UNIVERSIDADE FEDERAL FLUMINENSE

JOÃO VICTOR DAHER DAIBES

Implementação da Solução do Fluxo de Potência pelo Método de Newton-Raphson em uma Arquitetura Híbrida CPU-GPU

NITERÓI

2021

UNIVERSIDADE FEDERAL FLUMINENSE

JOÃO VICTOR DAHER DAIBES

Implementação da Solução do Fluxo de Potência pelo Método de Newton-Raphson em uma Arquitetura Híbrida CPU-GPU

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Orientadores:

Julio Cesar Stacchini de Souza
Milton Brown Do Coutto Filho

NITERÓI

2021

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

D129i Daher daibes, João Victor
Implementação da Solução do Fluxo de Potência pelo Método de Newton-Raphson em uma Arquitetura Híbrida CPU-GPU / João Victor Daher daibes ; Julio Cesar Stacchini De Souza, orientador ; Milton Brown Do Coutto Filho, coorientador. Niterói, 2021.
104 p. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2021.

DOI: http://dx.doi.org/10.22409/PGC.2021.m.15835254750

1. Unidade de processamento gráfico. 2. Fluxo de potência. 3. Sistemas de potência. 4. Processamento paralelo (Computador). 5. Produção intelectual. I. De Souza, Julio Cesar Stacchini, orientador. II. Do Coutto Filho, Milton Brown, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

João Victor Daher Daibes

Implementação da Solução do Fluxo de Potência pelo Método de Newton-Raphson em uma Arquitetura Híbrida CPU-GPU

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Aprovada em 18 de Março de 2021.

BANCA EXAMINADORA

JULIO CESAR STACCHINI DE SOUZA jstacchini@id.uff.br:94367612791
Assinado de forma digital por JULIO CESAR STACCHINI DE SOUZA jstacchini@id.uff.br:94367612791
Dados: 2021.03.17 11:03:25 -03'00'

Prof. Julio Cesar Stacchini de Souza - Orientador, UFF

Milton Brown Do Coutto Filho
Assinado de forma digital por Milton Brown Do Coutto Filho
Dados: 2021.03.18 15:23:16 -03'00'

Prof. Milton Brown Do Coutto Filho - Orientador, UFF

ESTEBAN WALTER GONZALEZ CLUA estebanclua@id.uff.br:19910039869
Assinado de forma digital por ESTEBAN WALTER GONZALEZ CLUA estebanclua@id.uff.br:19910039869
Dados: 2021.03.19 15:10:16 -03'00'

Prof. Esteban Esteban Walter Gonzalez Clua, UFF

Assinado de forma digital por Carmen Lucia Tancredo Borges
Dados: 2021.03.19 12:58:18 -03'00'

Prof. Carmen Lucia Tancredo Borges, UFRJ

Assinado digitalmente por RAINER ZANGHI rzanghi@id.uff.br:03420509740
Data: 2021.03.19 18:03:44-03'00'

Prof. Rainer Zanghi, UFF

Niterói

2021

Agradecimentos

Agradeço, primeiramente, a Deus, porque é pela Sua graça que o labor frutifica. Ele possibilitou que esse trabalho fosse concluído.

Agradeço aos meus país, Anna e Arthur, pelo apoio, compreensão, orientação e incentivo durante todo o curso.

Agradeço aos professores Milton e Julio pela orientação e cuidado que permitiram que esse trabalho se concretizasse.

Sou grato ao professor Rainer pela atenção e incentivo ao longo do trabalho e ao professor Esteban pelo interesse e pela instrução.

Também agradeço à CAPES e à FAPERJ pelo apoio financeiro prestado durante o curso.

Resumo

Esta Dissertação aborda um dos problemas mais importantes em sistemas elétricos de potência (SEPs), presente desde os primeiros estudos de análise de redes. Estuda-se a solução computacional do *Problema do Fluxo de Potência* (PFP), alcançada aplicando-se o método de Newton-Raphson. O método é algorítmico, portanto independente da plataforma de implementação. Um exame sobre o desenvolvimento alcançado pelos diferentes tipos de processadores aponta para a necessidade de revisão das metodologias de cálculos atuais. Provavelmente, melhorias de *hardware* não serão capazes de oferecer o desempenho exigido pelos cálculos do PFP nas chamadas redes elétricas inteligentes (*smart grids*). Arquiteturas computacionais contendo unidades de processamento gráfico (GPUs) têm sido consideradas promissoras para suplantar aquelas com CPUs. Nesse sentido, uma metodologia que se apoia na utilização de GPUs para a solução do PFP será aqui proposta e confrontada com processos afins que utilizam CPUs. Finalmente, construiu-se também uma abordagem híbrida que utiliza o processador mais adequado para cada etapa do processo de cálculo em questão de modo a produzir uma aceleração relevante frente aos métodos tradicionais. Diversas simulações foram realizadas usando-se o sistema de referência do IEEE 118 barras, que foi multiplicado para a criação de testes de grande porte, visando a avaliação do desempenho do tratamento computacional proposto. Resultados mostram que o uso de GPUs é capaz de promover aceleração quando comparado com implementações que adotam abordagem esparsa e paralelismo na CPU.

Palavras-chave: fluxo de potência, sistemas elétricos de potência, unidade de processamento gráfico, processamento paralelo.

Abstract

This Dissertation addresses one of the most important problems in the electric power systems (EPSs), present since the first network analysis studies. The computational solution of the Power Flow Problem (PFP), obtained by applying the Newton-Raphson method, is studied. The method is algorithmic, therefore independent of the implementation platform. The examination of the development achieved by different processors points to the need to review current calculation methodologies. Hardware improvements are unlikely to deliver the performance required by PFP calculations on so-called smart grids. Computational environments containing graphics processing units (GPUs) have been promising considerations to supplant those of CPUs. In this sense, a methodology that relies on GPUs for the PFP solution will be proposed here and confronted with similar processes that use CPUs. A hybrid approach that utilizes the most appropriate processor for each stage of the calculation process was also built, to produce a relevant acceleration compared to traditional methods. Several simulations were carried out using the IEEE 118 bus reference system, which was multiplied to create large test instances, aiming at evaluating the performance of the proposed computational approach. Results show that GPUs can speedup implementations that adopt a sparse approach and parallelism.

Keywords: power flow, electrical power systems, graphics processing unit, parallel processing.

Lista de Figuras

2.1 Convenções de sinais adotadas [47]. 8

2.2 Linha de transmissão [47]. 10

2.3 Transformador de potência. Fonte: Autor. Adaptado de Monticelli [47]. . . 10

2.4 Cálculo do fluxo de potência pelo método de Newton-Raphson[47]. 19

3.1 Dimensão dos menores componentes [49] de processadores de diversos fabricantes organizados por ano [12]. 25

3.2 Frequência de *clock* de processadores de diversos fabricantes por ano [11]. . 26

4.1 Principais estruturas de um programa CUDA C [53]. 33

4.2 Diagrama esquemático simplificado do chip GP100, de arquitetura Pascal [71]. 37

4.3 Diagrama esquemático simplificado do *Streaming Multiprocessor* do *chip* GP100, de arquitetura Pascal [71]. 38

5.1 Fluxograma simplificado do processo utilizado para solução do *problema do Fluxo de Potência*. 43

5.2 Fluxograma simplificado do laço iterativo executado no bloco "Método de Newton-Raphson" na Figura 5.1. 43

5.3 Fluxograma do processo utilizado para solução do *problema do Fluxo de Potência* na GPU. Fonte: Autor. 44

5.4 Principais variáveis relacionadas aos ramos, à organização do SEP e ao método iterativo. 46

5.5 Principais variáveis relacionadas às barras do SEP. 47

5.6 Percentual cumulativo do número de barras de IEEE118 segundo valor crescente de seus graus. 52

6.1 Numeração das barras de um sistema da forma ieee118xn. 61

6.2 Construção dos sistemas sintéticos ieee118x2 e ieee118x4. 62

6.3 Diagrama esquemático das linhas de transmissão que ligam os 32 sistemas IEEE 118 para formar o sistema IEEE 118x32. 62

6.4 Parcela do tempo total demandado por cada uma das etapas das implementações densas e esparsas paralelas para a CPU na solução do PFP. . . 68

6.5 Parcela do tempo total demandado por cada uma das etapas das implementações densas e esparsas para a GPU na solução do PFP. 69

6.6 Tempo (em milissegundos) demandado para a execução das etapas destacadas para cada um dos sistemas apresentados. 70

Lista de Tabelas

6.1 Instâncias de teste utilizadas. 63

6.2 Fração dos elementos da matriz de admitâncias nodais que é igual a zero. . 63

6.3 Tempo demandado pelos programas executados apenas na CPU. 66

6.4 Aceleração das Rotinas Esparsas Sequenciais frente às rotinas densas sequenciais e paralelas da CPU. 66

6.5 Aceleração das rotinas Esparsas Paralelas executadas na CPU frente às Esparsa Sequenciais executada na CPU. 67

6.6 Tempo demandado pelos programas que usam a GPU. 67

6.7 Tempo demandado pelos programas com abordagem híbrida GPU-CPU e suas acelerações frente aos programas para CPU paralelos e sequenciais. . . 71

6.8 Aceleração promovida em cada uma das principais etapas das implementações executadas na GPU frente ao programa segundo paradigma esparso executado na CPU com paralelismo e mesmo tipo de representação de valor de ponto flutuante. 72

A.1 Tempo demandado para a execução das etapas do programa com abordagem *densa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão dupla. 83

A.2 Tempo demandado para a execução das etapas do programa com abordagem *densa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão simples. 83

A.3 Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão dupla. 83

A.4 Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão simples. 84

A.5 Tempo demandado para a execução das etapas do programa com abordagem *densa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão dupla. 85

A.6 Tempo demandado para a execução das etapas do programa com abordagem *densa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão simples. 85

A.7 Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão dupla. 85

A.8 Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão simples. 86

A.9 Tempo demandado para a execução das etapas do programa com abordagem *densa* na GPU e que faz uso de valores de ponto flutuante com precisão dupla. 87

A.10 Tempo demandado para a execução das etapas do programa com abordagem *densa* na GPU e que faz uso de valores de ponto flutuante com precisão simples. 87

A.11 Tempo demandado para a execução das etapas do programa com abordagem *esparsa* na GPU e que faz uso de valores de ponto flutuante com precisão dupla. 87

A.12 Tempo demandado para a execução das etapas do programa com abordagem *esparsa* na GPU e que faz uso de valores de ponto flutuante com precisão simples. 87

A.13 Tempo demandado para a execução das etapas do programa com abordagem *híbrida* e que faz uso de valores de ponto flutuante com precisão dupla. 88

A.14 Tempo demandado para a execução das etapas do programa com abordagem *híbrida* e que faz uso de valores de ponto flutuante com precisão simples. 88

Lista de Abreviaturas e Siglas

API	:	Interface de Programação de Aplicativos;
CI	:	Circuito Integrado;
CMOS	:	Complementary Metal Oxide Semiconductor;
CPU	:	Unidade Central de Processamento;
CUDA	:	Compute Unified Device Architecture;
DRAM	:	Memória de Acesso Randômico Dinâmica;
GPGPU	:	Programação de Propósito Geral em GPUs;
GPU	:	Unidade de Processamento Gráfico;
IEEE	:	Instituto de Engenheiros Eletricistas e Eletrônicos;
PFP	:	Problema do Fluxo de Potência;
PFPS	:	Ponto Flutuante de Precisão Simples;
PFPD	:	Ponto Flutuante de Precisão Dupla;
SEP	:	Sistema Elétrico de Potência;
SIN	:	Sistema Interligado Nacional;
SM	:	Streaming Multiprocessor;

Sumário

1 Introdução **1**

1.1 Revisão Bibliográfica 2

1.2 Objetivos 4

1.3 Estrutura 5

2 Fluxo de Potência em Redes Elétricas **6**

2.1 Introdução 6

2.1.1 Convenções 7

2.1.2 Definição do Problema 8

2.2 Modelagem do Sistema Elétrico de Potência 9

2.2.1 Matriz de Admitâncias Nodais 9

2.2.2 Injeções de Potência 12

2.2.3 Fluxos de Potência em Ramos 14

2.3 Formulação do Problema Básico 15

2.3.1 Método Numérico de Newton-Raphson 16

2.4 Variações do Método de Newton Raphson 20

2.5 Explorando a Esparsidade 21

3 Evolução do Desempenho de Microprocessadores **23**

3.1 Introdução 23

3.2 Processadores Modernos e a Barreira da Potência 24

3.2.1 Potência Dissipada por um Processador 25

3.3 Barreira de Utilização 28

4 Unidade de Processamento Gráfico 29

4.1 Introdução 29

4.2 Programação em GPUs 30

4.2.1 Abordagem Histórica 30

4.2.2 APIs de Programação de Propósito Geral em GPUs 31

4.2.3 Modelo de Programação CUDA C/C++ 32

4.3 A Arquitetura Pascal 37

4.3.1 Elementos dos Streaming Multiprocessors 37

5 Metodologia 42

5.1 Introdução 42

5.2 Solução do PFP pelo Método de Newton-Raphson 42

5.2.1 Arquivo de Entrada e Estruturas de Dados 43

5.2.2 Construção da Matriz de Admitâncias Nodais 48

5.2.3 Cálculo da Injeção de Potência Nodal 49

5.2.4 Cálculo do Vetor de Resíduos 51

5.2.5 Cálculo da Matriz Jacobiano 52

5.2.6 Solução do Sistema Linear 57

5.2.7 Atualização das Tensões Nodais Complexas 57

5.2.8 Cálculo dos Fluxos de Potência Ativos e Reativos nos Ramos 57

5.2.9 Impressão dos resultados e conclusão da execução 58

6 Testes Numéricos 59

6.1 Introdução 59

6.2 Sistemas Teste 60

6.2.1 Construção de Sistemas 60

6.3 Resultados Numéricos 63

7 Conclusão **73**

7.1 Trabalhos Futuros 74

Referências **76**

Apêndice A – Tempos de Execução **82**

A.1 Programas que utilizam unicamente a CPU 83

A.1.1 Programas Sequenciais 83

A.1.2 Programas Paralelos 85

A.2 Programas que utilizam a GPU 87

Capítulo 1

Introdução

Sistemas elétricos de potência (SEPs) compreendem uma série de equipamentos e estruturas encarregadas de gerar, transportar e distribuir energia aos consumidores. Na análise de redes elétricas, foram primeiramente utilizados simuladores físicos – chamados analisadores de rede [63] – que consistiam em conjuntos de dispositivos analógicos que possibilitavam a modelagem de SEPs. Esses simuladores ocupavam grandes espaços e permitiam apenas simulações de sistemas de escala reduzida, requerendo diversas simplificações. Um único estudo do PFP em uma pequena rede podia levar horas.

O advento da computação digital ampliou a aplicação de vários métodos numéricos para a solução do PFP, estando sempre presente a meta de aumento de desempenho. Nesse contexto, inserem-se os intervalos de tempo demandados para sua execução tanto nas tarefas da operação (limites mais restritivos) como no âmbito do planejamento (limites considerados razoáveis para a análise de diversos cenários futuros).

Nos últimos anos, estudos com as novas *smart grids* [29] revelam uma tendência de ampliação do uso de fontes de geração de energia elétrica renovável, assim como distribuída, tendo ambas natureza estocástica. Diante deste cenário [77], passa a ser necessária consideração de soluções em um maior número de instâncias do problema computacional do fluxo de potência (PCFP), de modelagem mais complexa.

Os principais elementos utilizados para que sejam efetuados os cálculos do algoritmo que resolve o PCFP são as unidades centrais de processamento (CPUs). Nas últimas décadas, CPUs com núcleo único tiveram o desempenho continuamente aprimorado como fruto dos benefícios descritos pela *Lei de Moore* [48] e *dimensionamento de Dennard* [14]. Isso possibilitava que, sem que fossem feitas alterações nos programas, pudesse ser obtida uma melhoria no desempenho de aplicações quando executadas nas novas gerações de

processadores. Contudo, o dimensionamento de Dennard deixou de ser válido e levou a indústria à adoção de múltiplos núcleos homogêneos [69]. Além da exigência de alterações para que um programa utilize maior número de núcleos, restrições de dissipação de energia também impedem que a totalidade dos operadores disponíveis no circuito integrado (CI) das CPUs sejam, de fato, utilizados a todo tempo. A fração do processador que não pode ser utilizada por esses motivos é denominada *Silício Negro* (tradução livre da expressão em lingua inglesa *Dark Silicon*). Assim sendo, torna-se improvável que unicamente a evolução do *hardware* traga as melhorias exigidas para o processo de cálculo em questão. Uma das formas de evitar problemas de consumo e dissipação de energia é a adoção de processadores com arquitetura mais simples, que fazem melhor aproveitamento dos transistores disponíveis na área de CI utilizada para as funções que executam e são energeticamente mais eficientes. Atualmente, unidades gráficas de processamento (GPUs) são processadores programáveis, capazes de execução da rotinas de propósito geral, que se encaixam nesses requisitos.

Tendo em vista o acima exposto, mostra-se relevante o estudo de técnicas que permitam que um de programa destinado a solucionar o PCFP se valer do estado da arte em processamento de dados, com o objetivo de atender às demandas atuais dos SEPs.

1.1 Revisão Bibliográfica

Estudos mais recentes sobre SEPs têm apontado para o aperfeiçoamento das tarefas de *análise de redes, otimização e controle* [18]. Tendência de maior interligação de redes [3, 75], acomodação de fontes geradoras com características estocásticas e distribuídas são fatores importantes mencionados em relatórios dos operadores ligados ao sistema da América do Norte [5] e europeu [57]. Além disso, efeitos dessas mudanças nos processos de análise são frisados em relatório [75] de grupo de pesquisa conjunto de operadores de sistemas elétricos europeus. Também é ressaltada a necessidade de estudos de técnicas que promovam maior velocidade de cálculo do PFP em sistemas com elevado número de barras.

A aplicação de métodos de paralelização nos algoritmos do PFP vem sendo tratada há algum tempo, como demonstra o relatório [2] do Subcomitê de Métodos Analíticos e Computacionais do IEEE publicado em 1992 que abordou algumas das principais dificuldades desse processo de cálculo:

"Exceto pelos procedimentos analíticos que exigem soluções repetidas, como a aná-

lise de contingências, não há paralelismos óbvios inerentes à estrutura matemática dos problemas do sistema de potência. [...] O algoritmo [de solução do problema do fluxo de potência] usual, que faz uso de soluções matriciais iterativas, explora a extrema esparsidade advinda das conexões reais de rede para ganhar velocidade e poupar espaço de armazenamento. Algoritmos paralelos de processamento de matrizes densas não são competitivos com os métodos matriciais esparsos sequenciais. Além disso, como o padrão de acesso a elementos é irregular nos métodos matriciais esparsos, tem sido difícil de encontrar algoritmos paralelos análogos."

Contudo, tal concepção tem sido modificada com o surgimento de novas plataformas de cálculo como aquelas que usam a *programação de propósito geral em unidades de processamento gráfico* (GPGPU). Trabalhos como a simulação de estabilidade transitória em SEPs de grande porte feita por Jalili [36], o estudo do processo de *estimação de estado* pelo método dos mínimos quadrados ponderados de Karimipour [37] (que produziu aceleração de 38 vezes) e outros citados na revisão bibliográfica de Green [25] são exemplos do potencial desse campo de estudo.

A aplicação de técnicas da GPGPU ao PFP é um campo de pesquisa em plena atividade, como revelam os trabalhos que empregam algoritmos sofisticados para a solução do problema, entre estes os que buscam paralelismo em uma ou entre diversas instâncias do PFP, como mencionado por Yoon [80].

Os trabalhos de Liu [44], Robergue [65] e Zhou [81] são exemplos importantes de estudos que empregam paralelismo entre diferentes instâncias do PFP. Enquanto o primeiro levou a aceleração de até 24 vezes para a execução de várias instâncias do problema de Newton-Raphson Desacoplado Rápido, o segundo obteve aceleração de 45 vezes para o método de Gauss-Seidel e pouco menos que 18 vezes para o método de Newton-Raphson. Já o último conseguiu atingir aceleração de 57 vezes para processo de Análise de Segurança Estática, ao paralelizar a etapa de solução que envolve várias instâncias do problema do fluxo de potência. Todas essas comparações consideraram a característica esparsa do problema e foram feitas frente a programas sequenciais executados na CPU.

Também existem exemplos de estudos envolvendo a solução de apenas uma instância do problema. Um deles é o trabalho de Li [41] que faz uso de método de gradiente conjugado para resolver o sistema linear do fluxo de potência CC [47]. A maior aceleração obtida foi de 10,8 vezes. Alguns trabalhos abordam o *método de Newton-Raphson*, sem aplicação de técnicas de desacoplamento ou outras simplificações no cálculo da matriz jacobiano (apresentadas na Seção 2.4). Dentre eles, está o trabalho de Guo [28]. Nele, implementa-

ções densas que fazem uso da GPU, via API CUDA, foram construídas para solucionar o problema pelos métodos numéricos de Gauss-Seidel, Newton-Raphson e Newton-Raphson desacoplado. Enquanto a implementação do primeiro que faz uso de GPU passou a demandar tempo de execução maior frente à implementação que faz uso apenas da CPU, o uso da GPU levou a aceleração de 53,6 vezes para o método de Newton-Raphson e de 27,0 vezes para o método de Newton-Raphson desacoplado. Sistemas com até 974 barras foram considerados. Já no trabalho de Garcia [21], foi criado *kernel* CUDA capaz de solucionar os sistemas lineares das iterações do método de Newton-Raphson fazendo uso do método iterativo do gradiente biconjugado. O uso do *kernel* em implementação que explora a característica esparsa do problema levou a aceleração de 1,9 vezes, quando as opções de otimização dos compiladores não foram ativadas para sistema padrão do IEEE com 118 barras. O trabalho de Chen [8] trata do processo de Análise de Segurança Estática de SEPs. Esse tipo de estudo envolve solução de instâncias do PFP pelo método de Newton-Raphson, as quais tiveram aplicação de técnicas de GPGPU na solução do sistema linear. Isso produziu redução de até 40% no tempo de computação total do problema.

O único trabalho dentre os citados por Yoon que aborda uso de GPGPU a fim de acelerar a solução de uma instância do problema do fluxo de potência pelo método de Newton-Raphson e que não se restringe a implementações densas ou se concentra na etapa de solução do sistema linear foi o recente trabalho de Su [76], publicado em 2020. Nele, foram criadas rotinas densas e esparsas que envolvem manipulação de dados de forma vetorial e, para tal, é utilizada a biblioteca ArrayFire [45]. Apenas é apresentada aceleração frente à versão sequencial densa, sendo que a de maior valor obtida foi de 516 vezes.

Nesse contexto, devido ao elevado potencial de ganho computacional que as técnicas de paralelização, em especial as que fazem uso de GPUs apresentam, se faz relevante a avaliação de possíveis ganhos de desempenho fazendo uso de tecnologias que refletem o atual estado da arte do setor.

1.2 Objetivos

Esse trabalho faz parte de um esforço de pesquisa para que sejam encontradas técnicas que melhorem a eficiência computacional dos métodos de solução do PCFP em redes de grande porte. Para esse fim, será analisada a adequação desse problema ao modelo de programação de GPUs. Técnicas da computação paralela foram utilizadas para a criação

de algoritmos executáveis em CPUs e GPUs, usando os paradigmas de desenvolvimento conhecidos como denso e esparso (assim denominados em razão das características das matrizes encontradas no processo de cálculo do PCFP). Foram construídas implementações em linguagem C++ e CUDA C++, que utilizam representações de valores de ponto-flutuante de precisão simples e dupla, para a execução de testes, análise e avaliação do desempenho.

1.3 Estrutura

A seguir, apresenta-se a forma como o presente manuscrito está estruturado.

No Capítulo 2, define-se o PFP e alguns tópicos para facilitar a sua compreensão são apresentados. O principal método numérico usado para solucionar o PFP é descrito, assim como as características de esparsidade das matrizes envolvidas.

O Capítulo 3 trata da evolução dos microprocessadores ao longo do tempo e caracteriza alguns limitantes referentes ao desempenho de CPUs.

O Capítulo 4 apresenta a GPU como uma das ferramentas disponíveis a ser explorada no problema aqui tratado e aborda aspectos de sua arquitetura voltados programação de propósito geral.

O Capítulo 5 propõe abordagens para solução do PCFP, acompanhadas de breve análise. Já o Capítulo 6 reúne e compara os resultados das implementações construídas para os métodos de cálculo do PFP. Também é proposta uma abordagem híbrida que utiliza as melhores rotinas de cada uma das etapas necessárias para a solução do problema.

Por fim, o Capítulo 7 apresenta as principais conclusões alcançadas com a presente pesquisa e propõe sua continuidade.

Capítulo 2

Fluxo de Potência em Redes Elétricas

Embora o PFP seja amplamente conhecido, neste capítulo apresentam-se os principais elementos necessários à construção de algoritmos para a solução do referido problema. Serve também ao propósito de estabelecer a nomenclatura aqui adotada.

2.1 Introdução

O amplo uso de eletricidade de certa maneira pontua o estágio de evolução em que se encontra a sociedade moderna. Eletrodomésticos, veículos, iluminação, computadores estão entre os elementos indispensáveis às tarefas do dia-a-dia de todos.

A demanda pelo suprimento de energia elétrica é atendida por meio de SEPs, usualmente subdividos em sistemas de geração, transmissão e distribuição de energia. Inicialmente, existiam vários sistemas que operavam de forma isolada para atender a demandas regionais. Com o passar do tempo, tornaram-se claros os benefícios da interligação de sistemas, notadamente quanto à confiabilidade.

Atualmente, há SEPs que expandiram para além de fronteiras nacionais, adquirindo dimensões continentais. Um deles é a *Rede Elétrica da América do Norte* [5], que abrange grande parte dos Estados Unidos da América, Canada e do Estado mexicano da Baixa Califórnia. Segundo Overbye [60], "a rede elétrica da América do Norte realmente constitui um grande circuito". Outros exemplos são a *Área Síncrona da Europa Continental* [57] – que inclui diversos países europeus e do norte da África – e o *Sistema Elétrico Integrado - Sistema Elétrico Unido* (IPS/UPS) [3] que abrange países que constituíam a antiga União das Republicas Socialistas Soviéticas (URSS). A maior parte do território brasileiro é

atendida pelo *Sistema Interligado Nacional* (SIN) [59], que também abrange distâncias físicas continentais, comparáveis àquelas sistema europeu.

2.1.1 Convenções

Aqui, faz-se necessária a apresentação de alguns conceitos, como também as convenções adotadas na Dissertação.

Caso o leitor deseje, tratamento mais aprofundado dos conceitos de circuitos elétricos expostos nessa subseção pode ser encontrado no livro de Nilsson e Riedel [52] ou de Alexander e Sadiku [1].

Sistema de Transmissão

Esse trabalho se concentrará na análise de sistemas de transmissão [24] de energia de alta tensão e corrente alternada, operando de forma balanceada (i.e., correntes e tensões nas três fases de igual magnitude e defasadas de 120°). Assim, torna-se possível a utilização de circuito equivalente monofásico [24] do sistema elétrico de potência sob análise.

A ***Tensão Elétrica Complexa*** (ou *Diferença de Potencial Elétrico Complexa*) é referida pelo símbolo E e definida da seguinte forma:

$$E = V\underline{/\theta} = V \cdot e^{j \cdot \theta} \tag{2.1}$$

onde V é sua magnitude da tensão complexa, θ é o seu argumento ou ângulo de fase e $j = \sqrt{-1}$ é a unidade imaginária.

A ***Potência Elétrica Complexa*** é referida pelo símbolo S e definida conforme a seguinte expressão:

$$S = E \cdot I^* = P + j \cdot Q \tag{2.2}$$

onde E é a tensão complexa entre os terminais da carga elétrica, I é a corrente elétrica que a atravessa, j é a unidade imaginária e a operação $\bullet^*$ representa o conjugado [79] de um número complexo.

A impedância está relacionada com a generalização da ***lei de ohm*** [1] para circuitos que, além consumir *potência ativa*, também envolvem a circulação de *potência reativa*, sendo válido escrever:

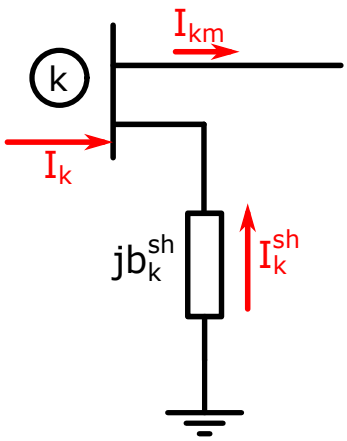

Figura 2.1: Convenções de sinais adotadas [47].

$$E = z \cdot I \tag{2.3}$$

onde E é a *tensão elétrica complexa* sob a qual os terminais elétricos de um elemento estão submetidos, z é a impedância desse elemento e I é a *corrente elétrica* complexa que o atravessa.

Convenções de Sinais

Na Dissertação, serão adotadas certas convenções como em Monticelli [47], representadas na Figura 2.1. Enquanto as injeções liquidas de potência são consideradas positivas quando entram na barra (i.e., geração) e negativas quando saem da barra (i.e., carga), os fluxos de potência são positivos quando saem da barra e negativos quando nela entram. Para os elementos *shunt* das barras é adotada a mesma convenção que para as injeções. As convenções de sentidos para o fluxo de potência ativa e reativa são as mesmas adotadas para correntes.

2.1.2 Definição do Problema

O funcionamento adequado de um SEP está relacionado ao atendimento de vários requisitos: não violação dos limites operacionais de seus variados elementos; satisfação da demanda de energia; garantia da integridade da rede quando submetida a condições adversas. Para que esse objetivo seja alcançado, são necessárias constantes ações de controle sobre o sistema, advindas de estudos de análise de redes [24], entre estes os que tratam do PFP [31], e.g., análise de contingências e transferência de potência entre áreas.

O *problema computacional* [10] *do PFP* pode ser definido como, dado modelo do SEP

em questão, despacho de potência ativa para as unidades geradoras do sistema e perfil de demanda das unidades consumidoras (i.e., quanta potência é consumida por elas), deve ser obtido o estado estático do sistema (suas tensões nodais complexas).

2.2 Modelagem do Sistema Elétrico de Potência

Utiliza-se no PFP o modelo conhecido por *barra-ramo*. Tal modelo focaliza as *barras* da rede (nós de um circuito elétrico), definidas como pontos de convergência de dois ou mais ramos da rede. Os *ramos* representam os elementos da rede que conectam duas barras (e.g., linhas de transmissão ou transformadores de potência).

2.2.1 Matriz de Admitâncias Nodais

Para o tratamento computacional, elementos da rede são representados matricialmente. No PFP, representa-se a rede pela denominada *matriz de admitâncias nodais*, cuja montagem será descrita a seguir.

Linhas de transmissão

As linhas de transmissão são representadas pelo denominado modelo π, como ilustra a Figura 2.2. Considere a linha que liga a barra k à barra m com susceptância shunt b_{km}^{sh} e admitância série:

$$y_{km} = g_{km} + jb_{km} = z_{km}^{-1} = \frac{r_{km}}{r_{km}^2 + x_{km}^2} - j\frac{x_{km}}{r_{km}^2 + x_{km}^2} \tag{2.4}$$

onde g_{km} é a condutância, b_{km} é a susceptância, r_{km} é a resistência e x_{km} é a reatância série da linha que liga a barra k à barra m.

A corrente elétrica que flui da barra k para a barra m pode ser obtida pela *lei de Ohm*:

$$I_{km} = y_{km}(E_k - E_m) + jb_{km}^{sh}E_k \tag{2.5}$$

onde y_{km} é a admitância série da linha de transmissão, E_k e E_m são, respectivamente, as tensões nodais complexas nas barras k e m e b_{km} é a susceptância *shunt* da linha.

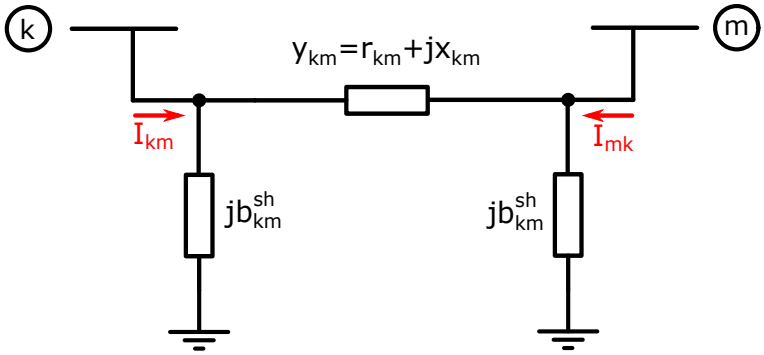

Figura 2.2: Linha de transmissão [47].

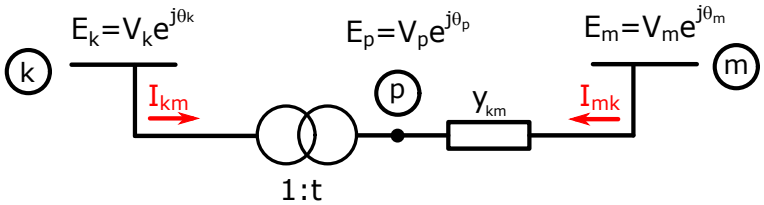

Figura 2.3: Transformador de potência. Fonte: Autor. Adaptado de Monticelli [47].

Transformadores

Transformadores são equipamentos que realizam a mudança do nível de tensão e corrente e/ou defasagem entre tais grandezas, nos seus terminais de saída com relação aos de entrada. A Figura 2.3 representa um transformador instalado entre as barras k e m como um transformador ideal (i.e., que não dissipa potência) em *série* com uma *admitância*. Esse transformador ideal possui relação de transformação complexa t_{km}, definida em função dos valores de tensão em seus terminais, descrita conforme a Expressão (2.6).

$$t_{km} = \frac{E_p}{E_k} \tag{2.6}$$

Onde E_p e E_k são, respectivamente, as tensões nodais complexas na barra k e no ponto p.

Como não há perdas em um transformador ideal e tomando como base a primeira igualdade de (2.2), vem:

$$E_k I_{km}^* + E_p I_{mk}^* = 0 \therefore \frac{E_p}{E_k} = -\frac{I_{km}^*}{I_{mk}^*} = -\left(\frac{I_{km}}{I_{mk}}\right)^* \tag{2.7}$$

Além disso, aplicando a definição (2.6) à última equação da relação (2.7), obtém-se:

$$\frac{I_{km}}{I_{mk}} = -t_{km}^{*} \tag{2.8}$$

A partir da Figura 2.3, pode-se formular a corrente I_{mk}:

$$I_{mk} = y_{km}(E_m - E_p) \tag{2.9}$$

Substituindo I_{mk} da relação (2.9) em (2.8) obtém-se:

$$I_{km} = -t_{km}^{*} y_{km}(E_m - E_p) \tag{2.10}$$

Substituindo E_p da equação (2.6) no segundo membro da equação (2.10):

$$I_{km} = |t_{km}|^2 y_{km} E_k + (-t_{km}^{*} y_{km}) E_m \tag{2.11}$$

onde t_{km} é a relação de transformação complexa de tensão do transformador entre as barras k e m.

Analogamente, pode-se escrever que:

$$I_{mk} = y_{km}(E_m - E_p) = (-t_{km} y_{km}) E_k + (y_{km}) E_m \tag{2.12}$$

onde t_{km} é a relação de transformação complexa de tensão do transformador entre as barras k e m.

Generalização e formulação matricial

Combinando as expressões (2.5) e (2.11), é possível generalizar a expressão que define a corrente que flui entre duas barras:

$$I_{km} = (|t_{km}|^2 y_{km} + j b_{km}^{sh}) E_k + (-t_{km}^{*} y_{km}) E_m \tag{2.13}$$

Segundo a primeira lei de Kirchhoff [52], a injeção líquida de corrente na barra k, conforme se vê na Figura 2.3, pode ser escrita como:

$$I_k + I_k^{sh} = \sum_{m \in \Omega_k} I_{km} \qquad , k \in [1, nb] \tag{2.14}$$

onde nb é o número de barras do sistema, Ω_k é o conjunto das barras ligadas (vizinhas) à barra k por meio de linha de transmissão e/ou transformador, I_k é a injeção líquida de corrente na barra k e I_k^{sh} é a corrente que flui pelo elemento *shunt* ligado à barra k.

Considerando a equação (2.14), é possível reescrever a equação (2.13) da seguinte forma:

$$I_k = \left[jb_k^{sh} + \sum_{m \in \Omega_k} (|t_{km}|^2 y_{km} + jb_{km}^{sh}) \right] \cdot E_k + \sum_{m \in \Omega_k} [(-t_{km}^* y_{km}) E_m] \tag{2.15}$$

Organizando os somatórios descritos conforme (2.15) na forma matricial para as nb barras, pode-se escrever:

$$\boldsymbol{I} = Y \boldsymbol{E} \tag{2.16}$$

onde $\boldsymbol{I} = \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_{nb} \end{bmatrix}$ é o vetor das injeções de corrente, $\boldsymbol{E} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_{nb} \end{bmatrix}$ é o vetor das tensões nodais e Y é a *matriz de admitâncias nodais*.

Com base nas relações (2.15) e (2.16), pode-se escrever as leis de formação dos elementos da matriz de admitâncias nodais da seguinte forma:

$$Y_{kk} = [jb_k^{sh} + \sum_{m \in \Omega_k} (|t_{km}|^2 y_{km} + jb_{km}^{sh})] \tag{2.17}$$

e

$$Y_{km} = -t_{km}^* y_{km} \tag{2.18}$$

2.2.2 Injeções de Potência

A potência gerada e a consumida (injeções líquidas) nas barras são os dados de entrada do PFP.

É possível reescrever a expressão (2.16) da seguinte forma:

$$I_k = \sum_{m \in K} Y_{km} E_m = \sum_{m \in K} (G_{km} + jB_{km}) V_m e^{j\theta_m} \tag{2.19}$$

onde K é o conjunto de todas as barras m adjacentes à barra k — i.e., todas as barras ligadas à barra k por meio de um ou mais ramos —, incluindo a própria barra k; G_{km} é o elemento de índices (k, m) de uma matriz que compreende a parte real dos elementos da matriz admitâncias nodais; B_{km} é o elemento de índices (k, m) de uma matriz que compreende a parte imaginária dos elementos da matriz admitâncias nodais; V_m é a magnitude do m-ésimo elemento do vetor das tensões nodais enquanto θ_m é o argumento desse elemento.

O conjugado da injeção de potência complexa na barra k, descrita conforme (2.2), é:

$$S_k^* = P_k - jQ_k = E_k^* I_k \tag{2.20}$$

Substituindo o valor de I_k definido no terceiro membro de (2.19) na expressão do terceiro membro de (2.20) e reescrevendo E_k como $e^{j\theta_k}$, tem-se:

$$S_k^* = V_k e^{-j\theta_k} \sum_{m \in K} (G_{km} + jB_{km}) V_m e^{j\theta_m} \tag{2.21}$$

Dos dois primeiros termos de (2.20) e de (2.21), obtém-se expressões para as injeções de potência ativa e reativa na barra k. Aplicando-se a *fórmula de Euler* [79] a elas, obtém-se as expressões abaixo:

$$P_k = V_k \sum_{m \in K} [V_m (G_{km} \cdot cos\theta_{km} + B_{km} \cdot sen\theta_{km})] \tag{2.22}$$

$$Q_k = V_k \sum_{m \in K} [V_m (G_{km} \cdot sen\theta_{km} - B_{km} \cdot cos\theta_{km})] \tag{2.23}$$

onde K é o conjunto de todas as barras m adjacentes à barra k — i.e., todas as barras ligadas à barra k por meio de uma ou mais linhas de transmissão —, incluindo a própria barra k; G_{km} é o elemento de índices (k, m) de uma matriz que compreende a parte real dos elementos da matriz admitância nodal; B_{km} é o elemento de índices (k, m) de uma matriz que compreende a parte imaginária dos elementos da matriz admitância nodal; V_m é a magnitude do m-ésimo elemento do vetor das tensões nodais enquanto θ_{km} é a diferença entre os argumentos das tensões nas barras k e m.

2.2.3 Fluxos de Potência em Ramos

Expressões para os fluxos de potência ativa P_{km} e reativa Q_{km} que fluem por ramo que liga a barra arbitrária k à barra arbitrária m, partindo da primeira para a segunda, podem ser deduzidas de forma condizente com o modelo adotado, a partir das equações para as correntes elétricas nos diversos elementos que compõem o sistema elétrico.

Linhas de Transmissão

Pode-se descrever o conjugado da potência complexa como:

$$S_{km}^{*} = P_{km} - j \cdot Q_{km} = E_k^{*} \cdot I_{km} \tag{2.24}$$

Substituindo a equação (2.5) no terceiro membro de (2.24) e expandindo-se E_k^{*} como $e^{-j\theta_k}$, obtém-se que:

$$S_{km} = y_{km} V_k e^{-j\theta_k} (V_k e^{-j\theta_k} - V_m e^{-j\theta_m}) + j b_{km}^{sh} V_k^2 \tag{2.25}$$

A partir do segundo membro de (2.24), reescreve-se (2.25) em termos da potência ativa e reativa:

$$\begin{aligned} P_{km} &= V_k^2 g_{km} - V_k V_m g_{km} cos\theta_{km} - V_k V_m b_{km} sen\theta_{km} \\ Q_{km} &= -V_k^2 (b_{km} + b_{km}^{sh}) + V_k V_m b_{km} cos\theta_{km} - V_k V_m g_{km} sen\theta_{km} \end{aligned} \tag{2.26}$$

onde g_{km} é a condutância série da linha de transmissão, b_{km} é a susceptância série da linha de transmissão, V_m é a magnitude do m-ésimo elemento do vetor das tensões nodais, θ_{km} é a diferença entre os argumentos das tensões nas barras k e m e b_{km} é a susceptância *shunt* da linha.

Transformadores

Analogamente, para a dedução das expressões da potência ativa e reativa que fluem por um transformador, substitui-se a expressão da corrente que flui por esse equipamento – Expressão (2.11) – na expressão do conjugado da potência aparente em função da corrente – terceiro membro de (2.24):

$$S_{km}^{*} = (|t_{km}| y_{km} E_k - t_{km}^{*} y_{km} Ekm) E^{*} \tag{2.27}$$

Aplicando-se a *fórmula de Euler* [79] à expressão (2.27), é possível separá-la em parte real e imaginária e escrever os fluxos de potência ativa e reativa, conforme indicam o primeiro e segundo termos da relação (2.24).

$$\begin{aligned} P_{km} =& |t_{km}|^2 V_k^2 g_{km} - |t_{km}| V_k V_m g_{km} cos(\theta_{km} + \varphi_{km}) \\ & - |t_{km}| V_k V_m b_{km} sen(\theta_{km} + \varphi_{km}) \\ Q_{km} =& - |t_{km}|^2 V_k^2 b_{km} + |t_{km}| V_k V_m b_{km} cos(\theta_{km} + \varphi_{km}) \\ & - |t_{km}| V_k V_m g_{km} sen(\theta_{km} + \varphi_{km}) \end{aligned} \tag{2.28}$$

onde g_{km} é a condutância série do transformador de potência entre as barras k e m, b_{km} é a susceptância série do transformador de potência entre as barras k e m, V_m é a magnitude do m-ésimo elemento do vetor das tensões nodais, θ_{km} é a diferença entre os argumentos das tensões nas barras k e m, t_{km} é a relação de transformação complexa de tensão do transformador entre as barras k e m e φ_{km} é a defasagem angular do transformador de potência entre as barras k e m.

Generalização

As equações (2.28) e (2.26) podem ser generalizadas e reescritas da seguinte forma:

$$\begin{aligned} P_{km} =& |t_{km}|^2 V_k^2 g_{km} - |t_{km}| V_k V_m g_{km} cos(\theta_{km} + \varphi_{km}) \\ & - |t_{km}| V_k V_m b_{km} sen(\theta_{km} + \varphi_{km}) \\ Q_{km} =& - |t_{km}|^2 V_k^2 (b_{km} + b_{km}^{sh}) + |t_{km}| V_k V_m b_{km} cos(\theta_{km} + \varphi_{km}) \\ & - |t_{km}| V_k V_m g_{km} sen(\theta_{km} + \varphi_{km}) \end{aligned} \tag{2.29}$$

2.3 Formulação do Problema Básico

Nessa seção, o PFP será formulado, considerando-se inicialmente a classificação das barras da rede, segundo o conhecimento das grandezas a elas referentes. Dessa forma, é estabelecida a seguinte classificação:

- ***barras PV*** são aquelas em que se conhece o valor da potência ativa líquida e magnitude da tensão. São usualmente barras de geração ou de compensação de reativos, onde existe controle da magnitude de suas tensões.

- ***barras PQ*** são aquelas em que apenas se conhece o valor das potências ativa e reativa líquidas (usualmente, barras de passagem ou de carga);
- ***barra de referência, $V\theta$, swing ou slack*** – cumpre o papel de realizar o balanço de potência ativa, levando em conta as perdas de transmissão, desconhecidas antes da solução do PFP. Estabelece a referência angular do sistema. O valor do módulo de sua tensão é controlado via injeção de potência reativa.

Os métodos numéricos iterativos que são ou já foram amplamente aplicados para a *solução do problema do fluxo de potência em SEPs* exigem que valores iniciais sejam atribuídos para as variáveis calculadas (i.e., módulos e ângulos das tensões nodais complexas). Quando se conhecem os valores das tensões nodais complexas do sistema em um ponto de operação semelhante ao atual, estes podem ser usados para iniciar o processo iterativo que calcula o novo fluxo de potência. Esse procedimento é referido como ***hot start***. Já quando não se possui esse tipo de informação, é válido iniciar o processo iterativo com valores de módulo da tensão de $1,0pu$[1] e ângulo de $0,0rad$, pois é mais provável que, em condições normais de operação, a maior parte das barras do sistema não se distancie muito desses valores. Esse procedimento é referido como ***flat start***.

Vale mencionar que o método numérico de Gauss-Seidel, proposto por Glimn e Stagg [23], tornou-se amplamente adotado durante a década de 1960 para a solução do PFP [40]. Alguns aspectos favoráveis à adoção do método são: não requer a fatoração de matrizes e, portanto, necessita de menos recursos computacionais [46] – e.g., memória [73], por não haver o conhecido efeito de preenchimento de matrizes esparsas durante o processo de fatoração.

2.3.1 Método Numérico de Newton-Raphson

O *método de numérico de Newton-Raphson* [6], ou simplesmente *método de Newton*, é capaz de obter a solução de sistemas de equações algébricas não lineares [79], representados por funções vetoriais reais – i.e., funções $\boldsymbol{f}$ tais que $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^n$ – diferenciáveis [27].

Este método se baseia na linearização sucessiva [72] da função $\boldsymbol{f}$. A linearização $\boldsymbol{y}(\boldsymbol{x})$ de $\boldsymbol{f}(\boldsymbol{x})$ no ponto $\boldsymbol{x}_0$ pode ser descrita por:

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_0) + J_{\boldsymbol{f}}(\boldsymbol{x} - \boldsymbol{x}_0) \tag{2.30}$$

[1] A presença da marcação *pu* junto a uma medida indica que ela foi normalizada segundo sistema *por unidade* [24].

onde $\boldsymbol{x}_0$ é um vetor pertencente ao $\mathbb{R}^n$ fixado e $J_{\boldsymbol{f}}$ é a *matriz jacobiano* da função $\boldsymbol{f}(\boldsymbol{x})$.

A *matriz Jacobiano* [6] de uma função vetorial $\boldsymbol{f}\colon \mathbb{R}^n \to \mathbb{R}^m$ pode ser definida como uma matriz com m linhas e n colunas cujo elemento (i, j), dado por $\partial \boldsymbol{f}_i / \partial \boldsymbol{x}_j$, representa a derivada parcial [27] da i-ésima componente da função $\boldsymbol{f}$ com relação a sua j-ésima variável.

A aproximação de (2.30) pode ser reescrita de forma que possa ser avaliada sucessivamente:

$$\boldsymbol{x}^{(\lambda)} = \boldsymbol{x}^{(\lambda-1)} - \left(J_{\boldsymbol{f}}\big|_{\boldsymbol{x}=\boldsymbol{x}^{(\lambda-1)}}\right)^{-1} \cdot \boldsymbol{f}(\boldsymbol{x}^{(\lambda)}) \tag{2.31}$$

onde $\boldsymbol{x}^{(\lambda)}$ é um vetor parcial da solução, obtida após execução da λ-ésima iteração.

Assim como para o método numérico de *Gauss-Seidel*, o uso do método de Newton exige que seja adotada aproximação inicial para a raiz que se deseja aproximar.

Aplicação ao problema do Fluxo de Potência

Pode-se afirmar que o método de Newton-Raphson e suas variações (e.g., *método de Newton desacoplado rápido*) se tornaram os algoritmos mais comumente [46, 24] utilizados para a solução desse problema, estando presentes em várias aplicações comerciais (e.g., PSLF [22], ETAP [16] e ANAREDE [34]) e não comerciais (e.g., MATPOWER [82]).

O sistema de equações a ser resolvido será descrito a seguir. Considere k tal que $1 < k < nb$, onde nb é o número de barras da rede sob análise. Como mencionado anteriormente, considera-se um problema em que sejam dados os valores de P_k e Q_k para toda barra k de tipo PQ, valores de V_k e P_k para barras k de tipo PV e valores de V_k e θ_k para a barra *swing*. Para cada barra k do sistema em que for conhecida a potência ativa (P_k), avalia-se (2.22) para gerar uma equação para o sistema a ser resolvido. De semelhante forma, avalia-se (2.23) para cada barra k do sistema em que a potência reativa (Q_k) líquida é conhecida. Desse modo, é criado um sistema de equações algébricas não lineares.

Encontrada a solução do problema, avalia-se (2.22) para a barra *swing* e (2.23) tanto para a barra de referência angular do sistema quanto para cada uma das barras de tipo PV. Finalmente, com os valores das tensões complexas em todas as barras do sistema, são calculados os valores das correntes elétricas em todos os ramos da rede por meio das equações 2.5, 2.11 e 2.12. Os fluxos de potência em todos os ramos podem ser obtidos

por meio de 2.29.

Para a solução do sistema, organiza-se cada uma das equações descritas da seguinte forma:

$$\Delta \boldsymbol{P}^{(n)} = \boldsymbol{P}_{conhecido} - \boldsymbol{P}^{(n)}_{calculado} = 0 \tag{2.32}$$

e

$$\Delta \boldsymbol{Q}^{(n)} = \boldsymbol{Q}_{conhecido} - \boldsymbol{Q}^{(n)}_{calculado} = 0 \tag{2.33}$$

onde $\boldsymbol{P}_{conhecido}$ e $\boldsymbol{Q}_{conhecido}$ são, respectivamente, os valores previamente conhecidos das potências ativa e reativa líquidas; $\boldsymbol{P}^{(n)}_{calculado}$ e $\boldsymbol{Q}^{(n)}_{calculado}$ são, respectivamente, as expressões dos segundos termos de (2.22) e (2.23) avaliadas na n-ésima iteração do processo iterativo. Além disso, vale ressaltar que marcação $(\bullet)^{(n)}$ indica que o valor foi calculado na n-ésima iteração e que $\boldsymbol{P}^{(n)}_{calculado}$ e $\boldsymbol{Q}^{(n)}_{calculado}$ são funções dos valores das magnitudes ($\boldsymbol{V}$) e ângulos de fase ($\boldsymbol{\theta}$) das tensões nodais das barras adjacentes à barra onde a potência está sendo calculada.

Em seguida, organizam-se as equações (2.32) e (2.33) como sendo as entradas de um vetor $\boldsymbol{g}$ e cada um dos valores de magnitude e de ângulo de fase das tensões nodais desconhecidos como entradas do vetor $\boldsymbol{x}$:

$$\boldsymbol{x}^{(n)} = \begin{bmatrix} \boldsymbol{\theta}^{(n)} \\ \boldsymbol{V}^{(n)} \end{bmatrix} \quad e \quad \boldsymbol{g}(\boldsymbol{x}^{(n)}) = \begin{bmatrix} \boldsymbol{\Delta P}^{(n)} \\ \boldsymbol{\Delta Q}^{(n)} \end{bmatrix} \tag{2.34}$$

Deseja-se encontrar o conjunto de valores $\boldsymbol{x}$ de forma que $\boldsymbol{g}(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{\Delta P} \\ \boldsymbol{\Delta Q} \end{bmatrix} = 0$. O método se baseia na linearização da função $\boldsymbol{g}$ no ponto $\boldsymbol{x}^{(n)}$, dada pelos dois primeiros termos da série de Taylor:

$$\boldsymbol{g}(\boldsymbol{x}^{(n)} + \Delta \boldsymbol{x}^{(n)}) \cong \boldsymbol{g}(\boldsymbol{x}^{(n)}) + J(\boldsymbol{x}^{(n)})\Delta \boldsymbol{x}^{(n)} \tag{2.35}$$

onde J é a matriz Jacobiano, cujo elemento (i, j) é a derivada parcial da i-ésima entrada do vetor $\boldsymbol{g}$ com relação à j-ésima entrada do vetor $\boldsymbol{x}$.

A cada iteração, busca-se obter o zero da função linearizada – i.e., $\boldsymbol{g}(\boldsymbol{x}^{(n)} + \Delta \boldsymbol{x}^{(n)}) = 0 = \boldsymbol{g}(\boldsymbol{x}^{(n)}) + J(\boldsymbol{x}^{(n)})\Delta \boldsymbol{x}^{(n)}$ – e, em seguida, calcula-se o vetor $\boldsymbol{x}^{(n)}$.

$$\underbrace{\begin{bmatrix} \Delta \boldsymbol{P} \\ \Delta \boldsymbol{Q} \end{bmatrix}}_{\boldsymbol{g}(\boldsymbol{x})} = \underbrace{\begin{bmatrix} \dfrac{\partial(\Delta \boldsymbol{P})}{\partial \boldsymbol{\theta}} & \dfrac{\partial(\Delta \boldsymbol{P})}{\partial \boldsymbol{V}} \\ \dfrac{\partial(\Delta \boldsymbol{Q})}{\partial \boldsymbol{\theta}} & \dfrac{\partial(\Delta \boldsymbol{Q})}{\partial \boldsymbol{V}} \end{bmatrix}}_{J(\boldsymbol{x})} \cdot \underbrace{\begin{bmatrix} \Delta \boldsymbol{\theta} \\ \Delta \boldsymbol{V} \end{bmatrix}}_{\Delta \boldsymbol{x}} \begin{matrix} \left.\vphantom{\begin{matrix} a \\ a \end{matrix}}\right\} nPQ + nPV \\ \left.\vphantom{\begin{matrix} a \\ a \end{matrix}}\right\} nPQ \end{matrix}$$

$$\underbrace{\qquad\qquad}_{nPQ + nPV} \quad \underbrace{\qquad}_{nPQ}$$

Figura 2.4: Cálculo do fluxo de potência pelo método de Newton-Raphson[47].

Os passos desse método são detalhados a seguir, conforme descrição de Monticelli [47]:

(i) Fazer $n = 0$ e escolher uma solução inicial $\boldsymbol{x} = \boldsymbol{x}^{(n)} = \boldsymbol{x}^{(0)}$

(ii) Calcular o valor da função $\mathbf{g}(\boldsymbol{x})$ no ponto $x = \boldsymbol{x}^{(n)}$

(iii) Comparar o valor calculado $\mathbf{g}(\boldsymbol{x}^{(n)})$ com a tolerância especificada ε: se $|\mathbf{g}(\boldsymbol{x}^{(n)})| \leq \varepsilon$ o processo convergiu para a solução $\boldsymbol{x}^{(n)}$ e o processo iterativo chegou ao fim. Caso contrário, deve-se prosseguir para o passo (iv).

(iv) Calcular a matriz Jacobiana $J(\boldsymbol{x}^{(n)})$. As equações (2.36) e (2.37), vide Figura 2.4, a definem. Essa matriz a ser invertida para a solução do sistema, conforme indicado no próximo tópico.

$$J(\boldsymbol{x}^{(n)}) = \begin{bmatrix} [H] & [N] \\ [M] & [L] \end{bmatrix} \tag{2.36}$$

$$
\begin{aligned}
H &= \begin{cases} H_{km} = \partial P_k/\partial\theta_m = V_k V_m(G_{km} sen\theta_{km} - B_{km} cos\theta_{km}) \\ H_{kk} = \partial P_k/\partial\theta_k = -Q_k - V_k^2 B_{kk} \end{cases} \\
L &= \begin{cases} L_{km} = \partial Q_k/\partial V_m = V_k(G_{km} sen\theta_{km} - B_{km} cos\theta_{km}) \\ L_{kk} = \partial Q_k/\partial V_k = (Q_k - V_k^2 B_{kk})/V_k \end{cases} \\
M &= \begin{cases} M_{km} = \partial Q_k/\partial\theta_m = -V_k V_m(G_{km} cos\theta_{km} + B_{km} sen\theta_{km}) \\ M_{kk} = \partial Q_k/\partial\theta_k = P_k - V_k^2 G_{kk} \end{cases} \\
N &= \begin{cases} N_{km} = \partial P_k/\partial V_m = V_k(G_{km} cos\theta_{km} + B_{km} sen\theta_{km}) \\ N_{kk} = \partial P_k/\partial V_k = (P_k + V_k^2 G_{kk})/V_k \end{cases}
\end{aligned}
\tag{2.37}
$$

A matriz Jacobiana é quadrada de ordem $2 \cdot nPQ + nPV$, onde nPQ é o número de barras de tipo PQ e nPV é o número de barras de tipo PV.

(v) Calcular novo vetor de correção $\Delta\boldsymbol{x}^{(n)} = [J(\boldsymbol{x}^{(n)})]^{-1}\mathbf{g}(\boldsymbol{x})$

(vi) Determinar nova solução $\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \Delta\boldsymbol{x}^{(n)}$

(vii) Fazer $n + 1 \rightarrow n$ e voltar para o passo (ii).

2.4 Variações do Método de Newton Raphson

Modificações podem ser aplicadas ao método de Newton-Raphson para solução do PFP, visando melhorar seu desempenho computacional [73]. As principais se referem à construção de matrizes Jacobiano e à solução dos sistemas lineares em que elas participam.

Os ***métodos desacoplados*** de cálculo do fluxo de potência [73] fazem uso de duas propriedades verificadas para redes de transmissão de *extra-alta tensão* ($V > 230kV$) e *ultra-alta tensão* ($V > 750kV$) [47]. A primeira delas diz respeito à sensibilidade da potência ativa liquida das barras com relação aos ângulos de fase das tensões nodais ser muito superior à sensibilidade desses valores de potência com relação ao módulo das tensões nodais (i.e., $\partial P/\partial\theta >> \partial P_k/\partial\theta$).E a segunda característica se refere à sensibilidade da potência reativa líquida das barras com relação aos ângulos de fase das tensões nodais ser muito superior à sensibilidade desses valores de potência com relação ao módulo das tensões nodais (i.e., $\partial Q/\partial V >> \partial Q/\partial\theta$).

Um exemplo de aplicação desse princípio de desacoplamento é o ***método de Newton-Raphson desacoplado***. Nele, o sistema linear, representado pela matriz J do processo iterativo – conforme ilustração na Figura 2.4 –, é substituído por dois sistemas lineares [47]: um representado pela matriz $H = \partial P/\partial\theta$ – relacionando o vetor $\boldsymbol{\Delta P}^{(n)}$ com o vetor $\boldsymbol{\theta}^{(n)}$, descritos junto à relação (2.34) – e outro representado pela matriz $L = \partial Q/\partial V$ – relacionando o vetor $\boldsymbol{\Delta Q}^{(n)}$ com o vetor $\boldsymbol{V}^{(n)}$, descritos junto à relação (2.34).

Mais uma modificação que pode ser aplicada ao método de Newton-Raphson é a adoção de aproximações constantes para as curvas dos hiperplanos tangentes de sua versão desacoplada. Isso dá origem ao ***método de Newton-Raphson Desacoplado Rápido*** [74].

Além disso, outro ponto relacionado ao sistema linear em que podem ser observadas variações é quanto ao método escolhido para sua solução. Tradicionalmente adota-se método direto de solução (e.g., via fatoração LU [64]). Recentes estudos encontraram conveniência na aplicação de fatoração QR [64] para esse fim quando é feito uso de GPU para computar as rotinas, uma vez que, apesar de exigir maior esforço computacional frente a outros métodos como a fatoração LU, a fatoração QR apresenta boa estabilidade numérica [81] mesmo quando não é feito processo prévio de *pivoteamento*. Também podem ser encontrados estudos com aplicação de ***métodos iterativos para solução do sistema linear*** [66] – e.g., método dos resíduos mínimos generalizado (GMRES) [19] – que possibilitam mais fácil implementação em computadores paralelos e menor custo computacional frente aos métodos exatos.

2.5 Explorando a Esparsidade

Na implementação de programa computacional que solucione o PFP por meio de qualquer um dos métodos numéricos apresentados aqui, dois *paradigmas de programação* diferentes podem ser considerados com respeito ao *armazenamento, padrão de acesso e cálculo dos valores envolvidos*. O primeiro paradigma envolve características densas, onde todos os elementos das estruturas necessárias, sejam esses nulos ou não, são presentes. O segundo paradigma envolve características esparsas, onde apenas elementos não nulos são considerados.

Enquanto o uso de ***forma densa de armazenamento*** geralmente tem a vantagem de permitir que os elementos armazenados sejam diretamente acessados, o uso de ***formato de armazenamento esparso*** se mostra vantajoso principalmente por duas razões. Uma

delas refere-se à economia de posições de memória utilizadas, o que permite que estruturas de dados compactadas possam ser armazenadas em memórias de um *nível hierárquico* [61] mais elevado – mais rápidas, mas com capacidade total de armazenamento mais restrita. A outra está relacionada ao fato de que *cálculos desnecessários*, devido à presença de elementos nulos, podem ser evitados.

No PFP pelo *método de Newton-Raphson*, as matrizes de admitâncias nodais e Jacobiano são responsáveis pela característica esparsa dos algoritmos que o solucionam. Enquanto que o padrão de preenchimento da primeira reflete a conectividade física entre ou barras da rede, o padrão de preenchimento da segunda se origina dos termos da matriz de admitâncias nodais usados para a sua construção (conforme (2.37)).

Capítulo 3

Evolução do Desempenho de Microprocessadores

De uma forma ou de outra, a computação está presente em muitas das atividades da vida moderna. Esse capítulo apresenta uma descrição breve da evolução no tempo no que diz respeito à capacidade de processamento de dados de dispositivos que estão no cerne dos equipamentos usados em computação para os mais diversos propósitos.

3.1 Introdução

A computação pode ser entendida abstratamente como um processo deliberado, bem definido [70], que ao realizar uma sequência de passos transforma um conjunto de entradas em um conjunto de saídas.

Desde tempos bem remotos, o ser humano processa dados [4] nas mais variadas tarefas que se defronta. Sistemas Numeração e a Aritmética se desenvolveram e com eles dispositivos que pudessem auxiliar na execução de processos numéricos de cálculo, como, por exemplo, ábacos, ossos de Napier, réguas de cálculo, etc [4].

Resultados concretos dos esforços para a construção de máquinas que fossem capazes de calcular são atribuídos a Schickard (em 1623), Pascal (em 1642), Morland (em 1666) e Leibniz (em 1671), sem contudo se tornarem suficientemente precisas e economicamente viáveis [51].

Entre 1802 e 1822, Charles Babbage projetou uma máquina mecânica de calcular diferenças. De seus trabalhos foi originada a ideia fundamental de computadores controlados por programas [51].

Com o desenvolvimento tecnológico, foi possível construir computadores eletrônicos, primeiramente usando relés eletromecânicos (década de 1940) e válvulas termiônicas (década de 1950) como elementos comutadores [39] para possibilitar o processamento de dados. No início da década de 1960, transistores passaram a ser utilizados nos circuitos dos computadores com essa finalidade. Foi no final dessa década que circuitos integrados (CIs) – que concentravam vários transistores, capacitores, resistores e suas interconexões em um só chip de silício – passaram a ser empregados. Em 1971, o primeiro microprocessador produzido comercialmente, o Intel 4004, foi lançado [17].

3.2 Processadores Modernos e a Barreira da Potência

Após o Intel 4004, até meados da década de 2000, vários processadores de núcleo único foram criados com os mais variados objetivos de projeto. Pode-se dizer que a melhora observada nas suas performances é justificada por dois fatores. O primeiro deles é a capacidade de reduzir as dimensões físicas dos transistores que compõem os CIs. Foi observado que, a cada dezoito meses, o número de transistores possíveis de serem criados em uma mesma unidade de área de um CI aproximadamente dobra. Esse fenômeno foi denominado *Lei de Moore* [48]. Tal padrão pode ser observado no gráfico da Figura 3.1, que organiza os tamanhos mínimos dos componentes [49] que formam processadores de diversos fabricantes em cada ano. O outro fator está relacionado com a constante capacidade de redução da tensão de operação dos transistores que compõem os CIs. Esse fenômeno é conhecido como *Dimensionamento de Dennard* [14]. Dessa forma, e conforme será melhor descrito adiante, foi possível manter a potência demandada para o funcionamento dos processadores em níveis aceitáveis, mesmo se elevando não apenas o número de transistores envolvidos, mas também suas frequências de *clock* – i.e., frequência de uma sequência de pulsos utilizados para a sincronização de eventos internos que ocorrem no hardware [61] – e complexidade de seus *pipelines* (e.g., implementando funções complexas de execução de instruções fora de ordem).

Apesar disso, restrições físicas impedem que a tensão de funcionamento dos transistores continue a ser reduzida. Isso significa que, em termos práticos, o dimensionamento de Dennard deixou de ser válido [13]. A principal consequência desse fato foi tornar a potência dissipada por processadores seu principal fator limitante de projeto. Isso fez com que a elevação da frequência de *clock* e da complexidade do *pipeline* não fossem mais movimentos desejados no projeto de processadores, uma vez que não são energeticamente eficientes [30]. Esse ponto se torna visível na Figura 3.2, que organiza as velocidades de

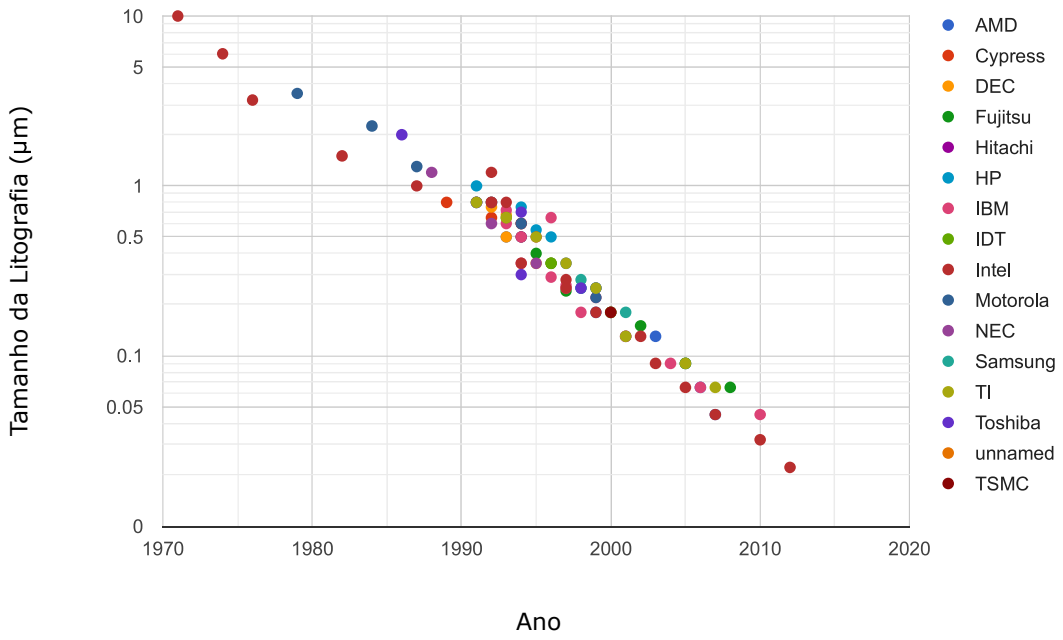

Figura 3.1: Dimensão dos menores componentes [49] de processadores de diversos fabricantes organizados por ano [12].

clock de processadores de vários fabricantes pelo ano de lançamento. Pode-se observar que até 2004 a velocidade de *clock* aumentou de forma aproximadamente constante. Contudo, a partir desse ano não ocorreram incrementos significativos nesse atributo.

3.2.1 Potência Dissipada por um Processador

Atualmente, pode-se dizer que *complementary metal–oxide–semiconductor* (CMOS) é a tecnologia dominante para circuitos integrados [61]. A Equação (3.1) descreve, de forma simplificada, a potência que é dissipada [50] em circuitos lógicos CMOS.

$$P = A \cdot C \cdot V^2 \cdot f + \tau \cdot A \cdot V \cdot I_{short} \cdot f + V \cdot I_{fuga} \tag{3.1}$$

O primeiro termo de (3.1) está relacionado com o consumo de potência dinâmica do circuito, que é devido à carga e descarga do componente capacitivo na saída de cada porta (*gate*). O consumo de potência dinâmica é diretamente proporcional à frequência de operação do sistema f, a um fator de atividade do sistema A (que representa a parcela das

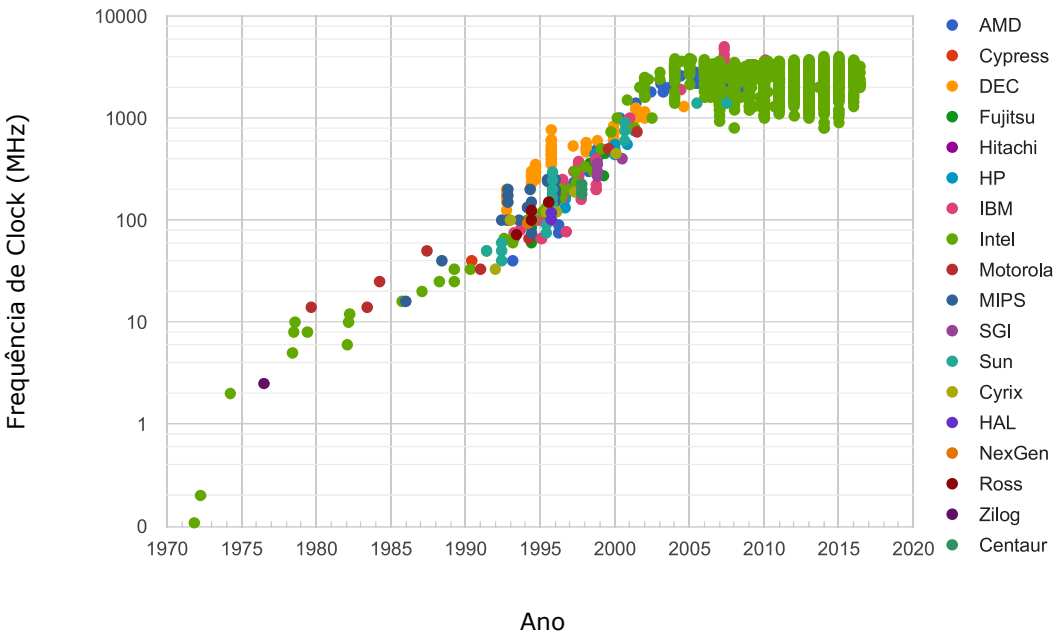

Figura 3.2: Frequência de *clock* de processadores de diversos fabricantes por ano [11].

portas que comutam em dado ciclo de clock), à capacitância das portas C e, ao quadrado da tensão de alimentação V. O segundo termo representa a potência transiente, resultado do fluxo da corrente I_{short} que surge, pelo período de tempo τ, quando a saída de uma porta lógica comuta. Por fim, a potência relacionada à corrente de fuga I_{fuga} no transistor, que independe da forma como se chaveia esse dispositivo, é representada pelo terceiro termo.

A potência transiente é desprezível para esse tipo de circuito lógico CMOS [30] e, para aplicações de núcleo único, a potência de fuga inicialmente também o era. Existe interdependência entre diferentes termos de (3.1). Uma delas é que a frequência de chaveamento do circuito possui um limite superior [50] f_{max} que, de forma aproximada, depende da tensão de alimentação V e da tensão de limiar V_{limiar}:

$$f_{max} \propto \frac{(V - V_{limiar})^2}{V} \tag{3.2}$$

Outra relação de interdependência, que relaciona a tensão de limiar à corrente de fuga I_{fuga}, é representada por:

$$I_{fuga} \propto exp\left(\frac{-V_{limiar}}{35mV}\right) \tag{3.3}$$

Como alcançar frequências de *clock* elevadas era um dos principais objetivos dentre projetistas de processadores de núcleo único, as tensões de (3.2) foram exaustivamente minimizadas e a frequência de operação f do processador projetado aproximada da máxima possível. Pode-se verificar em (3.3) que a corrente de fuga possui uma correlação inversa com a tensão de limiar. Dessa forma, os reduzidos valores alcançados para as tensões de limiar dos elementos dos circuitos lógicos acarretaram maiores correntes de fuga. Isso fez com que a parcela de (3.1) relativa à dissipação de potência devida à corrente de fuga também fosse elevada [50, 42]. Um exemplo disso pode ser observado no trabalho de Liao et al. [42], no qual verificou-se que essa fonte pode ser responsável por mais de 40% da dissipação total de potência em um processador.

Processadores de Vários Núcleos

Suponha que uma tarefa arbitrária que pode ser dividida em duas partes, executáveis concorrentemente. Tal tarefa pode ser executada sequencialmente em um processador com frequência de *clock* f_{max} – próxima do valor máximo possível, para aproveitar a maior produtividade que esse processador pode oferecer –, tensão de operação V e consumo de potência P. Caso a tensão de operação do processador seja reduzida pela metade, 3.2 indica que o limite máximo da frequência de operação também será reduzido proporcionalmente (para $f_{max}/2$). Como o número de instruções por ciclo de *clock* permanece constante com essas variações, o número de operações elementares que esse processador executa por unidade de tempo também será reduzido proporcionalmente. Desse modo, seria necessária utilização de 2 processadores iguais a esse para que essa tarefa fosse concluída no mesmo período de tempo. Contudo, a potência demandada por esses 2 processadores com tensão $V/2$ e frequência $f_{max}/2$ é, aproximadamente, a metade [50] da exigida pelo único processador que funciona sob a tensão V e frequência f_{max}.

Isso ilustra que, se o projeto de um processador de vários núcleos for feito de forma cuidadosa, ele possibilitará a execução de uma tarefa completamente paralelizável demandando o mesmo tempo para a conclusão, porém dissipando menor quantidade de energia para esse fim. Tal princípio propulsionou a criação de processadores de múltiplos núcleos até que eles se tornassem a nova regra.

3.3 Barreira de Utilização

A cada nova geração de processadores de múltiplos núcleos, projetistas usaram a maior parte dos transistores que passaram a estar disponíveis para a criação de mais núcleos, para manter a energia dissipada em níveis desejáveis (conforme ilustrado na Subseção 3.2.1). Contudo, diferentemente das melhorias de processadores com um único núcleo, ganhos de performance advindos da criação de novos núcleos exigem que programas estejam adequadamente adaptados para dividirem suas operações de processamento de dados em grupos passíveis de execução concomitante, de forma que seja possibilitada distribuição da carga de trabalho por todos os núcleos disponíveis. Apenas dessa forma a execução de um único programa pode, de fato, se beneficiar dessas melhorias.

Além disso, vários outros limitantes conduziram o desenvolvimento de processadores com núcleos homogêneos (i.e., todos os núcleos que esses processadores possuem são idênticos) para um ponto em que existe uma série de empecilhos para que seus níveis de processamento máximos sejam alcançados. Os três principais empecilhos [30] são: a falta de paralelismo de programa a ser executado, limitações da largura de banda do canal externo (*off-chip*) de comunicação e restrições de dissipação de energia. Essa questão é referida como a *barreira de utilização* e a fração do processador que fica inativa devido à barreira de utilização é denominada *silício negro*.

Uma solução [30] para tais limitações é a adoção de arquiteturas heterogêneas[1] e especializadas (portanto, mais simples), em que componentes podem executar um conjunto de tarefas de forma bem eficiente. As GPUs (*Unidades de Processamento Gráfico*) modernas se encaixam nesse perfil, podendo ser utilizadas como co-processadores capazes de acelerar aplicações de diversas áreas. GPUs serão melhor abordadas com mais detalhe no Capítulo 4.

[1] A *computação heterogênea* faz referência ao emprego mais de um tipo de núcleo de processamento em um mesmo sistema. Essa ideia se contrasta com a *computação homogênea*, onde todos os núcleos dos processadores presentes são iguais.

Capítulo 4

Unidade de Processamento Gráfico

Unidades de processamento gráfico (GPUs) estão entre os elementos comumente encontrados em computadores pessoais atuais. Aqui, a programação em GPUs para propósitos gerais será apresentada, com enfoque na API CUDA. Aspectos da arquitetura Pascal do fabricante Nvidia são apresentados.

4.1 Introdução

Aplicações de propósito geral implementadas em GPUs modernas podem apresentar aceleração de desempenho de várias ordens de grandeza [43] quando comparadas às que usam exclusivamente as tradicionais *unidades centrais de processamento* (CPUs). Além disso, redução da quantidade de energia demandada para a execução do processamento também pode ser observada. Isso é devido a diferenças fundamentais entre as filosofias de projeto [38] desses dois tipos de processadores.

CPUs têm como princípio de projeto a redução da latência individual de cada um dos *threads* (i.e., linhas de execução) por elas executados. Para concretizar esse requisito, grandes memórias de *cache* de último nível são construídas no *chip* para que sejam armazenados dados acessados com maior frequência. Quando isso é feito, acessos à memória de alta latência são substituídos por consultas rápidas ao cache. Além disso, unidades aritméticas e lógica de acesso a operandos também são projetadas de forma que suas latências sejam minimizadas. Contudo, esses três tipos de práticas consomem não somente área do *circuito integrado* (CI), mas também energia durante seu funcionamento. Contudo, essa área do CI e a referida energia poderiam ser utilizadas para que fossem realizadas mais operações aritméticas e acessos à memória.

Diferente das CPUs, os princípios de projeto de GPUs foram historicamente impulsionados pela indústria dos videogames e computação gráfica. Essas finalidades se beneficiam da execução de um grande número de operações a cada quadro de vídeo produzido. Isso exige que o projeto dos CIs de GPUs tenham como objetivo a maximização da vazão de cálculos. Essa prática é denominada *projeto orientado à vazão* [38]. A solução predominante para esse fim está relacionada à economia de área do CI e potência ao permitir que acessos à memoria e operações aritméticas tenham maior latência. Nesse sentido, a vazão de cálculo e de dados acessados da memória pode ser magnificada ao direcionar os recursos economizados para essa finalidade. Além disso, como existe elevada independência entre cálculos envolvidos no *pipeline gráfico*[1], é adotado design com grande número de núcleos e linhas de execução concomitantes. Esse grande número de *threads* concorrentes é capaz de esconder a latência dos componentes da GPU ao manter os elementos do *chip* sempre ocupados.

4.2 Programação em GPUs

4.2.1 Abordagem Histórica

Os computadores das décadas de 1970 e 1980 não tinham componentes como as GPUs modernas, mas sim controladores gráficos [61, 62]. Esses controladores consistiam, de forma simplificada, em um ou mais *chips* de *memória dinâmica de acesso aleatório* (DRAM) junto ao controlador de memória necessário e hardware capaz de mapear o conteúdo dessa memória para saída de vídeo. Nas décadas de 1980 e 1990, o hardware gráfico de maior desempenho possuía *pipelines* de função fixa configuráveis, mas não programáveis. Tais funções podiam ser acessadas via *interfaces de programação de aplicações* (APIs) como DirectX [62] – proprietária, disponibilizada pela Microsoft desde 1995 – ou OpenGL [68] – uma API de padrão aberto, lançada [62] em 1992 e suportada por diferentes fabricantes.

É desejável que algumas das tarefas [38] executadas no pipeline gráfico sejam passíveis de introdução de variabilidade, de acordo com as necessidades de cada projeto de computação gráfica. Tais características que motivaram que certas etapas fossem feitas programáveis, e não mais funções fixas. Contudo, algoritmos diferentes fazem uso dessas etapas de forma desigual, de modo que, enquanto a proporção dos recursos de hardware alocados para uma dada parte são abundantes para dado algoritmo, os recursos alocados

[1] O *pipeline gráfico* [38] pode ser definido como uma sequência de ações necessárias para transformar uma cena 3D em um conjunto de pixels, passiveis de exibição em uma tela 2D.

para outras são insuficientes. Isso motivou que fosse construída arquitetura unificada [38] que possibilitasse que os mesmos recursos de hardware fossem dinamicamente alocados para diferentes etapas do *pipeline* gráfico. Esses processadores gráficos unificados foram dotados de operadores de inteiros e de pontos flutuantes eficientes.

Essas novas características chamaram a atenção da comunidade científica para o surgimento de condições que possibilitariam a execução de rotinas de propósito geral em GPUs. Porém, com a tecnologia da época, isso não era simples de ser alcançado. Programadores precisavam encontrar, em APIs gráficas como OpenGL ou DirectX, funções equivalentes ao tratamento desejado para o problema computacional abordado. De forma simplificada, funções tinham que ser escritas como o processamento de *pixels* de imagens o que, mesmo fazendo uso de linguagens de alto nível, era muito difícil e limitante, pois forçava todo o código a se encaixar nas APIs projetadas para pintar pixels.

No projeto da arquitetura Tesla [38] da *Nvidia Corporation*, os processadores gráficos unificados foram feitos completamente programáveis, possuindo memória de instruções, *cache* de instruções e lógica de controle de sequenciamento de instruções. Muitos processadores de *shaders* [38] dividem seus *caches* de instruções e lógica de controle de sequenciamento de instruções, o que é conveniente para aplicações gráficas, uma vez que os mesmos programas precisam ser aplicados a vários vértices ou *pixels*. Ademais, foram acrescidas instruções de acesso e escrita de memória com capacidade de endereçamento de *bytes* aleatórios para suportar os requisitos de programas C compilados.

4.2.2 APIs de Programação de Propósito Geral em GPUs

Nesse contexto, estando notório o potencial do uso de GPUs como co-processadores para acelerar determinadas tarefas, várias APIs foram lançadas como meios para essa prática. Em 2006, a Nvidia lançou a API CUDA (inicialmente, um acrônimo para *Compute Unified Device Architecture*) [54, 67], de código fechado, que possibilita a programação de propósito geral em GPUs (GPGPU) produzidas pela fabricante. CUDA suporta linguagens de programação como C, C++ e FORTRAN. Outra API GPGPU importante é a OpenCL, um padrão aberto desenvolvido inicialmente pela *Apple Inc.* e refinado por equipes de diversas empresas relacionadas ao setor de GPUs. Essa API foi lançada em 2009 e, atualmente, é mantida pelo consórcio industrial *Khronos Group Inc.* [32, 67]. Produtos de diversas companhias [33] são conforme o padrão que define uma API passível de ser chamada de programas para a CPU nas linguagens de programação C e C++. A OpenACC é mais uma API relacionada ao processo de GPGPU. Seu nome é um acrônimo para *Open*

Accelerators. Ela se diferencia das duas últimas ao disponibilizar diretivas para o compilador, capazes de fazer com que código feito originalmente para ser executado na CPU execute de forma paralela em GPUs. Dessa forma, boa parte dos aspectos específicos da arquitetura heterogênea sendo utilizada são abstraídos, o que facilita a implementação e portabilidade do código tanto entre sistemas de diferentes fabricantes quanto distintas gerações de equipamentos da mesma empresa.

Apesar dessas APIs fornecerem algumas comodidades e abstraíram parte das características das plataformas paralelas heterogêneas, ainda é importante que programadores conheçam características das arquiteturas nas quais os programas serão executados para que, de forma efetiva, se alcance bom desempenho.

4.2.3 Modelo de Programação CUDA C/C++

A Figura 4.1 representa, de forma esquemática, como um programa CUDA C/C++ pode ser organizado. Nesse diagrama, cada linha ondulada representa uma *linha de execução* – ou, em inglês, *thread*. No meio da computação GPGPU existem dois termos muito comuns. O primeiro deles é *host* ou *hospedeiro* e faz menção à CPU e à memória DRAM principal do computador, que ela tem acesso. O outro é *device* ou *dispositivo* que referenciam a GPU e sua *memória global*, melhor descrita na Seção 4.2.3. Código comum em linguagem C cria uma única linha de execução. Caso seja utilizada API de processamento paralelo [69] – e.g., OpenMP [69] ou POSIX Threads [7] –, vários threads podem ser executados na CPU, conforme ilustrado na seção do programa do esquema da Figura 4.1 "Execução de Código Paralelo CPU".

Kernels

CUDA C++ estende a linguagem de programação C++, permitindo que o programador defina funções, denominadas *kernels*, que, quando chamadas, são executadas em paralelo por *N linhas de execução* diferentes. Um *kernel* é definido usando o especificador de declaração `__global__` [53]. A seguir está representado um trecho de código ilustrativo da declaração e chamada de um *kernel*.

```
1        // Definição do Kernel
2        __global__ void nomeDoKernel(tipo argumentosDoKernel /* ... */)
3        {
4                // código a ser executado na GPU
5        }
```

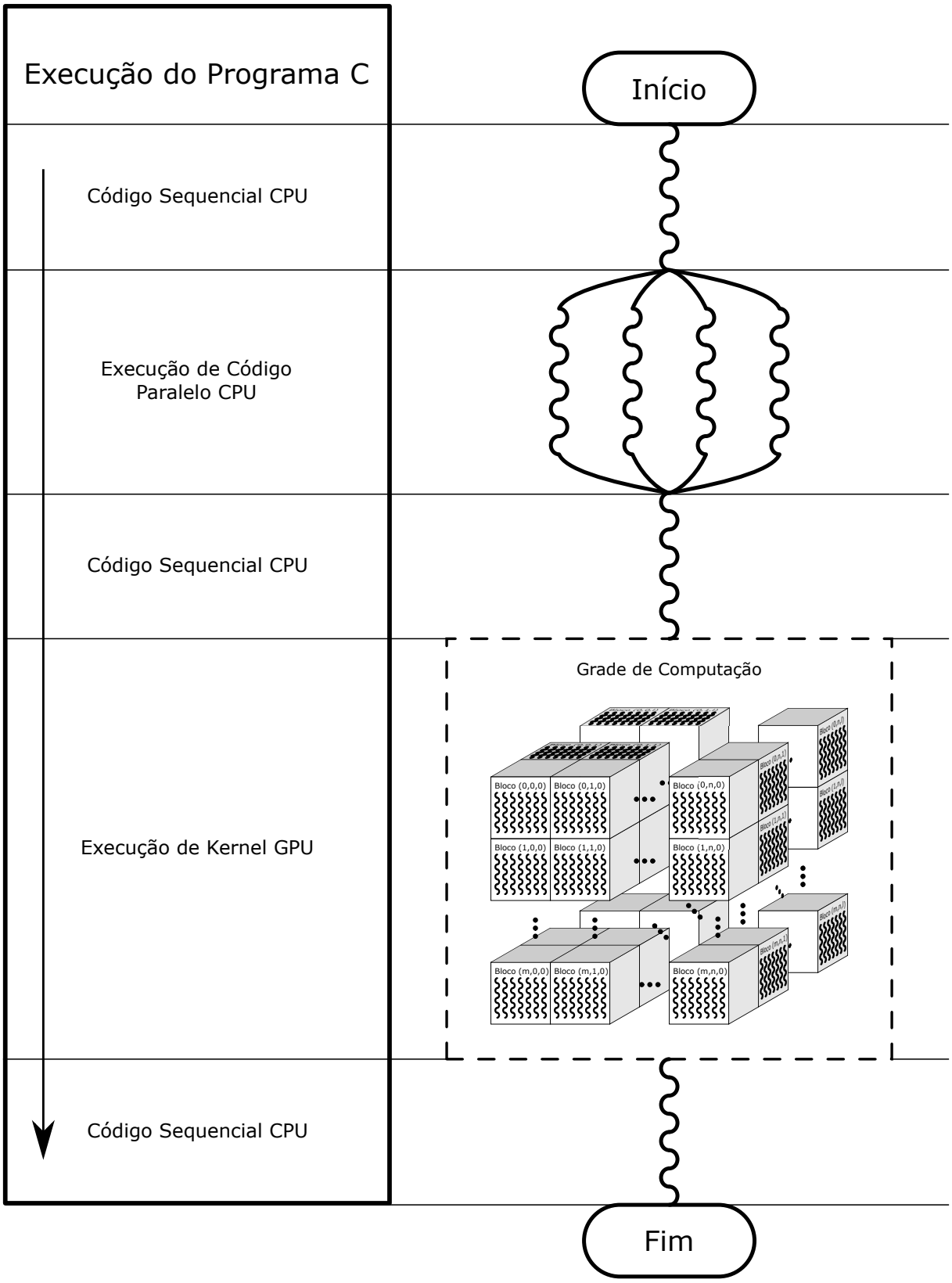

Figura 4.1: Principais estruturas de um programa CUDA C [53].

```
6
7         //...
8
9         int main()
10        {
11                //...
12                // Chamada do Kernel
13                nomeDoKernel<<<numeroDeBlocosDaGrade, numeroDeThreadsPorBloco,
14                numeroDeBytesDeMemoriaCompartilhadaPorBloco, streamAssociada>>>
15                (argumentosDoKernel);
16                //...
17        }
```

A execução de um kernel também foi ilustrada na Figura 4.1. Notar que, enquanto existem CPUs que podem executar centenas de *threads* ao mesmo tempo, o número máximo de linhas de execução simultâneas de uma única GPUs pode atingir a ordem de centenas de milhares de linhas de execução simultâneas.

Vale salientar que, como o *host* e o *device* possuem ***espaços de memória*** separados [38], faz-se necessária a transmissão de todos dos dados de todas as variáveis armazenadas na memória principal do computador que serão necessários para a execução de dado *kernel* para a memória do dispositivo antes que esse *kernel* possa ser executado. Esse processo é normalmente feito via interface PCIe (conforme ilustrado para a arquitetura Pascal na Seção 4.3). Além disso, todo produto da computação do *kernel* que será utilizado por rotinas executadas no *host* deverá, de forma semelhante, ser copiado da memória do *device* para a memória do *host*. Esse processo de cópia e transferência de dados via interface PCIe pode exigir tempo significativo frente ao tempo demandado para a execução do *kernel*. Por isso, sempre que possível, transferências entre o hospedeiro e o dispositivo devem ser minimizadas e o tempo demandado para elas deve ser considerado quando analisada a efetiva aceleração promovida por um *kernel* em contraste com sua versão executada em CPUs.

Blocks e grids

Os *threads* que executam em paralelo no *device*, chamados por um mesmo *kernel*, são subdivididos em *blocos* – em inglês, *blocks*. Por sua vez, um conjunto de *blocos* é organizado em uma *grade* – em inglês, *grid*.

Uma representação esquemática dessas estruturas pode ser observada na Figura 4.1. A afirmação do número de *threads* lançado por determinada chamada de um *kernel* é

definida [53] pelos parâmetros *número de blocos na grade* e *número de threads por bloco*, escritos entre os delimitadores "<<<" e ">>>" na chamada do *kernel*, conforme pode ser observado no trecho de código ilustrativo da Subsubseção 4.2.3.

Warps

GPUs produzidas pela Nvidia baseiam seu funcionamento em um modelo de execução denominado Instrução Única, Múltiplos Threads (SIMT) [53] que é uma extensão [61] do modelo Instrução Única, Múltiplos Dados (SIMD), um dos quatro *modelos taxonômicos de arquiteturas de computação de Flynn* [20]. Enquanto SIMD é tipicamente implementado com núcleos de processamento escalares, dotados de registradores e unidades de execução vetoriais, SIMT é um modelo em que *threads* múltiplos executam instruções comuns a dados arbitrários. A esse conjunto de *threads* que executam instruções comuns, dá-se o nome de *warp*. O número de *threads* adotado pelos projetistas das GPUs atuais da Nvidia para constituir um *warp* é 32 [53].

Eventualmente, podem existir instruções de desvio que fazem com que *threads* de um mesmo *warp* tomem caminhos de execução diferentes. Quando isso ocorre, os trechos dos caminhos de execução *divergentes* são percorridos de forma sequencial pelos *threads* de dado *warp*. Isso não é um comportamento desejável, uma vez que faz com que o desempenho da aplicação se afaste da melhor performance que o hardware pode oferecer.

Streams

Um *stream* é uma sequência de comandos (possivelmente emitida por diferentes *threads* em determinado *host*) que executam no *device* de forma ordenada. Diferentes *streams*, por outro lado, podem ter seus comandos executados fora de ordem, uma *stream* com relação à outra [53].

Apesar dos lançamentos de *kernels* na plataforma CUDA serem assíncronos com relação à execução de comandos no *hospedeiro*, todas as tarefas relacionadas à GPU colocadas em um mesmo *stream* são executadas uma após a outra. Contudo, *kernels* associados a *streams* diferentes potencialmente podem ser executados ao mesmo tempo. Quando o campo `streamAssociada` – representado no trecho de código ilustrativo da declaração e chamada de um *kernel* na Subseção 4.2.3 – é deixado em branco, todos os *kernels* do programa são associados a uma única *stream* [53] (i.e., à *stream padrão*).

Hierarquia de Memória

Threads CUDA possuem acesso a diferentes espaços de memória. Os principais serão apresentados a seguir.

A ***memória global*** é situada fora do CI do processador da GPU e implementada usando *chips* DRAM. Por isso, dentre todos os diferentes tipos de memória que um *thread* CUDA pode acessar, a *memória global* é a que possui maior capacidade de armazenamento, mas também maior latência de acesso e menor vazão. Todos os *threads* de todos os *blocos* e todas as *grades* podem acessar dados nessa memória. Além disso, é para ela que são transferidos dados vindos da memória da CPU e vice versa. Um fator importante relacionado à performance de *kernels* CUDA é o acesso à *memória global*. O acesso, por *threads* vizinhos, a vários elementos de memória *coalescidos* pode ser feito em uma única operação. Caso os elementos buscados estiverem em locais distantes da memória, são necessárias várias operações, o que prejudica a performance da aplicação.

Existem dois tipos principais de memória situadas dentro do CI do processador da GPU. Uma delas é a ***memória compartilhada***. Sua localização dentro dos *Streaming Multiprocessors* da arquitetura Pascal estão representados na Figura 4.3. O conteúdo dessa memória pode ser definido no programa CUDA e é comum para todos os *threads* em dado *bloco*. O outro tipo principal de memória dentro do CI da GPU são os ***registradores***. Eles comportam a maior parte das variáveis locais a cada um dos *threads*. Um ponto importante que tange o uso dos registradores e suas limitações é quando um limite superior de capacidade de registradores é atingido. Esses limites são específicos para cada arquitetura e, para a arquitetura Pascal, são expostos adiante. Quando isso ocorre, menos *blocos* são atribuídos aos *Streaming Multiprocessors* durante a execução, o que pode reduzir drasticamente o paralelismo da aplicação e, consequentemente, a performance alcançada.

Além disso, também existem vários outros tipos de memória, *caches* e *buffers* nos CIs das GPUs. Alguns dos presentes na arquitetura Pascal serão expostos na Seção 4.3. O leitor que desejar explicação mais detalhada do uso e aplicação dos tipos de memória citados nesse tópico, assim como da *memória de textura* ou da *memória constante*, pode recorrer ao livro de Kirk [38].

4.3 A Arquitetura Pascal

Nessa seção, serão expostos elementos da arquitetura Pascal da fabricante Nvidia. Isso será de grande valia para ilustrar, de modo geral, como uma GPU se organiza. Essa arquitetura foi escolhida por ser a arquitetura da GPU utilizada para os experimentos apresentados no Capítulo 6. Na Figura 4.2, apresenta-se um diagrama esquemático simplificado do chip GP100 [71], da arquitetura Pascal. Esse *chip* possui 60 *Streaming Multiprocessors* (SMs) organizados em 6 unidades *Graphics Processing Clusters* (GPCs) – i.e., cada GPC possui 10 SMs. O *Giga Thread Scheduler* (GTS) é a unidade responsável pela atribuição de *blocos* de *kernels* a cada um dos *Streaming Multiprocessors* organizados dentro de cada um dos GPCs. Além disso, o *chip* possui controlador PCIe 3.0 que possibilita comunicação entre o *host* e o *device*. Os controladores de memória são os componentes responsáveis por copiar dados da *memória global* – DRAM localizada fora do chip da GPU (apresentada na Subsubseção 4.2.3) – para o cache de nível 2 (L2$) e vice-versa. Por esse meio, *threads* são capazes de acessar dados armazenados na memória global.

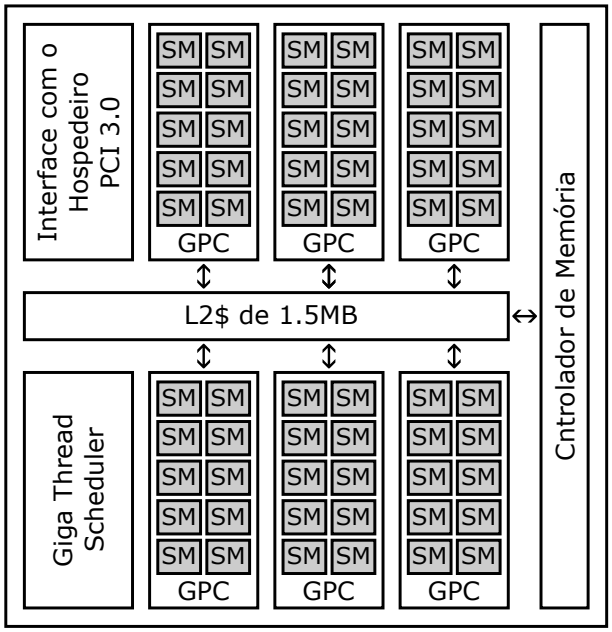

Figura 4.2: Diagrama esquemático simplificado do chip GP100, de arquitetura Pascal [71].

4.3.1 Elementos dos Streaming Multiprocessors

Na Figura 4.3, está representado diagrama simplificado da estrutura de um *Streaming Multiprocessor* (SM) do *chip* GP100, de arquitetura Pascal. Esse SM [71] possui duas subestruturas idênticas. Cada uma dessas subestruturas concentra 32 *núcleos*, 16 *unidades de precisão dupla* (DPUs), 8 *filas de carga/armazenamento de dados* (LDST) e

8 *unidades de funções especiais* (SFUs). Além disso, cada uma dessas unidades possui seu próprio *buffer de instruções*, *agendador de warps*, par de *despachantes* e *arquivo de registradores*. Essas duas unidades dividem *cache* de instruções, *cache* de nível 1 e memória compartilhada. Cada um desses componentes será melhor descrito nas subsubseções abaixo.

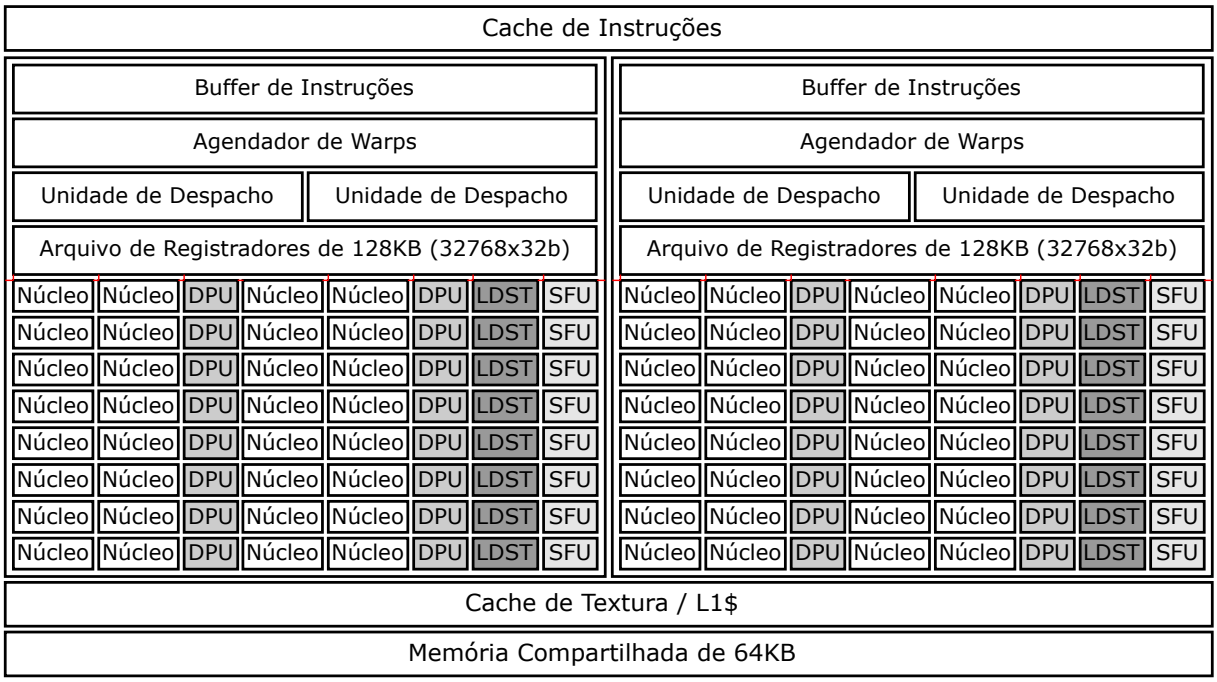

Figura 4.3: Diagrama esquemático simplificado do *Streaming Multiprocessor* do *chip* GP100, de arquitetura Pascal [71].

Núcleos da GPU

Núcleos da GPU são estruturas que possuem unidades capazes de operar valores inteiros e pontos flutuantes de precisão simples. Cada núcleo possui *porta de despacho*, por onde eles recebem a próxima instrução a ser executada, e *porta de coletora de operandos*, pela qual são recebidos os operandos antes armazenados no *Arquivo de Registradores*. Além disso, os núcleos também possuem capacidade de fazer operações de pontos flutuantes de precisão dupla. Contudo, exigem vários ciclos para executar esse tipo de tarefa [71].

Unidades de Precisão Dupla (DPUs)

Unidades de Precisão Dupla são estruturas capazes de realizar operações em valores de pontos flutuantes de precisão dupla com maior velocidade do que um núcleo da GPU é capaz de fazer. Elas exigem mais espaço físico no circuito integrado (CI) do que os

núcleos [71]. Isso está relacionado ao fato da área necessária para se construir um multiplicador crescer de forma quadrática com o número de bits envolvidos. Por esse motivo, alguns *chips* da arquitetura Pascal são desprovidos de DPUs. Isso acarreta uma vazão de operações sobre valores de ponto flutuante 32 vezes menor [71, 53] quando comparada à vazão de operações sobre aqueles de precisão simples.

Unidades de Funções Especiais (SFUs)

As *Unidades de Funções Especiais* possibilitam a execução eficiente de funções transcendentais como seno, cossenos, exponenciações, raízes quadradas ou logaritmos. São muito utilizadas não apenas por aplicações científicas, mas também por aplicações gráficas para executar, por exemplo, processo de rotação em objetos tridimensionais.

Arquivo de Registradores (RF)

Como pode ser observado na Figura 4.3, cada SM de uma GPU com *chip* GP100 possui arquivo de registros com 32.768 registradores de 32 bits, esses registradores totalizam $128kB$ de capacidade. Além disso, existem outas limitações de hardware [71] que tangem o número de registradores que um *thread* pode possuir. Em GPUs antigas (até a *capacidade de computação* [53] 3.0), esse número era 63 [71], em GPUs mais recentes, passou a ser 255 (i.e., até a *capacidade de computação* mais recente, 8.6 [53]). É importante notar que, como um valor de ponto flutuante de precisão dupla possui tamanho de 64 bits, são necessários dois registradores de 32 bits para seu armazenamento [53].

Filas de Carga/Armazenamento de Dados (LDST)

Filas de Carga e Armazenamento de dados – em inglês *Load/Store Queues* – são usadas para fazer o trânsito de informações entre os núcleos e a memória. Qualquer núcleo que precise fazer uma operação de carga ou armazenamento de dados da memória é faz um pedido que é enfileirado nessa estrutura. Enquanto o requerimento não é concretizado, outro(s) *warps* são agendados para execução [71] nesse ínterim. Esse conceito pode ser referenciado como *ocultação de latência* (ou *latency hiding*, em inglês). Sua exploração, ao buscar disponibilidade de *warps* independentes, passíveis de execução enquanto operações de carga ou armazenamento de dados não são finalizadas, é um ponto importante do desenvolvimento de um programa voltado para GPUs.

Cache de Nível 1 (L1$) e Cache de Textura

O *Cache de Nível 1* e de Textura é um *cache* unificado [53]. O cache de nível 1 é *controlado por hardware* [71] (i.e., o programador não tem controle de quais elementos são ou não armazenados nessa memória). A API CUDA suporta um subconjunto das funções do hardware de texturização [53] que a GPU usa para acessar essa memória, originalmente usadas para o *pipeline gráfico*.

Memória Compartilhada

A memória compartilhada é um cache *controlado por software* [71]. Dessa forma, é possível que um programa CUDA determine o qual informação o hardware do SM deve armazenar nessa memória.

Memória Constante

É um cache somente de leitura, compartilhado por todas as unidades funcionais e que acelera as velocidades de leitura do espaço de memória constante, uma fração da *memória global do dispositivo*.

Cache de Instruções

Esse *cache* [71] armazena instruções de um bloco em execução em dado SM. Toda vez que o GTE atribui um bloco a dado SM, instruções desse bloco também são carregadas nesse *buffer*.

Buffer de Instruções

Essa memória [71] atua como *cache* de instruções local de cada SM. Instruções armazenadas no *cache* de instruções são copiadas para esse *buffer*.

Agendador de Warps

O agendador de *warps* tem como função transformar cada bloco atribuído pelo GTS em um conjunto de *warps* e agendá-los para serem executados.

Unidade de Despacho

Uma vez que exista disponibilidade de recursos necessários para a execução de um *warp* que esteja agendado, a *Unidade de Despacho* o faz, enviando informações necessárias para os *núcleos*, DPUs, SFUs ou LDSTs cabíveis.

Capítulo 5

Metodologia

Nesse capítulo será apresentada a aplicação de técnicas de computação paralela para a solução do PFP. Após breve introdução sobre a eficiência computacional da implementação de métodos numéricos usados na solução do problema, apresenta-se a aplicação de GPU como ferramenta alternativa para a consecução deste objetivo.

5.1 Introdução

A construção de metodologia para a solução do PFP é tarefa complexa, uma vez que pequenos descuidos podem acarretar elevação do tempo computacional demandado. Cada etapa apresentada será acompanhada de detalhes da implementação que visam a obtenção de bom desempenho. Além disso, consideração da complexidade assintótica de tempo [10] das rotinas propostas se mostra relevante uma vez que, ao envolver sistemas de grande porte, más características de escalabilidade podem impactar significativamente o tempo demandado para execução.

5.2 Solução do PFP pelo Método de Newton-Raphson

Os diagramas das Figuras 5.1 e 5.2 ilustram as principais etapas que constituem as metodologias propostas para solucionar o problema com o uso da CPU. Já o diagrama da Figura 5.3 apresenta as principais etapas do método adotado adaptadas para uso da GPU. Todas as abordagens se baseiam no que foi descrito na Subseção 2.3.1.

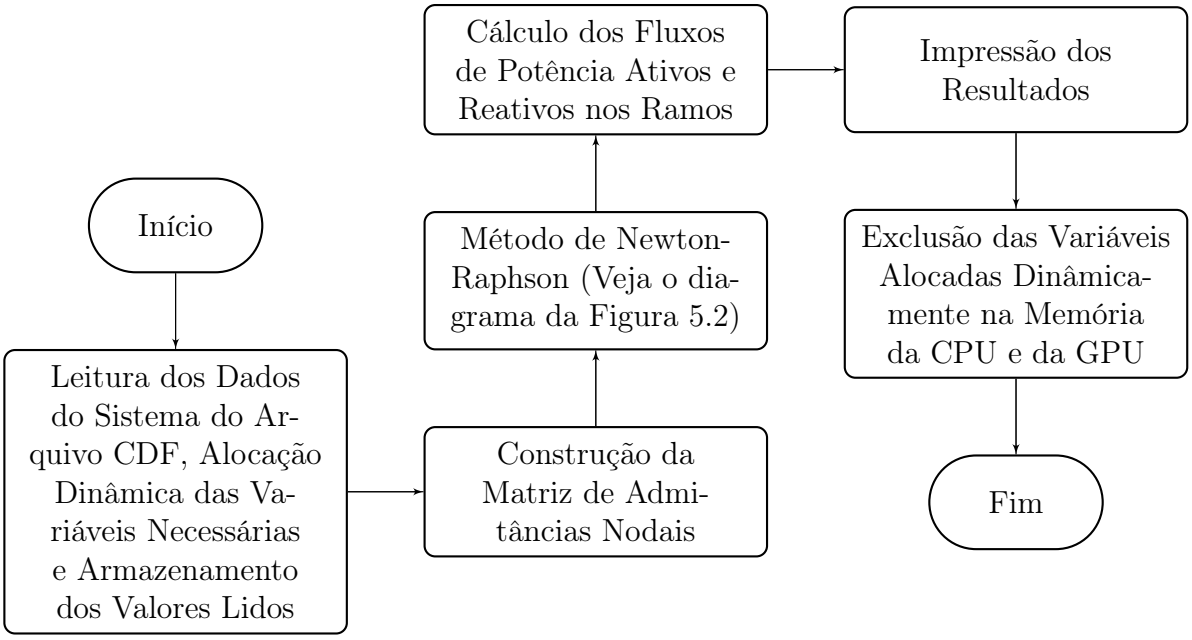

Figura 5.1: Fluxograma simplificado do processo utilizado para solução do *problema do Fluxo de Potência*.

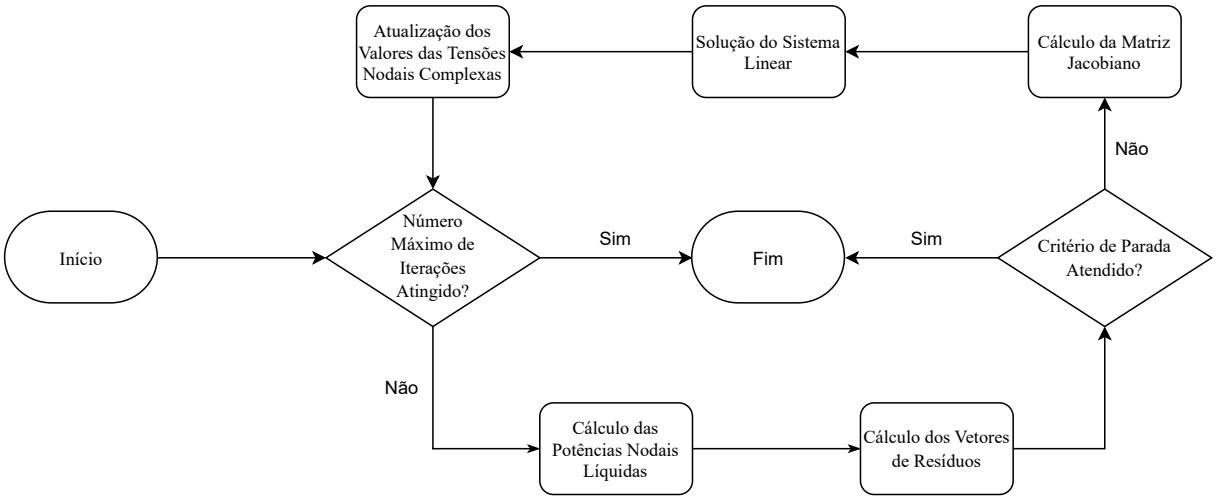

Figura 5.2: Fluxograma simplificado do laço iterativo executado no bloco "Método de Newton-Raphson" na Figura 5.1.

5.2.1 Arquivo de Entrada e Estruturas de Dados

Antes que as rotinas de cálculo possam ser executadas, faz-se necessária leitura dos dados do sistema sob análise. O formato do arquivo escolhido para armazenamento desses dados foi o *Common Document File*, ou CDF, que é um formato desenvolvido por um grupo de trabalho do IEEE para troca de dados do PFP [26].

Primeiramente, são lidos numero de barras (assim como seus tipos) e ramos da rede

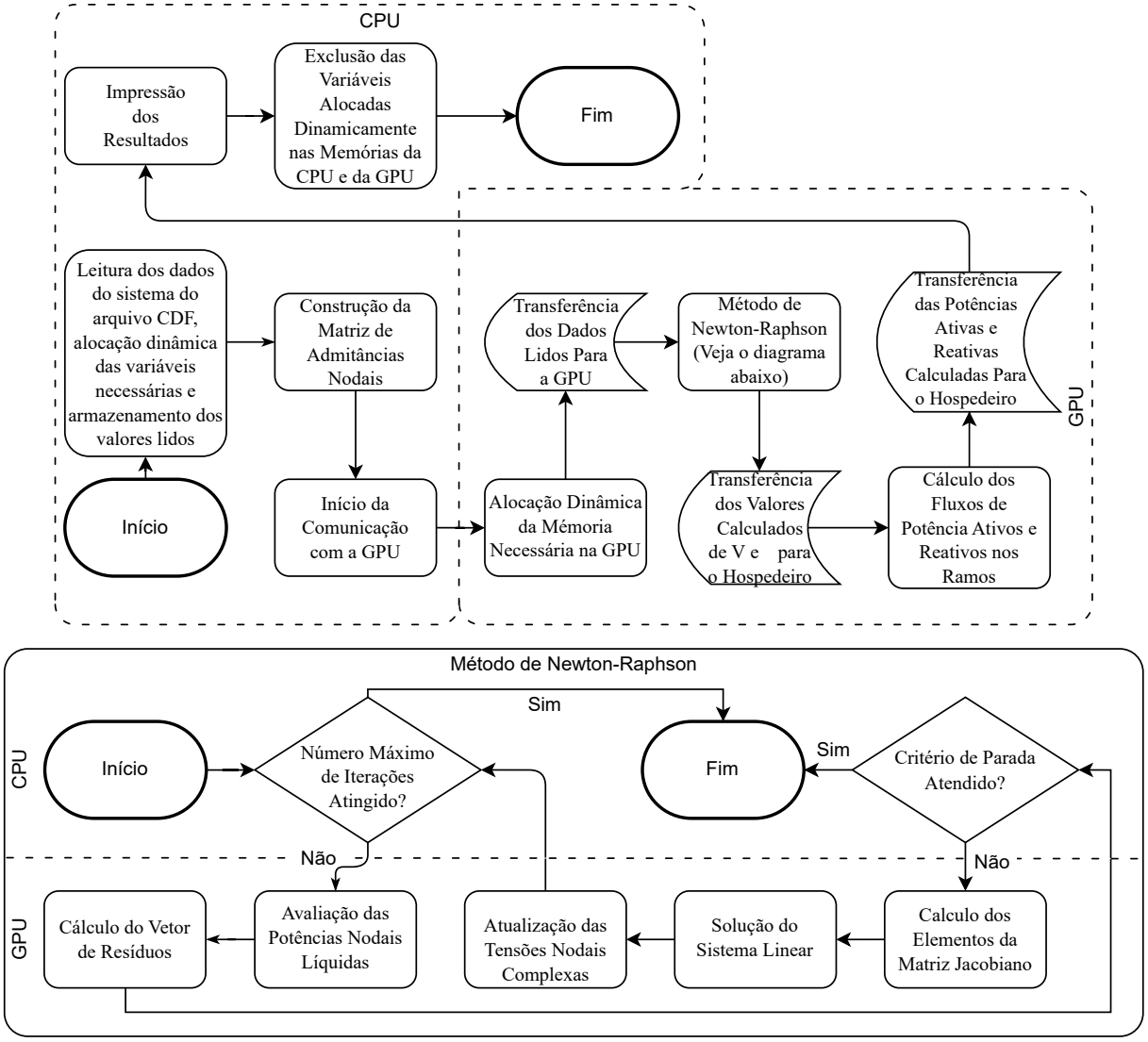

Figura 5.3: Fluxograma do processo utilizado para solução do *problema do Fluxo de Potência* na GPU. Fonte: Autor.

e, em seguida, é alocada memória para que possam ser armazenados os dados necessários. As principais variáveis utilizadas podem ser subdivididas de acordo com sua natureza. Elas são relativas a barras, a ramos, a todo o sistema ou ao método iterativo. As três últimas subdivisões estão representadas no diagrama da Figura 5.4 e a primeira na Figura 5.5. Enquanto valores que são lidos do arquivo de entrada estão organizados na divisão superior de cada uma das caixas desses diagramas, valores calculados durante a execução do programa estão na divisão intermediária. Cada uma das linhas contém breve descrição de uma variável alocada, seguida de seu tipo e dimensão. O número de posições de memória necessárias para armazenamento de valores inteiros e de ponto flutuante estão organizados na subdivisão inferior. Nesses diagramas nB, nL, $nnzY$ e $nnzJ$ são, respectivamente os números de barras, ramos, elementos não nulos da matriz de admitâncias

nodais e elementos não nulos da matriz jacobiano.

Alocação Dinâmica de Memória e Transferências entre *Host* e *Device*

Nas rotinas escritas em linguagens C++ e CUDA C++, as principais variáveis utilizadas na memória principal do computador são alocadas com o uso do comando `malloc` e todas as alocações dinâmicas de variáveis na memória global da GPU são feitas com o uso do comando `cudaMalloc`. A maior parte das transferências de memória foi feita com o uso da função `cudaMemcpy`. No caso das respostas intermediárias do problema, foi utilizada transferência assíncrona [53], por meio da rotina `cudaMemcpyAsync`. Desse forma, podem-se transferir dados entre as memórias do *host* e do *device* enquanto *kernels* são computados.

Armazenamento de Matrizes Esparsas

Matrizes que possuem grande parte das entradas nulas são denominadas *matrizes esparsas* [66]. Para que seja tirada vantagem dessa característica, faz-se necessária a adoção de uma forma própria para armazenar seus elementos. Essas técnicas evitam desperdício de memória por evitar o armazenamento de elementos nulos. Nas rotinas esparsas apresentadas nesse trabalho, será feito uso das duas formas de armazenamento descritas a seguir. Esse processo se torna mais claro com a apresentação de exemplos. Para essa finalidade, considere a matriz A. Sua forma de armazenamento *densa*, onde uma posição de memória é alocada para cada um de seus elementos, está representada em (5.1):

$$A = \begin{bmatrix} 7.5 & 6.2 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.1 & 2.0 & 0.0 & 0.0 \\ 4.0 & 0.0 & 0.0 & 1.9 & 9.4 \\ 0.0 & 0.0 & 8.3 & 0.0 & 5.0 \end{bmatrix} \tag{5.1}$$

Lista de Coordenadas (COO) [66]**.** Esse formato armazena a listagem dos elementos não nulos de uma matriz em conjunto com suas coordenadas de posição. Isso é feito com uso de três vetores: um contendo os valores não nulos da matriz, denominado *val*; outro cuja i-ésima entrada conterá o número da linha da i-ésima entrada da matriz, denominado *indiceDaLinha*; o terceiro arquiva na i-ésima entrada o número da coluna da i-ésima entrada da matriz e é denominado *indiceDaColuna*. A matriz A é escrita na forma COO do seguinte modo:

Dados do Sistema
Número de Barras PQ, *nPQ* (inteiro, *1*)
Lista com Índices das Barras PQ (inteiro, *nPQ*)
Número de Barras PV, *nPV* (inteiro, *1*)
Lista com Índices das Barras PV (inteiro, *nPV*)
Base de Potência Por Unidade (ponto flutuante, *1*)
Matriz de Admitâncias Nodais: • Métodos densos (ponto flutuante Complexo, *nB* × *nB*) Métodos Esparsos (forma de armazenamento CSR e COO): • Número de Elementos não Nulos de Y, *nnzY* (inteiro, *1*) • Vetor de Elementos não Nulos de Y (ponto flutuante complexo, *nnzY*) • Vetor de Ponteiros para Linhas (inteiro, *nB+1*) • Vetor de Índices das Linhas dos Elementos (inteiro, *nnzY*) • Vetor de Índices das Colunas dos Elementos (inteiro, *nnzY*)
Métodos Densos: • Posições inteiras: $2 + nPQ + nPV$ • Posições de ponto flutuante: $1 + 2 \cdot nB^2$ Métodos Esparsos: • Posições inteiras: $2 + nPQ + nPV + 4 \cdot nL + 3 \cdot nB + 2$ • Posições de ponto flutuante: $1 + 4 \cdot nL + 2 \cdot nB + 1$

Dados de Ramo
Admitância série (ponto flutuante, *nL*)
Susceptância shunt (ponto flutuante, *nL*)
Módulo da relação de transformação de tensão (ponto flutuante, *nL*)
Ângulo da relação de transformação de tensão (ponto flutuante, *nL*)
Barra de partida (inteiro, *nL*)
Barra de chegada (inteiro, *nL*)
Fluxos de potência: Pdp (ponto flutuante, *nL*) Ppd (ponto flutuante, *nL*) Qpd (ponto flutuante, *nL*) Qdp (ponto flutuante, *nL*)
Métodos Densos e Esparsos: • Posições inteiras: $2 \cdot nL$ • Posições de ponto flutuante: $8 \cdot nL$

Dados do Método Iterativo
Estrutura sem leitura de dados
Número da Iteração Atual (inteiro, *1*) Potência Ativa Calculada (ponto flutuante, *nB*) Potência Reativa Calculada (ponto flutuante, *nB*) Vetor de Resíduos/Correção (ponto flutuante, *nPV+nPQ•2*) **Matriz Jacobiano:** • Métodos Densos (ponto flutuante, *(nPV+nPQ•2)×(nPV+nPQ•2)*) Métodos Esparsos (forma de armazenamento CSR e COO): • Número de Elementos não Nulos de J, *nnzJ* (inteiro, *1*) • Vetor de Elementos não Nulos de J (ponto flutuante, *nnzJ*) • Vetor de Ponteiros para Linhas (inteiro, *nPV+nPQ•2+1*) • Vetor de Índices das Colunas dos Elementos (inteiro, *nnzJ*) • Vetor de Índices das Linhas dos Elementos (inteiro, *nnzJ*) • Número de Elementos não Nulos de H, *nnzH* (inteiro, *1*) • Vetor com as Posições do Vetor de Elementos não Nulos de J ocupadas por Elementos de H (inteiro, *nnzH*) • Número de Elementos não Nulos de L, *nnzL* (inteiro, *1*) • Vetor com as Posições do Vetor de Elementos não Nulos de J ocupadas por Elementos de L (inteiro, *nnzL*) • Número de Elementos não Nulos de M, *nnzM* (inteiro, *1*) • Vetor com as Posições do Vetor de Elementos não Nulos de J ocupadas por Elementos de M (inteiro, *nnzM*) • Número de Elementos não Nulos de N, *nnzN* (inteiro, *1*) • Vetor com as Posições do Vetor de Elementos não Nulos de J ocupadas por Elementos de N (inteiro, *nnzN*)
Métodos Densos: • Posições inteiras: 1 • Posições de ponto flutuante: $2 \cdot nB + nPV + 2 \cdot nPQ + (nPV + 2 \cdot nPQ)^2$ Métodos Esparsos: • Posições inteiras: $7 + nPV + 2 \cdot nPQ + 3 \cdot nnzJ$ • Posições de ponto flutuante: $2 \cdot nB + nPV + 2 \cdot nPQ + nnzJ$

Figura 5.4: Principais variáveis relacionadas aos ramos, à organização do SEP e ao método iterativo.

Dados de Barra
Identificador (inteiro, *nB*)
Magnitudes de Tensão Controladas (ponto flutuante, *nB*)
Potência Ativa Gerada (ponto flutuante, *nB*)
Potência Reativa Gerada (ponto flutuante, *nB*)
Potência Ativa Consumida (ponto flutuante, *nB*)
Potência Reativa Consumida (ponto flutuante, *nB*)
Potência Ativa Líquida (ponto flutuante, *nB*)
Potência Reativa Líquida (ponto flutuante, *nB*)
Base de Tensão Por Unidade (ponto flutuante, *nB*)
Susceptância Shunt (ponto flutuante, *nB*)
Condutância Shunt (ponto flutuante, *nB*)
Ponteiros para Barras PQ (inteiro, *nB*)
Magnitude da Tensão (ponto flutuante, *nB*)
Ângulo de Fase da Tensão (ponto flutuante, *nB*)
Métodos Densos e Esparsos: • Posições inteiras: $2 \cdot nB$ • Posições de ponto flutuante: $12 \cdot nB$

Figura 5.5: Principais variáveis relacionadas às barras do SEP.

$$
\begin{aligned}
valA &= \begin{bmatrix} 7.5 & 6.2 & 3.1 & 2.0 & 4.0 & 1.9 & 9.4 & 8.3 & 5.0 \end{bmatrix} \\
indiceDaLinhaA &= \begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 3 & 3 & 4 & 4 \end{bmatrix} \\
indiceDaColunaA &= \begin{bmatrix} 1 & 2 & 2 & 3 & 1 & 4 & 5 & 3 & 5 \end{bmatrix}
\end{aligned} \tag{5.2}
$$

A ordem na qual os elementos da matriz A são armazenados nas listas é arbitrária. Contudo, é muito comum organizá-los de forma *orientada a linha* (i.e., elementos das linhas com menor índice sempre são listados antes e elementos de uma mesma linha são ordenados de forma que a numeração de suas colunas seja crescente), como foi feito no exemplo. Esse formato é vantajoso pois permite que os elementos dessa matriz sejam diretamente percorridos (i.e., acesso direto ao n-ésimo elemento não nulo e suas coordenadas). Contudo, caso seja necessário acesso ao elemento de coordenadas (i, j), é preciso que os vetores de coordenadas sejam precorridos para determinar a posição desse elemento no vetor de valores val ou se ele é igual a zero. Dessa forma, acessar o elemento com coordenadas (i, j) em matriz armazenada nesse formato possui complexidade assintótica de tempo $O(nnz)$, onde nnz é o número de entradas não nulas da matriz.

Linha Esparsa Comprimida (CSR) [66]**.** Esse formato é semelhante ao COO com ordem *orientada a linha*. A diferença entre eles é que o CSR armazena, no lugar da lista

de índices de linhas, um vetor (denominado $vetorDePonteirosParaLinhas$) cuja i-ésima entrada é a posição do primeiro elemento da i-ésima linha da matriz original no vetor de valores (val). A matriz A é escrita na forma CSR do seguinte modo:

$$\begin{aligned} valA &= \begin{bmatrix} 7.5 & 6.2 & 3.1 & 2.0 & 4.0 & 1.9 & 9.4 & 8.3 & 5.0 \end{bmatrix} \\ vetorDePonteirosParaLinhasA &= \begin{bmatrix} 1 & 3 & 5 & 8 & 10 \end{bmatrix} \\ indiceDaColunaA &= \begin{bmatrix} 1 & 2 & 2 & 3 & 1 & 4 & 5 & 3 & 5 \end{bmatrix} \end{aligned} \tag{5.3}$$

A vantagem dessa forma de armazenamento está relacionada ao fato de que o acesso ao elemento de coordenadas (i, j) exige que seja percorrido apenas o trecho do vetor de coordenadas referente à linha i. Esse processo apresenta complexidade de tempo $O(log(\Delta))$, onde Δ é o maior número de elementos não nulos que uma linha da matriz possui. Contudo, acesso às coordenadas do n-ésimo elemento não nulo exige que o vetor de ponteiros para as linhas seja percorrido. Isso dá complexidade de tempo $O(log(nLinhas))$ para essa operação, onde $nLinhas$ é o número de linhas da matriz.

Conforme mencionado na Seção 2.5, as matrizes de admitâncias nodais e Jacobiano do problema estudado possuem característica esparsa relevante. A primeira precisará ser percorrida na etapa de cálculo da matriz Jacobiano descrita na Subseção 5.2.5, ter linhas específicas percorridas na etapa de cálculo das potências líquidas (descrita na Subseção 5.2.3) e ter elementos acessados pelas coordenadas na etapa de cálculo dos elementos da matriz Jacobiano (descrita na Subseção 5.2.5). A segunda precisará ser percorrida para cálculo de seus elementos (descrito na Subseção 5.2.5) e será utilizada para a solução do sistema no formato CSR (abordado na Subseção 5.2.6). Nesse sentido, se mostra conveniente a utilização dos vetores de índices dos formatos COO e CSR para o armazenamento dessas duas matrizes.

5.2.2 Construção da Matriz de Admitâncias Nodais

Após a leitura dos dados de entrada e respectivo armazenamento na memória principal do computador, a matriz de admitâncias nodais deve ser calculada. Para que isso seja feito, (2.17) e (2.18) foram reescritas de forma algorítmica. O Algoritmo 1 organiza a forma densa de construção dessa matriz. Essa rotina apresenta limite de tempo assintótico superior $O(nB + nL)$. Vale salientar que a forma de manejo, na implementação densa, da matriz com nB^2 posições pode impactar esse resultado.

Algoritmo 1 Cálculo da Matriz de Admitâncias Nodais Densa

Entrada: Número de ramos do SEP sob análise nL, vetor de barras de partida dos ramos $\boldsymbol{de}$, vetor de barras de chegada dos ramos $\boldsymbol{para}$, vetor de admitâncias dos ramos $\boldsymbol{y}$, vetor de relações de transformação de tensão complexas dos ramos $\boldsymbol{t}$, vetores de susceptâncias *shunt* de barras $\boldsymbol{b_b^{sh}}$ e de ramos $\boldsymbol{b_r^{sh}}$ e número de barras do SEP sob análise nB.

Saida: Matriz de admitâncias nodais Y.

1: **função** CalcAdmitância(nL, $\boldsymbol{de}$, $\boldsymbol{para}$, $\boldsymbol{y}$, $\boldsymbol{t}$, $\boldsymbol{b_b^{sh}}$, $\boldsymbol{b_r^{sh}}$, nB)
2: **para** $i \leftarrow 1$ **até** nL **faça**
3: $Y_{\boldsymbol{de}[i],\ \boldsymbol{para}[i]} \leftarrow Y_{\boldsymbol{de}[i],\ \boldsymbol{para}[i]} - (\boldsymbol{t}[i])^* \cdot \boldsymbol{y}[i]$
4: $Y_{\boldsymbol{para}[i],\ \boldsymbol{de}[i]} \leftarrow Y_{\boldsymbol{para}[i],\ \boldsymbol{de}[i]} - (\boldsymbol{t}[i]) \cdot \boldsymbol{y}[i]$
5: $Y_{\boldsymbol{de}[i],\ \boldsymbol{de}[i]} \leftarrow |\boldsymbol{t}[i]|^2 \cdot \boldsymbol{y}[i] + j \cdot \boldsymbol{b_r^{sh}}[i]$
6: $Y_{\boldsymbol{para}[i],\ \boldsymbol{para}[i]} \leftarrow \boldsymbol{y}[i] + j \cdot \boldsymbol{b_r^{sh}}[i]$
7: **fim para**
8: **para** $i \leftarrow 1$ **até** nB **faça**
9: $Y_{i,\ i} \leftarrow Y_{i,\ i} + j \cdot \boldsymbol{b_b^{sh}}[i]$
10: **fim para**
11: **retorna** Y
12: **fim função**

O Algoritmo 2 organiza a forma esparsa de construção da matriz Y. A $listaY$ tem tamanho previamente conhecido, por isso podem ser acrescentados elementos a ela sem que seja necessária realocação de memória (i.e., a operação possui complexidade de tempo constante). Além disso, as rotinas criaMatrizEsparsa e criaVetorDeIndicesCOO possuem limite assintótico superior de tempo $O(nnzY) = O(nB + nL)$, pelos mesmos motivos do caso anterior. $nnzY = nB + 2 \cdot nL$ é o número de elementos não nulos da matriz Y.

As estruturas de dados utilizadas para armazenamento dos valores dos elementos da matriz de admitâncias nodais o fazem ao armazenar parte real e imaginária de cada número complexo em posições consecutivas. Isso propicia a coalescência de dados acessados uma vez que são calculados de forma conjunta e, como será visto nas próximas subseções, é comum que tais valores sejam usados também em conjunto.

5.2.3 Cálculo da Injeção de Potência Nodal

O Algoritmo 3 calcula a injeção de potência ativa na k-ésima barra, conforme indica (2.22). Procedimento análogo também pode ser escrito para injeção de potência reativa descrita em (2.23). Na implementação *densa* do algoritmo, o conjunto de barras vizinhas K considerado contém todas as barras do SEP sob análise, enquanto, na implementação esparsa, o conjunto de barras vizinhas contém todas as barras da k-ésima linha da matriz de admitâncias nodais, armazenada na forma CSR (i.e., cujas colunas estão relacionadas

Algoritmo 2 Cálculo da Matriz de Admitâncias Nodais Esparsa

Entrada: Número de barras nB e ramos nL do SEP sob análise, vetor de barras de partida dos ramos $\boldsymbol{de}$, vetor de barras de chegada dos ramos $\boldsymbol{para}$, vetor de admitâncias dos ramos $\boldsymbol{y}$, vetor de relações de transformação de tensão complexas dos ramos $\boldsymbol{t}$ e os vetores de susceptâncias *shunt* de barras $\boldsymbol{b_b^{sh}}$ e de ramos $\boldsymbol{b_r^{sh}}$.

Saida: Matriz de admitâncias nodais $Y_{CSR+COO}$.

1: **função** CalcAdmitância(nB, nL, $\boldsymbol{de}$, $\boldsymbol{para}$, $\boldsymbol{y}$, $\boldsymbol{t}$, $\boldsymbol{b_b^{sh}}$, $\boldsymbol{b_r^{sh}}$)
2: **para** $i \leftarrow 1$ **até** nL **faça** // *inclusão do ramo i à lista de elementos listaY*
3: $listaY$.push($\boldsymbol{de}[i]$, $\boldsymbol{para}[i]$, $-(\boldsymbol{t}[i])^* \cdot \boldsymbol{y}[i]$)
4: $listaY$.push($\boldsymbol{para}[i]$, $\boldsymbol{de}[i]$, $-(\boldsymbol{t}[i]) \cdot \boldsymbol{y}[i]$)
5: $listaY$.push($\boldsymbol{de}[i]$, $\boldsymbol{de}[i]$, $|\boldsymbol{t}[i]|^2 \cdot \boldsymbol{y}[i] + j \cdot \boldsymbol{b_r^{sh}}[i]$
6: $listaY$.push($\boldsymbol{para}[i]$, $\boldsymbol{para}[i]$, $\boldsymbol{y}[i] + j \cdot \boldsymbol{b_r^{sh}}[i]$)
7: **fim para**
8: **para** $i \leftarrow 1$ **até** nB **faça**
9: // *inclusão da susceptância shunt da barra i*
10: $listaY$.push(i, i, $Y_{i,\,i} + j \cdot \boldsymbol{b_b^{sh}}[i]$)
11: **fim para**
12: // *ordena elementos de Y, soma entradas com mesmos*
13: // *índices e escreve no formato CSR e COO*
14: $Y_{CSR} \leftarrow$criaMatrizEsparsa(Y)
15: $Y_{CSR+COO} \leftarrow$ criaVetorDeIndicesCOO(Y_{CSR})
16: **retorna** $Y_{CSR+COO}$
17: **fim função**

às barras que estão realmente ligadas à barra por meio de um ramo). As rotinas das versões paralelas densa e esparsa que são executadas na CPU avaliam a função CalcP para várias barras em concomitância. Enquanto o cálculo das potências líquidas nas nB barras por meio das rotinas densas possuem complexidade assintótica $O\left(\left\lceil \frac{nB}{nT} \right\rceil \cdot nB\right)$, esse cálculo por meio das funções esparsas possui complexidade $O\left(\left\lceil \frac{nB}{nT} \right\rceil \cdot \Delta\right)$, onde nT é o número de linhas de execução concomitantes presentes durante o processamento e Δ é o maior número de elementos não nulos que uma linha que a matriz de admitâncias nodais possui.

O *kernel* denso consiste no lançamento de nB threads para a redução do laço **para** da Linha 3. Isso é feito com o uso de *memória compartilhada* da GPU para obtenção de melhor desempenho. Cada *thread* do *kernel* que implementa o pseudocódigo do Algoritmo 3 com abordagem esparsa se torna responsável pelo cálculo da potência líquida ativa ou reativa em uma barra do sistema (i.e., cada *thread* faz uma chamada da função CalcP). Esse tipo de abordagem se beneficia da adoção de forma de armazenamento CSR da *matriz de admitâncias nodais*, uma vez que o único loop **para** do pseudocódigo dessa rotina pode ser escrito como o percorrimento dos valores não nulos da k-ésima linha da matriz Y. Vale ressaltar que as rotinas de cálculo de potência ativa e reativa líquidas podem,

Algoritmo 3 Cálculo da injeção de potência ativa P_k na barra k

Entrada: Barra k, conjunto de suas barras vizinhas K, vetor de magnitudes das tensões nodais complexas $\boldsymbol{V}$, vetor dos argumentos das tesões nodais complexas $\boldsymbol{\theta}$ e partes real G e imaginária B da matriz de admitâncias nodais.

Saida: Valor da injeção de potência ativa na barra k.

1: **função** CALCP(k, K, $\boldsymbol{V}$, $\boldsymbol{\theta}$)
2: $acc \leftarrow 0$
3: **para todo** $m \in K$ **faça**
4: $\theta_{km} \leftarrow \boldsymbol{\theta}[k] - \boldsymbol{\theta}[m]$
5: $acc \leftarrow acc + \boldsymbol{V}[m](G_{km} \cdot cos\theta_{km} + B_{km} \cdot sen\theta_{km})$
6: **fim para**
7: **retorna** $\boldsymbol{V}[k] \cdot acc$
8: **fim função**

potencialmente, ser executadas em concomitância. Isso pode ser indicado para compilador CUDA utilizado com a atribuição de *streams* diferentes para os dois *kernels*. Um aspecto importante da implementação proposta para os *kernels* esparsos é relativa à propriedade de que a divergência entre *threads* será tão pequena quanto forem as diferenças entre os graus dos nós do SEP (i.e., o número de nós diretamente conectados a um determinado nó, por meio de ramos). A presença de pequenos desvios entre os graus dos nós de um SEP é um padrão comum. Exemplificação dessa característica para o sistema IEEE118 (melhor apresentado na Seção 6.2) pode ser observada no gráfico da Figura 5.6. Nele estão organizados os percentuais das barras que possuem cada grau. Para esse caso, menos de 8% do total de barras possui grau superior a 5. Essa característica ampara satisfatória regularidade entre *threads* concorrentes, propriedade importante para a obtenção de bom desempenho.

As complexidades assintóticas do cálculo com os dois *kernels* são, respectivamente, $O\left(nB \cdot \left\lceil \frac{nB}{nT} \right\rceil + nB \cdot \log_2 nB\right)$ e $O\left(\left\lceil \frac{nB}{nT} \right\rceil \cdot \Delta\right)$, onde nB é o número de barras do sistema, nT é o número de linhas de execução concomitantes presentes durante o processamento e Δ é o maior número de elementos não nulos que uma linha da matriz possui.

5.2.4 Cálculo do Vetor de Resíduos

Nessa etapa, o vetor de resíduos deve ser calculado conforme a segunda equação de (2.34). Isso é feito realizando $2 \cdot nPQ + nPV$ subtrações, onde nPQ é o número de barras de tipo PQ e nPV é o número de barras de tipo PV do SEP sob análise. Como os cálculos são independentes, eles podem ser distribuídos pelos núcleos da CPU em abordagem paralela e *kernel* com essa finalidade pode ser construído ao lançar $2 \cdot nPQ + nPV$ *threads*. A

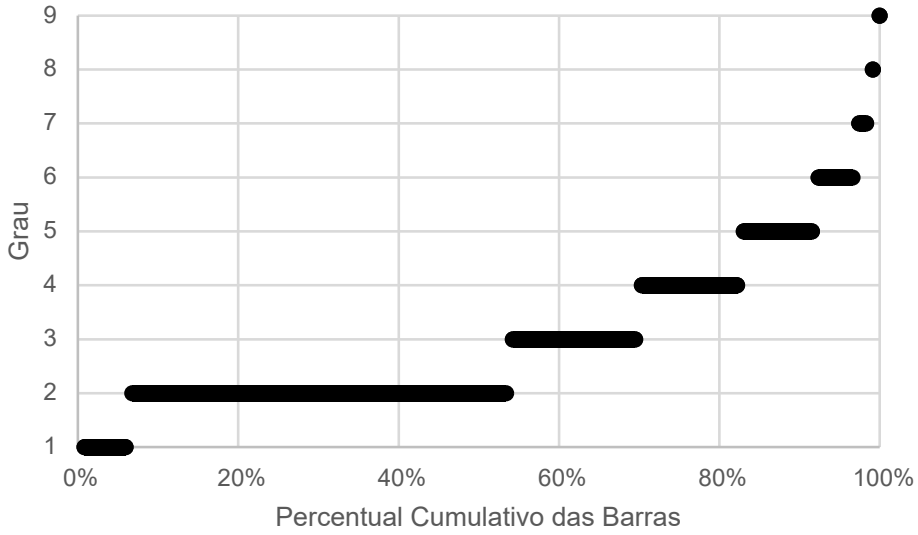

Figura 5.6: Percentual cumulativo do número de barras de IEEE118 segundo valor crescente de seus graus.

complexidade assintótica de tempo desse algoritmo é $O\left(\left\lceil \frac{nB}{nT} \right\rceil\right)$, onde nB é o número de barras do SEP e nT é o número de linhas de execução concomitantes presentes durante o processamento.

5.2.5 Cálculo da Matriz Jacobiano

O cálculo da matriz Jacobiano é feito conforme (2.36) e (2.37). Os Algoritmos 4 e 5 apresentam a forma como podem ser calculados os elementos das submatrizes H e M, respectivamente, usando abordagem densa. As submatrizes L e N podem ser calculadas de forma análoga. Versões paralelas desses algoritmos podem ser construídas executando-se as diferentes iterações do laço **para** mais interno das rotinas, na linha 6 de DNCALCH e na linha 8 de DNCALCM, em concomitância. *Kernel* também pode ser implementado lançando um *thread* para cada iteração desses laços **para** (i.e., um *thread* para cada elemento que a submatriz possuir). A *divergência* dos *threads* de *kernels* construídos dessa forma é minimizada uma vez que as expressões booleanas das linhas 3 e 7 do Algoritmo 4 e da linha 6 do Algoritmo 5 evitam desvios condicionais. Dessa forma, as complexidades de tempo das rotinas que calculam as submatrizes H, L, M e N serão proporcionais às suas dimensões, respectivamente: $O\left(\left\lceil \frac{(nPQ+nPV)^2}{nT} \right\rceil\right)$, $O\left(\left\lceil \frac{nPQ^2}{nT} \right\rceil\right)$, $O\left(\left\lceil \frac{(nPQ+nPV)\cdot nPQ}{nT} \right\rceil\right)$ e $O\left(\left\lceil \frac{nPQ\cdot(nPQ+nPV)}{nT} \right\rceil\right)$, onde nPQ é o número de barras de tipo PQ, nPV é o número de barras de tipo PV do modelo do SEP sob análise e nT é o número de linhas de execução concomitantes presentes durante o processamento.

Algoritmo 4 Cálculo da submatriz H densa

Entrada: Ponteiro para posição de memória onde deverá ser armazenada a matriz Jacobiano J. Vetor de magnitudes das tensões nodais complexas $\boldsymbol{V}$, vetor dos argumentos das tesões nodais complexas $\boldsymbol{\theta}$, vetor de injeções de potência reativa calculadas $\boldsymbol{Qcalc}$, número de barras de tipo PQ nPQ, número de barras de tipo PV nPV e número da barra $V\theta$ $barraV\theta$.

Saida: Elementos da submatriz H nas devidas posições da matriz jacobiano em J.

1: **função** DNCALCH($\boldsymbol{V}$, $\boldsymbol{\theta}$, $\boldsymbol{P}$, nPQ, nPV, $barraV\theta$)
2: **para** idx $\leftarrow 1$ **até** $nPQ + nPV$ **faça**
3: $offm \leftarrow idx \geq barraV\theta$ // *variável booleana igual a 1 se a barra idx possui*
4: // *número maior que a swing.*
5: $idx \leftarrow idx + offm$
6: **para** $idy \leftarrow 1$ **até** $nPQ + nPV$ **faça**
7: $offk \leftarrow idy \geq barraV\theta$ // *variável booleana igual a 1 se a barra idy possui*
8: // *número maior que a swing.*
9: $idy \leftarrow idy + offk$
10: **se** $idx = idy$ **então**
11: $\theta_{km} \leftarrow \boldsymbol{\theta}[k] - \boldsymbol{\theta}[m]$
12: $J_{idy-offk,\ idx-offm} \leftarrow \boldsymbol{V}[idy] \cdot \boldsymbol{V}[idx] \cdot Real(Y_{idy,\ idx}) \cdot sen(\theta_{km})$
13: $-Imaginario(Y_{idy,\ idx}) \cdot cos(\theta_{km})$
14: **senão**
15: $J_{idy-offk,\ idx-offm} \leftarrow -\boldsymbol{Qcalc}[idx] - \boldsymbol{V}[idx] \cdot \boldsymbol{V}[idx] \cdot$
16: $Imaginario(Y_{idx,\ idx})$
17: **fim se**
18: **fim para**
19: **fim para**
20: **fim função**

A rotina esparsa para cálculo da matriz Jacobiano se baseia na propriedade da sua lei de formação destacada na Seção 2.5: caso um elemento da matriz de admitâncias nodais $Y_{k,m} = G_{k,m} + j \cdot B_{k,m}$ (onde $j = \sqrt{-1}$ é a unidade imaginária e $k \neq m$) seja nulo, pode-se depreender de (2.37) que possível elemento relacionado aos índices k e m de cada uma das submatrizes também será nulo.

Nesse sentido, a execução de rotina esparsa com essa finalidade exige que a matriz de admitâncias nodais seja percorrida para que se determine os padrões de preenchimento das submatrizes do Jacobiano. Assim que a existência de dado elemento não nulo é determinada, os devidos cálculos podem ser efetuados. Esse raciocínio foi expressado no Algoritmo 6. Desse modo, esse algoritmo configura complexidade assintótica de tempo $O(nnzY)$, onde $nnzY$ é o número de elementos não nulos da matriz de admitâncias nodais. Conforme visto na Subseção 5.2.2, $nnzY = 2 \cdot nL + nB$. Assim, pode-se reescrever essa complexidade como $O(nB + nL)$.

Algoritmo 5 Cálculo da submatriz M densa

Entrada: Vetor de magnitudes das tensões nodais complexas $\boldsymbol{V}$, vetor dos argumentos das tesões nodais complexas $\boldsymbol{\theta}$, vetor de injeções de potência ativa calculadas $\boldsymbol{Pcalc}$, vetor que lista em ordem crescente as barras de tipo PQ $\boldsymbol{barrasPQ}$, número de barras de tipo PQ nPQ, número de barras de tipo PV nPV e número da barra $V\theta$ $barraV\theta$.
Saida: Submatriz M.

1: **função** DNCALCM(k, $\boldsymbol{V}$, $\boldsymbol{\theta}$, $\boldsymbol{Pcalc}$, nPQ, nPV)
2: **para** idx $\leftarrow 1$ **até** $nPQ + nPV$ **faça**
3: $offm \leftarrow idx \geq barraV\theta$ // *variável booleana igual a 1 se a barra idx*
4: // *possui número maior que a swing,*
5: // *0 caso contrário.*
6: $idx \leftarrow idx + offm$
7: $deslocamento \leftarrow nPQ + nPV$
8: **para** $idy \leftarrow 1$ **até** nPQ **faça**
9: **se** $idx = \boldsymbol{barrasPQ}[idy]$ **então**
10: $\theta_{km} \leftarrow \boldsymbol{\theta}[\boldsymbol{barrasPQ}[idy]] - \boldsymbol{\theta}[m]$
11: $J_{idy-deslocamento,\ idx-offm} \leftarrow -\boldsymbol{V}[barrasPQlim[idy]] \cdot \boldsymbol{V}[idx] \cdot (real(Y[barrasPQlim[idy], idx]) \cdot cos(aux) + imaginario(Y[barrasPQlim[idy], idx]) \cdot sin(aux))$
12: **senão**
13: $J_{idy+deslocamento,\ idx-offm} \leftarrow \boldsymbol{Pcalc}[idx] - \boldsymbol{V}[idx] \cdot \boldsymbol{V}[idx] \cdot$
14: $real(Y[idx, idx])$
15: **fim se**
16: **fim para**
17: **fim para**
18: **retorna** J
19: **fim função**

É muito importante que, quando implementada, a forma de percorrer os elementos de Y seja escolhida com cautela para que a complexidade do algoritmo não seja afetada. Enquanto o acesso de linhas relativas a barras PQ foi feito com o uso de listagem das barras com esse tipo e vetor de ponteiros do formato CSR de Y, a determinação se a coluna de um elemento de Y possui índice relacionado a barra PQ foi feita em tempo constante ao utilizar-se um vetor auxiliar. Se a i-ésima barra do sistema é a j-ésima barra de tipo PQ, a i-ésima entrada desse vetor é igual a j. Caso contrário, essa entrada será nula. Esse vetor auxiliar foi denominado *ponteiro para barras PQ* e também é utilizado para a determinação das coordenadas do elemento calculado na matriz jacobiano. Vale mencionar que o ponteiro para barras PQ pode ser criado em tempo linear.

Para que possa ser paralelizado, esse método deve ser subdividido em duas etapas. A primeira consiste na *análise simbólica da matriz de admitâncias nodais*. Durante esse processo, a matriz Y é percorrida de forma análoga ao feito no procedimento sequencial para que sejam construídas principalmente três tipos de estruturas: vetores com a in-

Algoritmo 6 Construção da matriz Jacobiano Esparsa

Entrada: Vetor de magnitudes das tensões nodais complexas $\boldsymbol{V}$, vetor dos argumentos das tesões nodais complexas $\boldsymbol{\theta}$, vetor de injeções de potência ativa $\boldsymbol{Pcalc}$, vetor de injeções de potência reativa $\boldsymbol{Qcalc}$, vetor que lista em ordem crescente as barras de tipo PQ $\boldsymbol{barrasPQ}$, vetor que lista em ordem crescente as barras de tipo PV $\boldsymbol{barrasPV}$, e número da barra $V\theta$ $barraV\theta$.

Saida: Matriz Jacobiano J nos formatos COO e CSR.

1: **função** SPJACOBIANO(Y, $\boldsymbol{barrasPV}$, $\boldsymbol{barrasPQ}$)
2: **para toda** linha i de Y relacionada a uma barra PQ ou PV **faça**
3: **para cada** elemento não nulo da linha i cuja coluna j está relacionada a uma barra PQ ou PV **faça**
4: *// Elemento da submatriz H*
5: adicione as coordenadas do novo elemento à listagem da matriz J
6: calcule valor segundo (2.37) e adicione à listagem.
7: **fim para**
8: **para cada** elemento não nulo da linha i cuja coluna j está relacionada à k-ésima barra PQ **faça**
9: *// Elemento da submatriz N*
10: adicione as coordenadas do novo elemento à listagem da matriz J
11: calcule valor segundo (2.37) e adicione à listagem.
12: **fim para**
13: **fim para**
14: **para cada** linha i de Y relacionada à k-ésima barra PQ **faça**
15: **para cada** elemento não nulo da linha i cuja coluna j está relacionada a uma barra PQ ou PV **faça**
16: *// Elemento da submatriz M*
17: adicione as coordenadas do novo elemento à listagem da matriz J
18: calcule valor segundo (2.37) e adicione à listagem.
19: **fim para**
20: **para cada** elemento não nulo da linha i cuja coluna j está relacionada à l-ésima barra PQ **faça**
21: *// Elemento da submatriz L*
22: adicione as coordenadas do novo elemento à listagem da matriz J
23: calcule valor segundo (2.37) e adicione à listagem.
24: **fim para**
25: **fim para**
26: constrói vetores de posições COO e ponteiros de linhas CSR de J
27: **fim função**

dexação dos elementos da matriz Jacobiano nos formatos COO e CSR, vetores com as indexações de cada uma das submatrizes no formato COO e vetores contendo as posições dos elementos de cada uma das submatrizes no vetor de entradas não nulas da matriz Jacobiano. Esses últimos vetores serão utilizados como *listas de trabalho* para os *kernels* que serão descritos a seguir. A segunda subdivisão compreende a *etapa numérica* na qual são calculados os elementos não nulos. Foram criados quatro *kernels*, um para o

cálculo dos elementos de cada uma das submatrizes. Um thread é lançado para calcular cada elemento não nulo. A ordem em que os threads de cada um desses *kernels* calcula esses elementos é ditada pela respectiva *lista de trabalho*. Para que sejam minimizadas divergências, os primeiros elementos das *listas de trabalho* compreendem componentes das submatrizes com os dois índices iguais (cuja fórmula de cálculo é diferente, segundo (2.37)). A forma como foram organizados esses *kernels* é exemplificada para a submatriz L no Algoritmo 7. Algoritmos para cálculo dos elementos não nulos de H, M e N podem ser escritos de forma análoga. Esses *kernels* também podem ser executados em concomitância. Mais uma vez, isso pode ser formalizado via adoção de *stream* CUDA diferente para cada *kernel*. As duas etapas de cálculo do Jacobiano paralelo apresentam complexidade de tempo $O\left(nnzJ + \left\lceil \frac{nnzJ}{nT} \right\rceil \cdot log_2(\Delta)\right)$, onde $nnzJ$ é o número de elementos não nulos da matriz jacobiano e nT é o número de linhas de execução concomitantes presentes durante o processamento.

É importante salientar que, após execução da rotina de análise simbólica da matriz Jacobiano, essa etapa só precisará ser executada em outra iteração se os tipos das barras do SEP forem alterados pelo efeito de algum ajuste de controle ou violação de limites físicos (e.g., violação do limite de injeção de potência reativa de qualquer uma das unidades geradoras do sistema).

Algoritmo 7 Cálculo da submatriz L esparsa

Entrada: Ponteiro para posição de memória onde deverão ser armazenados os elementos não nulos da matriz Jacobiano J. Vetor de magnitudes das tensões nodais complexas $\boldsymbol{V}$, vetor dos argumentos das tesões nodais complexas $\boldsymbol{\theta}$, vetor de injeções de potência ativa $\boldsymbol{P}$, lista de posições da submatriz L $\boldsymbol{Lpos}$, número de elementos não nulos da matriz L $nnzL$ e vetores de índices de linhas $\boldsymbol{cooRowIndSubMatJ}$ e colunas $\boldsymbol{cooColIndSubMatJ}$ das submatrizes.

Saida: Elementos da submatriz L nas devidas posições da matriz jacobiano em J.

1: **função** SPCALCL(J, $\boldsymbol{V}$, $\boldsymbol{\theta}$, $\boldsymbol{P}$, $\boldsymbol{Lpos}$, $nnzL$, $\boldsymbol{cooRowIndSubMatJ}$, $\boldsymbol{cooColIndSubMatJ}$)
2: **para** id $\leftarrow 1$ **até** $nnzL$ **faça**
3: $idx \leftarrow \boldsymbol{cooColIndSubMatJ}[\boldsymbol{Lpos}[id]]$
4: $idy \leftarrow \boldsymbol{cooRowIndSubMatJ}[\boldsymbol{Lpos}[id]]$
5: **se** $idx = idy$ **então**
6: $\theta_{km} \leftarrow \boldsymbol{\theta}[k] - \boldsymbol{\theta}[m]$
7: $J[Lpos[id]] \leftarrow \boldsymbol{V}[idy] \cdot (Real(Y_{idy,\ idx}) \cdot sen(\theta_{km})$
8: $-Imaginario(Y_{idy,\ idx}) \cdot cos(\theta_{km}))$
9: **senão**
10: $J[Lpos[id]] \leftarrow (Q[idx] - \boldsymbol{V}[idx] \cdot \boldsymbol{V}[idx] \cdot Imaginario(Y_{idx,\ idx}))\,/\boldsymbol{V}[idx]$
11: **fim se**
12: **fim para**
13: **fim função**

5.2.6 Solução do Sistema Linear

Para a execução dessa etapa foram empregadas bibliotecas de Álgebra Linear otimizadas. Quando executada na CPU, foram utilizadas rotinas da biblioteca Intel® MKL [78]. Métodos densos fazem uso de funções LAPACK e esparsos que utilizam rotinas PARDISO para solucionar o sistema linear via fatoração LU de forma sequencial e paralela.

Para execução na GPU foi adotada a biblioteca cuSolver [55], desenvolvida pela Nvidia. As rotinas densas escolhidas são da implementação LAPACK, análogas às chamadas na biblioteca MKL. Até o momento em que esse trabalho foi realizado (versão 11.2.0), não estão disponíveis rotinas esparsas na biblioteca cuSolver que fazem uso de decomposição LU para a solução de sistemas esparsos na GPU. Por esse motivo, foi chamada função com esse propósito que faz uso de decomposição QR.

5.2.7 Atualização das Tensões Nodais Complexas

Assim como no Cálculo dos Resíduos, essa etapa envolve $2 \cdot nPQ + nPV$ operações de adição. Os procedimentos, *kernels* e análise são análogos.

5.2.8 Cálculo dos Fluxos de Potência Ativos e Reativos nos Ramos

A última etapa da implementação proposta que envolve cálculo tem o objetivo de determinar a potência ativa e reativa que flui pelos ramos do sistema. Esse processo é realizado ao avaliar as fórmulas em (2.29). Vale ressaltar que, como existe consumo de potência interno nos ramos (representado nos modelos descritos na Seção 2.2 pelas suas admitâncias série e *shunt*), essas expressões devem ser avaliadas para os dois sentidos possíveis. Isso pode ser escrito para o fluxo de potência ativa conforme o Algoritmo 8. Algoritmo que calcula o fluxo de potência reativa pode ser escrito de forma análoga. Paralelismo pode ser obtido pela execução concomitante das iterações do laço **para** da linha 2. Desse modo, além da rotina paralela para execução na CPU, *kernel* pode ser criado com o lançamento de um thread para cada ramo do SEP. Além disso, o *kernel* que calcula o fluxo de potência ativa nos ramos é independente do que calcula o fluxo de potência reativa. Dessa forma eles podem ser executados em concomitância. Mais uma vez, isso pode ser informado ao compilador CUDA com o lançamento desses em dois *streams* diferentes. A principal distinção entre implementação densa e esparsa está no acesso ao elemento da matriz de admitâncias nodais: enquanto no primeiro caso pode ser feito de forma direta,

no segundo é necessário percorrer vetores da forma de armazenamento CSR. Nesse sentido, a complexidade assintótica de tempo para a forma densa dessa etapa é $O\left(\left\lceil \frac{nL}{Th} \right\rceil\right)$, enquanto para a forma esparsa é $O\left(\left\lceil \frac{nL}{Th} \right\rceil \cdot \Delta\right)$.

Algoritmo 8 Cálculo do Fluxo de Potência Ativa nos Ramos

Entrada: Ponteiros para posições de memória onde deverão ser armazenados os fluxos de potência ativa $\boldsymbol{P_{dp}}$ que flui no sentido em que o ramo foi definido e no sentido reverso $\boldsymbol{P_{pd}}$. Vetor de magnitudes das tensões nodais complexas $\boldsymbol{V}$, vetor dos argumentos das tesões nodais complexas $\boldsymbol{\theta}$, vetor de admitâncias série dos ramos $\boldsymbol{y}$, vetor de barras de partida dos ramos $\boldsymbol{de}$, vetor de barras de chegada dos ramos $\boldsymbol{para}$, vetor de relações de transformação de tensão complexas dos ramos $\boldsymbol{t}$, vetor de argumentos das relações de transformação de tensão dos ramos $\boldsymbol{\varphi}$ e número de ramos do SEP sob análise nL.

Saida: Fluxo de Potência nos Ramos em $\boldsymbol{P_{dp}}$ e $\boldsymbol{P_{pd}}$.

1: **função** CALCFLUXP($\boldsymbol{P_{dp}}$, $\boldsymbol{P_{pd}}$, $\boldsymbol{V}$, $\boldsymbol{\theta}$, $\boldsymbol{y}$, $\boldsymbol{de}$, $\boldsymbol{para}$, $\boldsymbol{t}$, $\boldsymbol{\varphi}$, nL)
2: **para** idx $\leftarrow$ 1 **até** nL **faça**
3: $k \leftarrow \boldsymbol{de}[idx]$
4: $m \leftarrow \boldsymbol{para}[idx]$
5: $\varphi_{km} \leftarrow \boldsymbol{\varphi}[idx]$
6: $\theta_{km} \leftarrow \boldsymbol{\theta}[k] - \boldsymbol{\theta}[m]$
7: $\boldsymbol{P_{dp}}[idx] \leftarrow |\boldsymbol{t}[idx]|^2 \cdot \mathbf{V}[k]^2 \cdot Real(\boldsymbol{y}[idx]) - |\boldsymbol{t}[idx]| \cdot \boldsymbol{V}[k] \cdot \boldsymbol{V}[m] \cdot Real(\boldsymbol{y}[idx]) \cdot$
8: $cos(\theta_{km} + \varphi_{km}) - |\boldsymbol{t}[idx]| \cdot \boldsymbol{V}[k] \cdot \boldsymbol{V}[m] \cdot Imaginário(\boldsymbol{y}[idx]) \cdot$
9: $sen(\theta_{km} + \varphi_{km})$
10: $\boldsymbol{P_{pd}}[idx] \leftarrow |\boldsymbol{t}[idx]|^{-2} \cdot \boldsymbol{V}[m]^2 \cdot Real(\boldsymbol{y}[idx]) - |\boldsymbol{t}[idx]|^{-1} \cdot \boldsymbol{V}[m] \cdot \boldsymbol{V}[k] \cdot Real(\boldsymbol{y}[idx]) \cdot$
11: $cos(-\theta_{km} - \varphi_{km}) - |\boldsymbol{t}[idx]|^{-1} \cdot \boldsymbol{V}[m] \cdot \boldsymbol{V}[k] \cdot Imaginário(\boldsymbol{y}[idx]) \cdot$
12: $sen(-\theta_{km} - \varphi_{km})$
13: **fim para**
14: **fim função**

5.2.9 Impressão dos resultados e conclusão da execução

Após o término do cálculo dos fluxos de potência nos ramos da rede , os resultados obtidos e número de iterações necessárias para a convergência são exibidos na tela. Por fim, as posições de memória alocadas dinamicamente são liberadas.

Capítulo 6

Testes Numéricos

Este capítulo objetiva apresentar os resultados numéricos de testes obtidos com programas em linguagem C++ e CUDA C++ desenvolvidos integralmente na presente pesquisa.

6.1 Introdução

Todos os testes foram executados em uma estação de trabalho com processador Intel Core i3 9400F, 16GB de memória, placa de vídeo Nvidia Gforce GTX1060 com 3 GB de memória global e sistema operacional Pop!_OS 20.04 – distribuição Linux baseada no núcleo de versão 5.4. Os compiladores adotados foram o g++ versão 9.3.0 para programas em linguagem C++ executados apenas na CPU e o nvcc versão 10.2.89, com o kit de desenvolvimento CUDA Toolkit versão 10.2, para versões em linguagem CUDA C++ que possuem alguma rotina executada na GPU. As bibliotecas cuSolver 10.2 e MKL 2020 foram ligadas às rotinas compiladas de forma dinâmica. Além disso, todo paralelismo de CPU foi obtido usando a API OpenMP[1] [69] ou rotinas da biblioteca MKL adotada para a solução direta do sistema linear na CPU. Rotinas da biblioteca Eigen [15] (versão 3.3) foram usadas para auxiliar em operações pontuais na CPU envolvendo matrizes esparsas e chamada de rotinas da biblioteca MKL. Vale ressaltar que, em nenhum dos casos, as opções de otimização do compilador foram ativadas.

[1] A versão utilizada no trabalho foi a implementação parcial do padrão da API OpenMP 5.0 [58], distribuída de forma integrada ao compilador g++.

6.2 Sistemas Teste

Para que testes possam ser executados, foram utilizados como instâncias de teste os sistemas padrão do IEEE de 14, 30, 57 e 118 barras. Esses sistemas representam partes do sistema de energia elétrica americano – localizadas no centro-oeste dos Estados Unidos – do início da década de 1960 [9]. Seus números de barras (total, de tipo PV e PQ) e ramos estão organizados na Tabela 6.1. Além disso, foram testados sistemas de maiores dimensões, com números de barras e ramos mais próximos dos utilizados nas aplicações reais de grande porte.Para tal, foram criados sistemas maiores, a partir da multiplicação de sistemas IEEE 118. Dessa forma, foram construídos sistemas com 236, 472, 944, 1.888, 3.776, 7.552 e 15.104 barras. Esses novos sistemas foram denominados IEEE 118x2, 118x4, 118x8, 118x16, 118x32, 118x64 e 118x128. A forma como isso foi feito será exposta a seguir.

6.2.1 Construção de Sistemas

Nesse contexto, será exposta a metodologia utilizada para criar novos modelos de sistemas, de maior porte, considerando um sistema-base. Esse processo foi concebido visando a manutenção da proporção de ramos e barras do novo sistema frente ao original, assim como do padrão de interligação das barras. Por isso, buscou-se minimizar o acréscimo de ramos ao novo sistema. Dessa forma, escolheu-se ligar vários sistemas base, formando um anel, para que o novo sistema fosse constituído.

Esse processo envolveu três pontos fundamentais. O primeiro é a forma como devem ser ligados os sistemas-base. Para isso, foram escolhidas duas barras suficientemente robustas e com níveis de tensão próximos no equilíbrio estático. Cada uma dessas barras em cada um dos sistemas-base presentes será conectada à outra em um outro sistema-base. O segundo ponto é a definição das características do ramo utilizado para que seja feita essa ligação. Foram adotadas linhas de transmissão arbitrárias, idênticas a outras que já tivessem ligadas às barras escolhidas. Já o terceiro ponto é a atitude frente às barras *swing* dos sistemas base. Enquanto a barra *swing* relativa ao primeiro sistema teve seu tipo preservado, as demais foram transformadas em barras PV. Além disso, para que o balanceamento de potências fosse preservado por todo o novo circuito, escolheu-se uma fração arbitrária das barras anteriormente *swing* dos sistemas-base, uniformemente distribuídas pelo anel formado, para serem ligadas à barra *swing* do novo sistema. Por fim, é necessário renumerar as barras dos sistemas-base a fim de constituir conjunto de

rótulos válidos. A forma como foi feita essa renumeração para a união dos sistemas ieee118 foi organizada no diagrama da Figura 6.1. Nela, as numerações dos n sistemas originais estão representadas em blocos. As modificações necessárias estão sobre setas que apontam para a numeração atribuída à respectiva barra que constituí o novo sistema, denominado ieee118xn, onde n é um número natural.

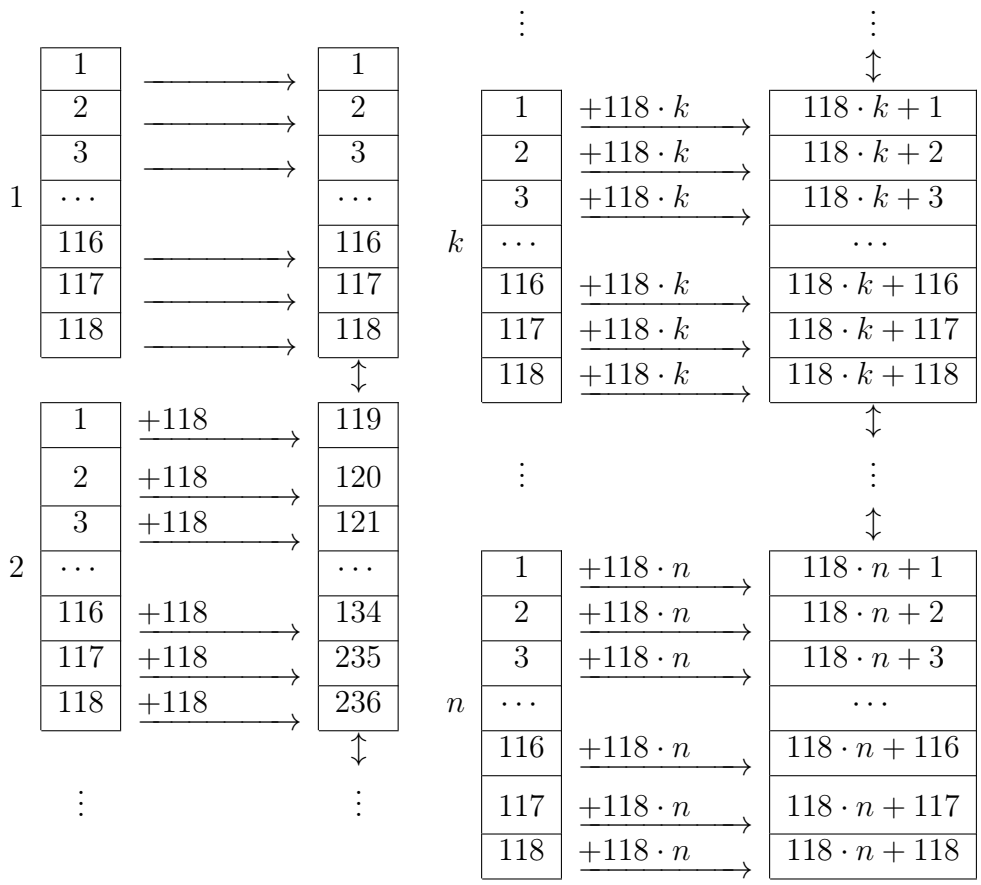

Figura 6.1: Numeração das barras de um sistema da forma ieee118xn.

Para que os sistemas base IEEE 118 sejam unidos, serão escolhidas as barras de números 66 e 89 e linhas de transmissão como as que interligam as barras 49 à 66 e 88 à 89. Esse processo foi ilustrado para sistemas constituídos por dois e quatro sistemas-base na Figura 6.2.

Além disso, foi adotada ligação de $\lfloor n/8 - 1 \rfloor$ barras anteriormente *swing* dos sistemas-base à barra *swing* do sistema IEEE 118xn construído. Linhas de transmissão como a que liga a barra 88 à 89 foi escolhida para concretizar esse ponto. Isso foi ilustrado na Figura 6.3. Nesse exemplo, cada um dos 32 círculos representa um sistema-base IEEE 118 e os traços representam as linhas de transmissão entre barras de diferentes sistemas-base. Os três traçados retos centrais interligam barras *slack* dos sistemas-base, já os demais interligam barras 66 ou 89 de sistemas-base diferentes.

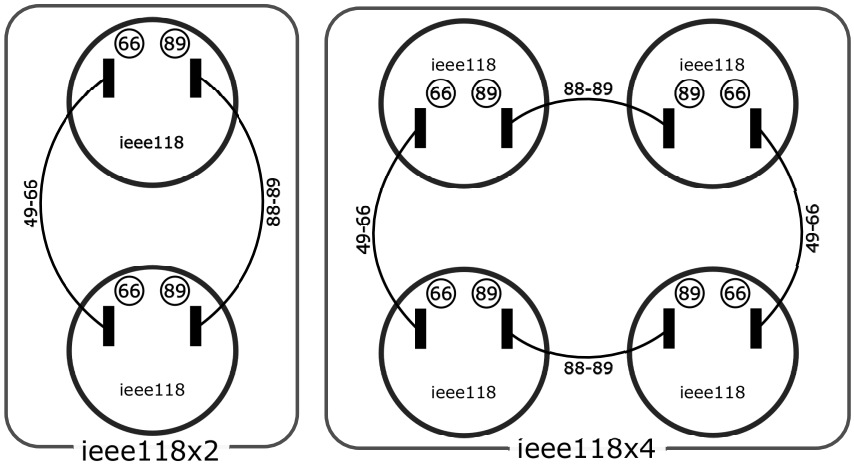

Figura 6.2: Construção dos sistemas sintéticos ieee118x2 e ieee118x4.

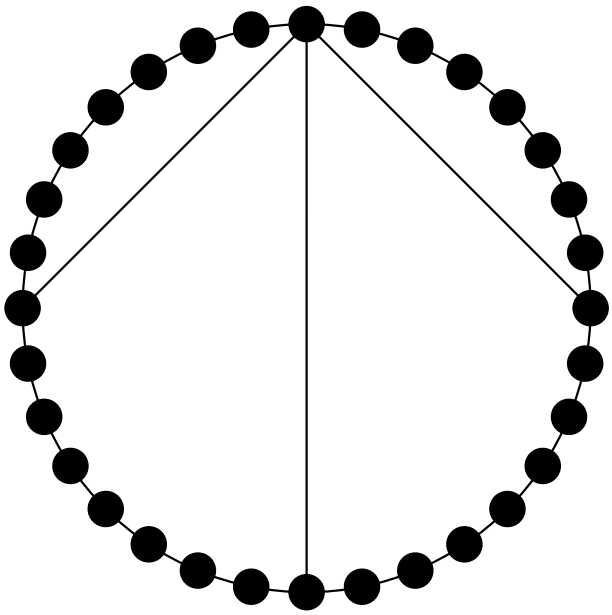

Figura 6.3: Diagrama esquemático das linhas de transmissão que ligam os 32 sistemas IEEE 118 para formar o sistema IEEE 118x32.

Dessa forma, foram construídos os sistemas IEEE 118x2, 118x4, 118x8, 118x16, 118x32, 118x64 e 118x128. Seus números de barras total, de tipo PV e PQ e número de ramos estão organizados na Tabela 6.1.

As frações dos elementos iguais a zero da matriz de admitâncias nodais dos sistemas considerados foram organizadas na Tabela 6.2. Pode-se perceber que esse percentual se torna maior com a elevação do número de barras dos sistemas listados. As respectivas matrizes Jacobiano apresentam semelhantes graus de esparsidade.

Tabela 6.1: Instâncias de teste utilizadas.

Nome IEEE	**Número de Barras**	**Número de Ramos**	**Número de Barras PV**	**Número de Barras PQ**
14	14	20	4	9
30	30	41	5	24
57	57	78	6	50
118	118	179	53	64
118x2	236	360	107	128
118x4	472	720	215	256
118x8	944	1440	431	512
118x16	1888	2881	863	1024
118x32	3776	5763	1727	2048
118x64	7552	11527	3455	4096
118x128	15104	23055	6911	8192

Tabela 6.2: Fração dos elementos da matriz de admitâncias nodais que é igual a zero.

Nome IEEE	**Matriz de Admitâncias Nodais**
14	72,45%
30	87,56%
57	93,44%
118	96,58%
118x2	98,28%
118x4	99,14%
118x8	99,57%
118x16	99,79%
118x32	99,89%
118x64	99,95%
118x128	99,97%

6.3 Resultados Numéricos

Nessa seção, serão apresentados os resultados alcançados com as simulações realizadas, com destaque para os tempos demandados por programas que adotam cada uma das abordagens apresentadas no Capítulo 5. Os tempos medidos não incluem a leitura de dados do sistema, nem aquele gasto na impressão dos resultados. É importante frisar que, além das etapas análogas às medidas para os programas executados unicamente na CPU, são consideradas nos programas que utilizam a GPU as etapas de alocação de memória global e todas as transferências de dados entre os espaços de memória do *host* e do *device* que são necessárias até que o resultado final seja obtido.

Para todos os casos, foram criadas implementações que utilizam valores de ponto flu-

tuante de precisão simples (PFPS) e dupla (PFPD). É importante observar que, desconsiderando pequenas variações nos erros gerados pelas diferentes sequências de operações de ponto flutuante realizadas em cada implementação, os resultados obtidos podem ser considerados numericamente idênticos. Além disso, a consistência dos resultados alcançados pelos programas desenvolvidos foi comprovada uma vez que, em todos os casos, os desvios máximos de potência foram inferiores à tolerância adotada (10^{-4}).

Exceto quando mencionado, foram medidos os tempos necessários para 100 execuções das implementações densas e esparsas, de forma que a média foi calculada para minimizar o possível impacto de outros processos em execução. Para que as medições fossem feitas, utilizou-se a biblioteca `chrono`, parte da biblioteca padrão da linguagem C++, junto às sincronizações com a GPU que se fizeram necessárias. No caso do tempo total demandado pelas rotinas que utilizam a GPU, utilizou-se eventos CUDA [56], uma alternativa mais leve quando comparada ao primeiro método. Análise estatística por meio do *desvio padrão percentual* ($\sigma_{\%}$) dos valores medidos é relevante para ilustrar o quão representativos são os valores médios apresentados.

$$\sigma_{\%} = \left(\frac{1}{\mu} \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x_i - \mu)^2} \right) \cdot 100\% \tag{6.1}$$

onde N é o número de valores de tempo medidos e μ é a esperança para os valores de x_i, tal que $\mu = \frac{1}{N} \cdot \sum_{i=1}^{N} x_i$.

Os valores de $\sigma_{\%}$ dos tempos medidos para todos os experimentos que usam a GPU apresentaram valor inferior a 10%. Nos experimentos sequenciais com sistemas com mais de 30 barras isso também se concretizou. Nos testes paralelos na CPU com mais de 118 barras os maiores valores encontrados foram inferiores a 20%. Era esperado que as rotinas paralelas para a CPU apresentassem maior desvio padrão percentual uma vez que, por utilizarem maior fração do potencial do processador, passariam a ser necessárias mais interrupções do sistema operacional. Os valores dos desvios padrão percentuais relativos aos resultados dos testes feitos estão organizados no Apêndice A, assim como os valores de tempo medidos para todos os testes apresentados nesse capítulo. Nesse contexto, a apresentação de valores médios se mostrou satisfatória para ilustrar o comportamento das rotinas propostas.

Uma definição importante para o desenvolvimento desse capítulo é a de *aceleração*. Aceleração de um programa A em relação a um programa B será a razão escrita a seguir. Nesta expressão, $T_{\Gamma}(n)$ é o tempo necessário para executar o programa Γ quando recebe

n como entrada.

$$S_{A,B} = \frac{T_A(n)}{T_B(n)} \tag{6.2}$$

A Tabela 6.3 organiza os tempos demandados para a execução das implementações densas e esparsas que fazem uso da CPU. Enquanto a presença do símbolo tipográfico do obelisco (†) em uma célula significa que o respectivo teste não pôde ser executado devido a limitação na capacidade da memória principal do computador, o asterisco (*) indica que foram executados 10 repetições para o teste. A implementação esparsa e sequencial executada na CPU para a solução do problema do FP em SEPs é considerada estado da arte [18]. Frente à implementação densa sequencial (mais trivial) a primeira apresenta grande aceleração, resultado confirmado pelas rotinas construídas. Foi obtida aceleração de até 1893 vezes para o programa sequencial esparso frente ao denso sequencial. Além disso, técnicas de computação paralela podem ser aplicadas às implementações densas. Contudo, isso não é o bastante para que seja obtida vantagem frente às rotinas esparsas sequenciais. Acelerações das rotinas esparsas sequenciais com relação às densas sequenciais e paralelas estão organizadas na Tabela 6.4.

Rotinas esparsas executadas na CPU também podem se beneficiar da adoção de paralelismo. Essa prática produziu aceleração de até 1,35 vezes frente às sequenciais. Todos os valores de aceleração obtidos para essas rotinas estão organizados na Tabela 6.5.

Note, na Tabela 6.3, que as diferenças entre os tempos demandados pelas versões densas que utilizam PFPS e PFPD são mais significativas do que as das respectivas versões esparsas. Isso está relacionado ao maior numero de operações envolvidas nas primeiras. Outro ponto relevante é relativo aos tempos demandados por implementações paralelas serem maiores do que os demandados para as respectivas implementações sequenciais quando o SEP analisado é suficientemente pequeno. Isso está relacionado ao tempo exigido para que os *threads* adicionais sejam criados. Uma vez que não exista quantidade suficiente de informação para que o ganho de velocidade de processamento com o uso de paralelismo supere esse custo inicial, performance inferior desse tipo de abordagem é esperada.

Os tempos demandados para a execução das rotinas que fazem uso da GPU estão organizados na Tabela 6.6. De forma semelhante ao ocorrido para as rotinas que executam na CPU, a presença do símbolo tipográfico do obelisco (†) em uma célula significa que o respectivo teste não pôde ser executado devido a limitação na capacidade da *memó-*

Tabela 6.3: Tempo demandado pelos programas executados apenas na CPU.

Instância de Teste	**Denso**				**Esparso**			
	Sequencial		**Paralelo**		**Sequencial**		**Paralelo**	
	PFPS	**PFPD**	**PFPS**	**PFPD**	**PFPS**	**PFPD**	**PFPS**	**PFPD**
14	1,34 ms	1,40 ms	14,6 ms	1,62 ms	1,96 ms	1,95 ms	12,4 ms	11,8 ms
30	2,15 ms	2,38 ms	14,3 ms	13,5 ms	2,59 ms	2,58 ms	12,5 ms	12,0 ms
57	4,73 ms	5,39 ms	18,4 ms	17,3 ms	3,29 ms	3,29 ms	13,4 ms	12,3 ms
118	15,3 ms	18,4 ms	32,4 ms	21,0 ms	5,23 ms	5,14 ms	15,9 ms	13,8 ms
118x2	56,4 ms	70,8 ms	30,3 ms	40,0 ms	8,51 ms	8,36 ms	18,4 ms	17,2 ms
118x4	229 ms	289 ms	89,1 ms	117 ms	15,7 ms	15,6 ms	24,4 ms	23,3 ms
118x8	998 ms	1,3 s	346 ms	493 ms	36,1 ms	29,8 ms	38,5 ms	33,3 ms
118x16	5,5 s	7,2 s	1,8 s	2,7 s	69,8 ms	70,0 ms	61,4 ms	62,0 ms
118x32	24,8 s	34,5	8,6 s	12,9 s	141 ms	140 ms	114 ms	114 ms
118x64	2 min 8 s	3 min 24 s	47,3 s	1 min 16 s	284 ms	284 ms	210 ms	214 ms
118x128	17 min 50 s*	†	6 min 3 s*	†	565 ms	556 ms	443 ms	457 ms

Tabela 6.4: Aceleração das Rotinas Esparsas Sequenciais frente às rotinas densas sequenciais e paralelas da CPU.

Sistema IEEE	**Rotina Densa**			
	Sequencial		**Paralela**	
	PFPS	**PFPD**	**PFPS**	**PFPD**
14	0,69	0,72	7,52	0,13
30	0,83	0,92	5,52	1,08
57	1,44	1,64	5,61	1,30
118	2,94	3,58	6,31	1,33
118x2	6,62	8,48	3,62	2,17
118x4	14,58	18,60	5,73	4,79
118x8	27,69	42,75	11,63	12,80
118x16	78,64	102,42	26,14	43,64
118x32	175,50	245,79	60,97	113,29
118x64	448,85	719,07	166,75	361,02
118x128	1893,67	†	652,41	†

ria global da GPU e a presença do asterisco (*) indica que os valores apresentados são médias de 10 medições. Diferença significativa no tempo demandado para execução dos métodos semelhantes (tanto *densos* quanto *esparsos*) que utilizam PFPS e PFPD pode ser observada. Isso é justificado uma vez que a placa gráfica usada não possui unidades de hardware específicas para a realização de operações aritméticas envolvendo PFPD, denominadas DPUs (Unidades de Precisão Dupla). Isso acarreta, para uma GPU com arquitetura Pascal, vazão de operações sobre PFPD 32 vezes menor [53, 71] do que de operações com PFPS.

Uma propriedade que pode ser observada nos programas que utilizam a GPU é a redução da carga de trabalho do *host*, uma vez que parte das funções passa a ser executada pelo *device*. Isso cria a possibilidade de funções como a de elevação dinâmica da taxa

Tabela 6.5: Aceleração das rotinas Esparsas Paralelas executadas na CPU frente às Esparsa Sequenciais executada na CPU.

Sistema IEEE	**PFPS**	**PFPD**
14	0,16	0,16
30	0,21	0,22
57	0,25	0,27
118	0,33	0,37
118x2	0,46	0,49
118x4	0,64	0,67
118x8	0,94	0,89
118x16	1,14	1,13
118x32	1,25	1,23
118x64	1,35	1,33
118x128	1,27	1,22

de frequência do processador (e.g., tecnologia Turbo Boost dos modelos da fabricante Intel [35]) e a maior disponibilidade da largura de banda do canal externo de comunicação da CPU beneficiarem em maior magnitude as frações do programa que são executadas nesse processador, quando confrontada com a versão análoga completamente executada na CPU. Nesse sentido, foi possível observar sutis melhorias no desempenho de algumas das frações dos programas que não foram modificadas para execução na GPU.

Tabela 6.6: Tempo demandado pelos programas que usam a GPU.

Instância de Teste	**Denso**		**Esparso**	
	PFPS	**PFPD**	**PFPS**	**PFPD**
14	234 ms	235 ms	357 ms	357 ms
30	237 ms	238 ms	357 ms	357 ms
57	243 ms	244 ms	360 ms	360 ms
118	261 ms	262 ms	362 ms	367 ms
118x2	293 ms	298 ms	374 ms	378 ms
118x4	359 ms	372 ms	402 ms	414 ms
118x8	493 ms	573 ms	499 ms	540 ms
118x16	935 ms	1,6 s	1,2 s	1,5 s
118x32	2,0 s	7,0 s	5,7 s	8,5 s
118x64	6 s	†	52 s	1 min 17 s
118x128	†	†	7 min 45 s*	11 min 23 s*

Tendo em vista os resultados mostrados nas Tabelas 6.3 e 6.6, torna-se evidente a necessidade de análise mais detalhada das implementações construídas. Pode-se depreender que as melhores rotinas que implementam as técnicas densas ou esparsas na CPU são as que fazem uso de paralelismo. Os gráficos da Figura 6.4 representam a fração do tempo total de processamento dessas rotinas que é empregado para cada uma das etapas

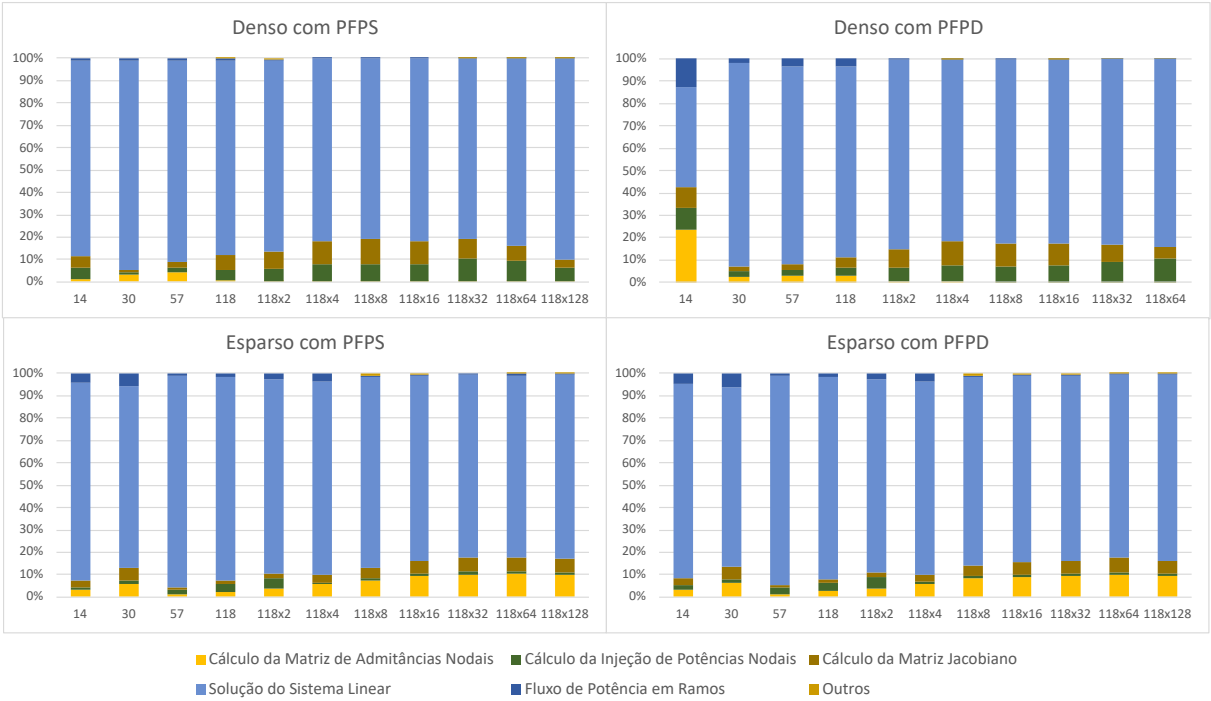

Figura 6.4: Parcela do tempo total demandado por cada uma das etapas das implementações densas e esparsas paralelas para a CPU na solução do PFP.

mencionadas no Capítulo 5. Gráfico análogo para os programas densos e esparsos que usam a GPU está representado na Figura 6.5. A etapa denominada *Inicialização da GPU* engloba o tempo demandado para o início da comunicação com o *device*, alocação memória e transferência dos dados lidos para a memória global, a etapa de *Inicialização da Biblioteca* é necessária para o uso das rotinas da biblioteca cuSolver. Consiste na criação de *handles* e *streams* para a biblioteca e inicialização de estrutura que descreve a forma das matrizes que definem os sistemas lineares a serem resolvidos [55].

É possível notar que as etapas de cálculo das *Cálculo das Injeções de Potência Nodais*, *Cálculo da Matriz Jacobiano* e *Solução do Sistema Linear* são dominantes frente às demais quando o SEP analisado é suficientemente grande. Nessa perspectiva, serão confrontados os tempos demandados por essas etapas em cada uma das implementações construídas. Para tal, organizou-se esses dados nos gráficos da Figura 6.6.

Pode-se constatar que, no caso da etapa de *Avaliação das Injeções de Potência Nodais*, os algoritmos esparsos que utilizam a GPU são superiores aos demais. Isso está relacionado à propriedade deles envolverem menor número de operações quando comparados aos densos, ao mesmo tempo que mantém satisfatória taxa de utilização da GPU. De semelhante forma, abordagem esparsa com o uso da GPU se mostrou mais propícia para o *Cálculo dos Elementos da Matriz Jacobiano*. Na etapa de *Solução do Sistema Linear*,

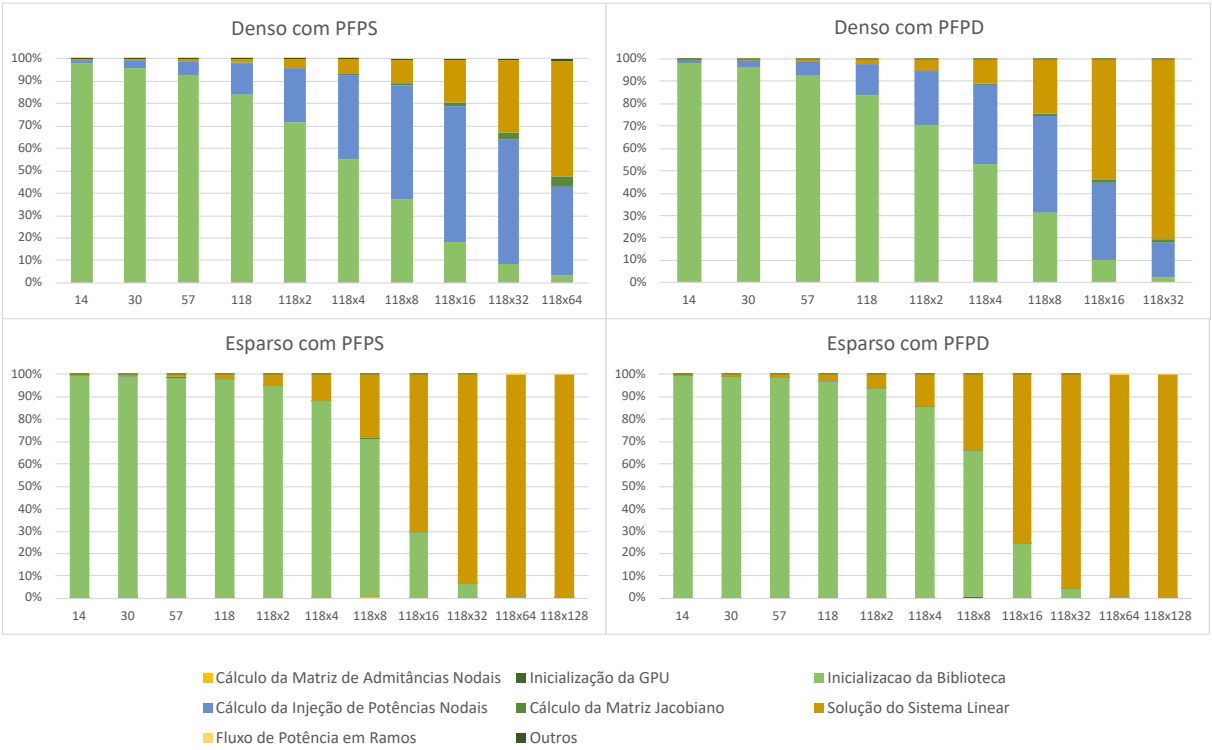

Figura 6.5: Parcela do tempo total demandado por cada uma das etapas das implementações densas e esparsas para a GPU na solução do PFP.

relação diferente foi observada: embora para instâncias de teste suficientemente pequenas os métodos de GPU densos tenham se mostrado mais rápidos, rotinas esparsas executadas na CPU se mostraram mais vantajosas para instâncias de teste maiores. Isso está relacionado aos desafios de paralelização de métodos esparsos para solução de sistemas lineares, principalmente em GPUs, que apresentam penalidades significativas quando há divergência entre *threads* concorrentes.

Nesse contexto, mostra-se pertinente a criação de implementação híbrida, que utilize as melhores rotinas para cada uma das etapas. Essa implementação se baseia nos programas esparsos executados na GPU para todas as etapas, com exceção da que soluciona o sistema linear, para a qual foi adotada a rotina paralela esparsa da CPU. Os tempos médios demandados para sua execução estão organizados na Tabela 6.7. O maior tempo demandado para a execução das rotinas híbridas em comparação com as respectivas rotinas esparsas sequenciais que executam na CPU, para sistemas suficientemente pequenos, pode ser justificado pelo tempo adicional necessário para a transferência inicial dos dados do problema para a GPU. Ele é compensado pela maior vazão de processamento da GPU quando existe quantidade suficiente de cálculos para serem executados. Além disso, o número máximo de linhas de execução concorrentes explorado cresce com o número de barras e ramos do SEP considerado, acarretando melhor aproveitamento do potencial da

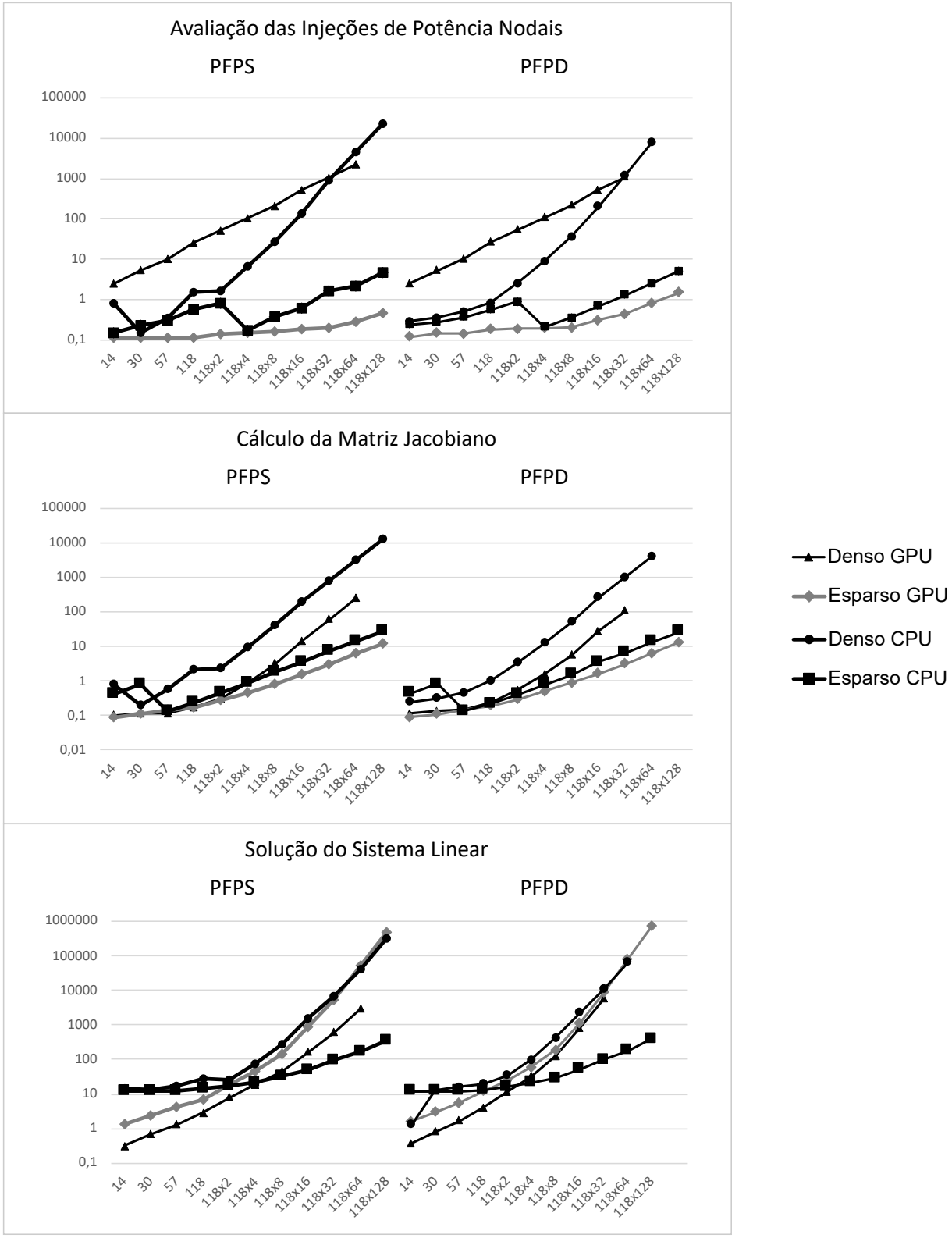

Figura 6.6: Tempo (em milissegundos) demandado para a execução das etapas destacadas para cada um dos sistemas apresentados.

GPU nos sistemas grandes. A Tabela 6.7 também organiza acelerações dos programas híbridos GPU-CPU frente às versões paralela (sob a coluna Aceleração Paralela) e sequencial (sob a coluna Aceleração Sequencial) esparsas que utilizam apenas a CPU. É possível observar que aceleração considerável, de 1,3 vezes, frente a ambos os casos sequenciais e de, aproximadamente, 1,1 frente aos paralelos, para os maiores sistemas testados.

Tabela 6.7: Tempo demandado pelos programas com abordagem híbrida GPU-CPU e suas acelerações frente aos programas para CPU paralelos e sequenciais.

Instância	**Tempo de Execução**		**Aceleração Paralela**		**Aceleração Sequencial**	
de Teste	**PFPS**	**PFPD**	**PFPS**	**PFPD**	**PFPS**	**PFPD**
14	3,62 ms	3,61 ms	3,43	3,27	0,54	0,54
30	4,17 ms	4,16 ms	3,01	2,88	0,62	0,62
57	4,66 ms	4,69 ms	2,87	2,63	0,71	0,70
118	5,34 ms	6,11 ms	2,97	2,26	0,98	0,84
118x2	8,49 ms	8,58 ms	2,17	2,01	1,00	0,97
118x4	13,9 ms	14,0 ms	1,76	1,67	1,13	1,11
118x8	22,8 ms	22,9 ms	1,69	1,45	1,58	1,30
118x16	41,0 ms	49,0 ms	1,50	1,27	1,70	1,43
118x32	97,6 ms	98,6 ms	1,16	1,16	1,45	1,42
118x64	192 ms	194 ms	1,09	1,10	1,48	1,46
118x128	410 ms	413 ms	1,08	1,11	1,38	1,35

Os efeitos de aceleração observados em frações do programa para as quais foram construídos *kernels* são aproximadamente constantes. Já sobre as demais partes são proporcionalmente menores, dependendo do porte do sistema envolvido. Isso justifica a redução da aceleração observada frente a implementação paralela.

Além disso, algumas distorções estão presentes nos resultados obtidos para os testes com PFPS em sistemas entre IEEE 118, 118x8 e 118x16. O ocorrido se deve à proximidade entre a precisão de alguns dos valores de ponto flutuante intermediários do problema e a tolerância exigida para convergência. Isso acarretou um número diferente de iterações para os programas, uma vez que as operações de ponto flutuante produziram valores intermediários diferentes.

Por fim, a Tabela 6.8 apresenta as acelerações observadas nas principais etapas das implementações que utilizam a GPU, com relação aos programas com mesma representação de ponto flutuante e que exploram esparsidade e paralelismo na CPU.

Tabela 6.8: Aceleração promovida em cada uma das principais etapas das implementações executadas na GPU frente ao programa segundo paradigma esparso executado na CPU com paralelismo e mesmo tipo de representação de valor de ponto flutuante.

	PFPS							PFPD						
Sistema IEEE	Total	Matriz de Admitân-cias Nodais	Injeção de Potências Nodais	Jacobiano	Sistema Linear	Fluxo de Potência em Ramos	Sistema IEEE	Total	Matriz de Admitân-cias Nodais	Injeção de Potências Nodais	Jacobiano	Sistema Linear	Fluxo de Potência em Ramos	
14	3,4	11,0	1,3	4,9	4,6	29,4	14	3,3	11,0	1,9	4,9	4,4	28,5	
30	3,0	11,6	1,9	7,8	3,8	46,3	30	2,9	12,0	2,1	7,8	3,7	49,2	
57	2,9	1,3	2,6	‡	3,6	7,2	57	2,6	1,3	2,6	‡	3,3	6,7	H
118	3,0	1,3	4,9	1,4	3,6	14,3	118	2,3	1,3	3,0	1,1	2,8	11,2	í
118x2	2,2	1,3	5,8	1,6	2,4	22,0	118x2	2,0	1,3	4,7	1,4	2,3	18,9	b
118x4	1,8	1,3	1,2	1,8	1,9	40,7	118x4	1,7	1,3	1,1	1,5	1,8	35,9	r
118x8	1,7	1,3	2,5	2,1	1,8	8,4	118x8	1,5	1,3	1,8	1,7	1,5	6,1	i
118x16	1,5	1,3	4,2	2,2	1,5	14,8	118x16	1,3	1,3	2,2	2,0	1,2	10,0	d
118x32	1,2	1,3	8,0	2,2	1,1	26,0	118x32	1,2	1,3	2,8	1,9	1,1	12,0	a
118x64	1,1	1,2	7,6	2,1	1,0	31,7	118x64	1,1	1,2	2,8	2,0	1,0	14,4	
118x128	1,1	1,2	10,1	2,1	1,0	38,5	118x128	1,1	1,2	3,3	2,0	1,1	15,8	
14	‡	11,0	1,3	4,8	8,7	29,2	14	‡	11,0	2,0	4,9	7,4	30,1	
30	‡	11,8	2,0	7,6	5,1	43,0	30	‡	12,0	1,9	7,4	4,0	44,6	
57	‡	1,3	2,6	‡	2,9	7,5	57	‡	1,3	2,6	‡	2,2	6,6	E
118	‡	1,3	5,0	1,3	2,1	13,0	118	‡	1,3	3,0	1,1	1,1	10,7	s
118x2	‡	1,3	5,8	1,6	‡	20,8	118x2	‡	1,3	4,7	1,4	‡	18,4	p
118x4	‡	1,3	1,1	1,8	‡	37,4	118x4	‡	1,3	1,1	1,6	‡	33,8	a
118x8	‡	1,3	2,4	2,2	‡	8,7	118x8	‡	1,3	1,8	1,8	‡	6,9	r
118x16	‡	1,3	3,3	2,3	‡	15,8	118x16	‡	1,3	2,3	2,1	‡	11,5	s
118x32	‡	1,3	7,7	2,3	‡	28,6	118x32	‡	1,3	2,9	2,1	‡	14,2	a
118x64	‡	1,2	7,6	2,3	‡	35,7	118x64	‡	1,2	3,1	2,1	‡	16,8	
118x128	‡	1,2	10,1	2,3	‡	36,3	118x128	‡	1,2	3,5	2,0	‡	16,6	
14	‡	54,5	‡	4,0	35,8	13,7	14	‡	43,9	‡	3,6	29,2	14,5	
30	‡	52,8	‡	7,3	17,2	20,0	30	‡	43,7	‡	5,8	13,7	22,3	
57	‡	5,3	‡	1,1	9,1	3,6	57	‡	4,4	‡	‡	6,7	3,5	
118	‡	5,1	‡	1,2	4,5	6,7	118	‡	4,2	‡	‡	3,1	6,1	D
118x2	‡	5,1	‡	1,4	1,9	11,4	118x2	‡	4,2	‡	‡	1,3	11,2	e
118x4	‡	4,9	‡	‡	1,1	20,3	118x4	‡	4,0	‡	‡	‡	19,9	n
118x8	‡	4,9	‡	‡	‡	4,2	118x8	‡	4,1	‡	‡	‡	3,5	s
118x16	‡	4,9	‡	‡	‡	7,0	118x16	‡	4,0	‡	‡	‡	5,9	a
118x32	‡	4,7	‡	‡	‡	11,9	118x32	‡	3,8	‡	‡	‡	8,2	
118x64	‡	4,4	‡	‡	‡	14,2	118x64	†	†	†	†	†	†	
118x128	†	†	†	†	†	†	118x128	†	†	†	†	†	†	

†	O teste não pode ser realizado por limitação da memória disponível.
‡	Não foi observada aceleração nessa etapa.

1,0 50,0

Capítulo 7

Conclusão

Sistemas elétricos de potência estão estruturados em partes que remetem à geração, transmissão, distribuição e utilização de energia elétrica. Muitas vezes, refere-se ao setor elétrico como provavelmente um dos de maior porte e complexidade do mundo atual. As atividades de operação e planejamento são realizadas para que se possa ofertar quantidades cada vez maiores de energia elétrica, de forma segura, com baixo impacto ambiental e economicamente viável. Compostos por geradores e cargas interconectados por linhas de transmissão e transformadores que formam uma rede elétrica, sistemas de potência trifásicos operam usualmente em condições de equilíbrio carga-geração, de modo a manter padrões bem estabelecidos de qualidade de fornecimento (respeitadas certas restrições operativas) e de continuidade de serviço.

Em vários estudos da área de análise de redes elétricas, o problema de fluxo de potência se impõe pela necessidade de se conhecer as tensões das barras (nodais) para uma determinada configuração da rede elétrica e das demais grandezas delas dependentes. Matematicamente, este problema envolve a solução de sistemas (de grande porte) de equações algébricas não lineares, (usualmente) pelo método de Newton-Raphson.

Esta Dissertação objetivou a obtenção de soluções para o problema de fluxo de potência em redes de grande porte, adotando-se o Método de Newton-Raphson e explorando-se a natureza das matrizes esparsas que fazem parte do cálculo dessas soluções. Técnicas da computação paralela foram utilizadas para a criação de algoritmos executáveis em CPUs e GPUs, implementados em linguagem C++ e CUDA C++. Testes foram executados com o uso de valores em ponto flutuante de precisão simples e dupla. Diversas simulações foram realizadas com base no sistema de referência do IEEE 118 barras, que foi multiplicado para a criação de testes de grande porte, para a avaliação do desempenho do tratamento computacional proposto.

Da pesquisa realizada, podem ser extraídas conclusões, sinteticamente descritas a seguir.

Os resultados obtidos pelas implementações executadas na CPU confirmam a maior eficiência computacional dos algoritmos que exploram esparsidade frente aos que realizam cálculos com matrizes tratadas como densas fossem. Aceleração de várias ordens de grandeza pôde ser observada entre os programas sequenciais. Além disso, os resultados indicam que pode ser obtida melhoria na performance do algoritmo com esparsidade, se aplicado o paralelismo. Isto foi observado nas simulações com os sistemas de mais de 1.888 barras. Confirmou-se, também, que a etapa de solução direta do sistema linear, obtida a cada iteração do método de Newton-Raphson, ocupa fração majoritária do tempo de execução dos programas na CPU.

Programas que utilizam a GPU mostraram bons resultados. As metodologias apresentadas para *Cálculo de Potências Nodais Líquidas*, *Cálculo da matriz Jacobiano* e *Cálculo do Fluxo de Potência em Ramos* que exploram esparsidade na GPU apresentaram aceleração significativa frente às análogas executadas na CPU. Também puderam ser observados ganhos no tempo computacional demandado pelas rotinas esparsas de solução direta de sistema linear com uso da biblioteca cuSolver (executadas na GPU) frente às da biblioteca MKL (executada na CPU) para sistemas de até 236 barras, quando o tempo demandado para inicialização das estruturas exigidas pela primeira biblioteca não é considerado. Uma abordagem híbrida que utiliza as melhores rotinas para a execução de cada uma das etapas foi proposta. Essa abordagem foi capaz de promover aceleração frente às melhores rotinas executadas na CPU para sistemas com mais de 472 barras.

7.1 Trabalhos Futuros

Tomando por base os presentes resultados da pesquisa, propõe-se os seguintes direcionamentos para trabalhos futuros:

- Embora as instâncias de teste sintéticas sejam úteis como prova de conceito sobre comportamento da metodologia proposta, a consideração de sistemas reais de grande porte deve ser explorada.
- Antes que uma implementação eficiente fosse alcançada para o programa desenvolvido, o tempo consumido pela etapa de *cálculo da matriz de admitâncias nodais* não representava porção significativa do tempo total de execução de nenhuma das abor-

dagens estudadas. Contudo, isso deixou de ser verdade após aplicação das técnicas de otimização propostas. Recomenda-se que seja estudada a aplicação de técnicas de paralelismo a essa etapa.

- Os estudos aqui realizados se restringiram ao uso de um modelo de CPU e placa gráfica. Novas pesquisas envolvendo outros modelos de processadores e GPUs, com especificações e características diferentes, devem ser considerados, para avaliação de desempenho da metodologia de cálculo proposta.

- Apesar de ter sido utilizada em *kernel* "denso", memória compartilhada não foi utilizada nos *kernels* "esparsos". Sugere-se um estudo que avalie essa possibilidade, uma vez que pode trazer ganhos significativos de performance.

- A etapa de análise simbólica da matriz Jacobiano adota uma forma como tal matriz será construída. A etapa de decomposição dessa matriz para solução direta do sistema linear que ela define envolve novo processo de análise simbólica [66]. Indica-se a realização de um estudo de viabilidade para a construção de técnica única para as duas etapas, assim como do uso de artifícios que tornem esse processo conjunto mais eficiente do que quando feito de forma individual.

Referências

[1] Alexander, C. K.; Sadiku, M. N. O. *Fundamentals of electric circuits*, 5th ed ed. McGraw-Hill, New York, NY, 2013.

[2] Alvarado, F.; Betancourt, R.; Clements, K.; Heydt, G. T.; Huang, G.; Ilic, M.; Scala, M. L.; Pai, M.; Pottle, C.; Talukdar, S.; Ness, J. V.; Wu, F. Parallel processing in power systems computation. *IEEE Transactions on Power Systems 7*, 2 (1992), 629–638.

[3] Arestova, A.; Häger, U.; Grobovoy, A.; Rehtanz, C. SuperSmart grid for improving system stability at the example of a possible interconnection of ENTSO-E and IPS/UPS. In *2011 IEEE Trondheim PowerTech* (June 2011), pp. 1–8.

[4] Aspray, W.; Bromley, A. G., Eds. *Computing before computers*. Iowa State University Press, Ames, Ia, 1990.

[5] Association, C. E. The north american grid: Powering cooperation on clean energy & the environment. Disponível em `https://electricity.ca/wp-content/uploads/2017/05/CEA_16-086_The_North_American_E_WEB.pdf`. Acessado em 26/12/2020.

[6] Burden, R. L.; Faires, J. D.; Burden, A. M. *Numerical analysis*, tenth edition ed. Cengage Learning, Boston, MA, 2016.

[7] Butenhof, D. R. *Programming with POSIX threads*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1997.

[8] Chen, Y.; Jin, H.; Jiang, H.; Xu, D.; Zheng, R.; Liu, H. Implementation and Optimization of GPU-Based Static State Security Analysis in Power Systems, Mar. 2017. ISSN: 1574-017X Pages: e1897476 Publisher: Hindawi Volume: 2017.

[9] Christie, R. D. University of washington power systems test case archive. Disponível em `https://labs.ece.uw.edu/pstca/`. Acessado em 20/10/2020.

[10] Cormen, T. H., Ed. *Introduction to algorithms*, 3 ed. MIT Press, Cambridge, Mass, 2009.

[11] CPUDB. Clock frequency. Disponível em `http://cpudb.stanford.edu/visualize/clock_frequency`. Acessado em 23/11/2020.

[12] CPUDB. Technology scaling by manufacturer. Disponível em `http://cpudb.stanford.edu/visualize/clock_frequency`. Acessado em 23/11/2020.

[13] Dennard, R. H.; Cai, J.; Kumar, A. A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics 51*, 4 (Apr. 2007), 518–525.

[14] Dennard, R. H.; Gaensslen, F. H.; Yu, H.; Rideout, V. L.; Bassous, E.; LeBlanc, A. R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (Oct. 1974), 256–268.

[15] Eigen. Eigen library. Disponível em `https://eigen.tuxfamily.org`. Acessado em 20/10/2020.

[16] etap. Energy management solutions to design, operate, and automate power systems. Disponível em `https://etap.com/`. Acessado em 20/03/2021.

[17] Faggin, F.; Hoff, M. E.; Mazor, S.; Shima, M. The history of the 4004. *IEEE Micro 16*, 6 (Dec. 1996), 10–20.

[18] Falcão, D. M. High performance computing in power system applications. In *Vector and Parallel Processing — VECPAR'96* (Berlin, Heidelberg, 1997), J. M. L. M. Palma and J. Dongarra, Eds., Lecture Notes in Computer Science, Springer, pp. 1–23.

[19] Flueck, A.; Chiang, H.-D. Solving the nonlinear power flow equations with an inexact Newton method using GMRES. *IEEE Transactions on Power Systems 13*, 2 (May 1998), 267–273. Conference Name: IEEE Transactions on Power Systems.

[20] Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers C-21*, 9 (Sept. 1972), 948–960. Conference Name: IEEE Transactions on Computers.

[21] Garcia, N. Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach. In *IEEE PES General Meeting* (July 2010), pp. 1–4. ISSN: 1944-9925.

[22] GE. Pslf simulation engine. Disponível em `https://www.geenergyconsulting.com/practice-area/software-products/pslf`. Acessado em 20/03/2021.

[23] Glimn, A. F.; Stagg, G. W. Automatic Calculation of Load Flows. *Transactions of the American Institute of Electrical Engineers. Part III: Power Apparatus and Systems 76*, 3 (Apr. 1957), 817–825. Conference Name: Transactions of the American Institute of Electrical Engineers. Part III: Power Apparatus and Systems.

[24] Grainger, J. J.; Stevenson, W. D.; Stevenson, W. D. *Power system analysis.* McGraw-Hill series in electrical and computer engineering. McGraw-Hill, New York, 1994.

[25] Green, R. C.; Wang, L.; Alam, M. High performance computing for electric power systems: Applications and trends. In *2011 IEEE Power and Energy Society General Meeting* (July 2011), pp. 1–8. ISSN: 1944-9925.

[26] Group, W. Common format for exchange of solved load flow data. *IEEE Transactions on Power Apparatus and Systems PAS-92*, 6 (1973), 1916–1925.

[27] Guidorizzi, H. L. *Um curso de cálculo, vol. 2 (5a. ed.).* Grupo Gen - LTC, 2000.

[28] Guo, C.; Jiang, B.; Yuan, H.; Yang, Z.; Wang, L.; Ren, S. Performance Comparisons of Parallel Power Flow Solvers on GPU System. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (Aug. 2012), pp. 232–239. ISSN: 2325-1301.

[29] Guérard, G.; Ben Amor, S.; Bui, A. A Complex System Approach for Smart Grid Analysis and Modeling. vol. 243, pp. 788–797.

[30] Hurson, A. R.; Sarbazi-Azad, H., Eds. *Dark silicon and future on-chip systems*, first edition ed. No. volume 110 in Advances in computers. Academic Press, an imprint of Elsevier, Cambridge, MA San Diego, CA Oxford London, 2018.

[31] IEEE. IEEE Recommended Practice for Conducting Load-Flow Studies and Analysis of Industrial and Commercial Power Systems. Tech. rep., IEEE, 2018. ISBN: 9781504452403.

[32] Inc, K. G. Opencl overview. Disponível em `https://www.khronos.org/opencl/`. Acessado em 20/12/2020.

[33] Inc, K. G. Opencl overview. Disponível em `https://www.khronos.org/conformance/adopters/conformant-companies#opencl`. Acessado em 20/12/2020.

[34] Intel. Intel turbo boost technology 2.0. Disponível em `http://www.cepel.br/pt_br/produtos/anarede-analise-de-redes-eletricas.htm`. Acessado em 20/03/2021.

[35] Intel. Intel turbo boost technology 2.0. Disponível em `https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`. Acessado em 10/01/2021.

[36] Jalili-Marandi, V.; Zhou, Z.; Dinavahi, V. Large-Scale Transient Stability Simulation of Electrical Power Systems on Parallel GPUs. *IEEE Transactions on Parallel and Distributed Systems 23*, 7 (July 2012), 1255–1266. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[37] Karimipour, H.; Dinavahi, V. Accelerated parallel WLS state estimation for large-scale power systems on GPU. In *2013 North American Power Symposium (NAPS)* (Sept. 2013), pp. 1–6.

[38] Kirk, D.; Hwu, W.-m. W. *Programming massively parallel processors: a hands-on approach*, 2. ed ed. Elsevier, Morgan Kaufmann, Amsterdam, 2013.

[39] Kuck, D. J. *The structure of computers and computations.* Wiley, New York, 1978.

[40] Kumar, R. J. R.; Rao, P. S. N. A new successive displacement type load flow algorithm and its application to radial systems. In *2014 IEEE 2nd International Conference on Electrical Energy Systems (ICEES)* (Jan. 2014), pp. 15–19.

[41] Li, X.; Li, F. GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method. *Electric Power Systems Research 116* (Nov. 2014), 87–93. Publisher: Elsevier.

[42] Liao, W.; Basile, J. M.; He, L. Leakage power modeling and reduction with data retention. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design* (New York, NY, USA, Nov. 2002), ICCAD '02, Association for Computing Machinery, pp. 714–719.

[43] Lin, C.; Liu, J.; Yang, P. Performance Enhancement of GPU Parallel Computing Using Memory Allocation Optimization. In *2020 14th International Conference on Ubiquitous Information Management and Communication (IMCOM)* (Jan. 2020), pp. 1–5.

[44] Liu, Z.; Song, Y.; Chen, Y.; Huang, S.; Wang, M. Batched Fast Decoupled Load Flow for Large-Scale Power System on GPU. In *2018 International Conference on Power System Technology (POWERCON)* (Nov. 2018), pp. 1775–1780.

[45] Malcolm, J.; Yalamanchili, P.; McClanahan, C.; Venugopalakrishnan, V.; Patel, K.; Melonakos, J. ArrayFire: a GPU acceleration platform. In *Modeling and Simulation for Defense Systems and Applications VII* (2012), E. J. Kelmelis, Ed., vol. 8403, International Society for Optics and Photonics, SPIE, pp. 49 – 56.

[46] Milano, F. *Power system modelling and scripting.* Power Systems. Springer, London, 2010.

[47] Monticelli, A. J. *Fluxo de Carga em Redes de Energia Elétrica.* Edgard Blücher Ltda, São Paulo, 1983.

[48] Moore, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (1965), 114–117.

[49] Moore, S. K. A better way to measure progress in semiconductors. Disponível em `https://spectrum.ieee.org/semiconductors/devices/a-better-way-to-measure-progress-in-semiconductors`. Acessado em 10/12/2020.

[50] Mudge, T. Power: A First Class Design Constraint for Future Architectures. In *High Performance Computing — HiPC 2000* (Berlin, Heidelberg, 2000), M. Valero, V. K. Prasanna, and S. Vajapeyam, Eds., Lecture Notes in Computer Science, Springer, pp. 215–224.

[51] Nijholt, A. *Computers and languages: theory and practice.* No. 4 in Studies in computer science and artificial intelligence. North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, Amsterdam ; New York : New York, N.Y., U.S.A, 1988.

[52] Nilsson, J. W.; Riedel, S. A. *Circuitos Elétricos*, 10 ed. Pearson Education do Brasil Ltda, São Paulo, 2016.

[53] Nvidia. Cuda c programming guide. Disponível em `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Acessado em 15/12/2020.

[54] Nvidia. Nvidia cuda compute unified device architecture programming guide version 1.0, 2007. Disponível em `http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`.

[55] Nvidia. cuSOLVER Library, Accessed November 16, 2020. Disponível em http://docs.nvidia.com/cuda/cusolver/index.html.

[56] Nvidia. Cuda runtime api, Accessed November 18, 2020. Disponível em https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[57] of Transmission System Operators for Electricity, E. N. Entsoe at a glance. Disponível em https://eepublicdownloads.entsoe.eu/clean-documents/Publications/ENTSO-E%20general%20publications/entsoe_at_a_glance_2015_web.pdf. Acessado em 25/12/2020.

[58] OMPstd. Openmp application programming interface. Disponível em https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Acessado em 20/03/2021.

[59] ONS. O sistema interligado nacional. Disponível em http://www.ons.org.br/paginas/sobre-o-sin/o-que-e-o-sin. Acessado em 20/03/2021.

[60] Overbye, T. J. Reengineering the electric grid. *American Scientist 88*, 3 (2000), 220.

[61] Patterson, D. A.; Hennessy, J. L. *Organização e projeto de computadores: interface hardware/software.* Elsevier, 2014.

[62] Peddie, J. *The history of visual magic in computers: how beautiful images are made in CAD, 3D, VR and AR.* Springer, London ; New York, 2013.

[63] Powell, L. *Power system load flow analysis.* McGraw-Hilll, New York; London, 2005.

[64] Press, W. H., Ed. *Numerical recipes: the art of scientific computing*, 3rd ed ed. Cambridge University Press, Cambridge, UK ; New York, 2007.

[65] Roberge, V.; Tarbouchi, M.; Okou, F. Parallel Power Flow on Graphics Processing Units for Concurrent Evaluation of Many Networks. *IEEE Transactions on Smart Grid 8*, 4 (July 2017), 1639–1648. Conference Name: IEEE Transactions on Smart Grid.

[66] Saad, Y. *Iterative methods for sparse linear systems*, 2nd ed ed. SIAM, Philadelphia, 2003.

[67] Sathre, P.; Gardner, M.; Feng, W. Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation. In *2012 41st International Conference on Parallel Processing Workshops* (Sept. 2012), pp. 89–96. ISSN: 2332-5690.

[68] Segal, M.; Akeley, K. The opengl graphics system: A specification, 2019. Disponível em https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf. Acessado em 18/12/2020.

[69] Silberschatz, A.; Gagne, G.; Galvin, P. B. *Operating System Concepts, Enhanced Edition.* Wiley., 2018.

[70] Sipser, M.; Queiroz, R. J. G. B. d.; Vieira, N. J. *Introdução à teoria da computação*. Thomson Learning, São Paulo, 2007.

[71] Soyata, T. *GPU parallel program development using CUDA*. CRC Press, Boca Raton, Florida, 2018.

[72] Stewart, J. *Multivariable calculus*, 7th ed ed. Brooks/Cole Cengage Learning, Belmont, CA, 2012.

[73] Stott, B. Review of load-flow calculation methods. *Proceedings of the IEEE 62*, 7 (July 1974), 916–929. Conference Name: Proceedings of the IEEE.

[74] Stott, B.; Alsac, O. Fast Decoupled Load Flow. *IEEE Transactions on Power Apparatus and Systems PAS-93*, 3 (May 1974), 859–869. Conference Name: IEEE Transactions on Power Apparatus and Systems.

[75] Stubbe, M.; Karoui, K.; Van Cutsem, T.; Wehenkel, L. Le projet PEGASE. *Revue E: Revue d'Electricité et d'Electronique Industrielle*, 4 (Dec. 2008).

[76] Su, X.; He, C.; Liu, T.; Wu, L. Full Parallel Power Flow Solution: A GPU-CPU-Based Vectorization Parallelization and Sparse Techniques for Newton–Raphson Implementation. *IEEE Transactions on Smart Grid 11*, 3 (May 2020), 1833–1844. Conference Name: IEEE Transactions on Smart Grid.

[77] Wan, Y.-h.; Parsons, B. K. Factors relevant to utility integration of intermittent renewable technologies. Tech. Rep. NREL/TP-463-4953, National Renewable Energy Lab., Golden, CO (United States), Aug. 1993.

[78] Wang, E.; Zhang, Q.; Shen, B.; Zhang, G.; Lu, X.; Wu, Q.; Wang, Y. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer International Publishing, Cham, 2014, pp. 167–188.

[79] Weisstein, E. W. *CRC concise encyclopedia of mathematics*, 2 ed. Chapman & Hall/CRC, Boca Raton, 2003.

[80] Yoon, D.-H.; Han, Y. Parallel power flow computation trends and applications: A review focusing on gpu. *Energies 13*, 9 (May 2020), 2147.

[81] Zhou, G.; Feng, Y.; Bo, R.; Chien, L.; Zhang, X.; Lang, Y.; Jia, Y.; Chen, Z. GPU-Accelerated Batch-ACPF Solution for N-1 Static Security Analysis. *IEEE Transactions on Smart Grid 8* (May 2017), 1406–1416. Conference Name: IEEE Transactions on Smart Grid.

[82] Zimmerman, R. D.; Murillo-Sánchez, C. E.; Thomas, R. J. MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education. *IEEE Transactions on Power Systems 26*, 1 (Feb. 2011), 12–19. Conference Name: IEEE Transactions on Power Systems.

APÊNDICE A – Tempos de Execução

Este apêndice traz tabelas que organizam medições tomadas durante os testes apresentados no Capítulo 5. Todos os valores de tempo estão em milissegundos e são médias de 100 medições, exceto nos casos onde o nome do sistema estiver marcado com um asterisco (*). As colunas $\boldsymbol{\sigma}_{\%}$ organizam os desvios padrão percentuais para todas as medidas de tempo totais por todo o trecho analisado, cujo valor médio é organizado sob as colunas **Total**. Colunas **Nº Iterações** organizam o número de iterações envolvido no processo iterativo. As demais colunas apresentam valores de tempo médio demandado. Enquanto a coluna **Ybus** organiza os valores relativos ao cálculo da matriz de admitãncias nodais, a coluna **calcPQ** é relativa à etapa de cálculo das injeções de potência ativa e reativa nodais. A coluna **Jacobiano** é referente ao tempo demandado para cálculo da matriz Jacobiano e a coluna **Sist. Lin.** refere à etapa de solução direta do sistema linear. O tempo médio demandado para o cálculo do fluxo de potência em ramos está sob as colunas **Fluxo**.

Nas planilhas referentes a programas que utilizam a GPU, as colunas **Init. GPU** são referentes à alocação inicial de memória na GPU e transferência dos dados iniciais para as posições alocadas. Já as colunas **Init. Lib.** contém os tempos médios demandados para a inicialização das estruturas exigidas pela biblioteca cuSolver.

A.1 Programas que utilizam unicamente a CPU

A.1.1 Programas Sequenciais

Tabela A.1: Tempo demandado para a execução das etapas do programa com abordagem *densa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	1,412212	11,54373	3	0,66741	0,40814	0,41825	1,283765	0,455908
30	2,328795	7,333478	3	0,13417	0,128026	0,180674	2,024345	0,720439
57	5,386792	5,333776	3	0,252738	0,379345	0,698807	4,313599	0,123501
118	18,97267	20,34707	4	0,53361	2,087207	3,035438	13,85438	0,262524
118x2	70,84646	1,465848	4	0,108684	8,559444	11,55007	50,58542	0,507871
118x4	289,2376	0,411587	4	0,229845	32,49622	46,69395	209,7256	0,784216
118x8	1271,801	0,356239	4	0,459584	133,205	188,5418	949,3766	0,206144
118x16	7172,096	0,870963	5	0,969761	710,4729	959,8556	5500,208	0,474542
118x32	34521,97	0,129142	5	1,950651	3797,776	3785,225	26935,67	1,357475
118x64	203928,3	0,192772	5	4,705779	29207,02	15138,88	159574,7	2,290952

Tabela A.2: Tempo demandado para a execução das etapas do programa com abordagem *densa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	1,34198	1,343146	3	0,652101	0,328479	0,419961	1,224593	0,390227
30	2,151199	5,70174	3	0,127218	0,184327	0,150364	1,823792	0,623107
57	4,733198	0,427754	3	0,238	0,315302	0,56969	3,780022	0,102406
118	15,34674	1,615501	4	0,536211	1,533943	2,162241	11,20971	0,218628
118x2	56,36381	0,565782	4	0,107876	6,045791	8,624348	41,55143	0,419655
118x4	228,6837	0,200341	4	0,219228	25,42166	34,74431	168,259	0,814339
118x8	998,3496	0,144923	4	0,448676	99,08894	145,1769	753,5433	0,16407
118x16	5485,945	0,189908	5	0,909359	523,6149	733,4191	4227,565	0,359994
118x32	24817,23	0,060568	5	1,860283	2334,193	2927,08	19553,01	0,844792
118x64	127693,4	0,120764	5	3,838316	15347,98	11685,54	100653,7	1,738179
118x128*	1070446	1,803723	5	8,713644	79561,93	46959,44	943909,1	5,178262

Tabela A.3: Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	1,945452	11,51782	3	0,460393	0,223172	0,670406	1,727791	0,126851
30	2,583604	5,993106	3	0,859456	0,337233	0,165805	2,220139	0,219845
57	3,285121	8,580848	3	0,17325	0,527111	0,28719	2,657279	0,374313
118	5,139965	6,689821	4	0,352152	0,140125	0,575809	3,865996	0,713262
118x2	8,35672	3,641298	4	0,700122	0,272576	1,545428	6,014512	0,17916
118x4	15,5513	3,150607	4	1,398201	0,52673	2,316701	10,94085	0,322878
118x8	29,75249	1,724835	4	2,803273	1,160579	4,271836	20,8842	0,630265
118x16	70,02365	1,234774	5	5,568045	2,410811	10,50015	50,11991	1,232178
118x32	140,4519	0,93627	5	11,26705	4,795238	20,74854	100,7437	2,535361
118x64	283,5986	0,851335	5	22,20276	9,476042	40,95365	205,3306	4,842146
118x128	555,7912	0,446439	5	44,18435	18,55355	82,53587	399,4092	9,542398

Tabela A.4: Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *sequencial* na CPU e que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	**Total**	$\sigma_{\%}$	**Nº Iterações**	**Ybus**	**calcPQ**	**Jacobiano**	**Sist. Lin.**	**Fluxo**
14	1,959061	11,1959	3	0,466339	0,164468	0,684947	1,744098	0,118044
30	2,587599	6,073796	3	0,861931	0,286446	0,187398	2,207071	0,225518
57	3,290292	8,356096	3	0,165294	0,463473	0,308844	2,660073	0,39748
118	5,228443	6,74439	4	0,370033	0,133131	0,636311	3,822863	0,716484
118x2	8,512515	4,680121	4	0,729043	0,251984	1,230462	6,073943	0,181674
118x4	15,68782	2,945396	4	1,446845	0,486041	2,399853	10,91792	0,339966
118x8	36,05704	1,401331	5	2,868341	1,566248	5,885908	25,35965	0,664679
118x16	69,7605	1,291058	5	5,660945	2,341426	11,52986	48,84924	1,331277
118x32	141,4053	1,135721	5	11,46233	4,335109	22,85669	99,83039	2,639462
118x64	284,4891	0,965422	5	22,5111	8,413663	45,17453	202,527	5,378586
118x128	565,2752	0,510612	5	44,81246	16,59315	90,68394	401,7239	10,37039

A.1.2 Programas Paralelos

Tabela A.5: Tempo demandado para a execução das etapas do programa com abordagem *densa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	1,61727	33,84962	3	0,669205	0,282384	0,251354	1,275336	0,356092
30	13,50072	17,84342	3	0,31774	0,344359	0,3121	12,50286	0,23933
57	17,32994	21,45378	3	0,499022	0,487911	0,459474	15,55784	0,615223
118	21,047	17,10228	4	0,670151	0,798432	1,031736	18,83572	0,703332
118x2	40,02675	5,601932	4	0,11151	2,484011	3,426758	33,99099	0,155688
118x4	116,9706	1,267704	4	0,231038	8,800605	12,33581	95,26854	0,27998
118x8	492,7795	1,171556	4	0,460156	35,72122	50,31445	406,0563	0,537622
118x16	2679,976	9,291398	5	0,938325	201,6624	257,3023	2217,842	0,203037
118x32	12865,69	0,932972	5	1,947471	1188,751	1001,814	10672,41	0,289276
118x64	75993,35	0,40688	5	4,681408	8065,237	4002,219	63919,87	0,599896

Tabela A.6: Tempo demandado para a execução das etapas do programa com abordagem *densa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	calcPQ	Jacobiano	Sist. Lin.	Fluxo
ieee14	14,6221	12,36052	3	0,223336	0,797966	0,800829	13,99762	0,127332
ieee30	14,26857	13,04292	3	0,441804	0,145379	0,188186	13,50833	0,150978
ieee57	18,43169	11,5223	3	0,823878	0,337386	0,548503	17,13269	0,1781
ieee118	32,41922	9,841689	4	0,196706	1,532686	2,156222	28,17997	0,2785
ieee118x2	30,28785	6,125927	4	0,113185	1,594643	2,325599	26,01086	0,133111
ieee118x4	89,13878	1,053272	4	0,219211	6,763523	9,222228	72,86867	0,237912
ieee118x8	346,1167	0,429573	4	0,450136	26,6356	38,90564	279,8619	0,449251
ieee118x16	1830,445	0,38723	5	0,899709	138,9943	193,1199	1497,061	0,900377
ieee118x32	8563,104	0,439141	5	1,840984	873,6238	768,8855	6918,237	0,204249
ieee118x64	47291,2	0,159665	5	3,81357	4452,309	3070,734	39763,06	0,46683
ieee118x128*	362606,1	0,263603	5	8,718737	22933,13	12298,31	327363,7	1,26403

Tabela A.7: Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	11,81389	18,6025	3	0,46238	0,233893	0,436411	11,51294	0,595784
30	11,99057	19,50256	3	0,886143	0,274774	0,823739	11,5816	0,933645
57	12,32851	21,48765	3	0,166155	0,355807	0,131599	11,79267	0,142626
118	13,80984	20,65403	4	0,360358	0,545664	0,213749	12,94616	0,25618
118x2	17,2061	14,66583	4	0,711962	0,86183	0,40484	15,67267	0,474613
118x4	23,25142	9,360426	4	1,379565	0,203685	0,768875	20,65433	0,876413
118x8	33,27755	6,541908	4	2,794642	0,348816	1,518747	28,13674	0,168636
118x16	62,04767	2,2742	5	5,590749	0,678475	3,465577	51,65686	0,324801
118x32	114,0636	1,243615	5	10,79632	1,260063	6,624993	94,35678	0,645804
118x64	213,7375	0,353852	5	21,40079	2,519084	13,44548	174,8234	1,263254
118x128	456,8282	2,227756	5	42,66878	5,093386	26,60008	379,2882	2,507564

Tabela A.8: Tempo demandado para a execução das etapas do programa com abordagem *esparsa* e *paralela* na CPU, que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	**Total**	$\sigma_{\%}$	**Nº Iterações**	**Ybus**	**calcPQ**	**Jacobiano**	**Sist. Lin.**	**Fluxo**
14	12,42715	16,14317	3	0,47748	0,145225	0,414037	12,11069	0,562727
30	12,54823	19,0375	3	0,899683	0,22919	0,810721	12,02921	0,840621
57	13,37149	18,19998	3	0,170344	0,301962	0,13226	12,68892	0,148133
118	15,86961	14,62518	4	0,367265	0,575392	0,222077	14,57982	0,280232
118x2	18,4478	15,06979	4	0,723158	0,815813	0,431361	16,39741	0,473514
118x4	24,44293	6,362537	4	1,431299	0,171344	0,822875	21,6904	0,864937
118x8	38,51336	6,348409	5	2,852325	0,372797	1,789874	32,9889	0,181255
118x16	61,41435	2,433476	5	5,650544	0,623046	3,57665	50,89654	0,336302
118x32	113,5642	0,911371	5	11,4357	1,582609	6,963912	93,41026	0,663468
118x64	210,4977	0,240533	5	21,82363	2,217601	13,59219	171,0677	1,334818
118x128	443,3651	0,839348	5	43,53727	4,59183	27,42631	364,5262	2,625849

A.2 Programas que utilizam a GPU

Tabela A.9: Tempo demandado para a execução das etapas do programa com abordagem *densa* na GPU e que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	Inic. GPU	Inic. Bib.	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	234,6287	1,368324	3	0,010541	0,293904	156,6657	2,491053	0,12167	0,393948	0,04104
30	237,8904	1,094663	3	0,020265	0,295409	156,4659	5,277361	0,142337	0,846072	0,041842
57	244,2487	1,310243	3	0,037934	0,298998	156,255	9,97557	0,1501	1,763737	0,041249
118	262,3533	1,073476	4	0,085145	0,299436	156,3187	25,85292	0,263108	4,222999	0,041999
118x2	297,7194	0,917446	4	0,17062	0,364314	156,3144	52,32476	0,571541	12,45968	0,042266
118x4	371,6638	0,969329	4	0,34216	0,470768	155,876	105,0818	1,594953	31,8449	0,044107
118x8	572,9656	0,51905	4	0,682687	0,682849	156,1046	214,6139	5,568014	118,4168	0,048064
118x16	1600,327	0,238389	5	1,386792	1,559291	155,2956	526,3396	26,76212	808,1006	0,055246
118x32	6993,124	0,083601	5	2,876652	4,989572	154,0538	1080,408	106,3025	5546,091	0,079139

Tabela A.10: Tempo demandado para a execução das etapas do programa com abordagem *densa* na GPU e que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	Inic. GPU	Inic. Bib.	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	234,2533	1,272279	3	0,008757	0,289203	156,5445	2,470528	0,104021	0,338619	0,0412
30	237,0388	1,188931	3	0,017033	0,290191	156,3325	5,265621	0,110418	0,700183	0,041927
57	242,8851	1,484134	3	0,032296	0,293281	156,5007	9,959189	0,115009	1,392781	0,041527
118	260,7865	1,215665	4	0,071947	0,302929	156,273	25,69697	0,18454	3,231149	0,041529
118x2	292,8882	1,012319	4	0,142918	0,30274	156,1794	51,68485	0,313813	8,711781	0,041713
118x4	358,868	0,87302	4	0,289455	0,377054	156,2339	105,0356	0,935695	19,78136	0,042587
118x8	493,3715	0,67724	4	0,577538	0,542595	155,9545	212,8985	3,026883	43,78456	0,043284
118x16	935,4378	0,407404	5	1,163523	0,985032	155,4336	523,4537	14,26354	161,6384	0,048044
118x32	1991,425	0,330408	5	2,423472	2,708919	154,8605	1070,894	59,31828	614,4685	0,055738
118x64	5601,334	0,253354	5	4,947685	9,60382	153,2124	2208,108	240,7418	2860,498	0,093949

Tabela A.11: Tempo demandado para a execução das etapas do programa com abordagem *esparsa* na GPU e que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	Inic. GPU	Inic. Bib.	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	356,7028	0,72742	3	0,042037	0,413435	354,1584	0,119027	0,088864	1,549897	0,019825
30	357,3488	0,600141	3	0,07372	0,416716	353,3931	0,146021	0,11071	2,878473	0,020911
57	359,6635	0,59772	3	0,129002	0,41617	353,1008	0,139362	0,144518	5,400144	0,021495
118	366,701	0,564352	4	0,270911	0,40897	353,5256	0,179612	0,191469	11,74861	0,024002
118x2	378,3057	0,688414	4	0,529358	0,413025	353,0405	0,183194	0,290645	23,44819	0,025807
118x4	414,3131	0,866958	4	1,043385	0,415012	353,2902	0,190241	0,484372	58,43898	0,025926
118x8	539,8807	0,436765	4	2,109092	0,425677	353,4218	0,198285	0,864135	182,3366	0,024371
118x16	1481,235	0,266647	5	4,269368	0,46224	354,0891	0,298858	1,638316	1119,797	0,028184
118x32	8452,502	0,210274	5	8,54452	0,605708	353,0358	0,43014	3,167745	8085,894	0,045469
118x64	77356,88	0,257766	5	17,15046	0,781055	353,2197	0,802837	6,332797	76978,73	0,075387
118x128	683701,2	0,112717	5	34,58023	1,159634	352,6239	1,44769	13,31628	683308,9	0,150732

Tabela A.12: Tempo demandado para a execução das etapas do programa com abordagem *esparsa* na GPU e que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	Total	$\sigma_{\%}$	Nº Iterações	Ybus	Inic. GPU	Inic. Bib.	calcPQ	Jacobiano	Sist. Lin.	Fluxo
14	356,6314	0,649326	3	0,043262	0,410895	354,2635	0,112773	0,087159	1,389688	0,019282
30	357,0267	0,57614	3	0,076183	0,412184	353,6139	0,113838	0,106673	2,374267	0,019544
57	359,6436	0,630352	3	0,133994	0,414402	354,1665	0,114447	0,139684	4,343166	0,019859
118	362,2648	0,651991	3	0,281526	0,410997	353,9376	0,114605	0,167233	7,011386	0,021607
118x2	373,6843	0,560835	4	0,55855	0,410533	353,3901	0,139584	0,27741	18,52372	0,022718
118x4	402,4032	0,59635	4	1,083266	0,411937	353,4723	0,149854	0,460557	46,39902	0,023153
118x8	499,278	0,569799	4	2,163632	0,419532	353,4049	0,155284	0,819399	141,839	0,020811
118x16	1213,195	0,26857	5	4,361256	0,463564	353,3289	0,186004	1,552386	852,7124	0,021308
118x32	5721,169	0,235016	5	8,72469	0,496111	357,4869	0,204242	2,979322	5350,608	0,023232
118x64	52224,56	0,488097	5	17,50426	0,682827	365,0725	0,291668	5,915983	51835,16	0,037354
118x128	465362	0,079983	5	34,99293	1,042039	352,8165	0,454134	12,144	464969,5	0,072431

Tabela A.13: Tempo demandado para a execução das etapas do programa com abordagem *híbrida* e que faz uso de valores de ponto flutuante com precisão dupla.

Sistema IEEE	**Total**	$\sigma_{\%}$	**Nº Iterações**	**Ybus**	**Inic. GPU**	**Inic. Bib.**	**calcPQ**	**Jacobiano**	**Sist. Lin.**	**Fluxo**
14	3,611457	1,650525	3	0,042159	0,38607	0,000762	0,121886	0,089698	2,630202	0,020939
30	4,161379	2,91223	3	0,073802	0,391216	0,000697	0,132809	0,105618	3,122397	0,018974
57	4,690145	9,819771	3	0,129565	0,386626	0,000697	0,137096	0,143164	3,548327	0,0213
118	6,107298	5,086444	4	0,273519	0,384277	0,000704	0,182361	0,191408	4,687831	0,022884
118x2	8,577658	3,555741	4	0,531526	0,384634	0,000694	0,184369	0,291104	6,778331	0,02512
118x4	13,95019	1,213484	4	1,046605	0,393372	0,000711	0,185966	0,49611	11,38897	0,024433
118x8	22,882	0,801684	4	2,117044	0,41363	0,000711	0,197558	0,914195	18,71615	0,027805
118x16	49,00588	0,286636	5	4,281586	0,444546	0,000707	0,305911	1,75349	41,50353	0,032599
118x32	98,62878	0,25383	5	8,585587	0,549988	0,000763	0,456166	3,402413	84,57927	0,053815
118x64	194,1982	0,313893	5	17,20024	0,745071	0,000734	0,887299	6,83557	166,8855	0,087561
118x128	412,61	2,425626	5	34,53859	1,116233	0,000721	1,535035	13,55537	359,1083	0,158742

Tabela A.14: Tempo demandado para a execução das etapas do programa com abordagem *híbrida* e que faz uso de valores de ponto flutuante com precisão simples.

Sistema IEEE	**Total**	$\sigma_{\%}$	**Nº Iterações**	**Ybus**	**Inic. GPU**	**Inic. Bib.**	**calcPQ**	**Jacobiano**	**Sist. Lin.**	**Fluxo**
14	3,619463	1,388036	3	0,043395	0,382214	0,000728	0,115731	0,084754	2,659577	0,019113
30	4,169179	2,143208	3	0,077624	0,381699	0,00071	0,118083	0,103564	3,154085	0,018148
57	4,660965	1,848281	3	0,134209	0,386638	0,000711	0,118166	0,136157	3,541926	0,020475
118	5,338658	1,159099	3	0,28276	0,384975	0,000708	0,11643	0,16432	4,04827	0,019596
118x2	8,48866	0,563635	4	0,551179	0,384291	0,000726	0,141803	0,276604	6,744852	0,021532
118x4	13,90041	3,487575	4	1,089688	0,391526	0,000724	0,143524	0,46028	11,40973	0,021231
118x8	22,78043	2,162861	4	2,174001	0,448844	0,000722	0,148456	0,870755	18,6834	0,021499
118x16	40,97295	0,357323	4	4,378841	0,44637	0,000761	0,149632	1,651554	33,80427	0,022692
118x32	97,59993	0,268128	5	8,741775	0,482702	0,00071	0,198893	3,237168	84,14436	0,025529
118x64	192,3364	1,068447	5	17,51015	0,632134	0,000757	0,293456	6,370931	166,3513	0,042042
118x128	409,8324	2,762274	5	35,18285	1,001983	0,000762	0,455261	12,90081	358,4126	0,068158