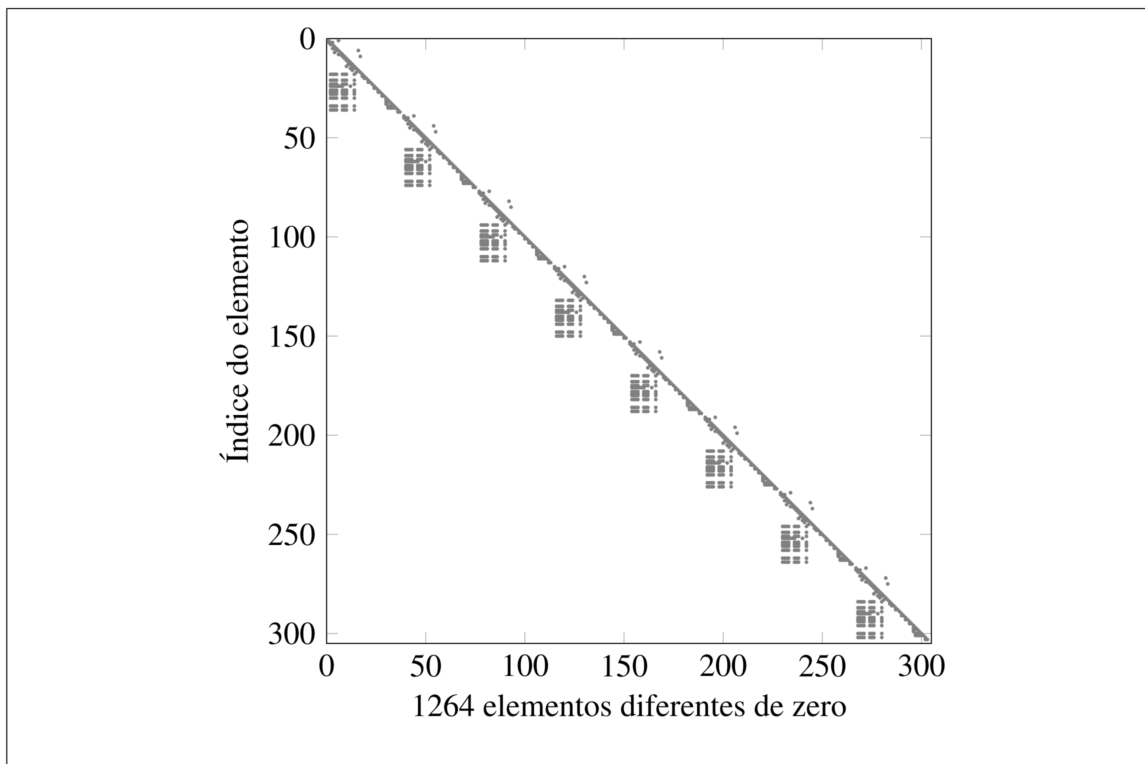

Figura 5.13: Matriz do CNFR - 8 nodos

Do fato de o nosso modelo de cálculo do problema da depleção ser semi-estático e como cada matriz de blocos forma um conjunto linearmente independente, ou seja, o sistema de equações de cada nodo é desacoplado um do outro, temos a base do modelo de paralelismo implementado nesta tese. Utilizam-se operações otimizadas de multiplicação e adição aplicadas a grandes matrizes e vetores, a partir da construção de um operador no qual cada nodo é independente. Assim, podemos explorar o paralelismo tanto da CPU, através do OPENMP (CHANDRA (2001)), quanto da GPU, ao construir um operador matricial que manipula uma grande matriz esparsa.

A matriz de depleção do problema sob teste neste trabalho é constituída por 2988 nodos de combustível e a matriz associada ao operador é constituída por uma matriz com mais de 107 mil linhas e aproximadamente 5 milhões de posições diferentes de zero. Estes números podem variar de acordo com a geometria do problema e com as concentrações de actinídeos, que por sua vez influenciam as concentrações dos produtos de fissão. Por esta razão o número de elementos diferentes de zero em cada nodo pode variar em função de sua posição geométrica e assim deve-se construir um novo operador de depleção a cada passo de queima do combustível. Por outro lado, como as matrizes de depleção de

cada nó, assim como a matriz construída através da *soma direta*, são matrizes esparsas constituídas por um grupo de três vetores. Como podemos observar anteriormente, a reserva de memória e sua construção dependem de índices que são obtidos de maneira recursiva se tornando um problema para o paralelismo. A solução encontrada consiste na construção da matriz em dois passos, no primeiro determinam-se, para cada nodo do combustível, os seguintes parâmetros:

- O número total de elementos não zero.
- O número de actinídeos.
- O número de produtos de fissão.
- Os indexadores de posição inicial e final de cada matrix-nodo.

Os parâmetro por nodo necessários a construção da matriz associada ao operador de depleção são: os parâmetros obtidos no primeiro passo, o fluxo de nêutrons por nodo, as concentrações de cada nuclídeo por nodo, as seções de choque (fissão, captura e $(n,2n)$) homogeneizadas por nodo e as constantes de decaimento.

Nas próximas seções serão abordados ambientes de preprocessamento paralelo utilizados nesta tese. O primeiro é denominado OpenMP (CHANDRA (2001)) e possui uma biblioteca e através de uma série de diretrizes de pré-processamento, que devem ser inseridas no código fonte, indica ao compilador as áreas de processamento paralelo

5.6 OpenMP

O OPENMP é um modelo de paralelismo baseado em divisão de tarefas, onde o processo principal, que denomina-se *master thread*, executa chamadas de sub-processos escravos e a tarefa a ser paralelizada é dividida entre eles. A este sub-processos denominam-se *threads* e são executados em uma fila que depende do números de núcleos de processamento presentes no sistema. Deve-se ressaltar que o ideal é que o números de *threads* em execução deve ser menor ou igual ao número de núcleos de processamento.

O modo de informar ao compilador que determinada parte do código foi marcada para funcionar em paralelo é realizada por diretrizes; em C ou C++ por exemplo, temos a instrução **#pragma**. O pré-processador então criará para cada *thread* uma seção de código de máquina que será distribuída para cada núcleo de processamento durante a execução do programa.

Cada um dos *threads* possui uma codificação de endereço, ou *id*. Este endereço pode ser obtido através de uma chamada de função e retorna um número inteiro. O *master thread* possui o endereço 0. Após a execução de cada *thread* o *master thread* assume o controle e continua o programa.

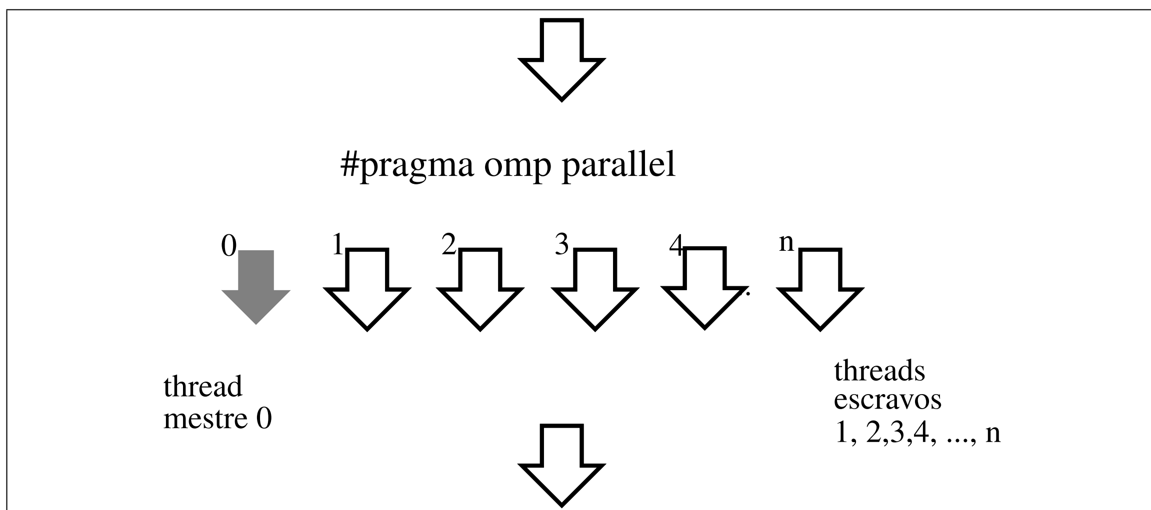

Figura 5.14: Distribuição de tarefas em OpenMP

O código foi implementado utilizando o padrão do OPENMP em todas as rotinas de manipulação de vetores e matrizes que utilizam a CPU, a adoção deste modelo de programação paralela permite tanto paralelismo de tarefas quanto paralelismo de dados, bem como a definição de área de memória compartilhada entre determinadas tarefas.

Deve-se ressaltar que o OpenMP se destina a utilização em topologias de memória compartilhada ou MIMD, neste caso os núcleos do processador devem compartilhar a memória fisicamente. Difere do MPI (GROPP *et al.* (1999)) por este ser mais utilizado em topologias de memória distribuída, ou seja, nem todos os núcleos de processamento pertencem ao mesmo *hardware físico*. Este paradigma permite a construção de *clusters*

de computadores com centenas de milhares de processadores como no supercomputador JAGUAR (BLAND *et al.* (2009)).

Uma diretiva é uma linha de código característica e deve fornecer informação adicional ao compilador. Basicamente o OPENMP distribui a execução do código entre os processadores através de diretivas e associadas á uma sentinela (*pragma*), que informam ao compilador de que forma este deve distribuir entre os *threads* da máquina.

5.6.1 Exemplos

Suponha que o programa principal, escrito em C, tenha como objetivo executar alguns procedimentos puramente sequenciais como **IF-THEN-ELSE** e outros que podem ser paralelizados como **FOR-THEN** e **DO-WHILE** como visto na figura 5.15.

```
int main{int argc, char *argv[] }{
// sequential
  for(int i=0;i < 1000;i++){
    if(i % 2)
      faca_algo_1(); // Se for par
    else
      faca_algo_2(); // Se for impar
      }
  }
// paralelo
#pragma omp parallel
  for(int i=0;i < 1000;i++){
    if(i % 2)
      faca_algo_1(); // Se for par
    else
      faca_algo_2(); // Se for impar
    }
}
```

Figura 5.15: OPENMP - Exemplo 1

Neste caso o escopo das funções dentro do laço são inteiramente privados, tornando-se necessário trazer o resultado de volta ao escopo do código principal.

No exemplo da figura 5.16 o laço do **FOR** é distribuído entre os *threads* do processador e cada um deles executa a instrução **if(i % 2) ... else ...**. Neste caso, a variável *i* é considerada automaticamente privada.

O OPENMP utiliza a mesma infra estrutura de *threads* do sistema operacional, as operações com *threads* incluem a criação, finalização, sincronização, agendamento, sinalização e interação com o processo *pai*. Por outro lado, os *threads* de um mesmo processo

```
#include <cmath>
#define SIZE 128
int main(){
    double cos_table[SIZE];
#pragma omp parallel for
    for(int i=0; i < SIZE; ++i)
      cos_table[i] = cos(2 * 3.1415 * (double)(i/SIZE));
}
```

Figura 5.16: OPENMP - Exemplo 2

compartilham os descritores de arquivos abertos, os gerenciadores de sinalização, as instruções, os dados e as variáveis de ambiente. Cada *thread* de um processo possui seu próprio conjuntos de registros e *stack pointer*, um *stack* para variáveis locais, ajuste de prioridade e retorna zero se não houver erro na execução.

```
#include <omp.h>
#include <stdio.h>
int main(){
  int x = 2;
// Apenas 2 threads
#pragma omp parallel num_threads(2) shared(x){
    if (omp_get_thread_num() == 0) // Se for o master
x = 8;
    else
printf("Passo 1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
#pragma omp barrier
    if (omp_get_thread_num() == 0)
      printf("Passo 2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
    else
      printf("Passo 3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  }
  return 0;
}
```

Figura 5.17: OpenMP, Exemplo 03

Na figura 5.17, pode-se observar o mecanismo denominado *condição de corrida*, é caracterizado por uma falha na programação. Neste caso dois processos independentes são criados e compartilham uma variável x, o primeiro processo tenta incrementar x enquanto o segundo espera mostra-lo na tela. Pode-se controlar este tipo de problema utilizando mecanismos de sincronização e barreira, que interrompem o paralelismo.

Nosso objetivo nesta seção é mostrar conceitualmente alguns poucos detalhes da programação em OPENMP na forma de exemplos de código comentado, uma vez que este trabalho não tem o objetivo de ser um tutorial sobre o assunto. Para um detalhamento minucioso desta ferramenta existe farta documentação (CHANDRA (2001)).

5.7 Computação de alto desempenho em GPU

O ambiente CUDA é basicamente uma arquitetura de *hardware* e *software* que possibilita que as placas de vídeo da NVIDIA executem programas escritos em C, C++, FORTRAN e etc. Um programa que utiliza este ambiente executa, em sua camada mais interna, chamadas em paralelo a funções denominadas *kernels*, e estas são sub-divididas em processos denominados *threads*. O modo como o programador particiona o problema é fundamental na programação que utiliza este paradigma, visto que a execução destes *threads* é organizada em blocos e em grades de blocos (KIRK (2007)), Observa-se que as figuras 5.18 e 5.20 foram obtidas em (KIRK (2007)).

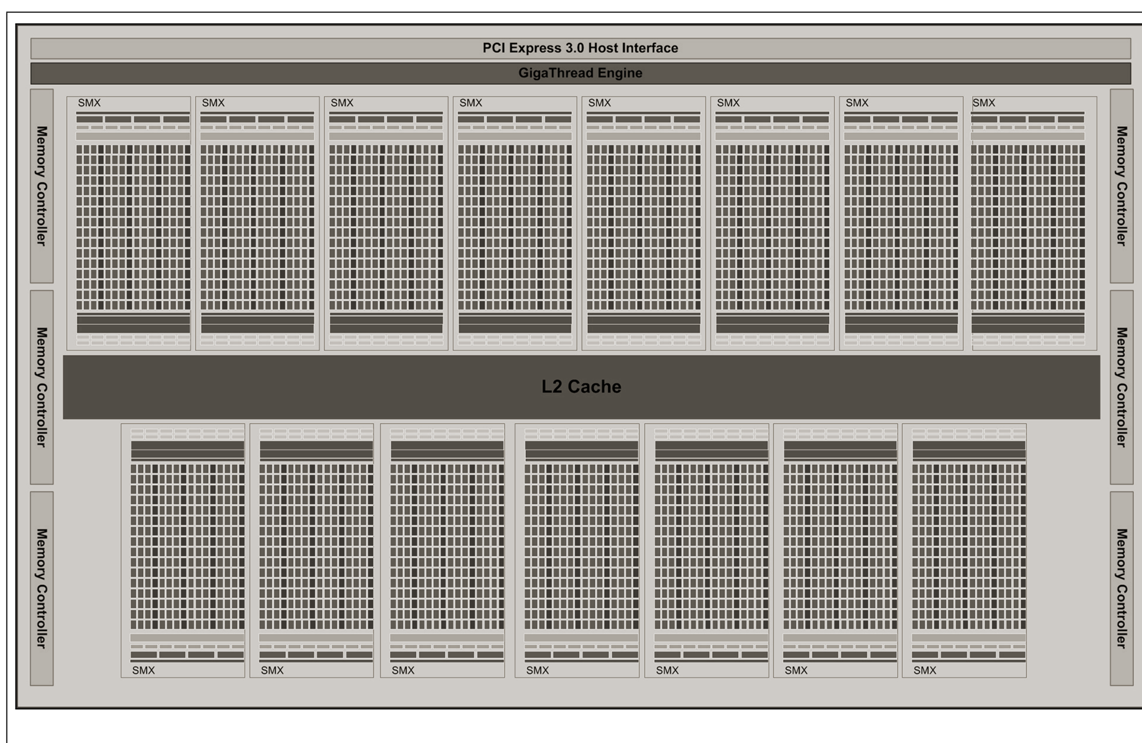

Figura 5.18: Diagrama de blocos da GPU Kepler

Uma GPU moderna da NVIDIA, baseada em uma arquitetura denominada *Kepler* (NVIDIA (2012)) possui em torno de 7 bilhões de transistores e um poder computacional de 1.3 TFlops de pico em dupla precisão e 5 TFlops em precisão simples. Na figura 5.18 podemos observar o seu diagrama de blocos.

5.7.1 *Processing units*

Uma GPU nada mais é do que um multi-processador constituído por um grande número de *Streaming Multiprocessors* (SMs). Onde cada SM é máquina com múltiplos processadores como visto na figura 5.19.

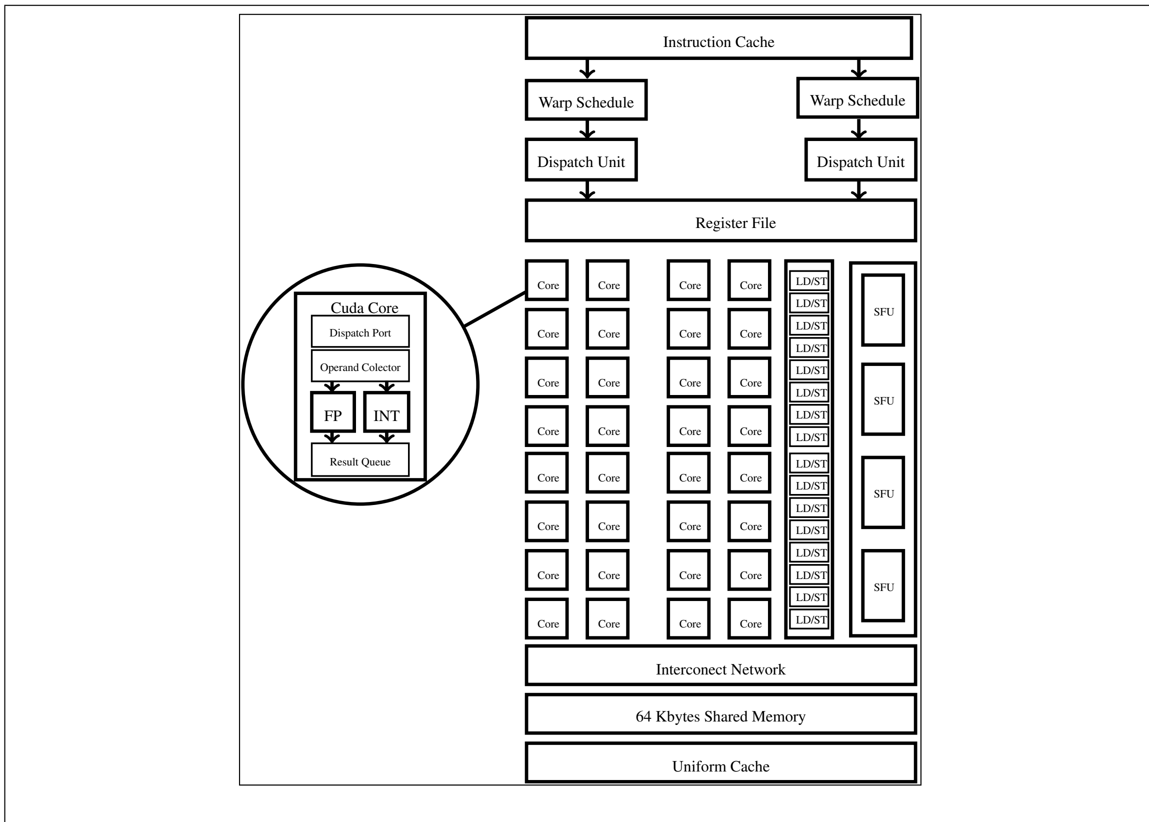

Figura 5.19: Diagrama de um *Stream Processor*

Cada SM é composta por várias unidades de *stream processors*(SPs) que são na realidade núcleos de processamento individuais orientados a *threads* ou blocos de código executável. Esta arquitetura impõe que dois processos ou *threads* localizados em diferentes SMs não podem ser sincronizados por um semáforo comum. De modo que a sincronização deve ser realizada pelas SMs. Se por um lado isto parece um desvantagem, a independência dos processos pode ser benéfica no sentido de que a aceleração pode ser melhor implementada em *hardware*.

5.7.2 A arquitetura SIMT

Ao escrever uma aplicação em CUDA devem-se particionar os processos ou *threads* em blocos de forma a que o *hardware* ao executa-lo circunscreva um bloco inteiro a uma simples SM, apesar de vários blocos poderem executar na mesma. O processador então divide cada bloco em unidades denominadas *warps*, composta por 32 *threads*. Deve-se compreender então este processo durante a escrita do código de modo a tirar vantagem desta característica. Nesta topologia, todos os *threads* de um *warp* executam juntos o mesmo código, isto é, durante o *fetch* de instruções da máquina a mesma instrução é distribuída. Podem ocorrer duas situações neste caso no ciclo de execução: cada *thread* executa uma instrução em particular ou nada faz (*nope*). A situação de *nope* ocorre quando existe divergências no *thread*, por exemplo **IF THEN ELSE**.

A taxionomia de Flynn caracteriza esta topologia como SIMD (*Single Instruction Multiple Data*) como foi visto em 5.1, ou ainda mais recentemente como SIMT (*Single Instruction Multiple Thread*).

5.7.3 As *Stream Machine eXtended*

A partir da arquitetura Kepler GK110 (NVIDIA (2012)) as *stream machines* foram aperfeiçoadas a um novo nível de desempenho e denominadas *Stream Machine eXtended* (SMXs). Cada SMX possui 192 núcleos CUDA de precisão simples, 64 unidades de dupla precisão, 32 *Special Function Units* (SFU), e 32 *Load/Store Units* (LD/ST). Podemos ver na figura 5.20 um diagrama.

Cada núcleo ou *Core*, possui uma unidade de ponto flutuante e uma unidade lógica aritmética de inteiros totalmente *pipelined*, ou seja, ela permite que a *Core* realize a busca de uma ou mais instruções além da próxima a ser executada. Além disso implementa completamente a norma IEEE 754-2008 (WHITEHEAD e FIT-FLOREA (2011)) e operações *Fused Multiply-Add* (FMA). Um dos principais ganhos da nova plataforma é que houve um incremento significativo da potência computacional em dupla precisão.

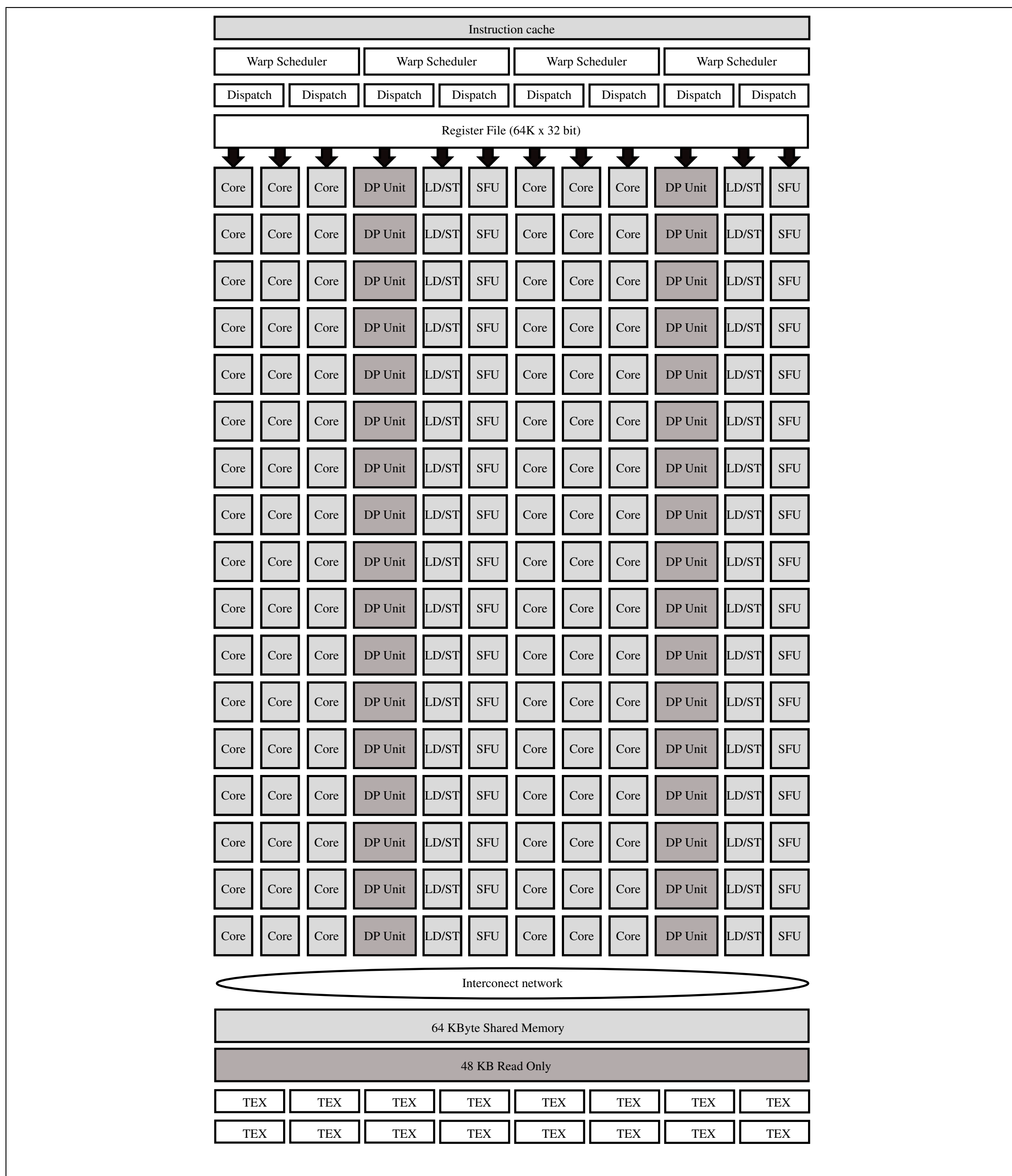

Figura 5.20: Diagrama de um *Stream MultiProcessor eXtended*

5.7.4 Estrutura de memória

A arquitetura de *hardware* desenvolvida nas GPUs é totalmente dedicada a aumentar o desempenho de aplicações que sejam intrinsecamente paralelizáveis. E de certa forma esta opção pode ser observado no modo como a memória é gerenciada. As GPUs possuem vários tipos de memória onde podemos destacar as do tipo *GLOBAL* e as do tipo *SHARED*.

O acesso as memórias do tipo *SHARED* garante a todos os *threads* em uma SM o compartilhamento de uma variável ou array definido como tal. Seu tempo de acesso é de 100 a 200 vezes mais veloz do que o acesso a memória *Global*, pois esta se localiza no *core* do processador da GPU. Por outro lado, possui apenas 32K ou 64 Kbytes nas novas GPUs com chipset Kepler (NVIDIA (2012)). Na figura 5.21se observa um fragmento de código em C que exemplifica a reserva de memória deste tipo:

```
__shared__ double vecD[1024]; // size = 1024*8 bytes = 8192 bytes
__shared__ float  vecF[1024]; // size = 1024*4 bytes = 4096 bytes
volatile __shared__ char vecC[1024]; // size = 1024*1 bytes = 1024 bytes
```

Figura 5.21: Declaração de variável do tipo *SHARED*

Observa-se que ao declarar uma variável ou *array* do tipo *SHARED* como volátil o compilador é livre para otimizar as posições de memória *SHARED* localizando seu apontador na memória de registro, esta deve ser a declaração padrão.

A memória do tipo *Global* é a memória geral da GPU e pode ser acessada por todos os *threads* de uma determinada aplicação. Atualmente existem GPUs com até 12 Gbytes de memória *Global*, contudo a velocidade de acesso é menor do que as memórias *shared* e único acesso a memória pode levar até centenas de *clocks* do processador.

As memória do tipo *Constant* são uma memória de apenas leitura acessível apenas a GPU e contém valores estáticos definidos no código em tempo de compilação.

A memória do tipo *Texture* é similar a memória de constantes e também é uma memória de apenas leitura, contudo pode apontar para uma posição de memoria *Global* e incrementar a performance da aplicação em certas circunstancias. Por exemplo a memória de textura pode ser utilizada para fazer *cache* no caso de uma multiplicação Matriz-Vetor $y = A * x$ onde x é o vetor sob *cache*, na figura 5.22 podemos observar o código necessário para apontar um determinado vetor (int, float ou double) para a memória de texturas.

A memória do tipo *Register* encontram-se fisicamente nas SMs. Cada GPU possui uma memória de registros de 32K *words* de 32 *bits* que são muito mais rápidos do que a memória *SHARED*. O compilador normalmente guarda as variáveis locais nos registros,

```
texture<float,1> tex_x_float;
texture<int2,1>  tex_x_double;
inline void bind_x(const float * x){
    size_t offset = size_t(-1);
    cudaBindTexture(&offset, tex_x_float, x);
}
// Use int2 to pull doubles through texture cache
inline void bind_x(const double * x){
    size_t offset = size_t(-1);
    cudaBindTexture(&offset, tex_x_double, x);
}
// Note: x is unused, but distinguishes the two unbind functions
inline void unbind_x(const float * x){
    cudaUnbindTexture(tex_x_float);
}
inline void unbind_x(const double * x){
    cudaUnbindTexture(tex_x_double);
}
__inline__ __device__ float fetch_x(const int& i, const float *x){
#ifdef UseCache
    return tex1Dfetch(tex_x_float, i);
#else
 return x[i];
#endif
}
__inline__ __device__ double fetch_x(const int& i, const double *x){
#ifdef UseCache
    int2 v = tex1Dfetch(tex_x_double, i);
    return __hiloint2double(v.y, v.x);
#else
    return inf; // Fail
#endif
}
```

Figura 5.22: Acesso a memória de texturas

com exceção de vetores, que podem ter índices como variáveis ou vetores muito grandes, como por exemplo na figura 5.23.

```
int a[20000], index;
index = 12000;
x = a[index]; // Falha
...
...
x = a[5];     // OK
```

Figura 5.23: Falha de indexação na compilação.

Como os registros não podem ser indexados por *hardware* o compilador não permite a alocação do vetor. Por outro lado, o compilador aceita o código para índices fixos.

A manipulação do fluxo de dados entre a GPU e CPU até a versão 6.0 do CUDA é baseada em reservar e guardar primeiramente os dados na CPU, copia-los para a memória *GLOBAL* da GPU, executar o programa *kernel* e então copia-los de volta a CPU. A movimentação entre *Host* e *Device* é realizada pela função *cudaMemcpy*, e faz uma copia entre a memória GLOBAL da GPU e a CPU, na figura 5.24.

```
int main(){
    const unsigned int N = 1024*1024*16; // 16M
    const unsigned int bytes = N * sizeof(int);
    int *host = (int*)malloc(bytes);
    int *device;
    cudaMalloc((int**)&device, bytes);
    memset(host, 0, bytes);
    cudaMemcpy(device, host, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(host, device, bytes, cudaMemcpyDeviceToHost);
    free(host);
    cudafree(device);
    return 0
}
```

Figura 5.24: Alocação de memória da GPU

A partir da versão 6.0 o CUDA disponibiliza um método no qual o espaço de endereçamento de memória é único chamado *Unified Memory Access* (UMA) e assim pode prescindir destes passos intermediários, se utilizando de uma extensão de memória realizada pelo *driver* da GPU. Todos os processadores envolvidos no UMA compartilha a memória uniformemente mas com tempos de acesso diferentes, de maneira similar a arquitetura NUMA (*Non Uniform Memory Access*) (NIEPLOCHA *et al.* (1996)).

Por outro lado, exite uma outra forma de reserva de memória denominada *pinned*, fazendo com que alocação de memória seja realizada pelo próprio *driver* da GPU na memória da CPU, e desta forma pode-se dispensar a copia entre *host*(CPU) e *device*(GPU) como vêm na figura 5.25.

```
int main()
{
  unsigned int N = 4*1024*1024;
  const unsigned int bytes = N * sizeof(double);
// host arrays
  double *h_aPageable;
  double *h_aPinned;
// allocate and initialize
  h_aPageable = (double*)malloc(bytes);      // host pageable
  cudaMallocHost((void**)&h_aPinned, bytes); // host pinned
  memcpy(h_aPinned, h_aPageable, bytes);
  // cleanup
  cudaFreeHost(h_aPinned);
  free(h_aPageable);
  return 0;
}
```

Figura 5.25: Alocação de memória *pinned*

5.7.5 Sincronização e barreiras

Ora, lembrando que centenas de *threads* executando em paralelo o mesmo código em algum momento tem de ser sincronizados. A gerência de conformidade é executada pela

função **__synchthreads();**. Outras formas de fazer a gerência dos *threads* no programa é utilizar os comandos **__threadfence**,

- **__threadfence_block();**: Determina que todos os *threads* atendam a sincronização no bloco.
- **__threadfence();**: Determina que todos os *threads* atendam a sincronização no *device*.
- **__threadfence_system();**: Determina que todos os *threads* atendam a sincronização no *device* e no *host*.

5.7.6 Extensões ao C/C++ do CUDA

A declaração de funções obedece a três tipos básicos:

- **__host__** : As funções deste tipo executam unicamente na CPU (*host*).
- **__global__** : As funções de *kernel*, ou seja as que serão distribuídas nos SMs das GPUs, são chamadas pelo *host* e executam no *device* (GPU).
- **__device__** : As funções que executam no *device* e são chamadas apenas por funções do tipo **__global__**.

De forma análoga a declaração das variáveis de programa :

- **__global__**
- **__device__**
- **__constant__**
- **dim3**
- **__shared__**

As variáveis do tipo **__shared__** tem seu conteúdo compartilhado entre todos os *threads* de um bloco. As variáveis definidas como **__constant__** são de apenas leitura, e

apenas pela GPU, sendo definidas em tempo de compilação. E as do tipo **dim3** definem a forma de organização dos *threads*, blocos e *grids*, por exemplo :

Ao especificarmos o tamanho do *grid*, ou seja, o número de linhas e colunas de blocos que compões um *grid* e o tamanho do bloco. O tamanho de um bloco é dado pelo número de linhas e colunas e *layers* de *threads* que compõe um bloco. Observe o código na figura 5.26.

```
dim3 dimGrid(n,1);
dim3 dimBlock(1,1,1);
SumAdd<<<dimGrid,dimBlock>>>(a,b,c);
```

Figura 5.26: Declaração de grid

Como podemos observar na figura 5.26, a função do tipo **__global__ SumAdd** é distribuída em um *grid*, que consiste de $n \times 1$ blocos e cada bloco possui apenas $1 \times 1 \times 1)$ *threads*.

5.7.7 *Grids*, blocos, *threads* e indexadores

O acesso aos *grids* e blocos de *threads* deve ser acessível por algum tipo de índice, ou ainda, como dizemos ao processador que um determinado processo (*thread*) deve acessar a posição $a_{i,j}$ de uma matriz ? Para tanto devem-se seguir alguns passos:

- Definir um determinado número de blocos (*blocksPerGrid*) e de processos por bloco (*threadsPerBlock*).

- Na chamada da função *kernel* que executará na GPU utiliza-se o operador

  func«< blocksPerGrid, threadsPerBlock»>(Parameters ...);

  onde **func** é a função *kernel*.

- No escopo da função **func** definimos o indexador através de uma aritmética simples.

Esta é uma maneira de proceder com a decomposição de domínio do problema, estratégia que é definida por alguns pesquisadores como *divide and conquer* (BENTLEY (1980)), e é claro não resolve o problema a ser atacado *per si*. Seguem então alguns tipos de indexadores possíveis.

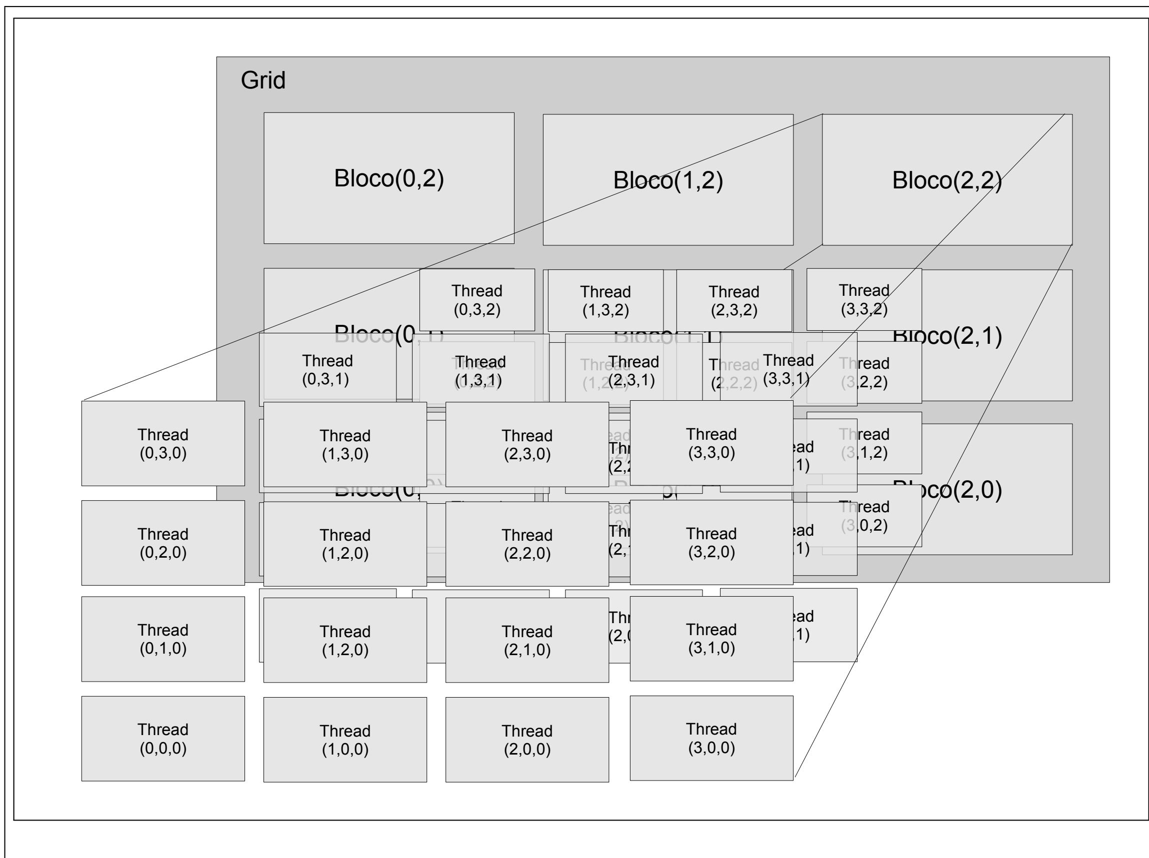

Figura 5.27: Grade e blocos de *threads*.

O indexador 1D

Tomemos como exemplo um array unidimensional de blocos:

- onde cada bloco tem um array de threads unidimensional

```
int blocksPerGrid, threadsPerBlock;
func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);
...
__global__ func(Parameters ...){
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x + threadIdx.x;
```

Figura 5.28: Array uni-dimensional de blocos 1D.

- onde cada bloco tem um array de threads bidimensional

- onde cada bloco tem um array de threads tridimensional

```
int blocksPerGrid;
dim3 threadsPerBlock(BLOCKX,BLOCKY);
...
__global__ func(Parameters ...){
func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);

UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y
     + threadIdx.y * blockDim.x + threadIdx.x;
```

Figura 5.29: Array uni-dimensional de blocos 2D.

```
int blocksPerGrid;
dim3 threadsPerBlock(BLOCKX,BLOCKY,BLOCKZ);
func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);
...
__global__ func(Parameters ...){
UniqueBockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z
     + threadIdx.z * blockDim.y * blockDim.x
     + threadIdx.y * blockDim.x + threadIdx.x;
```

Figura 5.30: Array uni-dimensional de blocos 3D.

Um indexador único de blocos 2D

- Um array bidimensional de blocos onde cada bloco tem um array de *threads* unidimensional.

```
int threadsPerBlock;
dim3 blocksPerGrid(BPG,BPG);
func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);
...
__global__ func(Parameters ...){
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.x + threadIdx.x;
```

Figura 5.31: Array bi-dimensional de blocos 1D.

- Um array bidimensional de blocos onde cada bloco tem um array de *threads* bidimensional

```
dim3 blocksPerGrid(BPG,BPG);
dim3 threadsPerBlock(BLOCKX,BLOCKY);
func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);
...
__global__ func(Parameters ...){
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.y * blockDim.x
     + threadIdx.y * blockDim.x + threadIdx.x;
```

Figura 5.32: Array bi-dimensional de blocos 2D.

- E também um array bidimensional de blocos onde cada bloco é constituído por um array de *threads* tridimensional, então:

```
dim3 blocksPerGrid(BPG,BPG);
dim3 threadsPerBlock(BLOCKX,BLOCKY,BLOCKZ);
func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);
...
__global__ func(Parameters ...){
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.z * blockDim.y * blockDim.x
     + threadIdx.z * blockDim.y * blockDim.z
     + threadIdx.y * blockDim.x + threadIdx.x;
```

Figura 5.33: Array bi-dimensional de blocos 3D.

5.7.8 Warps

Nesta seção sera feita uma pequena introdução ao modelo de execução adotado pelo CUDA. Serão discutidos como os *threads* são agrupados em blocos de *threads*, e como são destinados aos multiprocessadores no dispositivo. Deve-se deixar claro que as funções do tipo **__global__** executam unicamente na GPU. Em sua chamada especificam a distribuição da mesma em *grids*,*blocks* e *threads*, uma chamada tipica pode ser vista na figura 5.34.

```
dim3 blocksPerGrid(BPG,BPG);
dim3 threadsPerBlock(BLOCKX,BLOCKY);

func<<< blocksPerGrid, threadsPerBlock>>>(Parameters ...);

__global__ func(Parameters ...){

  // do somethink
}
```

Figura 5.34: Declaração de função global no CUDA.

Chamam-se de *warps* a um agrupamento em paralelo de *threads*. Durante a execução de uma função **__global__**, ocorre um agrupamento, ou *schedule*, dos *threads* em *warps* no qual os multiprocessadores na GPU executam instruções de cada *warp* em modo SIMD *Single Instruction Multiple Data*. O tamanho do *warp*, que efetivamente corresponde ao tamanho da SIMD, nas GPUs atuais é de 32 *threads*.

Os multiprocessadores ou *Stream MultiProcessors* das GPUs como a FERMI, possuem dois *schedulers* de *warps* e dois *instruction dispatch units* possibilitando a execução de dois *warps* concorrentemente. O modelo KEPLER o possui quatro *schedulers* de *warps* e dois *instruction dispatch units* possibilitando a execução de duas instruções independentes por *warp* a cada ciclo de máquina. Por outro lado, um excesso de cores pode ser utilizado para o agendamento de mais de um *warp* simultâneo.

Observe que o *scheduler* agenda *warps* apenas da mesma função *kernel*. Além disso cada bloco de *threads* é fixado a uma determinada *Stream MultiProcessor*, o que por sua vez implica na maneira com é feito o acesso a memória compartilhada, ou *shared memory*. Isso tem implicações fundamentais na maneira como são implementadas as rotinas de *kernel* (VOLKOV (2010)).

5.7.9 Coalescência e Conflitos no Acesso a memória Global

Nas seções anteriores foram discutidas as diversas maneiras de implementar a execução dos *threads* e mostrou-se que regiões diferentes de memória na GPU possuem tempos de acesso muito discrepantes entre si, o que retorna a questão de como otimizar a gerência da memória para melhorar o desempenho global do código. Um dos fatores principais é a coalescência no acesso a memória do tipo GLOBAL.

Ora, sempre que um *thread* executando na GPU lê ou escreve na memória GLOBAL ele sempre o faz acessando um grande pedaço de memória de cada vez, mesmo que efetivamente acesse apenas uns poucos registros. Se outros *threads* executam a mesma tarefa ou uma tarefa similar ao mesmo tempo então é interessante que essa memória que já está no *cache* seja utilizada por estes *threads*. Isto significa que o *hardware* foi construído de maneira a tornar a GPU mais eficiente quando os *threads* em execução acessam a memória GLOBAL em regiões contíguas, quando isto acontece dizemos que houve coalescência no padrão de acesso a memória.

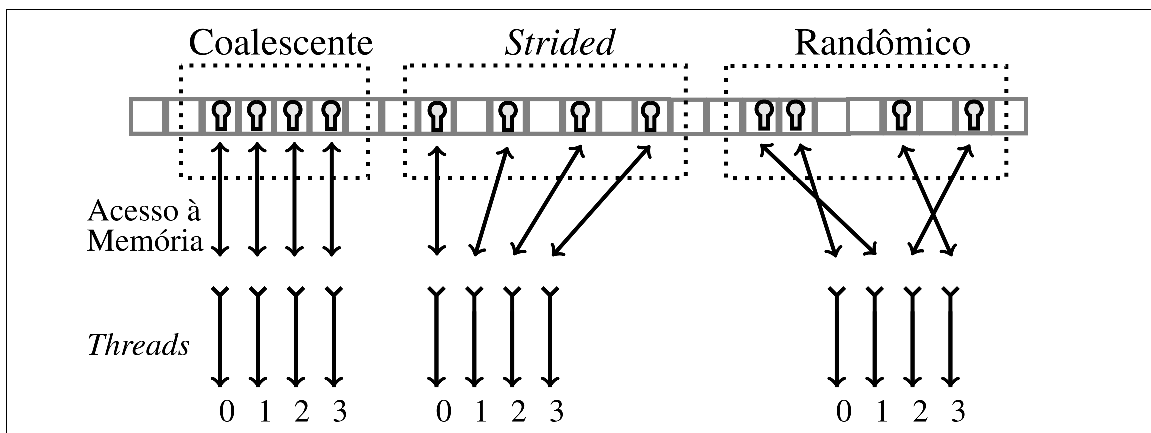

Figura 5.35: Diagrama de Coalescência

Na figura 5.35 podemos observar três exemplos, no exemplo de coalescência temos

que cada *thread* acessa a região de memória respectiva ao seu índice. Ao efetuar o acesso de leitura ou escrita na memória GLOBAL desta forma, obtemos o máximo desempenho possível.

No exemplo *Strided* cada *thread* acessa uma região de memória com uma lacuna de tamanho fixo entre cada pedaço. E apresenta um desempenho menor do que se o acesso fosse coalescente.

O pior de todos os cenários se apresenta quando não existe nenhuma ordem no acesso a memória global. Neste caso, o *hardware* pede um tempo de espera (*wait*) em todo o *WARP* e torna o acesso sequencial.

5.7.10 Operações com Vetores

Nesta seção foram empregados exemplos baseados em uma operação de soma entre dois vetores para esclarecer de que maneira diferentes implementações da mesma rotina podem ser efetuados na GPU, com incremento no desempenho.

Exemplo 1a

Neste exemplo, a soma de dois vetores, que engloba boa parte da ideia de paralelismo de dados que foi implementado nesta tese. A função *kernel_vector_sum* é descrita em linguagem C na figura 5.36. Observe que a variável **index** é um indexador unidimensional e que na linguagem C, a definição *float* * representa a posição de memória apontada pela variável **a**, **b** ou **out**.

```
__global__ void kernel_vector_sum(float *a, float *b, float *out, int N){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if(index < N)
    out[index] = a[index] + b[index];
}
```

Figura 5.36: Soma de vetores em **R** usando precisão simples.

Este tipo de chamada de *kernel* pressupõe que temos um único e grande *grid* de *threads* que manipula todo o vetor. Na figura 5.37 podemos observar a chamada da rotina pelo programa. Observe que as variáveis **blocksPerGrid** e **threadsPerBlock** definem o tamanho dos *grids* de blocos e o número de *threads* em cada bloco respectivamente.

```
void Vector_Add(int N, int *A, int *B, int* Out, int size){
  int blocksPerGrid=4096, threadsPerBlock=128;
  kernel_vector_sum<<< blocksPerGrid, threadsPerBlock>>>(N, A, B, Out, size);
}
```

Figura 5.37: Soma de vetores em **Z**

Exemplo 1b

A soma de dois vetores, vista no exemplo da figura 5.37, pode ser aprimorada na gpu se utilizarmos a técnica de *Grid-Stride Loops* como mostra a figura 5.38.

```
__global__ void kernel_vector_sum(float *a, float *b, float *c, int N){
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for (int index = idx; index < N; index += blockDim.x * gridDim.x) {
    c[index] = a[index] + b[index];
  }
}
```

Figura 5.38: Soma de vetores com *Stride*

Neste caso não se assume que o *grid* é suficientemente grande de forma a manipular todo o vetor. É criado então um laço com um passo de tamanho *blockDim.x* * *gridDim.x* e este é o número total de *threads* no *grid*. Deste modo se houverem 1024 *threads* no *grid*, o *thread* 0 efetuará a soma dos elementos $\{0, 1024, 2048, \cdots\}$. Outra vantagem está relacionado com o fate de este método manter a coalescência do acesso a memória.

Exemplo 1c

Nesta versão da soma de dois vetores em precisão simples, utilizou-se a passagem dos vetores através da memória **SHARED**. Na figura 5.39 podemos observar o código.

```
#define BLOCK_SIZE 64
__global__ void kernel_vector_sum(float *a, float *b, float *c, int s)
{
  volatile __shared__ float sdata_a[BLOCK_SIZE];
  volatile __shared__ float sdata_b[BLOCK_SIZE];
  int tid = threadIdx.x ;
  unsigned int idx = blockIdx.x * BLOCK_SIZE + tid;
  if(idx < s){
    sdata_a[tid] = a[idx];
    sdata_b[tid] = b[idx];
    __syncthreads();
    c[idx] = sdata_a[tid] + sdata_b[tid];
  }
}
```

Figura 5.39: Soma de vetores usando a memória **SHARED**

Exemplo 1d

Nesta versão da soma de dois vetores em precisão simples. Utilizou-se a passagem dos vetores através da **SHARED MEMORY** e a técnica de **double buffering** com o intuito de melhorar o desempenho da rotina. Na figura 5.40 podemos observar o código.

```
#define BLOCK_SIZE 64
#define BLOCK_SIZE_2 BLOCK_SIZE*2

__global__ void kernel_vector_sum(float *a, float *b, float *c, int s)
{
  volatile __shared__ float sdata_a0 [BLOCK_SIZE];
  volatile __shared__ float sdata_b0 [BLOCK_SIZE];
  volatile __shared__ float sdata_a1 [BLOCK_SIZE];
  volatile __shared__ float sdata_b1 [BLOCK_SIZE];

  int tid = threadIdx.x ;
  unsigned int idx0 = blockIdx.x * BLOCK_SIZE + tid;
  unsigned int idx1 = BLOCK_SIZE  + blockIdx.x * BLOCK_SIZE + tid;
  if(idx1 < s){
    sdata_b0[tid] = b[idx0];
    sdata_a0[tid] = a[idx0];
    c[idx0] = sdata_a0[tid] + sdata_b0[tid];
    idx0 += BLOCK_SIZE_2;
    __syncthreads();

    sdata_b1[tid] = b[idx1];
    sdata_a1[tid] = a[idx1];
    c[idx1] = sdata_a1[tid] + sdata_b1[tid];
    idx1 += BLOCK_SIZE_2;
    __syncthreads();
  }
}
```

Figura 5.40: Soma de vetores usando a memória **SHARED** e **Double Buffering**

Exemplo 1e

Na figura 5.41 temos a implementação final da rotina de soma de vetores, utilizada como uma função virtual do C++. Deste modo podemos operar com qualquer tipo de *ValueType* dentro do escopo { **float**, **double**, **int** } implementado até este momento.

No futuro podem ser incluídos objetos para operações com *quatérnios* e números complexos no escopo da biblioteca. A biblioteca de álgebra linear implementada compreende diversas funções, tais como: produto interno, soma de vetores, soma de múltiplos vetores e produto por escalar. De modo geral, podemos utilizar todas as funções da biblioteca cuBLAS da NVIDIA de maneira simples.

Na figura 5.42 podemos observar uma chamada da rotina *kernel_vector_sum* para os três tipos de dados definidos no escopo de dados.

```
template <typename ValueType, typename IndexType,int BLOCK_SIZE>
__global__ void kernel_vector_sum(ValueType *a, ValueType *b,
      ValueType *c, IndexType s)
{
  volatile __shared__ ValueType sdata_a0 [BLOCK_SIZE];
  volatile __shared__ ValueType sdata_b0 [BLOCK_SIZE];
  volatile __shared__ ValueType sdata_a1 [BLOCK_SIZE];
  volatile __shared__ ValueType sdata_b1 [BLOCK_SIZE];

  IndexType tid = threadIdx.x ;
  IndexType idx0 = blockIdx.x * BLOCK_SIZE + tid;
  IndexType idx1 = BLOCK_SIZE  + blockIdx.x * BLOCK_SIZE + tid;
  if(idx1<s){
    sdata_b0[tid] = b[idx0];
    sdata_a0[tid] = a[idx0];
    c[idx0] = sdata_a0[tid] + sdata_b0[tid];
    idx0 += (BLOCK_SIZE<<1);
    __syncthreads();

    sdata_b1[tid] = b[idx1];
    sdata_a1[tid] = a[idx1];
    c[idx1] = sdata_a1[tid] + sdata_b1[tid];
    idx1 += (BLOCK_SIZE<<1);
    __syncthreads();
    }
}
```

Figura 5.41: Soma de vetores genérica usando a memória *SHARED* e *Double Buffering*

```
  int GPU_block_size = 64;
  Vector<double> din1, din2, dout;
  Vector<float>  fin1, fin2, fout;
  Vector<int>    iin1, iin2, iout;
  dim3 BlockSize(GPU_block_size);
  dim3 GridSize(Vector_size/ GPU_block_size + 1);
// double sum of vectors
  cudaDeviceSynchronize();
  kernel_vector_sum<double, unsigned int, GPU_block_size>
  <<<GridSize, BlockSize>>>
  (din1->Value, din2->Value, dout->Value, Vector_size);
// float sum of vectors
  cudaDeviceSynchronize();
  kernel_vector_sum<float, unsigned int, GPU_block_size>
  <<<GridSize, BlockSize>>>
  (fin1->Value, fin2->Value, fout->Value, Vector_size);
// int sum of vectors
  cudaDeviceSynchronize();
  kernel_vector_sum<int, unsigned int, GPU_block_size>
  <<<GridSize, BlockSize>>>
  (iin1->Value, iin2->Value, iout->Value, Vector_size);
```

Figura 5.42: Soma de vetores Genérica em C++ STL e CUDA

Nos exemplos **1a** até **1e** foram brevemente descritas maneiras de aumentar o desempenho da GPU ao efetuar a soma de dois vetores. Além disso, ao utilizar a memória de texturas, cujo tempo de acesso é significativamente menor do que a memória global é feito um *cache* dos vetores de entrada que aumenta mais ainda o desempenho da operação.

5.7.11 Operações com matrizes

As operações de soma e multiplicação de matrizes esparsas são implementadas na biblioteca *cuSparse* da NVIDIA, contudo outros procedimentos foram implementados neste trabalho. Como por exemplo : A norma de Frobenius, o valor do Traço, criação de matriz Identidade e círculos de Gershgorin (WEISSTEIN (2003)).

Nesta seção serão utilizados exemplos relacionados a multiplicação da matriz esparsa por um vetor (SpMv) para demonstrar o potencial da GPU neste caso.

Exemplo 2a

Na figura 5.43 pode-se observar o código fonte da rotina que efetua o produto $A \cdot x = b$, onde A é a matriz esparsa definida por três vetores.

$$A = \begin{cases} \text{col} & \text{ponteiro para o vetor de colunas} \\ \text{row_offset} & \text{ponteiro para o vetor de linhas} \\ \text{val} & \text{ponteiro para o vetor de valores} \end{cases}$$

Este é o modo mais simples de encontrar o vetor b, contudo não apresenta um bom desempenho.

Exemplo 2b

No exemplo de código mostrado na figura 5.44 foram aplicadas as técnicas de utilização da memória *SHARED* e vetorização dos *threads*.

Na figura 5.45 podemos observar a chamada da função *spmv_csr_vector_kernel* utilizando vetorização por textura.

```
template <typename ValueType, typename IndexType>
__global__ void kernel_csr_spmv_scalar(
   const IndexType nrow, const IndexType *row_offset,
    const IndexType *col, const ValueType *val,
    const ValueType *in, ValueType *out) {
  IndexType ai = blockIdx.x*blockDim.x+threadIdx.x;
  IndexType aj;
  if (ai <nrow) {
    out[ai] = ValueType(0.0);
    for (aj=row_offset[ai]; aj<row_offset[ai+1]; ++aj) {
      out[ai] += val[aj]*in[col[aj]];
    }
  }
}
```

Figura 5.43: Produto matriz Vetor

```
template <typename ValueType, typename IndexType, IndexType BLOCK_SIZE, IndexType WARP_SIZE>
__launch_bounds__(BLOCK_SIZE * BLOCK_SIZE,1)
__global__  void kernel_csr_spmv_fast(
     const IndexType nrow, const IndexType *row_offset,
     const IndexType *col, const ValueType *__restrict__ val,
     const ValueType *x,ValueType *out){
  __shared__ ValueType sdata[BLOCK_SIZE + 16];
  __shared__ IndexType ptrs[BLOCK_SIZE/WARP_SIZE][2];
// global thread index
  const IndexType thread_id = BLOCK_SIZE * blockIdx.x + threadIdx.x;
// thread index within the warp
  const IndexType thread_lane = threadIdx.x & (WARP_SIZE-1);
// global warp index
  const IndexType warp_id = thread_id / WARP_SIZE;
// warp index within the CTA
  const IndexType warp_lane = threadIdx.x / WARP_SIZE;
// total number of active warps
  const IndexType num_warps = (BLOCK_SIZE / WARP_SIZE) * gridDim.x;
// use two threads to fetch row_offset[row] and row_offset[row+1]
// this is considerably faster than the straightforward version
  for(IndexType row = warp_id; row < nrow; row += num_warps){
    if(thread_lane < 2)
      ptrs[warp_lane][thread_lane] = row_offset[row + thread_lane];
//same as: row_start = vecPointers[row];
    const IndexType row_start = ptrs[warp_lane][0];
//same as: row_end = vecPointers[row+1];
    const IndexType row_end   = ptrs[warp_lane][1];
// compute local sum
    float sum = 0;
    for(int j = row_start + thread_lane; j < row_end; j += WARP_SIZE)
sum += val[j] * x[col[j]];
// reduce local sums to row sum (ASSUME: warpsize 32)
    sdata[threadIdx.x] = sum;
    sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x + 16];__syncthreads();
    sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x + 8]; __syncthreads();
    sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x + 4]; __syncthreads();
    sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x + 2]; __syncthreads();
    sdata[threadIdx.x] = sum = sum + sdata[threadIdx.x + 1]; __syncthreads();
// first thread writes warp result
    if (thread_lane == 0){
      out[row]+= sdata[threadIdx.x];
    }
  }
}
```

Figura 5.44: Produto matriz-vetor com vetorização

```
  int GPU_block_size = 64;
  dim3 BlockSize(GPU_block_size);
  dim3 GridSize(Vector_size/ GPU_block_size + 1);
  int VECTORS_PER_BLOCK = 2, THREADS_PER_VECTOR = 32;
  if(UseCache)
    bind_x(cast_in->vec_);
  spmv_csr_vector_kernel
  <UseCache, double, int, VECTORS_PER_BLOCK, THREADS_PER_VECTOR>
  <<<GridSize, BlockSize>>>
// Onde A é uma matriz CSR
  (Nrows, A.row_offset, A.col, A.val, Inp_vector, Out_vector);
  if(UseCache)
    unbind_x(cast_in->vec_);
    }
```

Figura 5.45: Produto matriz Vetor Vetorizado

5.7.12 Computação heterogênea

Usualmente o termo **computação heterogênea** está relacionado a sistemas computacionais que utilizam mais de um tipo de processador na execução de seus processos. Como visto na seção 5.1 ao incluir em um mesmo sistema processadores SIMD e MIMD, em nosso caso as GPU da NVIDIA e os processadores multi-núcleo da CPU, construímos um sistema de computação heterogêneo. Outros tipos podem incluir FPGA (BINESH MARVASTI e SZYMANSKI (2015)) ou processadores de sinais DSP (SARMA e SARMA (2015)).

Algumas bibliotecas de álgebra linear como a MAGMA (AGULLO *et al.* (2009)) e a PLASMA (PAI *et al.* (2010)) se propões a implementar uma programação de código distribuído entre a GPU e a CPU.

Neste trabalho os algorítimos foram implementados em linguagem C++ STL (MUSSER *et al.* (2009)) através da qual foi produzindo código genérico no qual o paralelismo é obtido distribuindo as tarefas mais próximas a camada de *hardware* entre os processadores e também utilizando as bibliotecas de álgebra linear BLAS, MKL e cuBLAS. A distinção entre a execução do código na CPU e na GPU é realizada através do mecanismo de polimorfismo do C++ se valendo do modelo de declarações de memória, *i.e.*, se o vetor ou matriz for declarado na CPU o código executa em CPU e caso contrário ele executa em GPU. Como inicialmente todos os vetores e matrizes são declarados em CPU utiliza-se procedimentos do tipo *MoveToDevice* para mover os dados para a GPU e simultaneamente indicar ao código que este deve executar as funções na GPU.

5.7.13 Polimorfismo, herança e funções virtuais

A linguagem C++ representou um novo paradigma de programação baseada em objetos (STROUSTRUP (1986)). Uma característica em especial é muito útil na programação de código híbrido CPU e GPU e é chamada de polimorfismo. De modo geral o polimorfismo ocorre entre classes relacionadas através de uma herança.

Uma das características-chave da herança de classe é que um ponteiro para uma classe derivada é *type compatible* com um ponteiro para a sua classe base. O polimorfismo é um recurso sofisticado e versátil da linguagem C++, nas figura 5.46 e 5.47 podemos observar um exemplo de código com a sua utilização.

```
class poligono {                  // pointers to base class
  protected:
    int width, height;
  public:
    void set_values (int a, int b){ width=a; height=b; }
};
class retangulo: public poligono {
  public:
    int area()                    { return largura*altura; }
};
class triangulo: public poligono {
  public:
    int area()                    { return largura*altura/2; }
};
```

Figura 5.46: Definição da classe Base no arquivo *base.hpp* em C++

Na figura 5.47 pode-se observar o código fonte para teste do polimorfismo. Neste teste declaram-se dois ponteiros para polígono ($poly1$ e $poly2$) nos endereços *ret* e *tri* respectivamente para os objetos do tipo $retangulo$ e $triangulo$, uma vez que ambos são objetos do tipo $poligono$.

```
#include <iostream>
#include base.hpp
using namespace std;
int main () {
  retangulo          ret;
  triangulo           tri;
  poligono  *poly1 = &ret;
  poligono  *poly2 = &tri;
  poly1->set_values (5,6);
  poly2->set_values (4,4);
  cout << ret.area() << '\n';
  cout << tri.area() << '\n';
  return 0;
}
```

Figura 5.47: Programa de teste do polimorfismo.

```
class poligono {                    // pointeiro para a classe base
  protected:
    int largura,altura;
  public:
    void set_values (int a, int b){ largura=a; altura=b; }
    virtual int area()              { return 0; }
};
```

Figura 5.48: Definição da classe Base com função virtual.

Ora, *poly*1 e *poly*2 são ponteiros do tipo *poligono* então apenas o membros herdados desta classe podem ser acessados e não aqueles herdado de *triangulo* and *retangulo*. Este incomodo pode ser resolvido atribuindo a palavra chave *virtual* a função *area()* agora declarada na classe *poligono*.

Basicamente, a palavra-chave *virtual* , como vistro na figura 5.48, permite que um membro de uma classe derivada com o mesmo nome que um da classe base seja apropriadamente chamado através de um ponteiro. Mais exemplos e farta literatura são encontrados facilmente (DEITEL e DEITEL (2008)).

Na figura 5.49 pode-se observar a forma de construção deste tipo de modelo baseado em funções virtuais aplicado no código desta tese.

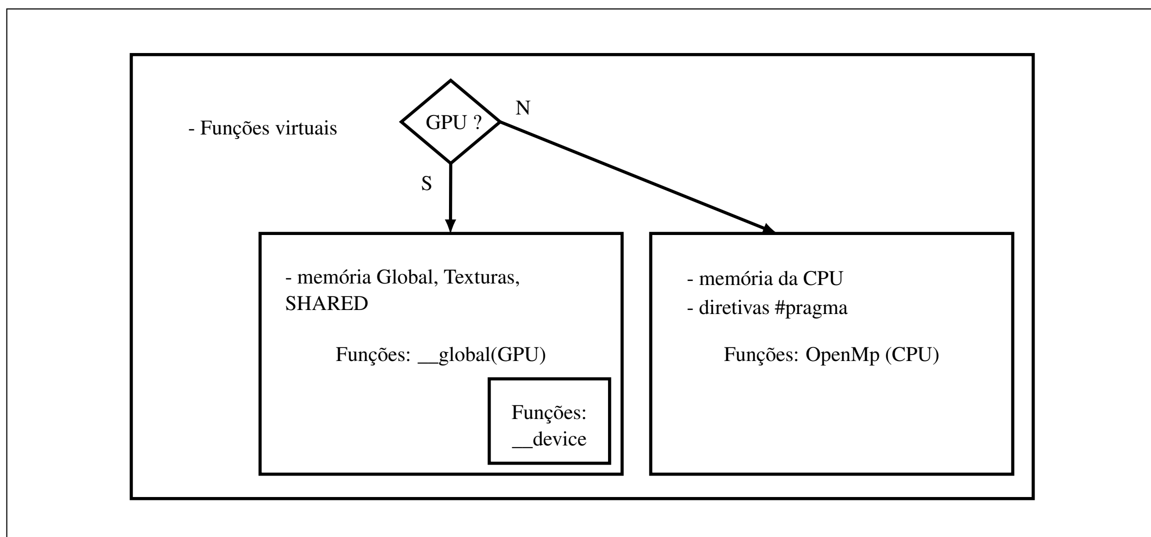

Figura 5.49: Diagrama de bloco

Além de direcionar a função correta, GPU ou CPU, de acordo com o tipo de memória reservada, este modelo permite implementar um código multi precisão baseado no tipo dos vetores e matrizes utilizados (*double*,*float*,*complex*, ou ainda *quad double* se este for

implementado). Neste caso para cada tipo temos de escrever a função adequada e neste caso obtemos uma função dita generalizada ou *functor* (ALEXANDRESCU (2001)).

Capítulo 6

Resultados, conclusão e proposta de trabalhos futuros

Neste capítulo são discutidos os resultados comparativos, de desempenho computacional e precisão, entre os algorítimos implementados nesta tese e o método sequencial de Gauss-Jacobi de ordem 3 implementado atualmente no sistema GEDAR.

Além disso no apêndice A pode-se observar os gráficos das concentrações dos nuclídeos analisados neste trabalho, bem como o desvio das concentrações em função do método de calculo empregado.

Contudo, é necessário um preambulo para compreender o modo como o sistema GEDAR calcula e atualiza o inventário do combustível.

O sistema GEDAR utiliza o método do ponto médio, ou preditor-corretor, para estimar a concentração média de cada nuclídeo dentro de cada passo de queima. No primeiro passo de queima o sistema GEDAR utiliza como condição inicial o combustível novo, ou seja, apenas com o U^{234}, U^{235} e U^{238}. Após efetuar o cálculo da queima, reserva-se a solução (preditora) e esta é realimentada no sistema como condição inicial. O programa então recalcula o fluxo de nêutrons, a realimentação termohidráulica, as seções de choque homogêneas para cada nodo em função da temperatura e das concentrações de Boro e Xenônio. A partir destes novos parâmetros o cálculo da queima é novamente realizado e obtêm-se a nova solução (corretora). A média entre soluções do corretor e preditor é

considerada como a nova solução do sistema de EDO (corrigido).

Assim, os cálculos da queima necessitam de duas soluções do sistema de EDO para cada passo, impactando a performance total do sistema GEDAR. O método denominado $\star$RKF representa a avaliação da queima pelo Runge-Kutta-Fehlberg sem a aplicação do preditor-corretor.

6.1 Analise do desvio na avaliação da sequência de queima

A avaliação da precisão entre cada método foi realizada utilizando um perfil de queima padrão do reator nuclear de Angra II, incluindo as transmutações induzidas por reações $(n,2n)$ e decaimento α. Este procedimento produz uma avaliação da queima e do inventário do combustível no reator através de 32 *steps* de queima em um total de 498 dias.

O método de média quadrática, *Root Mean Squared Error* (**RMSE**) (MENDENHALL e SINCICH (2006)), foi utilizado para medir a distância entre cada sequência de resultados e sua formula pode ser observada na equação (6.1).

Este método avalia a distancia euclideana entre uma sequência de resultados padrão e os resultados obtidos por cada método sob teste e obtém a diferença percentual total. Assim, a concentração do nuclídeo k após cada passo de queima é avaliada pelo método $m = \{$Padé, JAC2, JAC3, RKF, $\star$RKF$\}$, a concentração de cada nuclídeo y_i, no conjunto de dados de referência $Y_k^m = \{y_i, i = 1, 2, \cdots, N\}$ corresponde aos resultados obtidos pelo método de colocação de Gauss-Jacobi na forma sequencial produzida pelo código sistema GEDAR.

$$\textbf{RMSE} = 100 \times \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2}}{(Max\{y_i\} - Min\{y_i\})}, \quad x_i \in X_k \text{ and } y_i \in Y_k^m \tag{6.1}$$

Foi escolhido um nodo representativo para efetuar a comparação entre os métodos paralelos implementados da GPU e os resultados apresentados pelo método sequencial

fornecido pelo sistema GEDAR.

A figura 6.1 mostra o desvio aferido pelo método *RMSE* referente a cada método implementado neste trabalho. Nesta figura o conjunto de dados de referência X_k contém a concentração do nuclídeo i no passo de queima k e o conjunto Y_k contém a concentração do nuclídeo i no passo de queima k calculado pelo método mostrado na legenda.

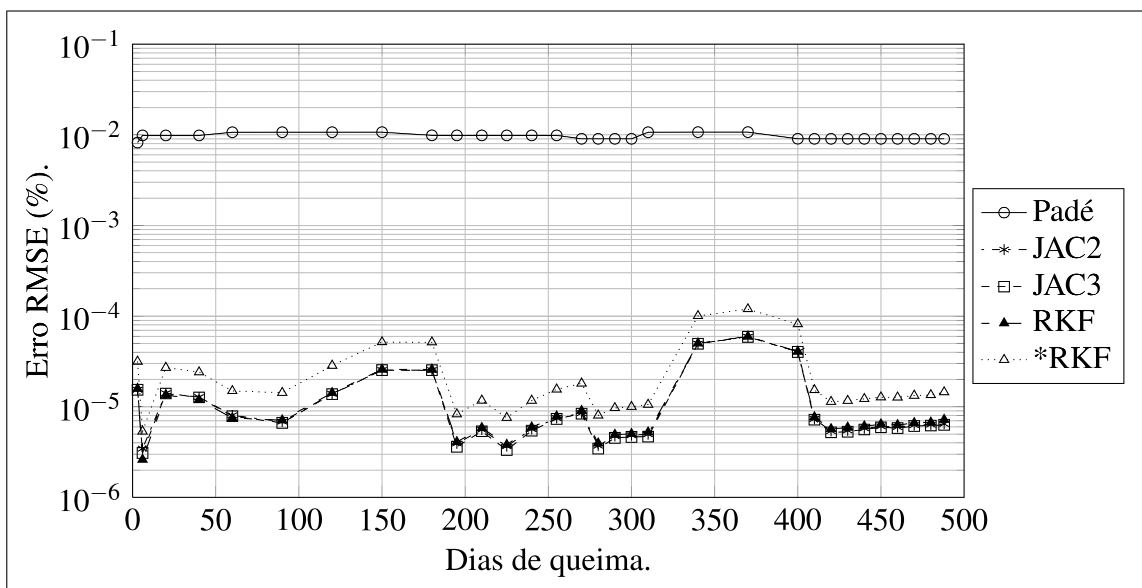

Figura 6.1: Desvio *RMSE* (%) *vs* Queima.

Nas tabelas 6.1, 6.2, 6.3 a primeira linha indica os métodos sob análise: RKF, Padé, e segunda e terceira ordem do método de colocação de Gauss-Jacobi (*JAC*2 and *JAC*3).

A tabela 6.1 apresenta o desvio aferido pelo método *RMSE* para os isótopos do *Urânio*, *Plutônio* , *Netúnio* , *Americio* and *Curio*.

Tabela 6.1: Erro percentual *RMSE* dos isótopos Am, Cm, U, Pu e Np.

Nuc	RKF	Padé	JAC3	JAC2
Am^{241}	0.001	0.070	0.002	0.002
Am^{242m}	0.026	0.062	0.025	0.025
$Am^{242,}$	0.146	0.085	0.140	0.140
Am^{243}	0.028	0.069	0.027	0.027
Cm^{242}	0.034	0.059	0.033	0.033
Cm^{244}	0.027	0.075	0.025	0.025
U^{234}	0.025	0.074	0.025	0.025
U^{235}	0.026	0.308	0.026	0.026
U^{236}	0.025	0.046	0.025	0.025

Nuc	RKF	Padé	JAC3	JAC2
U^{238}	0.024	0.372	0.023	0.023
Np^{237}	0.024	0.047	0.023	0.023
Pu^{238}	0.031	0.052	0.030	0.030
Np^{239}	0.53	0.33	0.52	0.529
Pu^{239}	0.018	0.049	0.018	0.018
Pu^{240}	0.032	0.048	0.031	0.031
Pu^{241}	0.024	0.057	0.024	0.024
Pu^{242}	0.033	0.060	0.032	0.032

Como observado no capítulo 3.3, a unidade de ponto flutuante das CPU Intel apresenta seu resultados com um arredondamento de 80 *bits* de precisão, contudo, as uni-

dade de ponto flutuante da GPU NVIDIA fornecem um arredondamento nas operações de apenas 64 *bits* conforme as especificações da norma IEEE-754 (WHITEHEAD e FIT-FLOREA (2011)). O combustível novo no início do ciclo apresenta apenas os isótopos U^{234}, U^{235} and U^{238}. Os actinídeos e alguns produtos de fissão apresentam um crescimento de suas concentrações muito pequeno nestes passo de queima iniciais, e por serem representados em ponto flutuante, e as operações de multiplicação e soma com diferentes precisões entre a CPU e a GPU, esta diferença pode ser considerada responsável pela diferença das concentrações nos passos de queima iniciais entre os métodos paralelos e o método sequencial baseado no método de colocação de Gauss-Jacobi de ordem 3 embutido no sistema GEDAR.

A tabela 6.2 apresenta o desvio aferido pelo método *RMSE* para os isótopos dos produtos de fissão *Samário*, *Ródio* and *Promécio*.

Tabela 6.2: Erro percentual *RMSE* das cadeias Mo, Rh, Pm e Sm.

Nuc	RKF	Padé	JAC3	JAC2
Zr^{95}	0.092	0.076	0.092	0.092
Nb^{95}	0.024	0.049	0.024	0.024
Mo^{95}	0.000	0.057	0.000	0.001
Ru^{103}	0.121	0.098	0.120	0.120
Rh^{03}	0.006	0.054	0.005	0.005
Rh^{05}	0.408	0.178	0.401	0.401

Nuc	RKF	Padé	JAC3	JAC2
Pm^{147}	0.030	0.046	0.030	0.030
Pm^{148m}	0.070	0.054	0.069	0.069
Pm^{148}	0.189	0.128	0.188	0.188
Sm^{147}	0.001	0.057	0.001	0.001
Pm^{149}	0.500	0.299	0.493	0.493
Sm^{149}	0.056	0.099	0.055	0.055

A tabela 6.3 apresenta o desvio aferido pelo método *RMSE* para os isótopos dos produtos de fissão *Neodímio*, *Praseodímio*, *Gadolínio* e *Európio*.

Tabela 6.3: Erro percentual *RMSE* das cadeias Nd, Gd, Xe.

Nuc	RKF	Padé	JAC3	JAC2
Pr^{143}	0.251	0.212	0.250	0.250
Nd^{143}	0.012	0.051	0.012	0.012
Eu^{155}	0.026	0.048	0.026	0.026
Gd^{155}	0.159	0.196	0.159	0.159

Nuc	RKF	Padé	JAC3	JAC2
I^{131}	0.306	0.246	0.304	0.304
Xe^{131}	0.016	0.050	0.016	0.016
I^{135}	0.883	0.942	0.829	0.829
Xe^{135}	0.988	2.171	1.048	1.048

Adicionalmente, a figura 6.2 apresenta um gráfico do tipo *boxplot* CHANG (2012) no qual o desvio RMSE é calculado sob outra perspectiva. O gráfico mostra o desvio RMSE acumulado por todos os passos de queima em função de cada método empregado na avaliação. O conjunto de dados X_k contém a concentração do nuclídeo k no passo

de tempo i calculado atualizando o procedimento sequencial e o conjunto Y_k^m contém a concentração de nuclídeo k no passo de tempo i calculado pelo método m. Aplicam-se estes dois conjuntos de dados na equação (6.1) e obtêm-se o desvio entre cada método paralelo e sequencial.

O resultado é mostrado na figura 6.2, neste caso pode-se observar que o desvio médio de cada método se concentra abaixo de 0.5 % com alguns pontos acima deste número em todos os métodos avaliados exceto o $\star$RKF que apresenta o desvio RMSE menor do que 2.0 %.

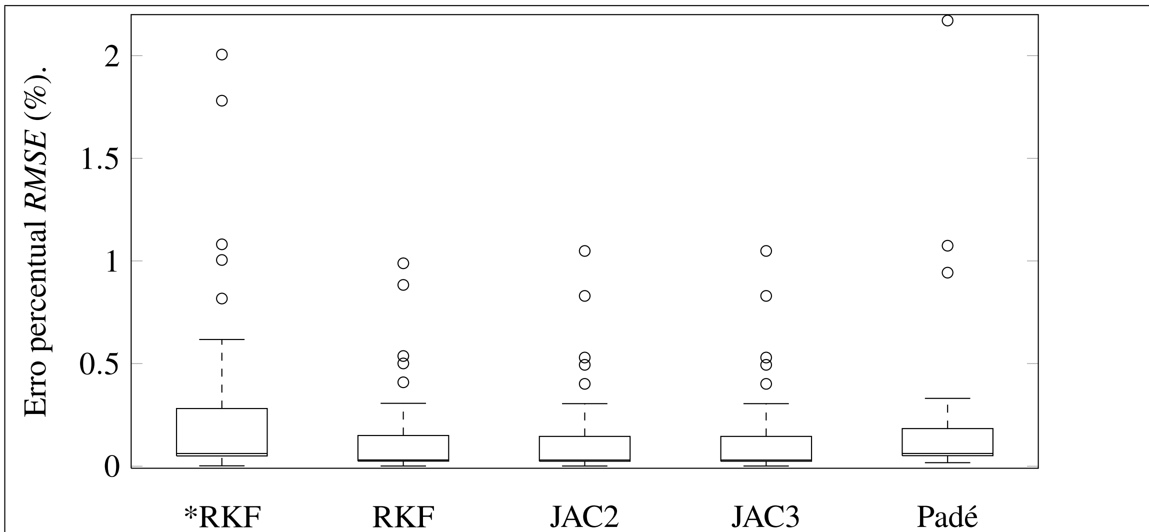

Figura 6.2: Desvio RMSE (%) *vs* método.

6.2 Análise de performance

Na simulação da queima avaliada nesta tese o código sistema GEDAR a avalia utilizando uma versão sequencial do método de Gauss-Jacobi de ordem 3. Utilizando um computador INTEL I7-860 com 8 Gbytes de memória RAM o tempo necessário para obter o cálculo da queima utilizando 32 passos é de 150 segundos aproximadamente. Os algorítimos implementados nesta tese foram avaliados utilizando uma GPU NVIDIA TITAN-BLACK, uma placa baseada no *chipset* Kepler GK-110 (NVIDIA (2012)).

A tabela 6.4 apresenta o tempo total em segundos necessário para avaliar a queima *versus* o passo de tempo em dias. Observa-se que a coluna ΔT apresenta o tamanho de cada passo de queima e a coluna *Total* apresenta o tempo de queima acumulado. Além

disso, as colunas *JAC2* e *JAC3* apresentam o tempo despendido para cada passo de queima se utilizado o método de colocação de Gauss-Jacobi de ordem 2 e 3 respectivamente. Nas coluna seguinte, RKF representa o tempo despendido ao utilizar o método de Runge-Kutta-Fehlberg.

Tabela 6.4: Tempo dispendido em segundos *vs* Método *vs* Passo de queima.

Passo	ΔT	Total	RKF	JAC2	JAC3	Passo	ΔT	Total	RKF	JAC2	JAC3
1	3	3	0.025	0.091	0.113	17	10	290	0.023	0.079	0.101
2	3	6	0.025	0.079	0.103	18	10	300	0.025	0.078	0.100
3	14	20	0.023	0.079	0.102	19	10	310	0.025	0.077	0.098
4	20	40	0.023	0.078	0.102	20	30	340	0.025	0.078	0.100
5	20	60	0.023	0.079	0.105	21	30	370	0.024	0.078	0.101
6	30	90	0.025	0.080	0.102	22	30	400	0.023	0.078	0.100
7	30	120	0.023	0.079	0.101	23	10	410	0.023	0.078	0.098
8	30	150	0.024	0.079	0.104	24	10	420	0.025	0.078	0.098
9	30	180	0.023	0.079	0.100	25	10	430	0.025	0.078	0.097
10	15	195	0.023	0.079	0.102	26	10	440	0.025	0.077	0.098
11	15	210	0.024	0.078	0.100	27	10	450	0.023	0.077	0.098
12	15	225	0.023	0.078	0.100	28	10	460	0.024	0.076	0.102
13	15	240	0.024	0.078	0.100	29	10	470	0.024	0.077	0.102
14	15	255	0.023	0.078	0.100	30	10	480	0.024	0.076	0.098
15	15	270	0.025	0.078	0.099	31	10	490	0.024	0.076	0.098
16	10	280	0.025	0.079	0.108	32	8	498	0.024	0.079	0.103

A tabela 6.5 apresenta o tempo total despendido para completar o ciclo de queima em função do método sob análise. Neste caso o método $\star$RKF representa a avaliação da queima sem a utilização do método preditor-corretor.

Tabela 6.5: Tempo total dispendido (segundos) para um ciclo de queima de 498 dias.

Método	Tempo dispendido (segundos)
$\star$Runge-Kutta-Fhelberg	0.343
Runge-Kutta-Fhelberg	0.786
Colocação Jacobi de ordem 2	2.529
Colocação Jacobi de ordem 3	3.246
Diagonal de Padé	57
Colocação Jacobi de ordem 3 CNFR	150

O resultados de desempenho do método de Padé não são mostrados na tabela 6.4 porque comparados aos obtido pelos métodos de Runge-Kutta-Fehlberg e de colocação de Gauss-Jacobi são muito inferiores. Nos testes realizados este método gastou aproximadamente 57 segundos para efetuar a queima.

6.3 Conclusões

O objetivo desta tese de doutorado é desenvolver uma solução de alto desempenho para o cálculo da queima do combustível no reator PWR e sua interligação ao código CNFR. Nesta tese foram implementados três algoritmos em CPU multi-núcleo e GPU para o cálculo da queima do combustível de um reator PWR. Foram alcançados os objetivos de desempenho e exatidão.

Os resultados dos testes de avaliação destes três métodos númericos foram compilados na forma de um artigo e publicados em revista indexada (HEIMLICH *et al.* (2016)).

Além disso, nesta tese foi demonstrado que a programação paralela aplicada em GPU incrementa em muito o desempenho computacional do cálculo da queima do combustível em um reator nuclear. A isto se deve não apenas ao desempenho do *hardware* em questão mas também aos procedimentos de modelagem computacional no código escrito para a GPU.

A utilização do modelo de funções virtuais generalizadas mostrado na seção 5.7.13, associada aos métodos de matrizes esparsas, apresentados na seção 5.3, e a utilização da soma direta como potencializadora da arquitetura SIMD, apresentada na seção 5.5, viabiliza a aplicação deste modelo em outros algorítmos da área de física de reatores.

O desempenho obtido é evidente nos resultados para o ciclo de queima do combustível do reator nuclear da usina nuclear de ANGRA II, como pode ser observado na seção 6.2.

Na seção 4.2.1 apresentou-se o método Runge-Kutta-Fehlberg, que supera o procedimento sequencial empregado no sistema GEDAR por um fator de mais de *duzentas vezes* no tempo gasto no processamento computacional, com o desvio RMSE perto de 1%. Além disso, como mencionado na seção dos resultados , o método ⋆Runge-Kutta-Fehlberg executa sem o esquema de preditor-corretor e mantém o desvio menor do que 2% mas é duas vezes mais rápido.

Na seção 4.3 apresentou-se o método de colocação de Gauss-Jacobi, que apresenta um desvio RMSE menor do que 1%. Por outro lado, a utilização do método de segunda ordem apresenta um melhor desempenho com uma exatidão semelhante. Neste caso, o método de segunda ordem mostra um relação de desempenho *sessenta vezes mais rápido*

em relação ao método sequencial empregado no CNFR.

Na seção 4.5 apresentou-se o método diagonal de Padé, que apresenta um desvio RMSE estável, apesar de modificações de tamanho de passo de tempo. O seu desempenho porém é pior do que os outros métodos e deve ser descartado como uma solução de alto desempenho.

Conclui-se ao analisar os resultados que a utilização de processamento em GPU impacta positiva e substancialmente na performance em todos os três métodos avaliados, obtendo resultados com a exatidão necessária. Podemos destacar que os métodos numéricos empregados na tese constituem um importante degrau na utilização da computação paralela baseada em GPU e OpenMP aplicada a física de reatores. Além disso, foi desenvolvido um modulo de queima que pode ser acoplado ao sistema GEDAR.

Além do desenvolvimento do código para a execução dos três métodos numéricos implementados neste trabalho, a base de algebra linear associada a manipulação de matrizes esparsas do tipo CSR e COO em CPU e GPU e sua interligação ao CNFR foi encapsulada em uma biblioteca denominada de Grephy *General Reactor Physics* conforme apresentado no anexo B.

6.4 Proposta de trabalhos futuros

Um dos possíveis trabalhos futuros consiste em fazer uma análise para otimizar o tamanho dos passos de tempo de queima, de maneira a empregar um método misto de forma a maximizar o desempenho minimizando o desvio total.

Um dos possíveis trabalhos futuros consiste na implementação em GPU de novos métodos para solução de sistemas de equações diferenciais não lineares, não homogêneos ou *stiff* (SHAMPINE (1977)) relacionados à queima do combustível ou à outras áreas da engenharia.

Entre as novas técnicas, a implementação de métodos que usam de múltiplos passos (DORMAND e PRINCE (1980)), métodos implícitos como o RADAU (CHIPMAN (1971)) ou CVODE (COHEN e HINDMARSH (1996)), a paralelização *multi-level* (HERMANN *et al.* (2010)) e métodos de precondicionamento do problema. Em uma outra linha

de ação, analisar desempenho computacional do método denominado CRAM (*Chebychev Rational Approximation Method*) (PUSA e LEPPÄNEN (2010)) implementado em GPU.

Uma das técnicas de programação que deve ser aplicada em todos os estudos futuros é a utilização de precisão mista (CLARK *et al.* (2010)) que pode incrementar de maneira significativa a performace do código, uma vez que na GPU TITAN-X existem três vezes mais processadores de precisão simples do que os de dupla precisão.

Os métodos desenvolvidos nesta tese utilizam um *container* C++ que encapsula os procedimentos de manipulação das matrizes esparsas e de algebra linear. Esta implementação é expansível para novos actinídeos e cadeias de fragmentos de fissão e está sendo desenvolvido um programa que faz a leitura das tabelas NUDAT (KINSEY *et al.* (1996)) e utilizando a tabela de seções de choque microscópicas (fissão, (n,2n), captura, etc) cria os grafos correspondentes as cadeias com *threshold* de tempo ajustável. Com o objetivo de eliminar etapas serão utilizadas as bibliotecas baseadas em C++ *template* BOOST (SCHÄLING (2011)) no desenvolvimento do código, por já possuir os algorítimos de otimização de grafos.

A grande performance das GPU está associada ao trabalho com grandes vetores e operadores matriciais, justamente o que é necessário a um código para solução da equação de difusão multigrupo 3D por diferenças finitas em malha fina. O salto de desempenho que pode ser obtido, inclusive com a utilização de multiplas GPU em *cluster*, justifica o desenvolvimento do código para queima pino a pino.

O desenvolvimento de um código de transporte de nêutrons multigrupo em 3D utilizando o método das características (MOC) e diferenças finitas em malha adaptativa, tal como código MPACT (*Michigan parallel characteristics transport code*) (ZHU *et al.* (2015)) executando em *cluster* de GPU. No caso do MPACT, o MOC resolve a equação de transporte de nêntrons bidimensional por diferenças finitas em malha grossa e os planos são acoplados por uma solução unidimensional P_3 ou difusão. O acoplamento termohidráulico é realizado de um modelo simplificado.

Referências Bibliográficas

ABRAM, T., ION, S., 2008, "Generation-IV nuclear power: A review of the state of the science", *Energy Policy*, v. 36, n. 12, pp. 4323–4330.

AGULLO, E., DEMMEL, J., DONGARRA, J., et al., 2009, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series*, v. 180, p. 012037. IOP Publishing.

ALEXANDRESCU, A., 2001, *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley.

ALMEIDA, A. A. H., 2009, *Desenvolvimento de algoritmos paralelos baseados em GPU para solução de problemas na área nuclear*. Tese de Doutorado, Dissertação de Mestrado em Engenharia de Reatores. PPGIEN/CNEN.

ALVIM, A. C. M., DA SILVA, F. C., MARTINEZ, A. S., 2006, *Especificação funcional do sistema de geração de dados do reator (GEDAR) do Código Nacional de Fí sica de Reatores (CNFR)*. Relatório técnico, COPPE/UFRJ, Programa de Engenharia Nuclear.

ALVIM, A. C. M., DA SILVA, F. C., MARTINEZ, A. S., 2010, "Depletion Calculation for a Nodal Reactor Physics Code". In: *18th International Conference on Nuclear Engineering*, pp. 311–316. American Society of Mechanical Engineers.

AMDAHL, G. M., 2007, "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.

ANDERSON, E., BAI, Z., DONGARRA, J., et al., 1990, "LAPACK: A portable linear algebra library for high-performance computers". In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pp. 2–11. IEEE Computer Society Press.

BAHADIR, T., LINDAHL, S.-Ö., 2009, “Studsvik's next generation nodal code SIMULATE-5”, *Advances in Nuclear Fuel Management IV (ANFM 2009)*.

BATEMAN, H., 1910, “The solution of a system of differential equations occurring in the theory of radioactive transformations”. In: *Proc. Cambridge Philos. Soc*, v. 15, pp. 423–427.

BATHE, K.-J., WILSON, E. L., 1976, “Numerical methods in finite element analysis”, .

BAYS, S., PIET, S., POPE, M., et al., 2009, “Dynamics: Impacts of Multi-Recycling on Fuel Cycle Performances”, .

BAYS, S., PIET, S., DUMONTIER, A., 2010, “Fuel Cycle Isotope Evolution by Transmutation Dynamics over Multiple Recycles”. In: *Proceedings of ICAPP*, v. 10.

BELLMAN, R., BELLMAN, R. E., BELLMAN, R. E., et al., 1970, *Introduction to matrix analysis*, v. 960. SIAM.

BENTLEY, J. L., 1980, “Multidimensional divide-and-conquer”, *Communications of the ACM*, v. 23, n. 4, pp. 214–229.

BINESH MARVASTI, M., SZYMANSKI, T. H., 2015, “An Analysis of Hypermesh NoCs in FPGAs”, *Parallel and Distributed Systems, IEEE Transactions on*, v. 26, n. 10, pp. 2643–2656.

BLAND, A. S., KENDALL, R. A., KOTHE, D. B., et al., 2009, “Jaguar: The world?s most powerful computer”, *Memory (TB)*, v. 300, n. 62, pp. 362.

BLAND, B., 2012, “Titan-early experience with the titan system at oak ridge national laboratory”. In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 2189–2211. IEEE.

BONDYOPADHYAY, P. K., 1998, “Moore’s law governs the silicon revolution”, *Proceedings of the IEEE*, v. 86, n. 1, pp. 78–81.

BURSTALL, R., 1979, *FISPIN-A computer code for nuclide inventory calculations*. Relatório técnico, UKAEA Risley Nuclear Power Development Establishment.

BURTAK, F., HAASE, H., VELDE, A., 1996, “SAV95-the new Siemens code system for PWR nuclear core design”. In: *Annual meeting on nuclear technology’96. Proceedings*.

BUTCHER, J. C., 1963, “Coefficients for the study of Runge-Kutta integration processes”, *Journal of the Australian Mathematical Society*, v. 3, n. 02, pp. 185–201.

CAYLEY, A., 1858, “A memoir on the theory of matrices”, *Philosophical transactions of the Royal society of London*, pp. 17–37.

CHANDRA, R., 2001, *Parallel programming in OpenMP*. Morgan Kaufmann.

CHANG, W., 2012, *R graphics cookbook*. "O'Reilly Media, Inc.".

CHEN, G.-S., 1990, “Asymptotic behavior of multigroup diffusion equations with Xe feedback”, *Annals of Nuclear Energy*, v. 17, n. 8, pp. 415–425.

CHENEY, E., KINCAID, D., 2012, *Numerical mathematics and computing*. Cengage Learning.

CHIHARA, T. S., 2011, *An introduction to orthogonal polynomials*. Courier Dover Publications.

CHIPMAN, F., 1971, “A-stable Runge-Kutta processes”, *BIT Numerical Mathematics*, v. 11, n. 4, pp. 384–388.

CLARK, M. A., BABICH, R., BARROS, K., et al., 2010, “Solving Lattice QCD systems of equations using mixed precision solvers on GPUs”, *Computer Physics Communications*, v. 181, n. 9, pp. 1517–1528.

COHEN, S. D., HINDMARSH, A. C., 1996, “CVODE, a stiff/nonstiff ODE solver in C”, *Computers in physics*, v. 10, n. 2, pp. 138–143.

CROFF, A. G., 1983, “ORIGEN2: a versatile computer code for calculating the nuclide compositions and characteristics of nuclear materials”, .

CUTHILL, E., 1972, “Several strategies for reducing the bandwidth of matrices”. In: *Sparse Matrices and Their Applications*, Springer, pp. 157–166.

CUTHILL, E., MCKEE, J., 1969, “Reducing the bandwidth of sparse symmetric matrices”. In: *Proceedings of the 1969 24th national conference*, pp. 157–172. ACM.

DATTA, K. B., DATTA, K., MOHAN, B., 1995, *Orthogonal Functions in Systems and Control*. Advanced Series in Mathematical Physics. World Scientific. ISBN: 9789810218898. Disponível em: <http://books.google.com.br/books?id=mWS0Skwk4k4C>.

DAVIS, T. A., HU, Y., 2011, “The University of Florida sparse matrix collection”, *ACM Transactions on Mathematical Software (TOMS)*, v. 38, n. 1, pp. 1.

DE MOURA MENESES, A. A., SCHIRRU, R., 2015, “A cross-entropy method applied to the In-core fuel management optimization of a Pressurized Water Reactor”, *Progress in Nuclear Energy*, v. 83, pp. 326–335.

DE OLIVEIRA, I. M. S., SCHIRRU, R., 2011, “Swarm intelligence of artificial bees applied to in-core fuel management optimization”, *Annals of Nuclear Energy*, v. 38, n. 5, pp. 1039–1045.

DEITEL, P. J., DEITEL, H. M., 2008, *C++ how to program.* PearsonPrentice Hall.

DO NASCIMENTO ABREU PEREIRA, C. M., SCHIRRU, R., MARTINEZ, A. S., 1999, “Basic investigations related to genetic algorithms in core designs”, *Annals of Nuclear Energy*, v. 26, n. 3, pp. 173–193.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., et al., 1990, “A set of level 3 basic linear algebra subprograms”, *ACM Transactions on Mathematical Software (TOMS)*, v. 16, n. 1, pp. 1–17.

DORMAND, J. R., PRINCE, P. J., 1980, “A family of embedded Runge-Kutta formulae”, *Journal of computational and applied mathematics*, v. 6, n. 1, pp. 19–26.

DUFF, I. S., ERISMAN, A. M., REID, J. K., 1986, *Direct methods for sparse matrices*. Clarendon Press Oxford.

DURAN, A., KLEMM, M., 2012, “The Intel® many integrated core architecture”. In: *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pp. 365–366. IEEE.

ELMROTH, E., GUSTAVSON, F., JONSSON, I., et al., 2004, “Recursive blocked algorithms and hybrid data structures for dense matrix library software”, *SIAM review*, v. 46, n. 1, pp. 3–45.

ENGLAND, T., RIDER, B., 1994, “LA-UR-94-3106”, *ENDF-349*.

FINNEMANN, H., BENNEWITZ, F., WAGNER, M., 1977, “Interface current techniques for multidimensional reactor calculations”, *INIS*.

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

FOX, G., HIRANANDANI, S., KENNEDY, K., et al., 1990, “Fortran D language specification”, .

GARFINKEL, D., MARBACH, C. B., SHAPIRO, N. Z., 1977, “Stiff differential equations”, *Annual review of biophysics and bioengineering*, v. 6, n. 1, pp. 525–542.

GEORGE, A., LIU, W.-H., 1975, “A note on fill for sparse matrices”, *SIAM Journal on Numerical Analysis*, v. 12, n. 3, pp. 452–455.

GEORGE, J. A., 1971, *Computer implementation of the finite element method*. Relatório técnico, DTIC Document.

GROPP, W., LUSK, E., SKJELLUM, A., 1999, *Using MPI: portable parallel programming with the message-passing interface*, v. 1. MIT press.

GRUMMER, R., MERK, S., FINNEMANN, H., et al., 2000, “SIEMENS'INTEGRATED CODE SYSTEM CASCADE-3D FOR CORE DESIGN AND SAFETY ANALYSIS”, *Proc. Advances in Reactor Physics, and Mathematics and Computation into the Next Millennium (Physor 2000)*, pp. 7–11.

HAIRER, E., WANNER, G., 1996, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, v. 14, *Springer Series in Computational Mathematics*. 2nd ed. Berlin, Springer. doi: 10.1007/978-3-642-05221-7.

HAIRER, E., LUBICH, C., ROCHE, M., 1989, “The numerical solution of differential-algebraic systems by Runge-Kutta methods”, .

HAMMARLUND, P., MARTINEZ, A. J., BAJWA, A. A., et al., 2014, “Haswell: The Fourth-Generation Intel Core Processor”, *IEEE Micro*, v. 34, n. 2, pp. 6–20.

HEIMLICH, A., MOL, A., PEREIRA, C., 2011, “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.

HEIMLICH, A., SILVA, F., MARTINEZ, A., 2016, “Parallel GPU implementation of PWR reactor burnup”, *Annals of Nuclear Energy*, v. 91, pp. 135–141.

HERMANN, E., RAFFIN, B., FAURE, F., et al., 2010, “Multi-GPU and multi-CPU parallelization for interactive physics simulations”. In: *Euro-Par 2010-Parallel Processing*, Springer, pp. 235–246.

HIGHAM, N. J., 2008, *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA, Society for Industrial and Applied Mathematics. ISBN: 978-0-898716-46-7.

HIGHAM, N. J., 2005, “The scaling and squaring method for the matrix exponential revisited”, *SIAM Journal on Matrix Analysis and Applications*, v. 26, n. 4, pp. 1179–1193.

HUSARCEK, J., 2004, "Safety margins and improved plant performance", *Implications of power uprates on safety margins of nuclear power plants*, p. 133.

INSULANDER BJOERK, K., FHAGER, V., 2009, "Comparison of thorium-plutonium fuel and MOX fuel for PWRs", .

INTEL, M., 2007. "Intel math kernel library". .

KERNIGHAN, B. W., RITCHIE, D. M., EJEKLINT, P., 1988, *The C programming language*, v. 2. prentice-Hall Englewood Cliffs.

KIM, T. K., 2013, "GEN-IV Reactors". In: *Nuclear Energy*, Springer, pp. 175–201.

KINSEY, R., DUNFORD, C., TULI, J., et al., 1996, "The NUDAT/PCNUDAT program for nuclear data", *Cross Sections*, v. 1, pp. 4.

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

KROPACZEK, D. J., TURINSKY, P. J., 1991, "In-core nuclear fuel management optimization for pressurized water reactors utilizing simulated annealing", *Nuclear Technology*, v. 95, n. 9, pp. 9–31.

KUMAR, V., GRAMA, A., GUPTA, A., et al., 1994, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA.

LANG, P., 1991, "Further burnup extension-Results of IAEA's WREBUS study", *Transactions of the American Nuclear Society;(United States)*, v. 63, n. CONF-910603–.

LEVESQUE, J. M., 2012, "Application Development for Titan-A Multi-Petaflop Hybrid-Multicore MPP System". In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 1731–1821. IEEE.

LIMA, E. L., 1976, *Curso de Análise, vol. 2*.

LIMA, E. L., 1981, "Curso de Análise, v. 2", *Projeto Euclides/IMPA*.

LIMA, E. L., 2004, *Curso de análise, Volume 1*. Rio de Janeiro: Instituto Nacional de Matemática Pura e Aplicada.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., et al., 2008, "NVIDIA Tesla: A unified graphics and computing architecture", *Ieee Micro*, v. 28, n. 2, pp. 39–55.

MARTIN, K., HOFFMAN, B., 2010, *Mastering CMake*. Kitware.

MENDENHALL, W., SINCICH, T., 2006, *Statistics for Engineering and the Sciences*. Prentice-Hall, Inc.

METCALF, M., REID, J. K., COHEN, M., 2004, *Fortran 95/2003 Explained*, v. 416. Oxford University Press Oxford.

MOLER, C., VAN LOAN, C., 1978, "Nineteen dubious ways to compute the exponential of a matrix", *SIAM review*, v. 20, n. 4, pp. 801–836.

MOLER, C., VAN LOAN, C., 2003, "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later", *SIAM review*, v. 45, n. 1, pp. 3–49.

MUSSER, D. R., DERGE, G. J., SAINI, A., 2009, *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional.

NICHOLS, A., ALDAMA, D., VERPELLI, M., 2008, "Handbook of nuclear data for safeguards: database extensions, August 2008", *IAEA INDC (NDS)-0534*.

NICKOLLS, J., DALLY, W. J., 2010, "The GPU computing era", *IEEE micro*, v. 30, n. 2, pp. 56–69.

NIEPLOCHA, J., HARRISON, R. J., LITTLEFIELD, R. J., 1996, "Global arrays: A nonuniform memory access programming model for high-performance computers", *The Journal of Supercomputing*, v. 10, n. 2, pp. 169–189.

NVIDIA, C., 2012, *NVIDIAs next generation CUDA compute architecture: Kepler GK110*. Relatório técnico, Technical report, 2012.[28] http://www.top500.org.

NVIDIA, C., 2008a, "Cublas library", *NVIDIA Corporation, Santa Clara, California*, v. 15.

NVIDIA, C., 2008b. "Programming guide". b.

NVIDIA, C., 2010. "CUFFT library". .

NVIDIA, T., 2012, "K20-K20X GPU Accelerators Benchmarks", *Application Performance Technical Brief., Nvidia*.

OVERTON, M. L., 2001, *Numerical computing with IEEE floating point arithmetic*. Siam.

PAI, S., GOVINDARAJAN, R., THAZHUTHAVEETIL, M., 2010, "PLASMA: portable programming for simd heterogeneous accelerators", *HPCA/PPoPP'10*.

PARSHALL, K. H., 2006, *James Joseph Sylvester: Jewish Mathematician in a Victorian World*. JHU Press.

PELLERIN, D., THIBAULT, S., 2005, *Practical fpga programming in c*. Prentice Hall Press.

PEREIRA, C. M., LAPA, C. M., 2003, "Coarse-grained parallel genetic algorithm applied to a nuclear reactor core design optimization problem", *Annals of Nuclear Energy*, v. 30, n. 5, pp. 555–565.

PISSANETZKY, S., 1984, *Sparse matrix technology*. Academic Press.

PRATA, F. S., SILVA, F. C., MARTINEZ, A. S., 2013, "Solution of the isotopic depletion equations using decomposition method and analytical solutions", *Progress in Nuclear Energy*, v. 69, pp. 53–58.

PRICE, D., CLARK, M., BARSDELL, B., et al., 2014, "Optimizing performance per watt on GPUs in High Performance Computing: temperature, frequency and voltage effects", *arXiv preprint arXiv:1407.8116*.

PUSA, M., LEPPÄNEN, J., 2010, "Computing the matrix exponential in burnup calculations", *Nuclear science and engineering*, v. 164, n. 2, pp. 140–150.

RAMIREZ, W. F., 1997, *Computational methods for process simulation*. Butterworth-Heinemann.

RANGARAJAN, G., 1991, "A library implementation of POSIXthreads", *Masterś Project Report, Florida State University Department of Computer Science*.

REINDERS, J., 2007, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.".

RENAULT, C., HRON, M., KONINGS, R., et al., 2009, "The Molten Salt Reactor (MSR) in generation 4: overview and perspectives", .

RICE, J. R., 1981, *Matrix computations and mathematical software*. McGraw-Hill New York.

SAAD, Y., SCHULTZ, M. H., 1986, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", *SIAM Journal on scientific and statistical computing*, v. 7, n. 3, pp. 856–869.

SAAD, Y., 1994. "SPARSKIT: a basic tool kit for sparse matrix computations". .

SAAD, Y., 2003, *Iterative methods for sparse linear systems*. Siam.

SALVATORES, M., 2005, "Nuclear fuel cycle strategies including partitioning and transmutation", *Nuclear Engineering and Design*, v. 235, n. 7, pp. 805–816.

SARMA, D., SARMA, K. K., 2015, "Multicore Parallel Computing and DSP Processor for the Design of Bio-inspired Soft Computing Framework for Speech and Image Processing Applications". In: *Recent Trends in Intelligent and Emerging Systems*, Springer, pp. 125–134.

SATISH, N., KIM, C., CHHUGANI, J., et al., 2010, "Fast sort on cpus, gpus and intel mic architectures", *Intel Labs*.

SCHÄLING, B., 2011, *The boost C++ libraries*. Boris Schäling.

SERP, J., ALLIBERT, M., BENEŠ, O., et al., 2014, "The molten salt reactor (MSR) in generation IV: Overview and perspectives", *Progress in Nuclear Energy*, v. 77, pp. 308–319.

SHAMPINE, L. F., 1977, "Stiffness and nonstiff differential equation solvers, II: detecting stiffness with Runge-Kutta methods", *ACM Transactions on Mathematical Software (TOMS)*, v. 3, n. 1, pp. 44–53.

SIMON, B., 2009, *Orthogonal polynomials on the unit circle*. American Mathematical Soc.

SINAP, A., VAN ASSCHE, W., 1994, "Polynomial interpolation and Gaussian quadrature for matrix-valued functions", *Linear algebra and its applications*, v. 207, pp. 71–114.

STROUSTRUP, B., 1986, *The C++ programming language*. Pearson Education India.

TOROKHTI, A., HOWLETT, P., 1971, *The Padé approximant in theoretical physics*, v. 71. Academic Press.

TURINSKY, P. J., KELLER, P. M., ABDEL-KHALIK, H. S., 2005, "Evolution of nuclear fuel management and reactor operational aid tools", *Nuclear Engineering and Technology*, v. 37, n. 1, pp. 79–90.

VILLADSEN, J., STEWART, W., 1967, "Solution of boundary-value problems by orthogonal collocation", *Chemical Engineering Science*, v. 22, n. 11, pp. 1483–1501.

VOLKOV, V., 2010, "Better performance at lower occupancy". In: *Proceedings of the GPU Technology Conference, GTC*, v. 10. San Jose, CA.

WAGNER, J., 2001, *Computational Benchmark for Estimation of Reactivity Margin from Fission Products and Minor Actinides in PWR Burnup Credit*. Relatório técnico, Oak Ridge National Lab., TN (United States). Funding organisation: Nuclear Regulatory Commission (United States).

WAINTRAUB, M., LAPA, C., MOL, A., et al., 2011, "THEWASP library. Thermodynamic water and steam properties library in GPU", .

WEISSTEIN, E. W., 2003, "Gershgorin Circle Theorem", .

WESTLAKE, J. R., 1968, *A handbook of numerical matrix inversion and solution of linear equations*, v. 767. Wiley New York.

WHALEY, R. C., DONGARRA, J. J., 1998, "Automatically tuned linear algebra software". In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 1–27. IEEE Computer Society.

WHITEHEAD, N., FIT-FLOREA, A., 2011, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs", *rn (A+ B)*, v. 21, pp. 1–1874919424.

WORRALL, A., 2009, "Potential Plutonium Utilization in Future UK PWRs". In: *Proceedings of the Institute of Nuclear Materials Management 50 th Annual Meeting, Tucson, AZ.*

YANNAKAKIS, M., 1981, "Computing the minimum fill-in is NP-complete", *SIAM Journal on Algebraic Discrete Methods*, v. 2, n. 1, pp. 77–79.

YOU, H., BUDIARDJA, R., BETRO, V., et al., 2013, *Performance Comparison of Scientific Applications on Cray Architectures*. Relatório técnico, Oak Ridge National Laboratory (ORNL).

ZHU, A., XU, Y., GRAHAM, A., et al., 2015, "Transient Methods for Pin-resolved Whole Core Transport using the 2D–1D Methodology in MPACT". In: *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (M&C 2015), American Nuclear Society, Nashville, TN, USA, April*, pp. 19–23.

Apêndice A

Resultados

Com o objetivo de ilustrar o comportamento das concentrações dos nuclídeos avaliados neste trabalho apresentam-se as figuras a seguir, construídas utilizando os valores obtidos pelo método de colocação de Gauss-Jacobi de ordem 3 em sua versão sequencial. Na figura A.1 observa-se o comportamento da concentração dos isótopos de Urânio ao longo do ciclo.

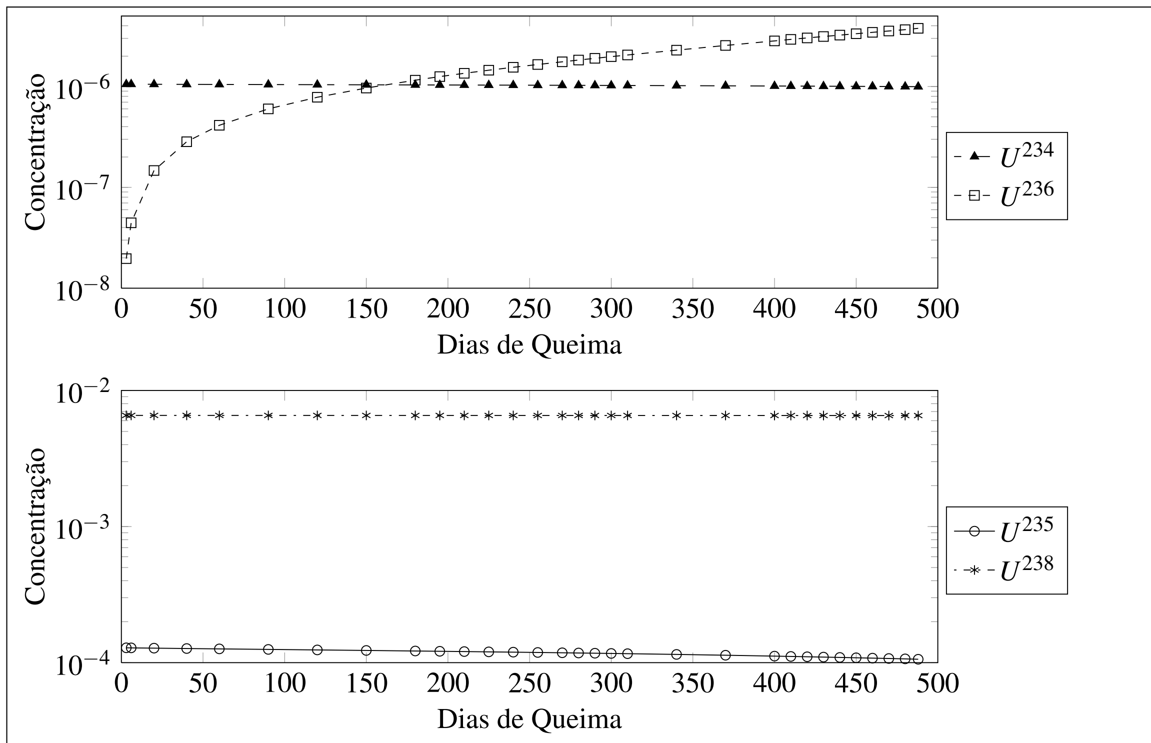

Figura A.1: Concentrações dos Isótopos de Urânio *vs* Tempo de Queima.

O desvio relativo percentual $\delta = 100 \cdot \sum_{i=1}^{n} \frac{|x_i - y_i|_1}{max\{x_i, y_i\}}$, visto na definição (3.8) foi

utilizado na avaliação da diferença entre os resultados do CNFR (y_i) e os métodos sob teste (x_i). Nas figuras A.2, A.3, A.4 e A.5 podem-se observar o comportamento do desvio percentual dos isótopos U^{234}, U^{235}, U^{236} e U^{238} respectivamente em relação ao padrão.

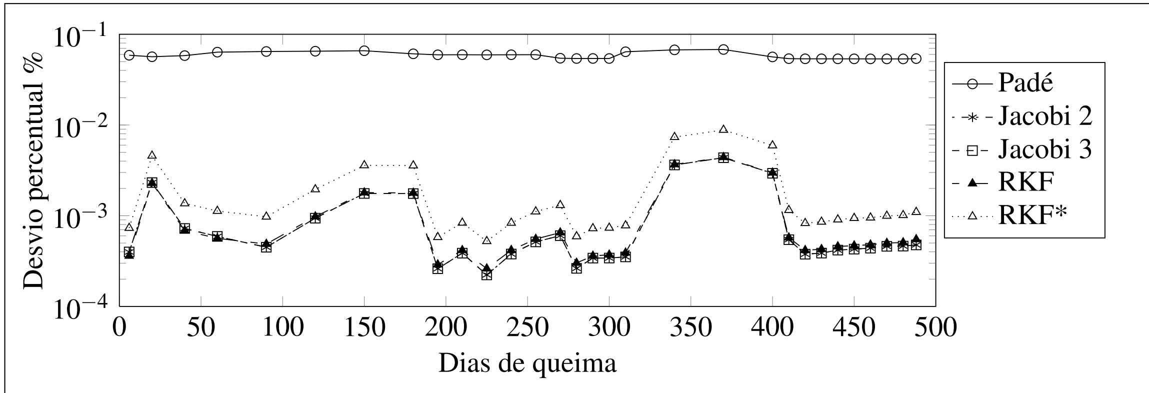

Figura A.2: Desvio na avaliação do U^{234}.

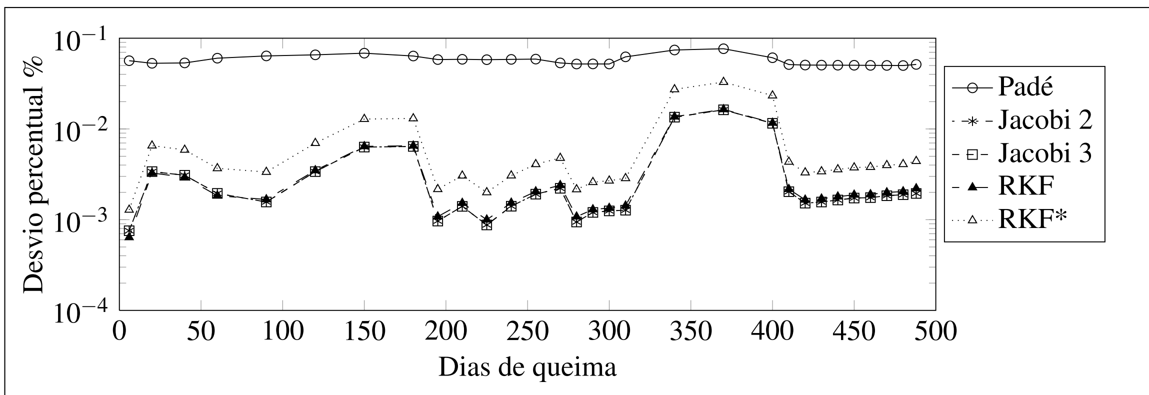

Figura A.3: Desvio na avaliação do U^{235}.

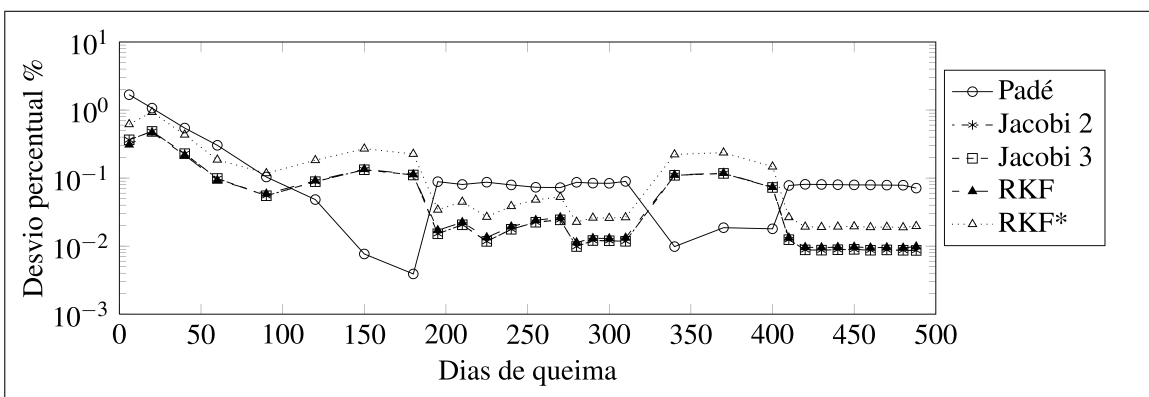

Figura A.4: Desvio na avaliação do U^{236}.

Na figura A.6 observa-se o comportamento da concentração dos isótopos de Netúnio ao longo do ciclo.

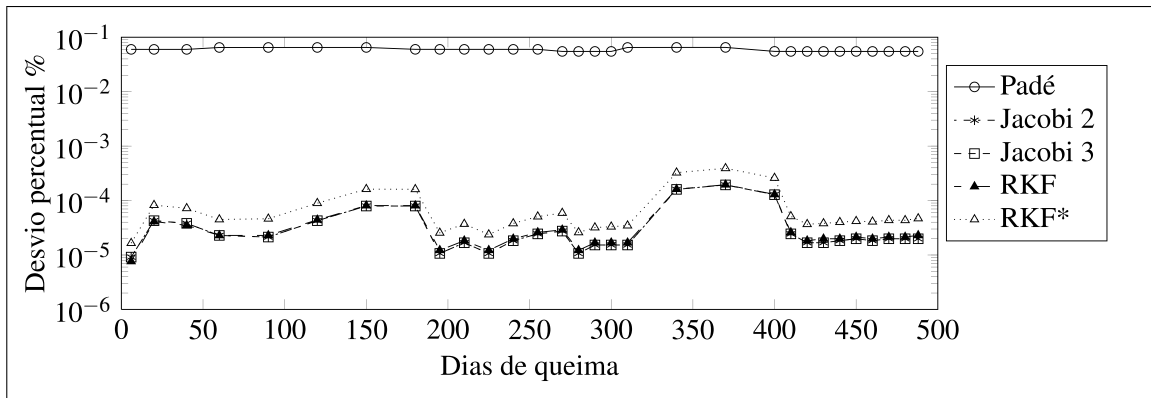

Figura A.5: Desvio na avaliação do U^{238}.

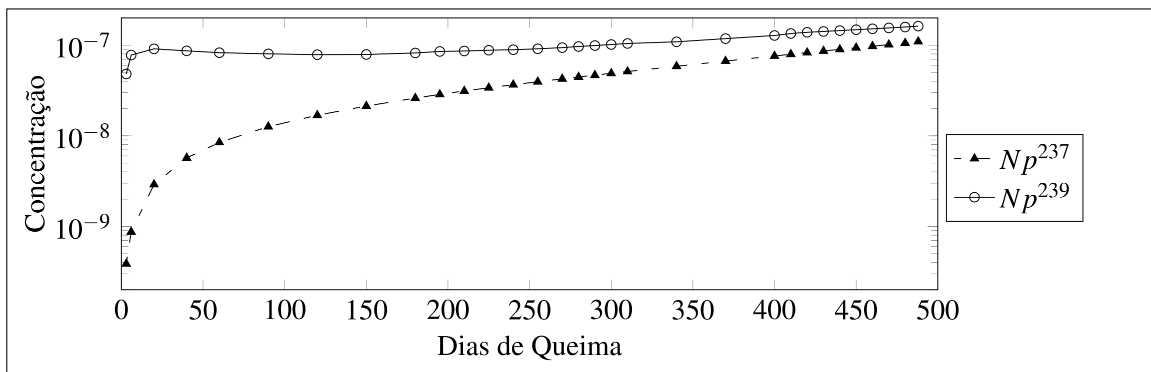

Figura A.6: Concentrações dos Isotopos de Netúnio *vs* Tempo de Queima.

Nas figuras A.7 e A.8 podem-se observar o comportamento do desvio percentual dos isótopos de Np^{237}, Np^{239}, respectivamente.

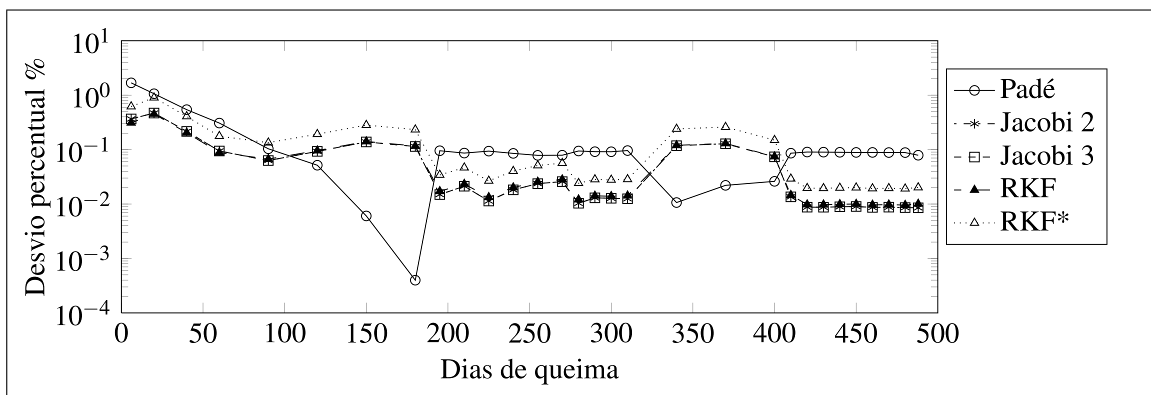

Figura A.7: Desvio na avaliação do Np^{237}.

Na figura A.9 observa-se o comportamento da concentração dos isótopos de Plutônio ao longo do ciclo.

Nas figuras A.10, A.11,A.12, A.13, e A.14 podem-se observar o comportamento do desvio percentual dos isótopos Pu^{238}, Pu^{239}, Pu^{240}, Pu^{241} e Pu^{242} respectivamente.

Na figura A.15 observa-se o comportamento da concentração dos isótopos de Amerí-

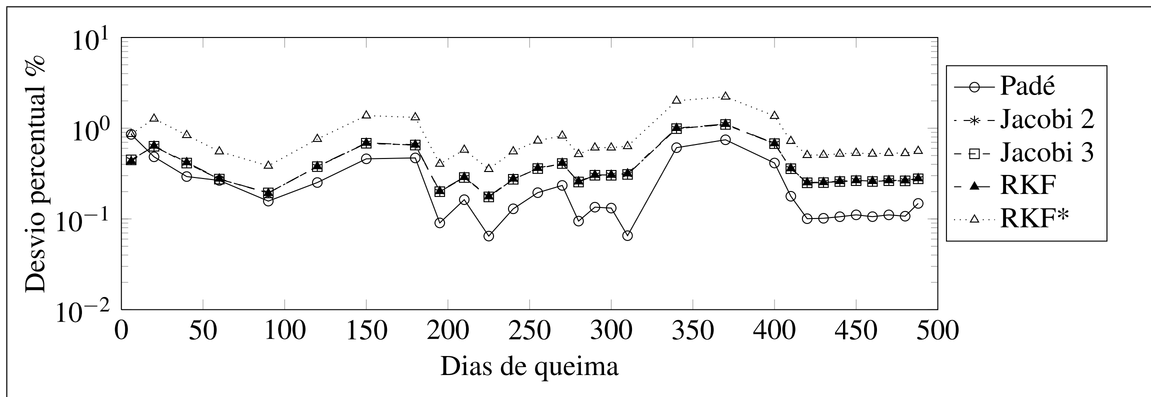

Figura A.8: Desvio na avaliação do Np^{239}.

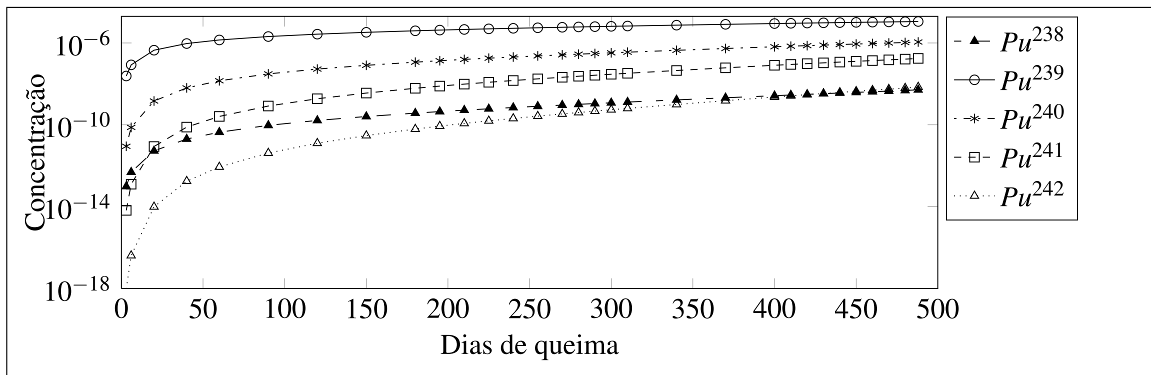

Figura A.9: Concentrações dos Isótopos de Plutônio *vs* Tempo de Queima.

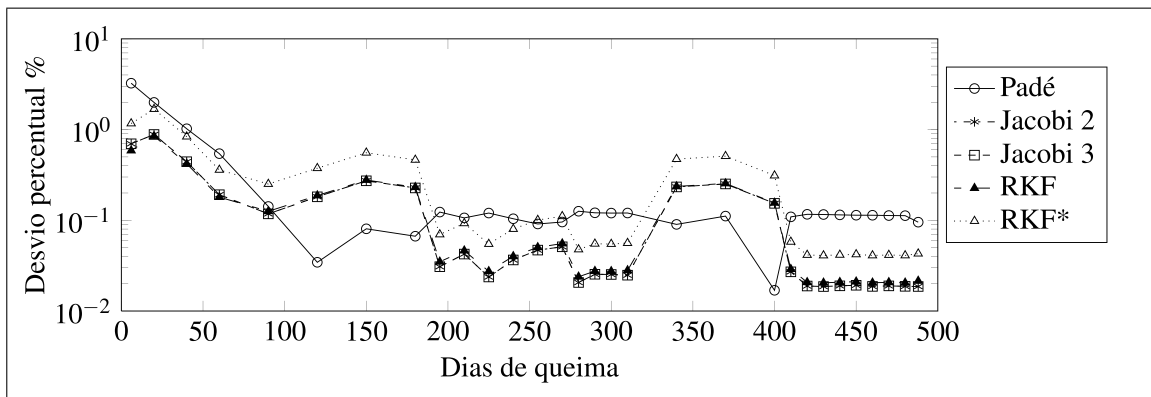

Figura A.10: Desvio na avaliação do Pu^{238}.

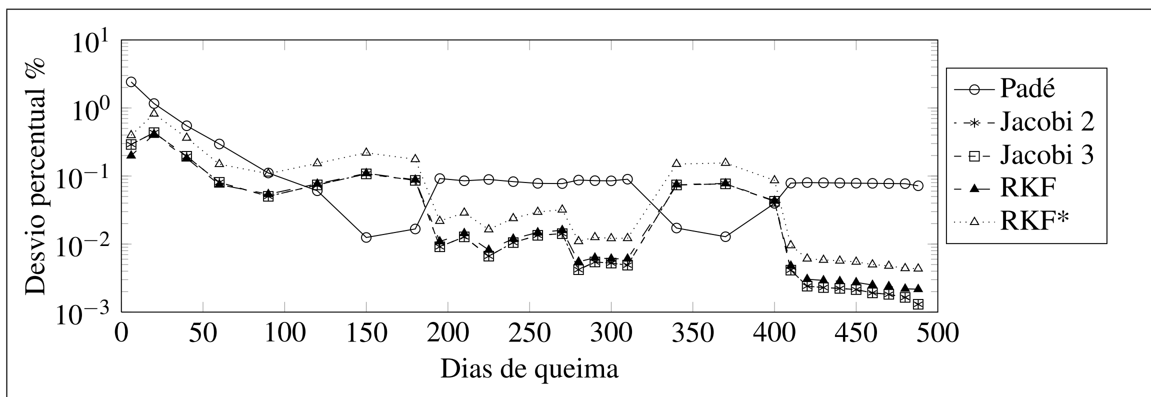

Figura A.11: Desvio na avaliação do Pu^{239}.

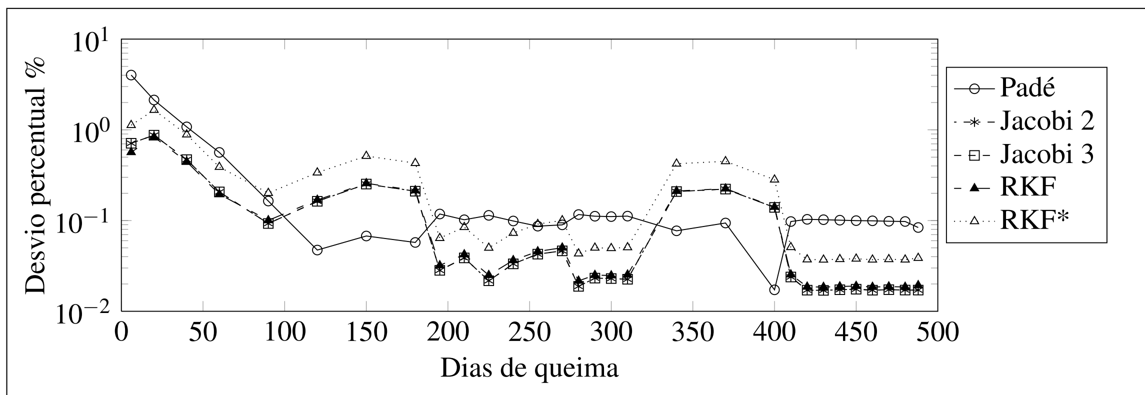

Figura A.12: Desvio na avaliação do Pu^{240}.

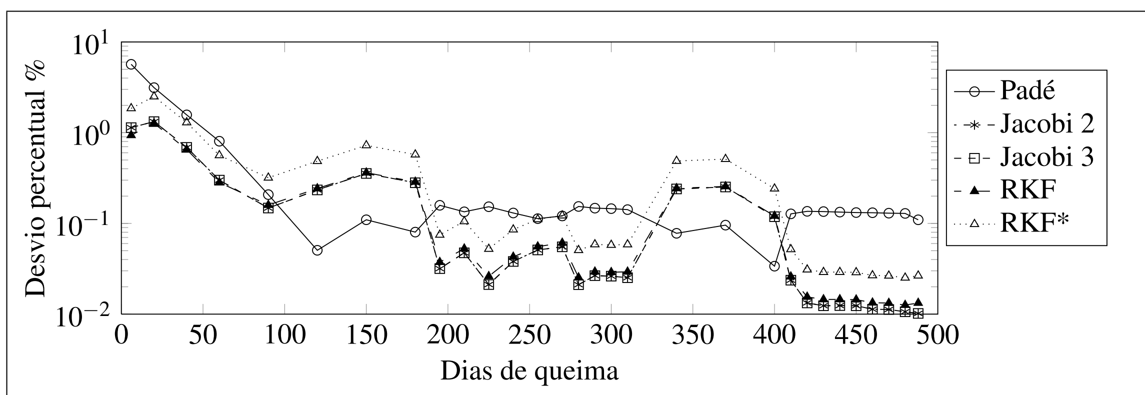

Figura A.13: Desvio na avaliação do Pu^{241}.

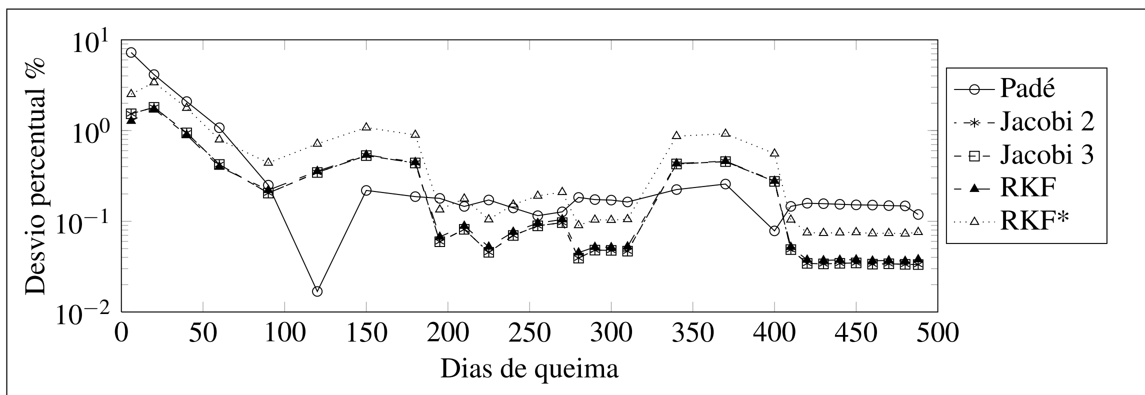

Figura A.14: Desvio na avaliação do Pu^{242}.

cio ao longo do ciclo.

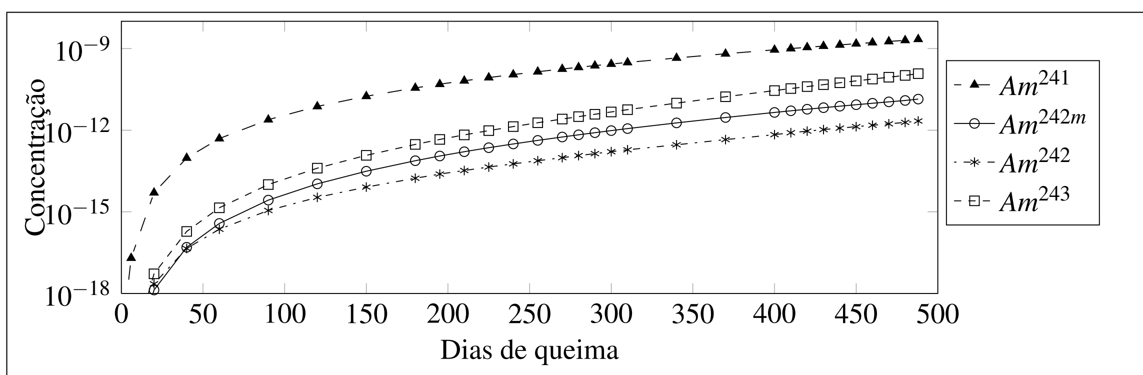

Figura A.15: Concentrações dos Isótopos de Amerício *vs* Tempo de Queima.

Nas figuras A.16, A.17,A.18 e A.19 podem-se observar o comportamento do desvio percentual dos isótopos de Am^{241}, Am^{242}, Am^{242m} e Am^{243} respectivamente.

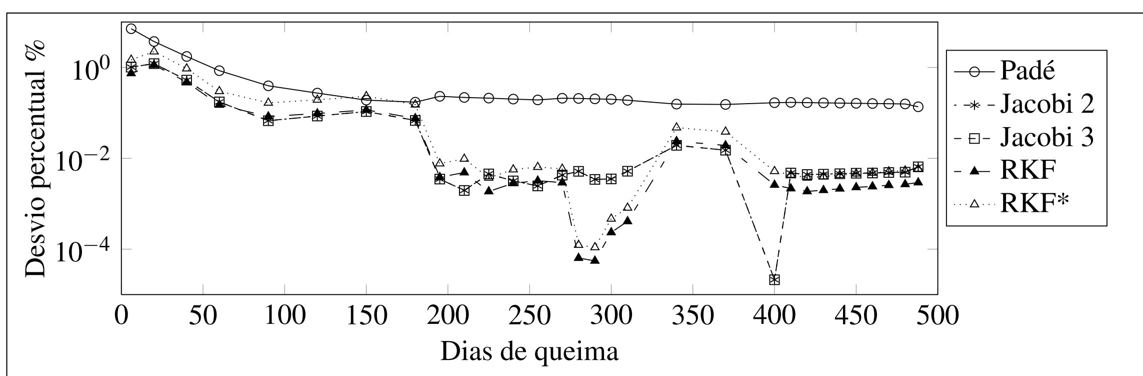

Figura A.16: Desvio na avaliação do Am^{241}.

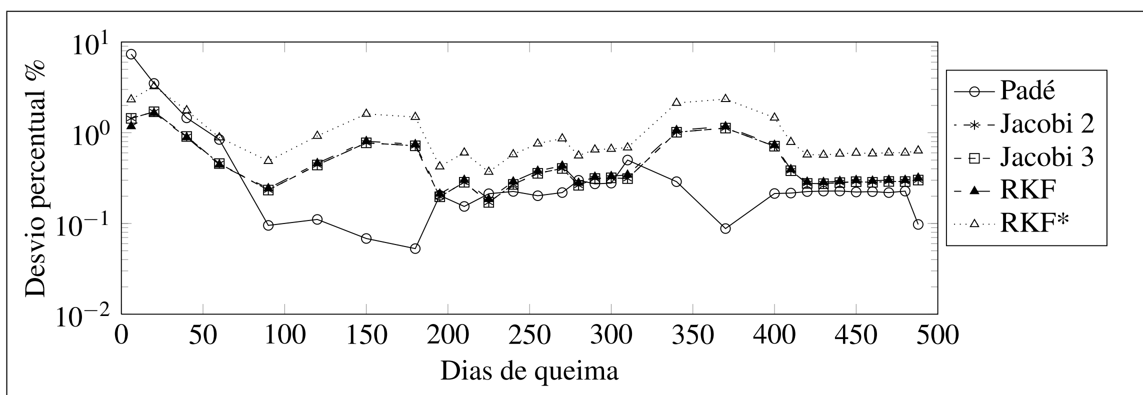

Figura A.17: Desvio na avaliação do Am^{242}.

Na figura A.20 observa-se o comportamento da concentração dos isótopos de Cúrio ao longo do ciclo.

Nas figuras A.21 e A.22 podem-se observar o comportamento do desvio percentual dos isótopos Cm^{242} e Cm^{244} respectivamente.

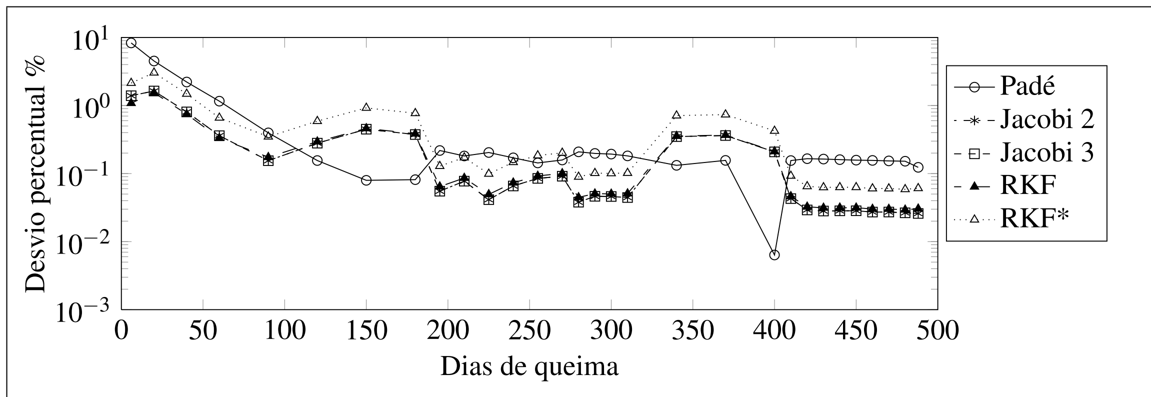

Figura A.18: Desvio na avaliação do Am^{242m}.

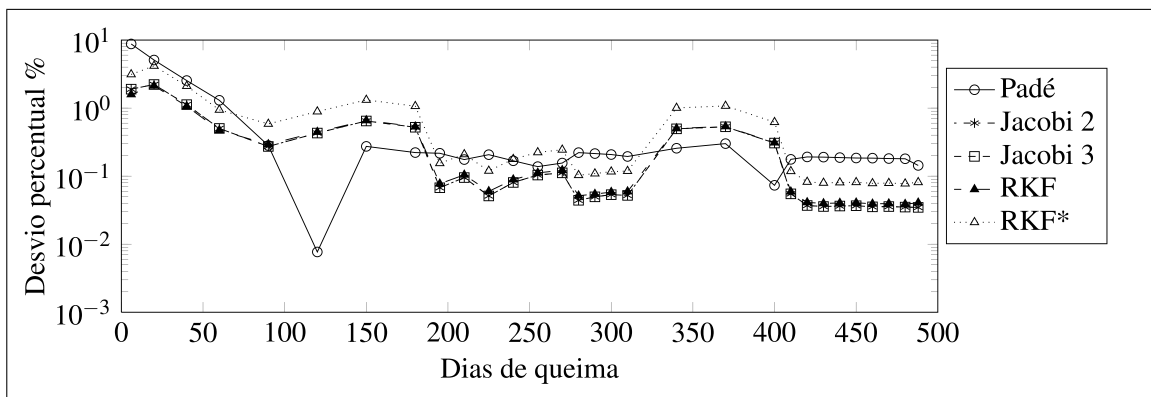

Figura A.19: Desvio na avaliação do Am^{243}.

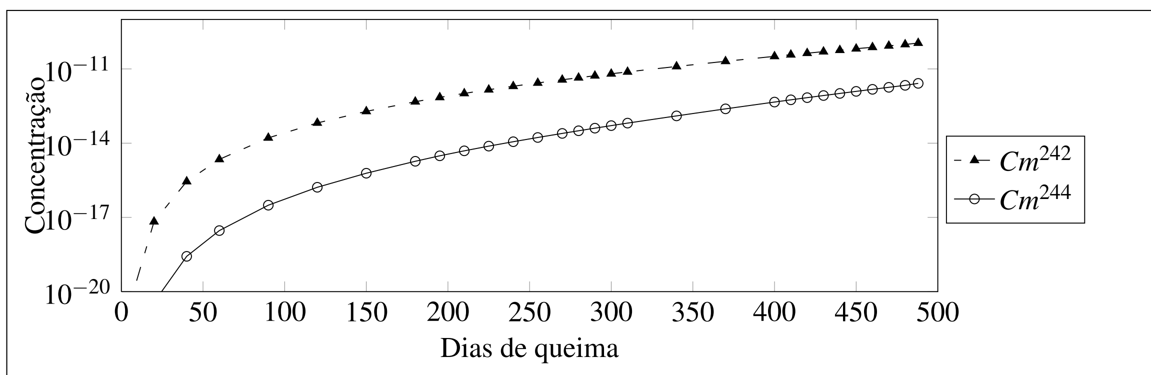

Figura A.20: Concentrações dos Isotopos Cúrio *vs* Tempo de Queima.

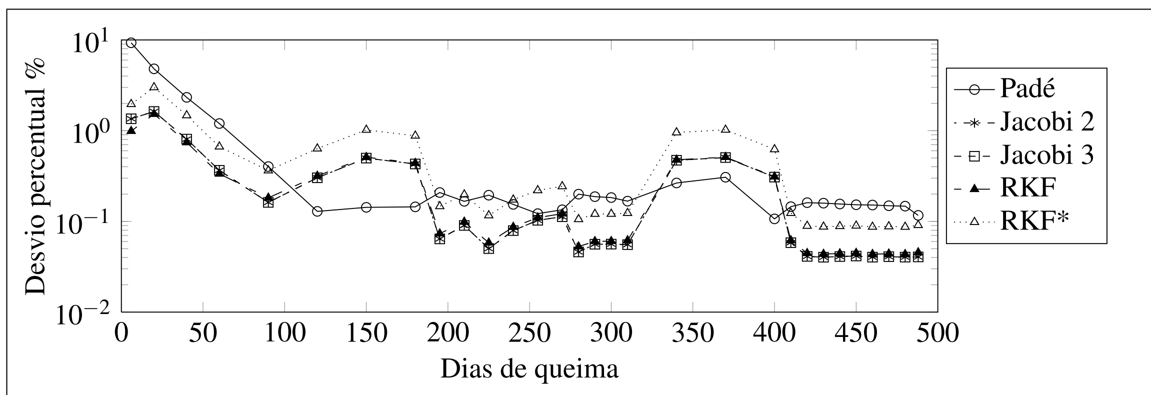

Figura A.21: Desvio na avaliação do Cm^{242}.

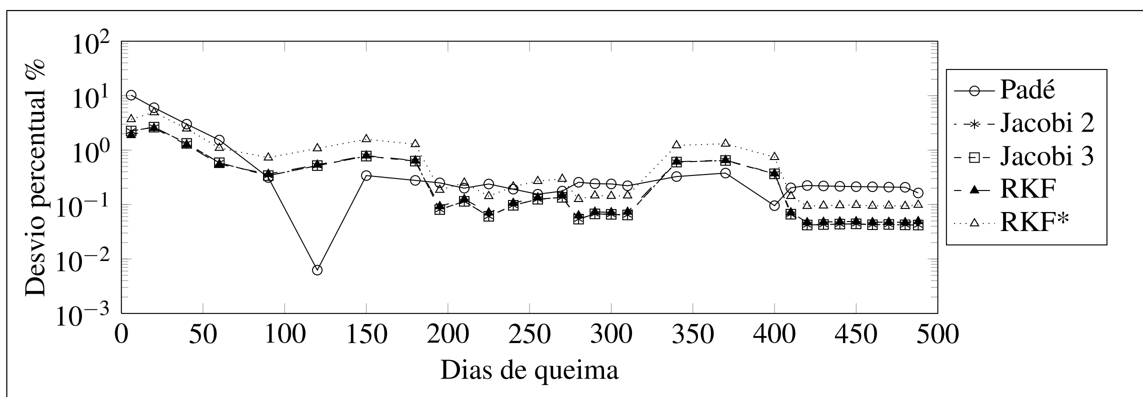

Figura A.22: Desvio na avaliação do Cm^{244}.

Na figura A.23 observa-se o comportamento da concentração dos isótopos Zr^{95}, Mo^{95}, Nb^{95} e Ru^{103} ao longo do ciclo.

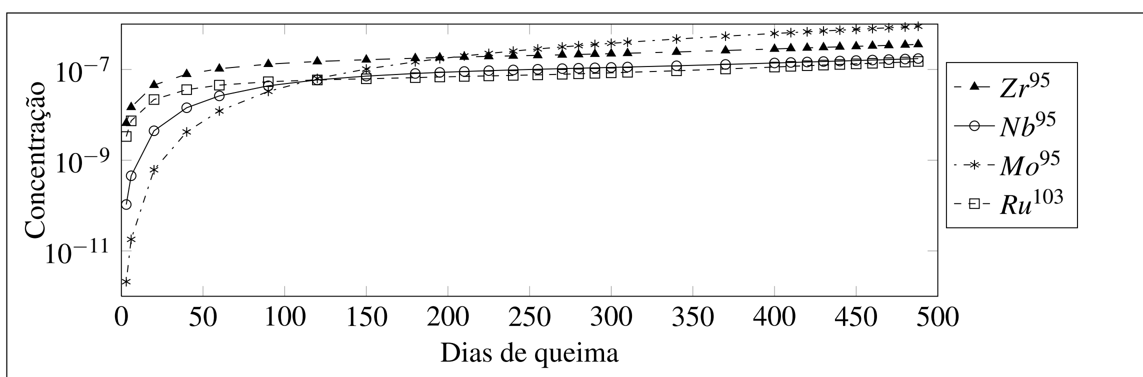

Figura A.23: Concentrações dos Isotopos Zr^{95}, Mo^{95}, Nb^{95} e Ru^{103} *vs* Tempo de Queima.

Nas figuras A.24,A.25,A.26 e A.27 podem-se observar o comportamento do desvio percentual dos isótopos Zr^{95}, Mo^{95}, Nb^{95} e Ru^{103} respectivamente.

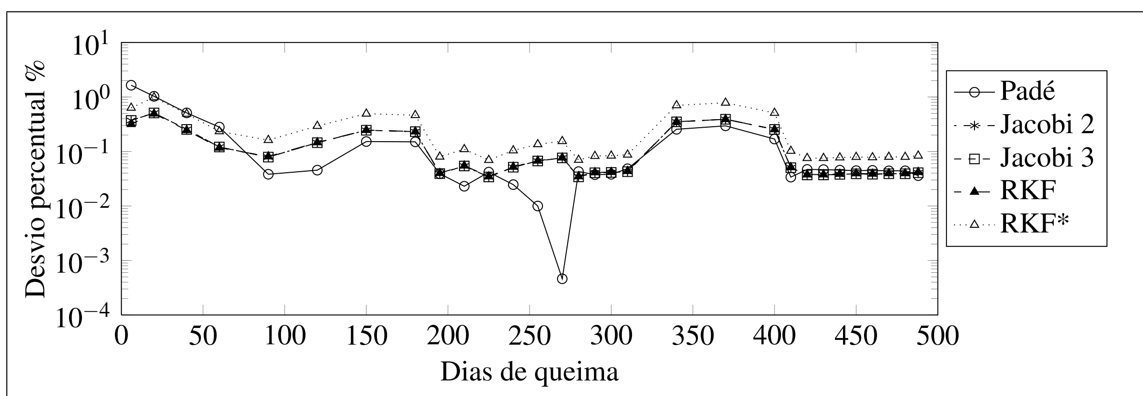

Figura A.24: Desvio na avaliação do Zr^{95}.

Na figura A.28 observa-se o comportamento da concentração dos isótopos de Ródio ao longo do ciclo.

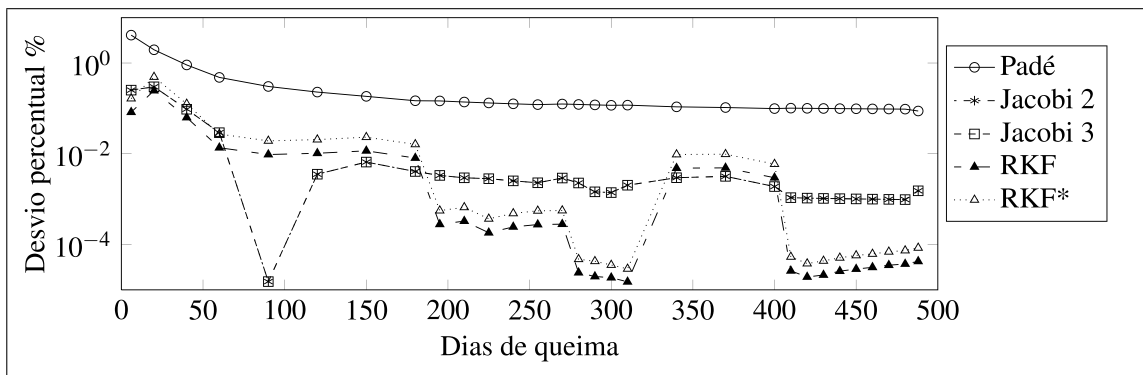

Figura A.25: Desvio na avaliação do Mo^{95}.

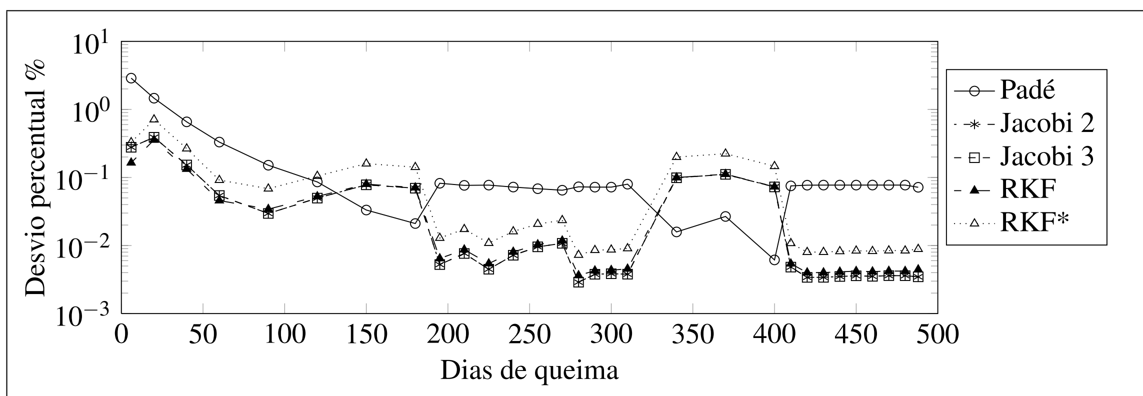

Figura A.26: Desvio na avaliação do Nb^{95}.

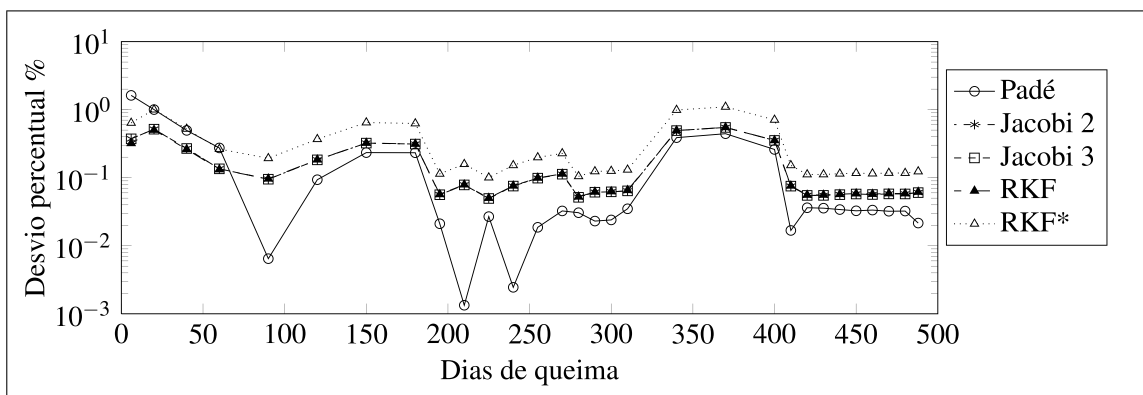

Figura A.27: Desvio na avaliação do Ru^{103}.

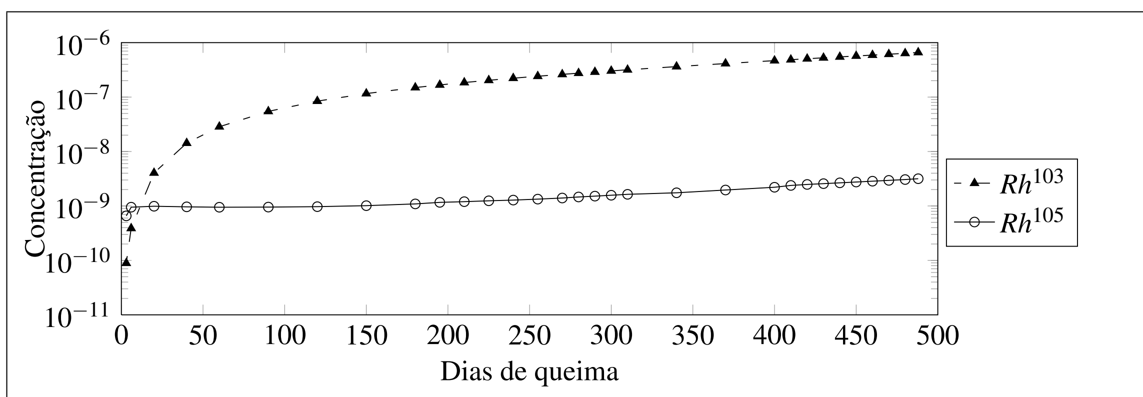

Figura A.28: Concentrações dos Isotopos de Ródio *vs* Tempo de Queima.

Nas figuras A.29 e A.30 podem-se observar o comportamento do desvio percentual dos isótopos Rh^{103} e Rh^{105} respectivamente.

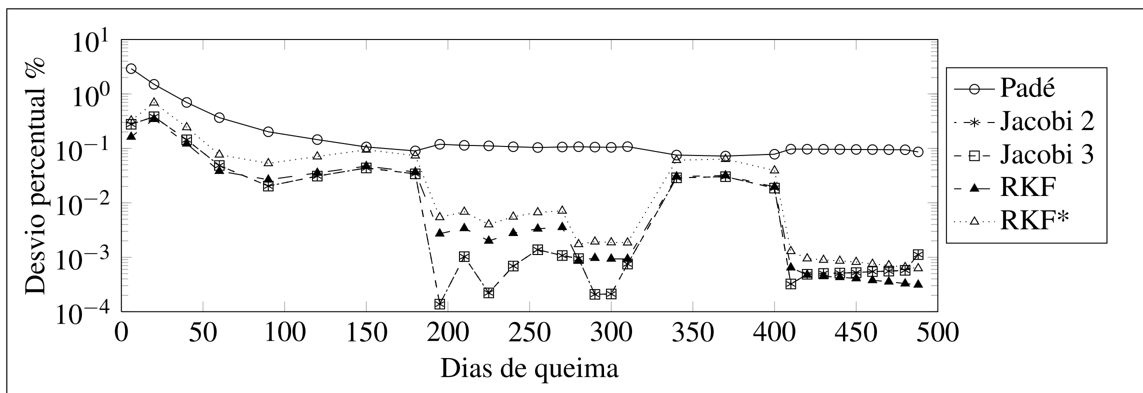

Figura A.29: Desvio na avaliação do Rh^{103}.

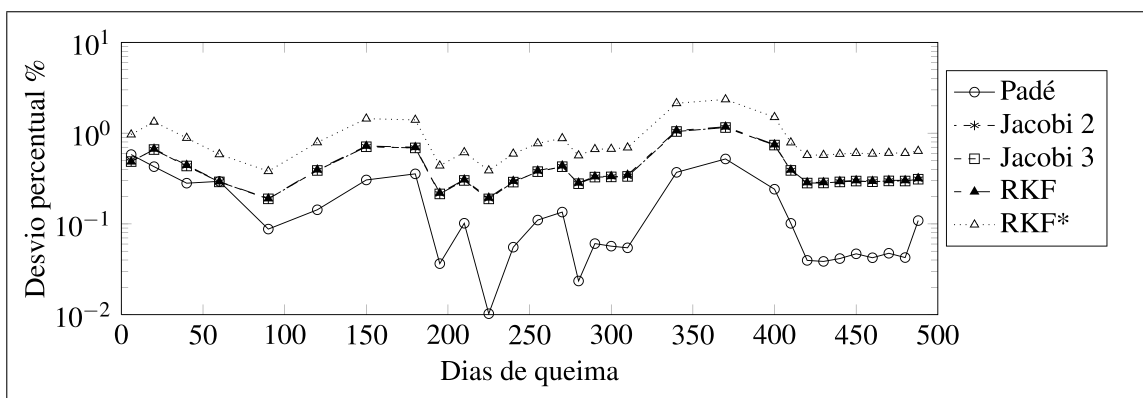

Figura A.30: Desvio na avaliação do Rh^{105}.

Na figura A.31 observa-se o comportamento da concentração dos isótopos de Iodo e Xenônio ao longo do ciclo.

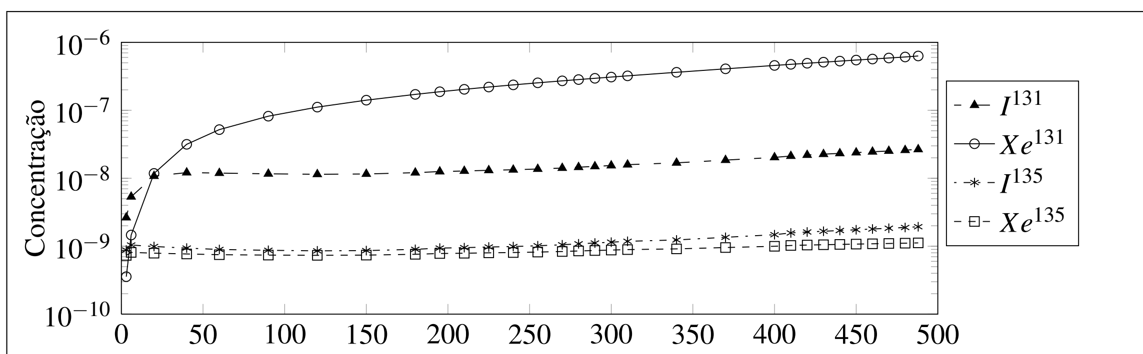

Figura A.31: Concentrações dos Isotopos de Iodo e Xenônio *vs* Tempo de Queima.

Nas figuras A.32, A.33,A.34 e A.35 podem-se observar o comportamento do desvio percentual dos isótopos I^{131}, Xe^{131}, I^{135} e Xe^{135} respectivamente.

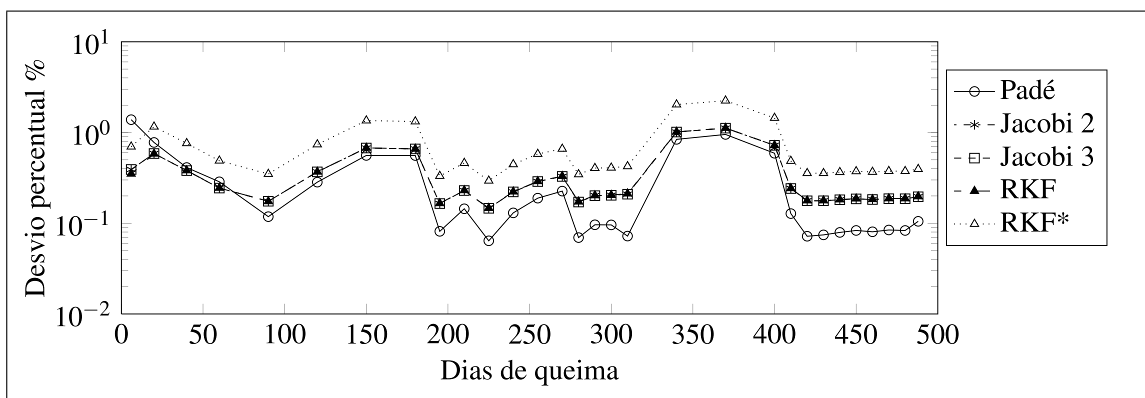

Figura A.32: Desvio na avaliação do I^{131}.

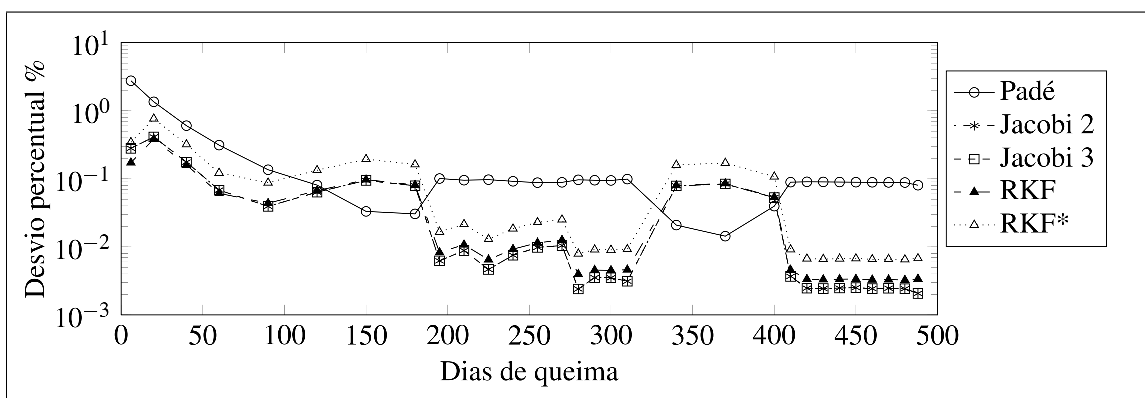

Figura A.33: Desvio na avaliação do Xe^{131}.

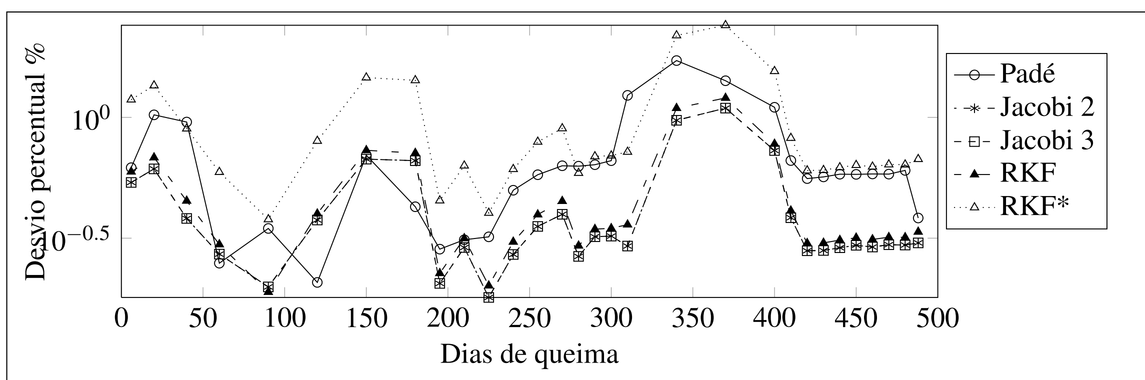

Figura A.34: Desvio na avaliação do I^{135}.

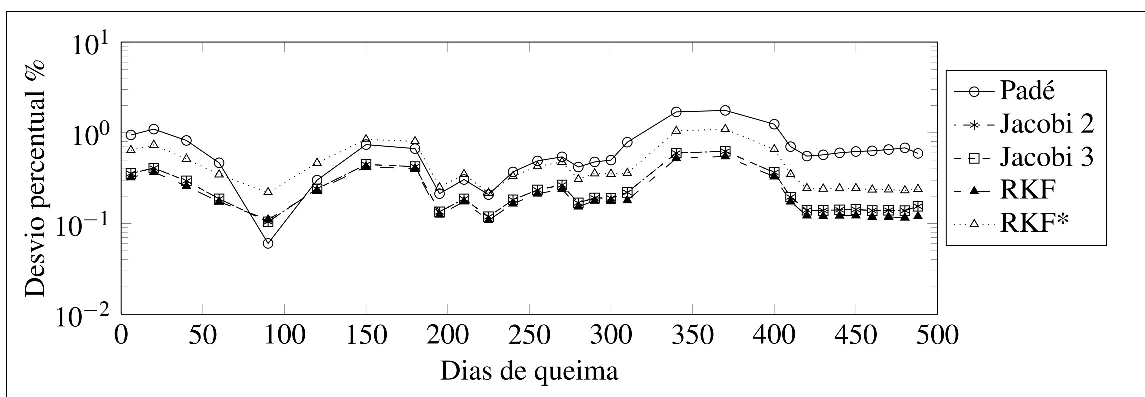

Figura A.35: Desvio na avaliação do Xe^{135}.

Na figura A.36 observa-se o comportamento da concentração dos isótopos Pr^{143} e Nd^{143} ao longo do ciclo.

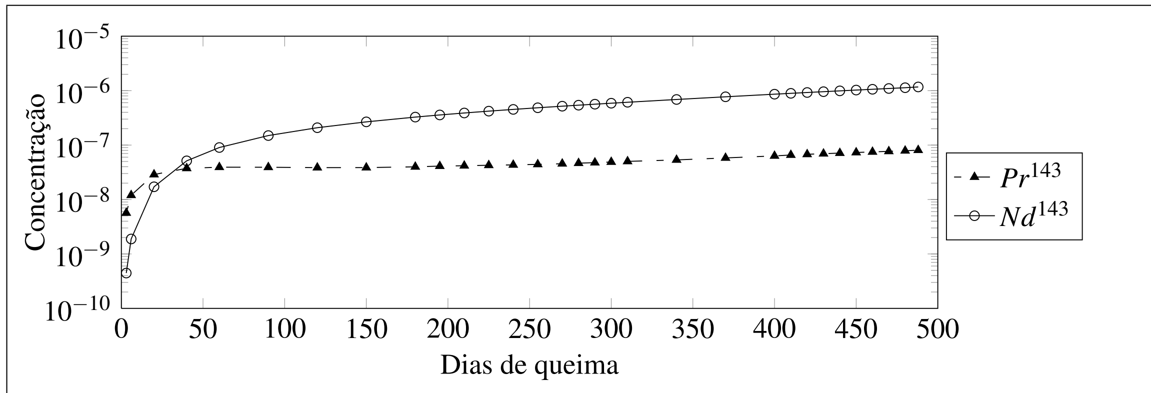

Figura A.36: Concentrações dos Isotopos Pr^{143} e Nd^{143} *vs* Tempo de Queima.

Nas figuras A.37 e A.38 podem-se observar o comportamento do desvio percentual dos isótopos Pr^{143} e Nd^{143} respectivamente.

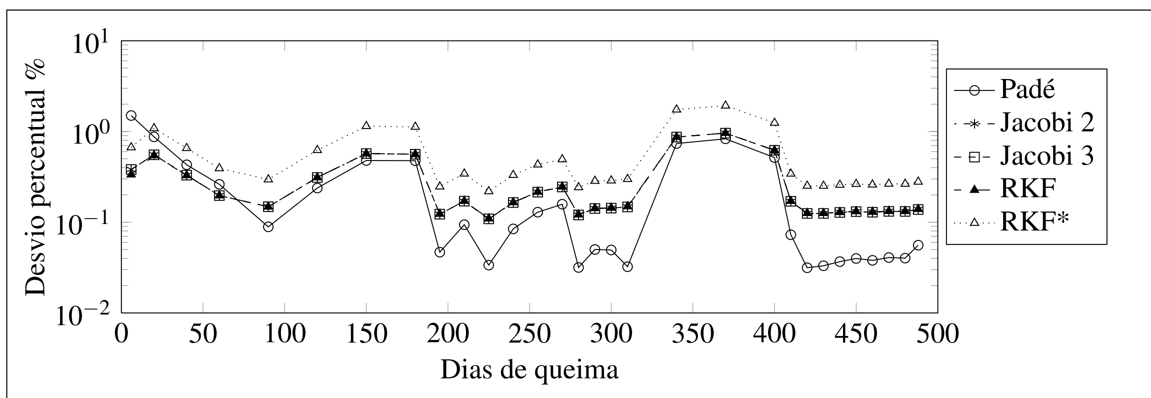

Figura A.37: Desvio na avaliação do Pr^{143}.

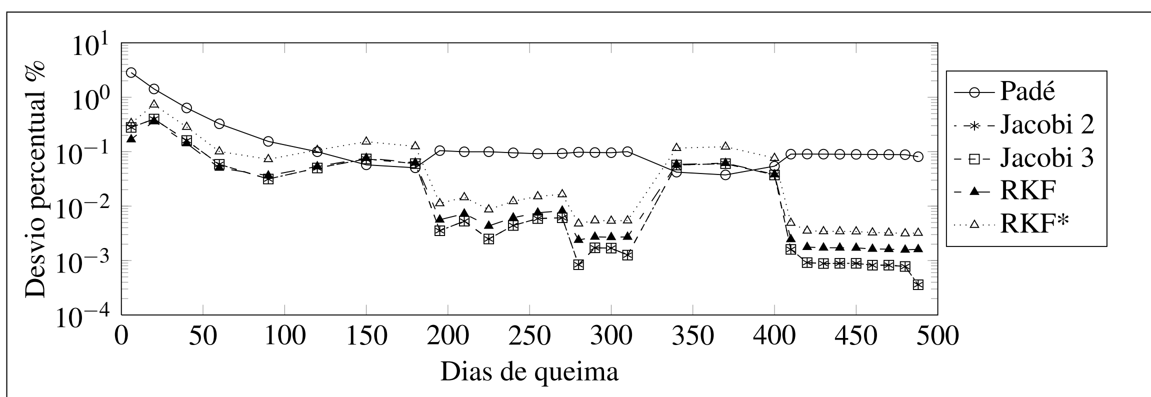

Figura A.38: Desvio na avaliação do Nd^{143}.

Na figura A.39 observa-se o comportamento da concentração dos isótopos de Promécio ao longo do ciclo.

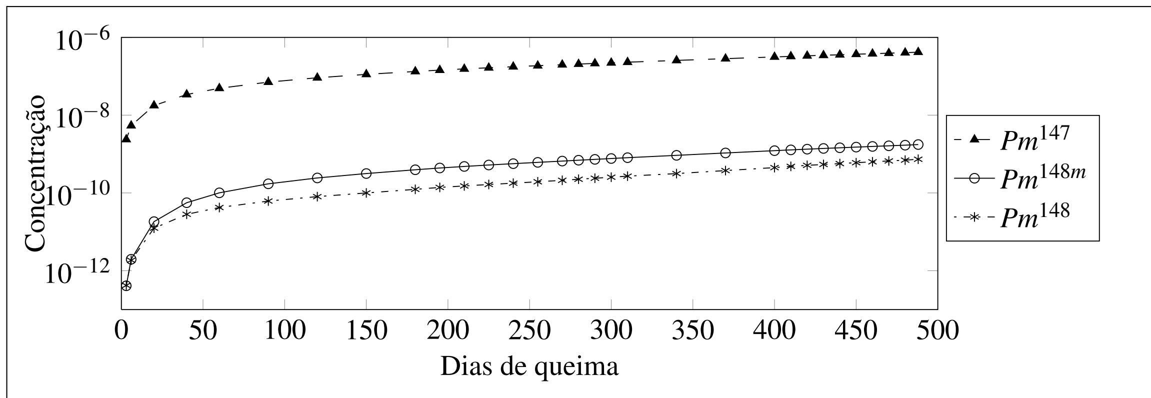

Figura A.39: Concentrações dos Isotopos de Promécio *vs* Tempo de Queima.

Nas figuras A.40, A.41 e A.42 podem-se observar o comportamento do desvio percentual dos isótopos Pm^{147},Pm^{148m} e Pm^{148} respectivamente.

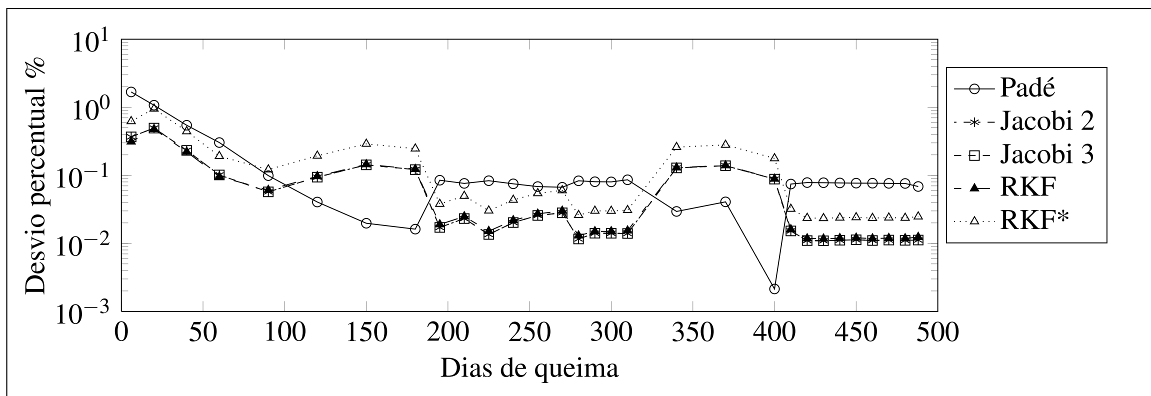

Figura A.40: Desvio na avaliação do Pm^{147}.

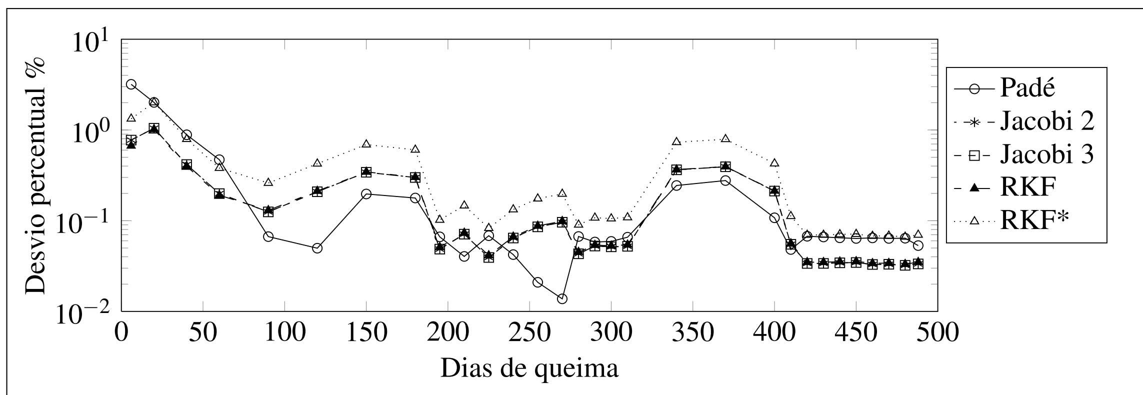

Figura A.41: Desvio na avaliação do Pm^{148m}.

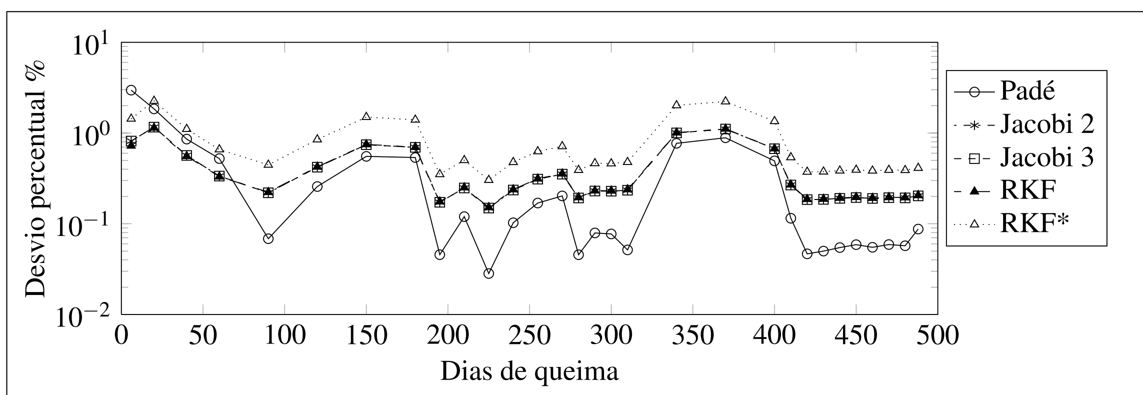

Figura A.42: Desvio na avaliação do Pm^{148}.

Na figura A.43 observa-se o comportamento da concentração dos isótopos de Samário ao longo do ciclo.

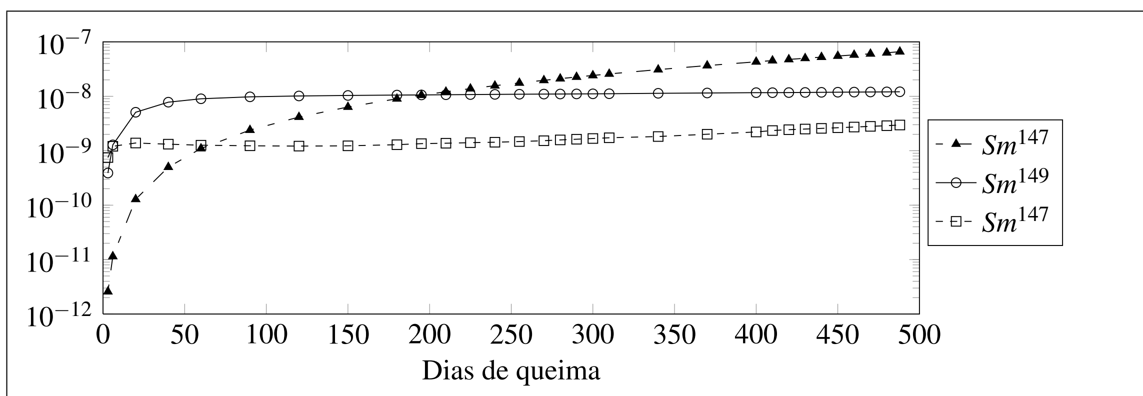

Figura A.43: Concentrações dos Isotopos de Samário *vs* Tempo de Queima.

Nas figuras A.44, A.45 e A.46 podem-se observar o comportamento do desvio percentual dos isótopos Sm^{147},Pm^{149} e Sm^{149} respectivamente.

Na figura A.47 observa-se o comportamento da concentração dos isótopos Eu^{155} e Gd^{155} ao longo do ciclo.

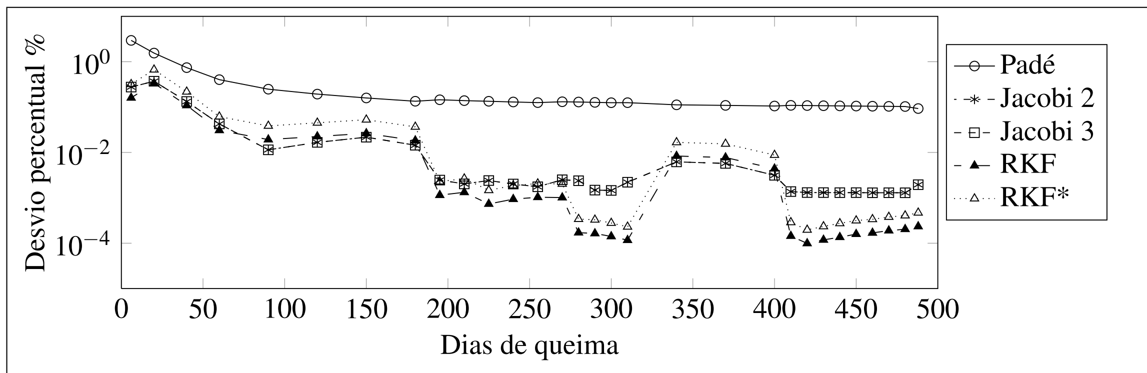

Figura A.44: Desvio na avaliação do Sm^{147}.

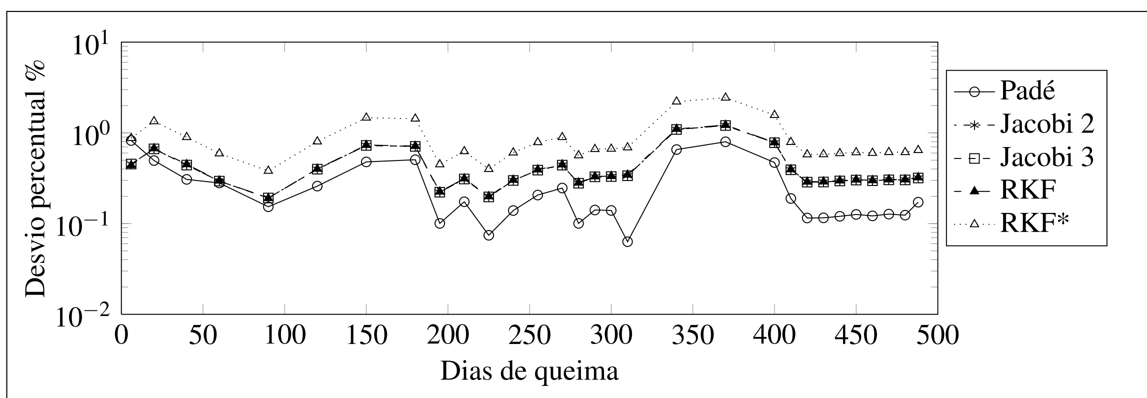

Figura A.45: Desvio na avaliação do Pm^{149}.

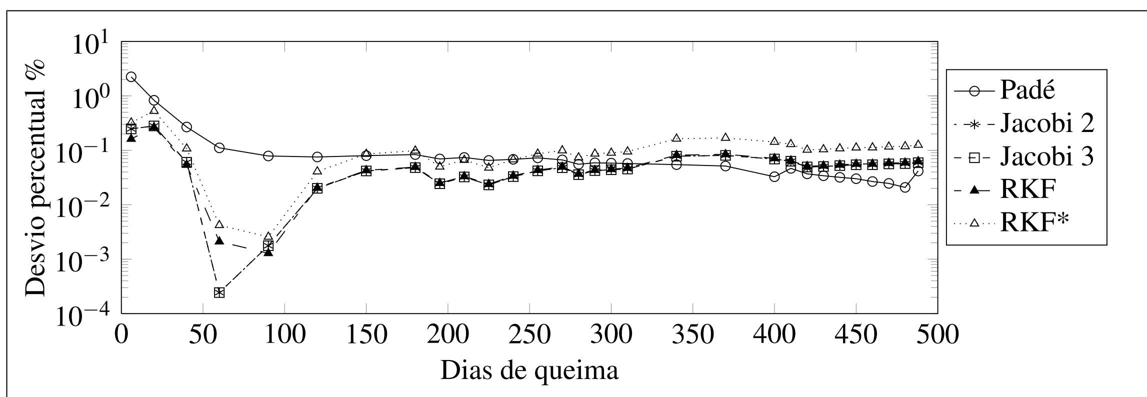

Figura A.46: Desvio na avaliação do Sm^{149}.

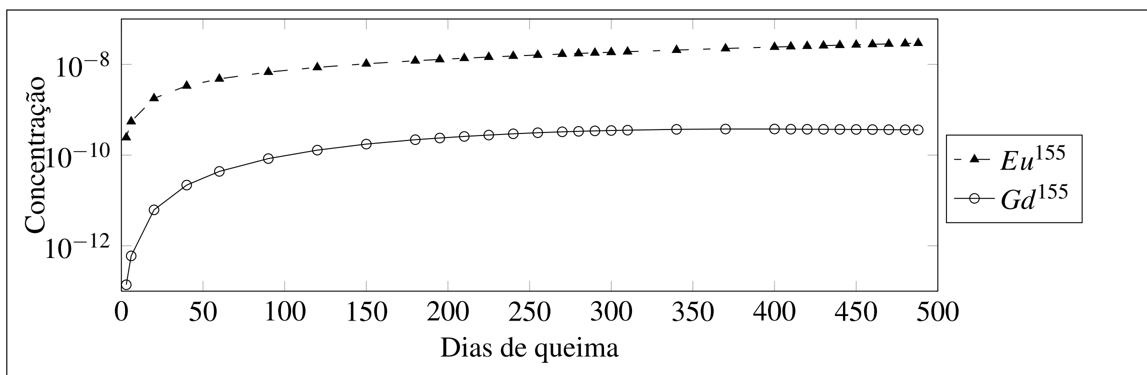

Figura A.47: Concentrações dos Isotopos Eu^{155} e Gd^{155} *vs* Tempo de Queima.

Nas figuras A.48 e A.49 podem-se observar o comportamento do desvio percentual dos isótopos Eu^{155} e Gd^{155} respectivamente.

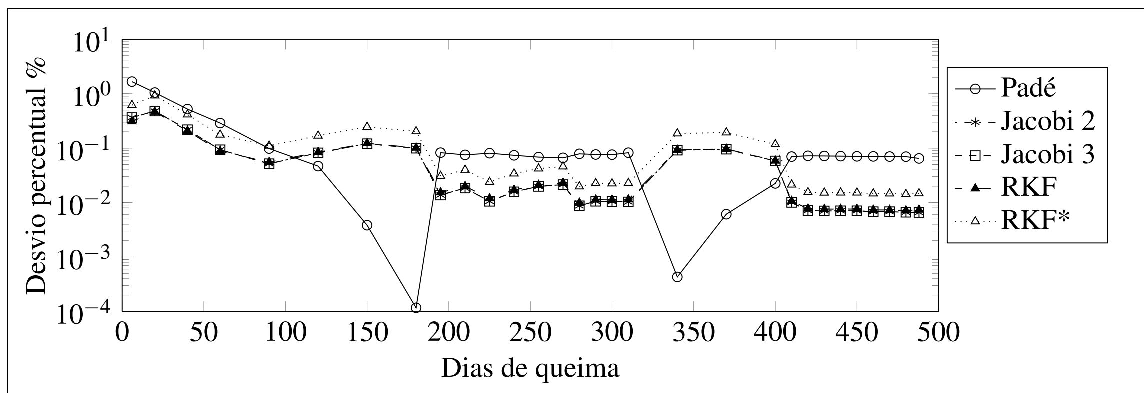

Figura A.48: Desvio na avaliação do Eu^{155}.

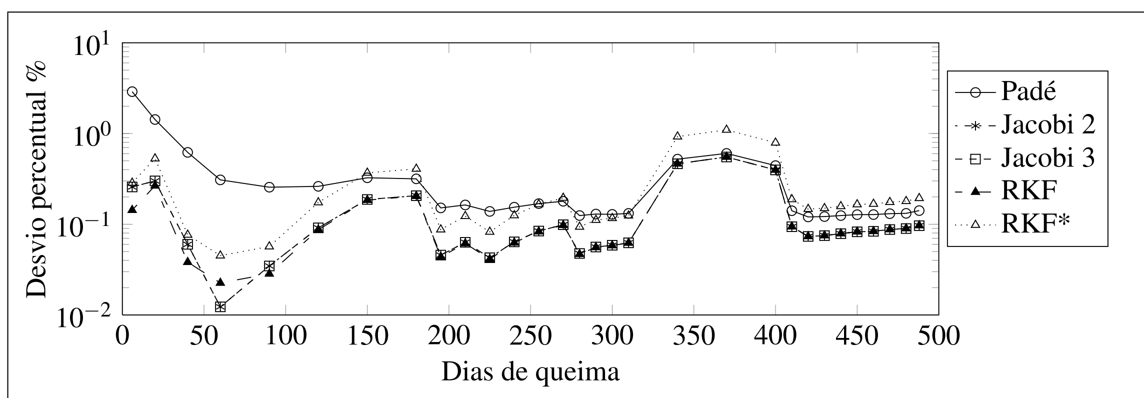

Figura A.49: Desvio na avaliação do Gd^{155}.

Apêndice B

Implementação e códigos

Neste capítulo são abordados alguns conceitos elaborados nesta tese que são fundamentais na construção da interface entre as rotinas executadas em CUDA e OPENMP e o usuário, escrevendo seu fonte em C++, C ou FORTRAN. Um destes conceitos consiste em utilizar ferramentas de código livre (*opensource*) em sua elaboração e que seja compatível com sistemas operacionais Windows do fabricante Microsoft e com os sistemas Linux com o intuito de no futuro portar o sistema para *clusters* de computadores. Assim optou-se por utilizar um ambiente de compilação baseado em CMAKE e construir uma biblioteca, denominada GREPHY (***G****eneral* ***Re****actor* ***Phy****sics Library*) que possa ser compilada no sistema operacional adotado. No anexo B.1 mostra-se um exemplo de configuração do ambiente CMAKE.

B.1 Ambiente de compilação CMake

Como foi mencionado anteriormente, foi utilizado o ambiente compilação denominado CMAKE (MARTIN e HOFFMAN (2010)) para construção de uma biblioteca de *link* dinâmico, denominada GREPHY, que encapsula todo o escopo de programação necessário para o acesso aos objetos e procedimentos implementados na tese. O arquivo de configuração principal é o *CMakeLists.txt* no qual definimos os pacotes que serão ligados a biblioteca, de forma geral o CMAKE procura arquivos *CMakeLists.txt* em cada subdiretório do código fonte escutando as rotinas de compilação encontradas em cada um.

```
cmake_minimum_required(VERSION 2.6)
project(GREPHY)
set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
if (${CMAKE_SOURCE_DIR} MATCHES ${CMAKE_BINARY_DIR})
  message(SEND_ERROR)
  message("-- ERROR: in-tree-build not allowed.")
  message("-- TRY: rm -f CMakeCache.txt; mkdir -p build; cd build; cmake ..; make")
  return()
endif()
set(GREPHY_BUILD_VERSION 1)
set(GREPHY_VERSION "${GREPHY_MAJOR_VERSION}.
    ${GREPHY_MINOR_VERSION}.${GREPHY_BUILD_VERSION}")
set(GREPHY_LIBRARY grephy)
set(BIN_BUILD_DIR ${CMAKE_BINARY_DIR}/bin)
set(INC_BUILD_DIR ${CMAKE_BINARY_DIR}/inc)
set(LIB_BUILD_DIR ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${BIN_BUILD_DIR})
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${LIB_BUILD_DIR})
SET(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${LIB_BUILD_DIR})
set(UTILS_DIR ${PROJECT_SOURCE_DIR}/src/utils)
# Determine if library should be build shared or static
option(BUILD_SHARED "Build the GREPHY library shared." ON)
if(BUILD_SHARED)
  SET(LIBRARY_TYPE SHARED)
else()
  SET(LIBRARY_TYPE STATIC)
endif()
```

Figura B.1: Arquivo CMakeLists.txt - Principal

Neste arquivo devem ser declaradas as bibliotecas que serão ligadas ao código, como as de álgebra linear ou as bibliotecas da NVIDIA. Por exemplo na figura B.2 podemos observar o trecho de configuração do arquivo *CMakeLists.txt* responsável por configurar a biblioteca MKL (INTEL (2007)).

```
 find_package(MKL)
if (NOT MKL_FOUND)
  message("-- MKL not found. Compiling WITHOUT MKL support.")
else()
  if (NOT SUPPORT_OCL)
    # Check for Intel Compiler
    find_program(CMAKE_C_COMPILER NAMES icc)
    find_program(CMAKE_CXX_COMPILER NAMES icpc)
    if (CMAKE_C_COMPILER MATCHES CMAKE_C_COMPILER-NOTFOUND OR
        CMAKE_CXX_COMPILER MATCHES CMAKE_CXX_COMPILER-NOTFOUND)
      message("-- MKL only supported by Intel Compiler. Compiling WITHOUT MKL support.")
    else()
      option(SUPPORT_MKL "Compile WITH MKL support." OFF)
      if (SUPPORT_MKL)
        include(CMakeForceCompiler)
        CMAKE_FORCE_C_COMPILER(icc "Intel C Compiler")
        CMAKE_FORCE_CXX_COMPILER(icpc "Intel C++ Compiler")
        set(MKL_CXX_FLAGS -DSUPPORT_MKL)
      endif()
    endif()
  else()
    message("-- MKL not supported while OpenCL is enabled.")
  endif()
endif()
```

Figura B.2: Arquivo CMakeLists.txt - Configuração do MKL

De maneira análoga, na figura B.3 pode-se observar o trecho de configuração com as atribuições necessária a configuração do CUDA.

```
find_package(CUDA)
if (NOT CUDA_FOUND)
  message("-- CUDA not found. Compiling WITHOUT CUDA support.")
else()
  option(SUPPORT_CUDA "Compile WITH CUDA support." ON)
  if (SUPPORT_CUDA)
    if ("${CMAKE_BUILD_TYPE}" STREQUAL "Debug" OR "${CMAKE_BUILD_TYPE}" STREQUAL "debug")
        set(CUDA_NVCC_FLAGS -g -G -O0 -ftz=false -gencode arch=compute_30,code=sm_30
        -gencode arch=compute_20,code=sm_20 --compiler-options -Wall -D__WARMUP_CALL__)
        set(CUDA_CXX_FLAGS -DSUPPORT_CUDA)
    else()
        set(CUDA_NVCC_FLAGS -O3 -gencode arch=compute_30,code=sm_30 -gencode arch=compute_20,
        code=sm_20 -gencode arch=compute_35,code=sm_35 --compiler-options -Wall -use_fast_math
        -D__WARMUP_CALL__)
        set(CUDA_CXX_FLAGS -DSUPPORT_CUDA)
    endif()
    # Call CUDA hardware parameter script
    execute_process(COMMAND ${CUDA_NVCC_EXECUTABLE} ${UTILS_DIR}/gpu_check_hw.cu
                    -o ${UTILS_DIR}/gpu_check_hw)
    execute_process(COMMAND ${UTILS_DIR}/gpu_check_hw)
  endif()
endif()
```

Figura B.3: Arquivo CMakeLists.txt - Configuração do CUDA

O ambiente CMAKE é muito utilizado tanto em sistemas operacionais do tipo UNIX quanto no Windows e baseado no conceito de *software* livre. Além disso possui um grande número de tutoriais e uma extensa documentação, como no livro (MARTIN e HOFFMAN (2010)), neste trabalho não se propõe escrever uma descrição mais elaborada, mas apenas ilustrar o seu uso e potencial.

B.2 Programa de teste em FORTRAN

Nesta seção apresenta-se o programa de teste implementado para comparar os vetores de concentrações encontrados por cada método e o resultado padrão fornecido pelo código CNFR, além disso este programa em FORTRAN testa a viabilidade da interligação entre o C++ e o CNFR. Na figura B.4 pode-se observar o fluxograma do programa de teste e dada bloco de execução é explicitado na forma de código fonte nas figuras B.5, B.6, B.7 e B.8.

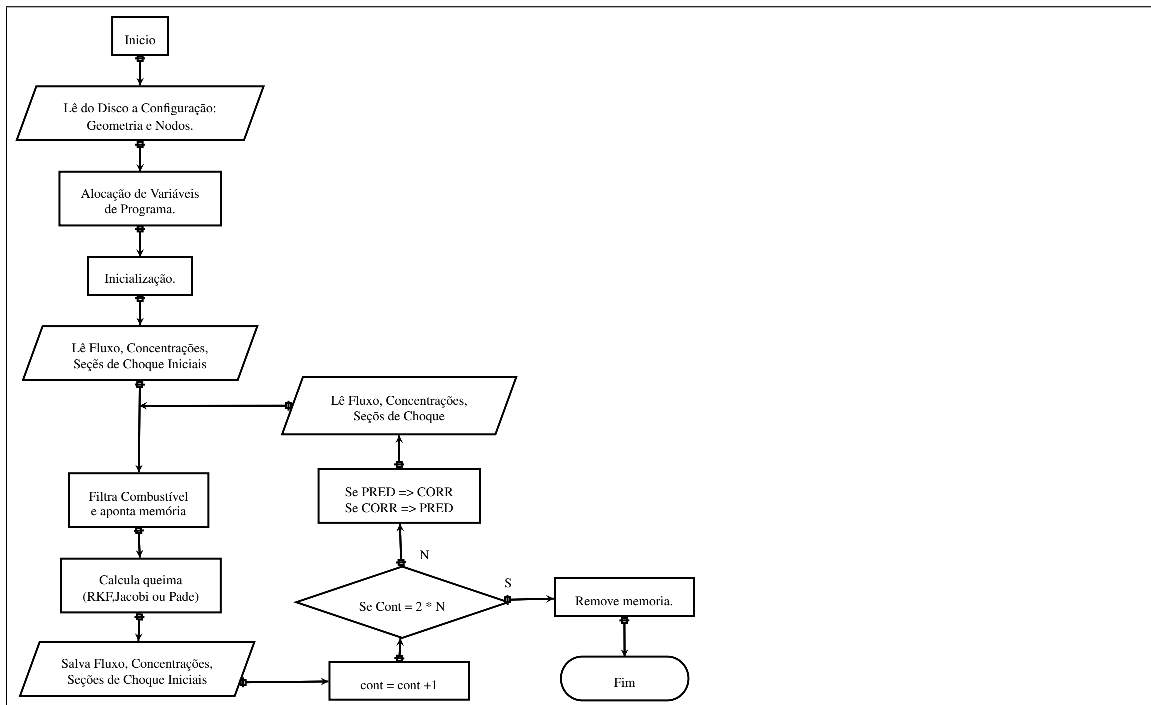

Figura B.4: Fluxograma do programa de avaliação da queima.

```
program deplet
    use Fluxos  use Mapa_Nucleo  use SC_Microscopicas  use Parametros_Fisicos
    use Informacoes_Gerais  use DadosGerais_TipoEC  use Nu_SCMF_Medias_Nodo
    use Energia_SCMF_Media_Nodo  use Intervalos_Tempo_Queima
    use Dados_Simulacao_Usuario  use Constante_Normalizacao
    use Concentracoes_Isotopicas  use Depletion_module  use Vect_SC_Microscopicas
    use, INTRINSIC :: ISO_C_BINDING
    implicit none
!   código ...
end program grephy_deplet
```

Figura B.5: Inicio do programa *deplet*

Na figura B.5 pode-se observar a declaração dos modulos em FORTRAN utilizados no código de teste. Na figura B.6 pode-se observar a declaração dos tipos de dados em FORTRAN utilizados no código de teste, note que a variável **dp** é declarada como um ponteiro para estrutura do tipo **Depletion_type** como visto na seção B.4 e na figura B.10.

```
type(Depletion_type)         :: dp_address
type(c_ptr) ,pointer         :: dp     ! declara ponteiro para dp
real(kind=C_DOUBLE), POINTER :: f1(:), f2(:)
integer                      :: Linha, Coluna, Divisao_Axial, Instante, Grupo, Nuclideo, Nodo,&
                                Opcao, counter, Disco, nact, nfy, cores, err, NNTC
real(kind=C_DOUBLE)          :: Ti, Tf, tolerance
real                         :: dt
call LeDadosEntrada                   call Retorna_Numero_Total_Nodos_Combustivel
call LeConcentracoesIniciais          NTNC = Numero_Total_Nodos_Combustivel
allocate(SCMicro_Captura(2,Numero_Total_Nuclideos, Numero_Total_Nodos),stat=ierr)
allocate(SCMicro_Fissao(2,Numero_Actinideos, Numero_Total_Nodos),stat=ierr)
allocate(vect_Abundância_Tipo(Numero_Actinideos * Numero_Produtos_Fissao),stat=ierr)
allocate(SCMicro_N2n(Numero_Actinideos,Numero_Total_Nodos),stat=ierr)   ! apenas o grupo rápido
                  ! Nestes vetores reservamos somente o que é combustível, de acordo com o mapa
allocate(Concentracao_Media(Numero_Total_Nuclideos, Numero_Total_Nodos),stat=ierr)
allocate(Fluxos_Nodo(2,Numero_Total_Nodos),stat=ierr)
allocate(vect_SCMicro_Fissao(2,Numero_Actinideos * NTNC),stat=ierr)
allocate(vect_SCMicro_Captura(2,Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_SCMicro_N2n(Numero_Actinideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_init(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_aux(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_pred(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_corr(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_media(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_pred_disc(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(vect_Concentracao_Isotopica_corr_disc(Numero_Total_Nuclideos * NTNC),stat=ierr)
allocate(Fluxos_Combustivel(2,NTNC),stat=ierr)
```

Figura B.6: Alocação de Variáveis de programa e memória

Na figura B.7 podem-se observar as rotinas de inicialização.

```
call InitializeGrephy()                                       ! Carrega a biblioteca de programas
call cDepletion_new(dp_address)                               !      Cria o operador de deplecao
    !        Carrega parametros fisicos
call cDepletion_setPhysicsParameters(dp,Percentual_Producao_Pm148_Tipo, &
     Percentual_Producao_Sm149_Tipo, Percentual_Producao_Am242_Tipo, &
     Percentual_Producao_Cm242_Tipo)
call cDepletion_setGeneralParameters(dp,Numero_Total_Nodos_Combustivel, Numero_Actinideos, &
     Numero_Produtos_Fissao, Numero_Total_Nuclideos, Nuclideo_Produzido_N2n_Tipo, &
     Nuclideo_Produzido_Captura_Tipo, Nuclideo_Produzido_decaimento_Tipo, &
     Constante_decaimento_Tipo, Codigo_Nuclideo_Arquivo_TipoEC)
call cDepletion_setFYields(dp, Abundância_Tipo(:,:,1))
```

Figura B.7: Inicialização do programa

Finalmente, na figura B.8 pode-se observar o laço utilizado na execução dos 32 passos de queima utilizando o método de preditor-corretor, ou seja, estima-se a concentração média de cada nuclídeo dentro de cada passo de queima.

```
do Instante=2, Numero_Intervalos_Tempo_Queima
  Ti = Tf                  Tf = Intervalo_Tempo_Queima(Instante)*86400 + Ti
  do Disco= 1, 0, -1                                        ! Começa com o Calculo = 1
    if(Disco==1)then
      do Opcao= 1, 0, -1
        call LeConcentracoes(Instante, Opcao)   call LeDadosNucleares(Instante, Opcao)
        call LeFluxo(Instante, Opcao)           call FiltraCombustivel
        call cDepletion_setCn(dp,Constante_Normalizacao_Fluxo)
        if(Opcao==1) then                                           ! Se for preditor
          vect_Concentracao_Isotopica_pred_disc = vect_Concentracao_Isotopica
          call Escreve_Concentracao_Combustivel_No(Instante,'__pred_dsk',1)
        else
          vect_Concentracao_Isotopica_corr_disc = vect_Concentracao_Isotopica
          call Escreve_Concentracao_Combustivel_No(Instante,'__corr_dsk',1)
          vect_Concentracao_Isotopica = (vect_Concentracao_Isotopica_corr_disc + &
                                         vect_Concentracao_Isotopica_pred_disc)/2.0
          vect_Concentracao_Isotopica_aux = vect_Concentracao_Isotopica
          call Escreve_Concentracao_Combustivel_No(Instante,'__AVG__dsk',1) ! FuelNode = 1
        endif
      end do
    else
      do Opcao= 1, 0, -1! Comeca com o preditor = 1, corretor = 0
call LeDadosNucleares(Instante+1, Opcao)
call LeFluxo(Instante+1, Opcao)         call FiltraCombustivel
if(Opcao==1)then
  vect_Concentracao_Isotopica = vect_Concentracao_Isotopica_aux
else
  vect_Concentracao_Isotopica = vect_Concentracao_Isotopica_pred_disc
endif
call cDepletion_resetFlags(dp)          call CopyPointers(dp)
call cDepletion_setCn(dp,Constante_Normalizacao_Fluxo)
        call cDepletion_calcN(dp, 'JACOBI2'// C_NULL_CHAR, 'BiCGStab'// C_NULL_CHAR, &
                          'None'// C_NULL_CHAR, Ti, Tf, tolerance, 4, 'GPU'// C_NULL_CHAR, err)
        if(Opcao==1) then
          call cDepletion_copyToNf(dp,vect_Concentracao_Isotopica)
          vect_Concentracao_Isotopica_pred = vect_Concentracao_Isotopica
          call Escreve_Concentracao_Combustivel_No(Instante+1,'__pred_cal',1)     ! Salva nodo 1
        else
          call cDepletion_copyToNf(dp,vect_Concentracao_Isotopica)
          vect_Concentracao_Isotopica_corr = vect_Concentracao_Isotopica
          call Escreve_Concentracao_Combustivel_No(Instante+1,'__corr_cal',1)
          vect_Concentracao_Isotopica = (vect_Concentracao_Isotopica_corr + &
          vect_Concentracao_Isotopica_pred)/2.0
          call Escreve_Concentracao_Combustivel_No(Instante+1,'__AVG__cal',1)     ! Salva nodo 1
        endif
      end do endif end do end do
```

Figura B.8: Laço de cálculo dos passos de queima

No primeiro passo de queima o código utiliza como condição inicial o combustível novo. Após efetuar o cálculo da queima, reserva-se a solução (preditora), faz-se a leitura do arquivo de dados no passo corretor e então recalcula-se a queima novamente e obtêm-se a nova solução (corretora). A média entre soluções do corretor e preditor é considerada como a nova solução.

B.3 Programa de interface com o FORTRAN

A opção pela construção da biblioteca em forma de objetos no C++ tornou necessária a inclusão de uma interface entre o código CNFR, escrito em FORTRAN, e a mesma. Isto se deve ao fato de como, de maneira diversa, os compiladores de C++ e FORTRAN organizam as estruturas de dados e as chamadas e retornos de funções. A partir das versões de compiladores compatíveis com a versão FORTRAN 2003 (METCALF *et al.* (2004)) é possível utilizar o mecanismo denominado *C Bindings*. Além disso o compilador do C++ cria símbolos externos que combinam os nomes das classes e os métodos em um processo conhecido como *mangling*. Neste caso deve-se forçar o compilador a gerar os símbolos com nomes que sejam reconhecidos pelo FORTRAN. Na figura B.9 pode-se observar de que forma isto é implementado.

```
#ifdef __cplusplus
  extern"C" {
#endif
  .
  Declaração de código
  .
#ifdef __cplusplus
  }
#endif
```

Figura B.9: Interface entre o FORTRAN e o C++

Aplicou-se esta orientação ao compilador, definindo no programa de interface as funções e apontadores para os objetos relacionados a alocação de memória, criação da matriz de depleção e a execução dos métodos de solução implementados nesta tese.

B.4 FORTRAN e C *Bindings*

Como vimos na seção B.3, as implementações de compiladores FORTRAN após o ano 2003 obedecem ao padrão de troca de informações com bibliotecas em C++ denominado de C *Bindings*. Pode-se observar na figura B.10 um fragmento da listagem do código.

```
! depletion_module.f90
module Depletion_module
  use, intrinsic :: ISO_C_Binding
  implicit none
  private
  type Depletion_type ! Declaração do objeto de acesso as rotinas
    type(C_ptr) :: object = C_NULL_ptr
  end type Depletion_type
  interface                                          !Chamada em C
                                                     !Rotina que insere as secoes de choque
 subroutine cDepMatrix__setSigmas (this, n2n, fission, capture) bind(C,name="DepMatrix_setSigmas")
      import
      type(C_ptr), intent(in) :: this
      real(C_DOUBLE), intent(inout), DIMENSION(*) :: n2n, fission, capture ! double pointer
 end subroutine cDepMatrix__setSigmas
 public :: cDepletion_setSigmas, Depletion_new, Depletion_type ! , & ... e muitas outras
 contains                              ! Chamadas em FORTRAN para a interface em C
 subroutine cDepletion_new(this)
    type(Depletion_type), intent(out)  :: this
    this%object = cDepMatrix__new()
 end subroutine cDepletion_new
 subroutine cDepletion_setSigmas (this, n2n, fis1, fis2, cap1, cap2)
      type(C_ptr), intent(in) :: this
      REAL(C_DOUBLE) , DIMENSION(*) :: n2n, fis1, fis2, cap1, cap2  ! double pointer
      call cDepMatrix__setSigmas(this, n2n, fis1, fis2, cap1, cap2) !
 end subroutine cDepletion_setSigmas
```

Figura B.10: Interface com o FORTRAN via C *Bindings*

O arquivo fonte denominado *Depletion_module* contém as descrições das rotinas em FORTRAN e sua conexão com as chamadas das rotinas em C. As rotinas em C que efetuam o acesso a biblioteca estão descritas em um arquivo fonte em separado e declaradas como tipo *extern*, um trecho de código pode ser vista na figura B.11.

```
#include <iostream>
#include <cassert>
#include <grephy.hpp>
using namespace std;
/* C wrapper interfaces to C++ routines */
extern "C" {
  void grephy_initialize(void){ //   Initialize library grephy
    grephy::init_grephy();
    grephy::info_grephy();
  }
  void DepMatrix_setSigmas(grephy::DepMatrix< grephy::LocalVector<double>, double> *This,
   double *n2n, double *fissao, double *captura, int grupo){
       This->SetSigmasPtr(n2n, fissao, captura, grupo-1);
  }
  ...
}
```

Figura B.11: Interface C de acesso ao C++

Observa-se que todas as rotinas do FORTRAN que acessam a biblioteca onde estão definidos os objetos e programas que são tratados nesta tese devem possuir um equivalente ou *wrapper* neste arquivo fonte.

B.5 Construção da matriz de depleção

É necessária a construção da matriz de depleção a cada passo de queima pois no início do ciclo ocorrem alterações na concentração tanto dos produtos de fissão quanto dos actinídeos menores que alteram o total de elementos não zero da matriz esparsa associada a queima, como pode ser observado na seção 5.3. Desta forma podem ser evitados duas dificuldades no cálculo numérico: o armazenamento e multiplicação de elementos de valor nulo na matriz esparsa e ainda o problema de *fill-in* (GEORGE e LIU (1975)). Este inconveniente está relacionado as operações de multiplicação e soma de matrizes esparsas, nas quais podem ocorrer variações para mais ou para menos do número elementos não zero após tais operações. E pode ocorrer no caso dos algorítimos empregados nesta tese que utilizam potências do operador matricial, ou seja, os métodos de colocação de Gauss-Jacobi e o método diagonal de Padé. Na figura B.12 podemos observar o efeito de efetuar a potência de uma matriz esparsa característica do processo de queima em um nodo.

Inicialmente, a construção da matriz de transição apenas em CPU representou um gargalo no cálculo da queima. A rotina inicial apresentava um tempo de 45 mS por passo, o que tornava o cálculo total demasiadamente lento, pois seriam perdidos praticamente 2.8 segundos apenas neste passo. A rotina de construção em GPU apresenta um resultado muito melhor: 1.2 mS para a construção por passo e cerca de 0.1 mS para a cópia dos vetores de concentrações, fluxo de nêutrons e matriz de rendimento dos produtos de fissão.

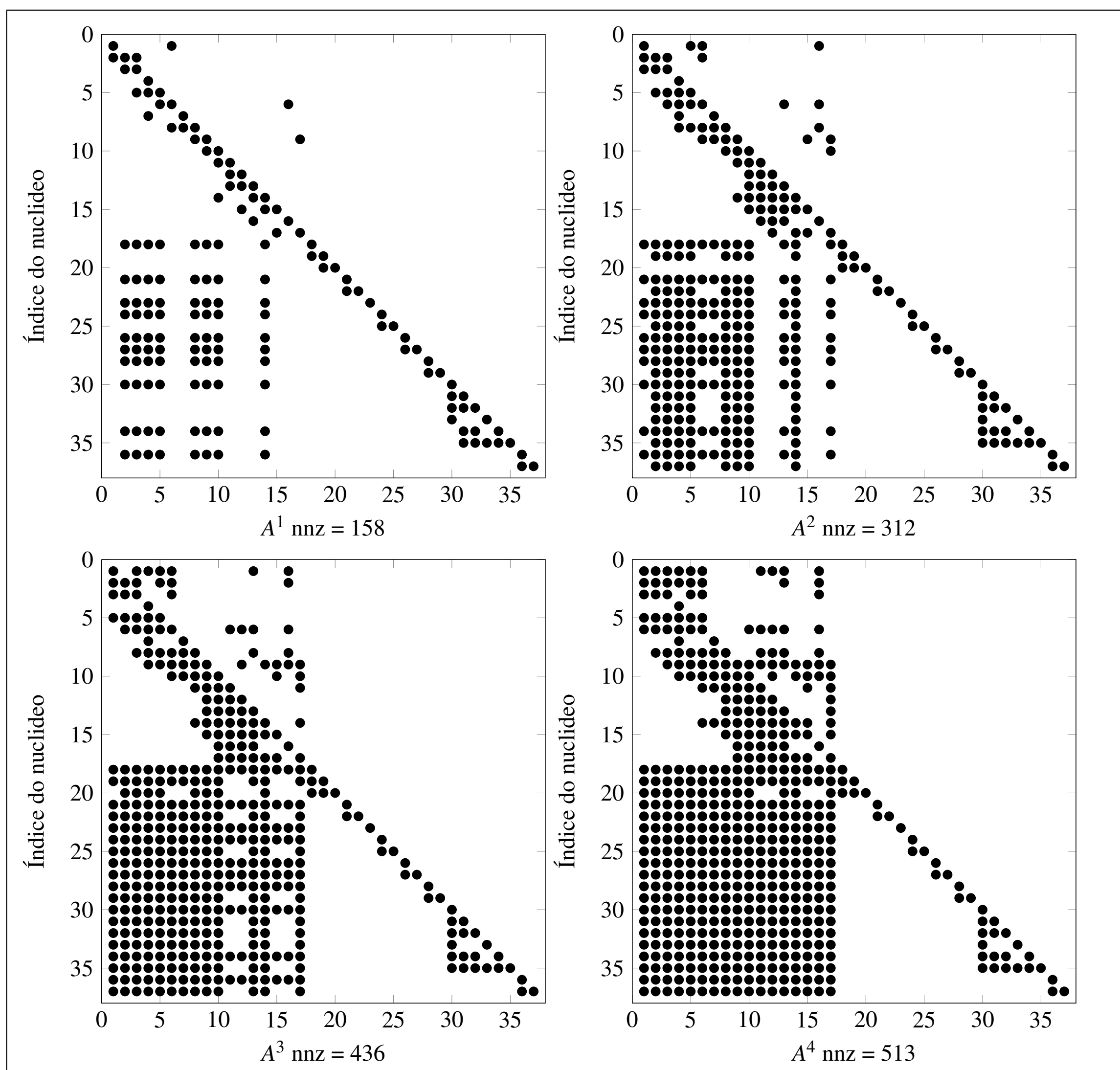

Figura B.12: Potências da matriz de depleção

Apêndice C

Construção dos operadores de queima

C.1 Operadores de queima

Na figura C.1 pode-se observar o código escrito em linguagem C necessário a construção da matriz de depleção.

```
grephy::DepMatrix< grephy::LocalVector<double>, double> *dp;
// dp é o objeto "matriz de deplecao" do tipo "double"
dp->resetFlags();                           // Antes de atualizar deve-se reiniciar
dp->SetNumActFY(Numero_Actinideos,Numero_Produtos_Fissao,Numero_Total_Nuclideos);
dp->Nuclideo_Produzido_N2n_Tipo_.SetDataPtr(&Nuclideo_Produzido_N2n_Tipo,"npN2N",
                                           Numero_Total_Nuclideos);
dp->Nuclideo_Produzido_Captura_Tipo_.SetDataPtr(&Nuclideo_Produzido_Captura_Tipo,"npCap",

                                               Numero_Total_Nuclideos);
dp->Nuclideo_Produzido_decaimento_Tipo_.SetDataPtr(&Nuclideo_Produzido_decaimento_Tipo,
                                                 "npDecay", Numero_Total_Nuclideos);
dp->NumActinides_     = Numero_Actinideos;
dp->NumFissionYields_ = Numero_Produtos_Fissao;
dp->NumElements_      = Numero_Total_Nuclideos;
dp->SetPhi(Phi_1,Phi_2);                    // Fluxo termico e rapido
dp->SetCn(Cn);                              // Constante de Normalizacao
dp->MoveToDevice();
dp->TestBuildGPU();     // Verifica o Tamanho
dp->Allocate(); // Aloca em memoria
dp->BuildDepletionGpu();                    // Constroi a matrix de Deplecao
```

Figura C.1: Construção da matriz de depleção

Os operadores de queima implementados nesta tese, **rkf_**, **jacobi_** e **pade_**, recebem a matriz de queima por atribuição, esta matriz está encapsulada em um objeto tipo *DepMatrix*.

C.2 RKF

A figura C.2 mostra a declaração do objeto **rkf_** no programa de interface entre o FORTRAN e o C++. As variáveis de entrada são: **Ni**, **MaxSteps**, **Tolerance**, **init_t**, **end_t** e o ponteiro para **dp**, observe que o operador de depleção ***dp** é definido na seção B.5.

```
void DepMatrix_calcN(grephy::DepMatrix< grephy::LocalVector<double>, double> *dp,
      double init_t, double end_t, double tol,int cont){
  grephy::RK<grephy::DepMatrix< grephy::LocalVector<double>, double>,
            grephy::LocalVector<double>, double > rkf_;

  rkf_.SetMaxSteps(3600); // Maximo de passos
  rkf_.SetTolerance(tol); // Tolerancia
  rkf_.SetCounter(cont); // Número maximo de passos
  rkf_.SetTime(init_t, end_t);  // tempo inicial e final
  rkf_.SetOperator(*dp);
  rkf_.Build(dp->GetNumNodes(),dp->GetNumElements());
  rkf_.MoveToDevice();          // Move para a GPU
// dp->Ni_ é a concentracao inicial e dp->Nf_ é o resultado
  rkf_.SolveRkFhel_One(&dp->Ni_,&dp->Nf_);
  dp->MoveToHostNf_();
}
```

Figura C.2: Construção do operador RKF

C.3 Método de colocação de Gauss-Jacobi

A figura C.3 mostra a declaração do objeto **jacobi_** no programa de interface entre o FORTRAN e o C++. Observe que o operador de depleção ***dp** é definido na seção B.5. As variáveis de entrada são: α e β, **M** que é a ordem do método, e além disso o número de nodos e elementos do nodo (**nnodos**, **neleme**). É possível escolher o método de solução para o sistema linear, se GMRES (SAAD e SCHULTZ (1986)), BICG (SAAD (2003)) ou GAUSSELI (RICE (1981)) (obs : A eliminação de Gauss foi implementada preliminarmente para teste e tem um limite de 1 nodo).

```
void DepMatrix_calcN(grephy::DepMatrix< grephy::LocalVector<double>, double> *dp,
      double init_t, double end_t, double tol,int solver){
grephy::JACOBI<grephy::DepMatrix< grephy::LocalVector<double>, double>,
               grephy::LocalVector<double>, double > jacobi_;
  jacobi_.Clear(); // jacobi_ eh o operador
  jacobi_.SetOperator(*dp);
  jacobi_.SetAlphaBeta(2.0f,1.0f);
  jacobi_.SetM(2); // Informa a ordem
  jacobi_.SetTime(init_t,end_t);
  int nnodos = dp->GetNumNodes();
  int neleme = dp->GetNumElements()
  jacobi_.SetBlock(neleme,nnodos);
  jacobi_.Build();
  jacobi_.MoveToDevice();              // Move para a GPU
  jacobi_.CalcPowers_();               // Calcula
  switch(psolver){
    case BlockGaussEli:
      if(pprocessor == CPU)
         jacobi_.Solve_GE(dp->Ni_,&dp->Nf_);
      break;
    case BiCGStab:
        jacobi_.Solve_Bic(dp->Ni_,&dp->Nf_);
        break;
      case GMRES:
        jacobi_.Solve_GMRES(dp->Ni_,&dp->Nf_);
      break;
  };
  dp->MoveToHostNf_();
}
```

Figura C.3: Construção do operador de colocação de Gauss-Jacobi

C.4 Método diagonal Padé

A figura C.4 mostra a declaração do objeto **ssdpade_** no programa de interface B.9 entre o FORTRAN e o C++. Observe que o operador de depleção ***dp** é definido na seção B.5.

```
void DepMatrix_calcN(grephy::DepMatrix< grephy::LocalVector<double>, double> *dp,
      double init_t, double end_t, double tol,int cont){
grephy::DiagonalPade<grephy::DepMatrix< grephy::LocalVector<double>, double>,
                     grephy::LocalVector<double>, double > ssdpade_;
  ssdpade_.Clear();
  ssdpade_.SetOperator(*dp);              // Operador de Deplecao
  ssdpade_.SetTime(init_t,end_t);         // tempo inicial e final
  ssdpade_.MoveToDevice();   // Move para a GPU
  ssdpade_.Build();
  ssdpade_.Expm(dp->Ni_,&dp->Nf_);
  dp->MoveToHostNf_();
}
```

Figura C.4: Construção do operador diagonal de Padé