UNIVERSIDADE FEDERAL FLUMINENSE

Ayres Nishio da Silva Junior

Uso de GPUs para Identificação Concorrente de Criticalidades em Sistemas de Medição

NITERÓI

2021

UNIVERSIDADE FEDERAL FLUMINENSE

Ayres Nishio da Silva Junior

Uso de GPUs para Identificação Concorrente de Criticalidades em Sistemas de Medição

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: Computação Científica e Sistemas de Potências

Orientadores:
Milton Brown Do Couto Filho e Julio Cesar Stachinni De Souza

NITERÓI

2021

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

S586u Silva junior, Ayres Nishio da
Uso de GPUs para Identificação Concorrente de Criticalidades em Sistemas de Medição / Ayres Nishio da Silva junior ; Milton Brown Do Coutto Filho, orientador ; Julio Cesar Stachinni De Souza, coorientador. Niterói, 2021.
96 f. : il.

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

DOI: http://dx.doi.org/10.22409/PGC.2021.m.15107816762

1. Criticalidades. 2. GPU. 3. Combinatorial. 4. Estimação de Estados. 5. Produção intelectual. I. Do Coutto Filho, Milton Brown, orientador. II. De Souza, Julio Cesar Stachinni, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecário responsável: Sandra Lopes Coelho - CRB7/3389

Ayres Nishio da Silva Junior

Uso de GPUs para Identificação Concorrente de Criticalidades em Sistemas de Medição

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: Computação Científica e Sistemas de Potência

Aprovada em 11 de fevereiro de 2021.

BANCA EXAMINADORA

Prof. Milton Brown Do Coutto Filho - Orientador, UFF

Prof. Julio Cesar Stachinni de Souza - Orientador, UFF

Prof. Esteban Walter Gonzalez Clua, UFF

Prof. Djalma Mosqueira Falcão, COPPE/UFRJ

Niterói

2021

Agradecimentos

Agradeço a meus pais Ayres e Ana Cristina por todo suporte, amor e inspiração. Agradeço a meus orientadores Milton Brown e Julio Stachinni por toda disposição e ensinamentos. Agradeço ao meu supervisor Luiz Adriano pela ideia de utilizar GPUs, pela confiança e o auxílio financeiro pela bolsa do Cepel. Agradeço a todos os meus companheiros de mestrado, especialmente Rafael e Vinícius, por todos os assistência e conversas. Por fim, agradeço também aos meus amigos próximos do "Galere"por todo suporte e por tornar a vida mais divertida durante toda caminha deste trabalho

Resumo

A Estimação de Estado é uma das principais funções da operação em tempo real de sistemas elétricos de potência. Porém, para que a estimação ocorra adequadamente torna-se necessário um sistema de medição capaz de observar a rede elétrica como um todo, considerando indisponibilidades de seus elementos. Em uma etapa que precede a estimação em si (filtragem do estado), a análise de criticalidades visa identificar vulnerabilidades no esquema de medição, distinguindo combinações de medidas, formando tuplas de diferentes cardinalidades, que sejam essenciais para a observabilidade da rede. No entanto, a identificação das criticalidades é um problema de natureza combinatorial, que requer um tempo computacional elevado devido ao grande número casos a serem analisados. Contudo, deseja-se uma busca rápida, em um tempo viável, por conta da atuação das funções de operação em tempo real do sistema. Objetivando a agilização do processo de busca por criticalidades de diferentes cardinalidades, esta Dissertação propõe uma abordagem que utiliza a programação paralela de CPU e GPU, em que combinações de medidas são avaliadas concorrentemente. Também são apresentadas adaptações da abordagem proposta para otimizar o desempenho do algoritmo implementado para a análise de criticalidades. Resultados numéricos de simulações envolvendo sistemas para testes apontados na literatura como referência evidenciam a eficiência da abordagem proposta em GPUs, em que *speed-ups* de até 40x foram alcançados, quando comparados à implementação sequencial clássica que utiliza unicamente CPUs. Isto posto, este trabalho demonstra a aptidão da programação paralela em GPUs para tratar o problema da análise de criticalidades, servindo como um ponto de partida para pesquisas futuras.

Palavras-chave: Estimação de Estado, observabilidade, criticalidades, medidas, programação paralela, GPU.

Abstract

State Estimation (SE) is one of the main functions necessary for electric power systems' real-time operation. It requires a measurement plan capable of adequately observing the electrical network, even under its elements' unavailability. In a previous estimation step (before state filtering), criticality analysis aims to identify vulnerabilities in the metering system, characterized by combinations of measurements forming tuples of different cardinalities, essential for the network's observability. To identify criticalities is a combinatorial nature problem that requires sizeable computational time due to many possible tuples to analyze. Faster criticality analyses are always desirable, considering that SE is in a real-time operating environment. This Dissertation aims to speed-up criticality analysis by proposing an approach via parallel programming in CPU and GPU, where the combinations of measurements are evaluated concurrently. Adaptations of the proposed approach are also presented to optimize the algorithm's performance for criticality analysis. Numerical results of simulations involving benchmark systems show the efficiency of the proposed approach in GPU, in which speed-ups of up to 40x were achieved compared with the original sequential implementation in CPU only. The present study demonstrates the viability of parallel programming in GPUs to address criticality analysis, serving as a starting point for future researches.

Keywords: State Estimation, observability, criticalities, measurements, parallel programing, GPUs.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

COS	:	Centros de Operação de Sistemas;
SGE	:	Sistemas de Gerenciamento de Energia;
UM	:	Unidade Medição;
UTR	:	Unidade Terminal Remota;
DEI	:	Dispositivo Eletrônico Inteligente;
UMF	:	Unidade de Medição Fasodial;
SCADA	:	*Surpevisory Control and Data Acquisition*;
EE	:	Estimação de Estado;
CAD	:	Computação de Alto Desempenho;
GPU	:	Unidade de Processamento Gráfico;
SEP	:	Sistemas Elétricos de Potência;
EG	:	Erro Grosseiro;
C_{meas}	:	Medida Crítica;
C_k	:	K-Tupla Crítica;
B&B	:	*Branch and Bound*
CPU	:	Unidade Central de Processamento;
CUDA	:	Arquitetura Unificada de Dispositivos;
ALU	:	Unidades Lógicas Aritméticas;
SM	:	*Streaming Multiprocessos*;

Sumário

Capítulo 1

Introdução

1.1 Considerações Gerais

A operação de um sistema moderno de energia elétrica é realizada de forma hierarquizada em Centros de Operação de Sistemas (COS), com o auxílio de avançadas ferramentas computacionais reunidas em um Sistema de Gerenciamento de Energia (SGE) [1].

Em um regime de operação caracterizado pelo equilíbrio entre carga e geração, o estado operativo do sistema é definido pelas tensões nodais complexas, expressas em módulos e ângulos de fase, correspondentes a uma determinada configuração da rede elétrica [3].

Usualmente, o estado não é observado diretamente, e sim calculado a partir de um conjunto de medidas redundantes, convencionalmente conhecidas por medidas convencionais, coletadas através de Unidades de Medição (UMs) denominadas Unidades Terminais Remotas (UTRs) e Dispositivos Eletrônicos Inteligentes (DEIs), que reúnem comumente medidas de fluxo/injeção de potência ativa/reativa e magnitudes de tensão. Existem também as Unidades de Medição Fasorial (UMFs) que recolhem os valores de tensões nodais e correntes nos ramos da rede, expressos por seus ângulos de fase e magnitudes, referidas como medidas fasoriais ou sincrofasores.

As medidas convencionais coletadas são reunidas pelo sistema SCADA (*Supervisory Control and Data Acquisition*) e fornecidas ao COS, onde devem ser processadas e averiguadas de modo a atestar sua consistência. Para este fim, dentre os aplicativos que compõem o SGE, o Estimador de Estado é responsável por filtrar erros estatisticamente pequenos inerentes à medição e detectar/identificar medidas portadoras de erros grosseiros (EGs). O Estimador fornece como resultado o estado de operação mais provável, para uma determinada configuração da rede, que será utilizado pelas demais funções integran-

tes do SGE, dentre estas a análise de contingências, fluxo de potência ótimo e despacho econômico, como se vê na Figura 1.1.

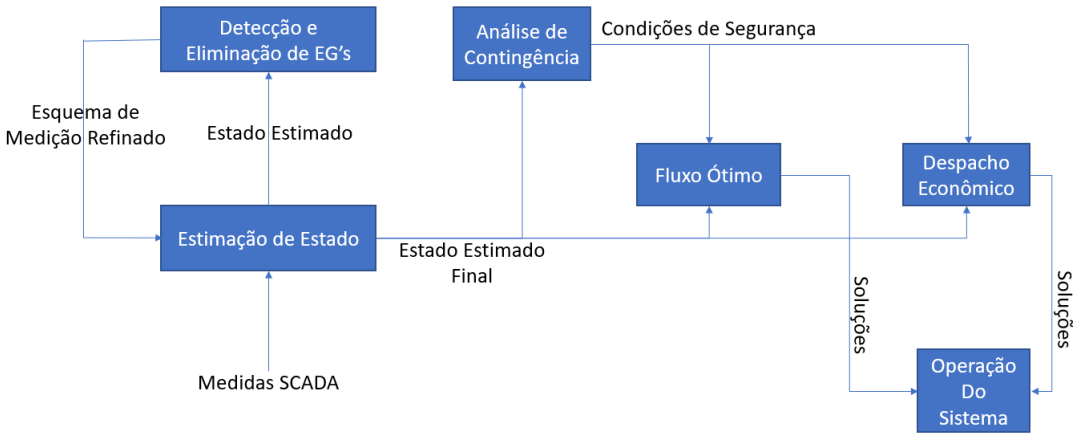

Figura 1.1: Funções do SGE [28]

Devido ao fato de a estimação de estado em sistemas de potência (EE) comumente adotar o método dos mínimos quadrados ponderados (MQP) [1], a qualidade dos resultados alcançados é primordialmente influenciada pela quantidade de medidas redundantes para a obtenção do estado da rede. Sendo assim, para garantir um bom desempenho da EE, o esquema de medição deve atender os seguintes requisitos [3]:

1. Observabilidade - Garantir que o estado de toda rede seja estimado
2. Qualidade - Permitir que a estimação seja alcançada com adequada precisão.
3. Confiabilidade - Possibilitar a detecção, identificação e substituição de medidas espúrias.
4. Robustez - Assegurar que os requisitos anteriores sejam atendidos, mesmo que haja indisponibilidade de medidas.

Apesar da busca permanente pelo atendimento de todos estes requisitos, restrições financeiras e operacionais podem acarretar condições adversas que impossibilitem o suprimento de medições, que consequentemente, venham comprometer o processo de EE. Dentre estas possíveis adversidades, pode-se citar a indisponibilidade de UMs, perda de canais de comunicação, ataques cibernéticos, desastres naturais e alterações topológicas.

Em um processo prévio à EE propriamente dita (etapa de filtragem), a análise convencional de observabilidade visa determinar se o conjunto de medidas disponíveis é suficiente ou não (análise binária) para que o estado do sistema seja estimado como um todo. Sendo assim, nenhuma conclusão pode ser alcançada em relação à qualidade do esquema de medição e a confiabilidade da EE. Alternativamente, uma análise mais ampla pela identificação da criticalidade de elementos da rede, se apresenta como uma forma de quantizar a capacidade de observação do plano de medição, trazendo informações sobre quão próximo um sistema se encontra de uma possível perda de observabilidade e que capacidade apresenta para tratar EGs.

Partindo deste princípio, a análise de criticalidades tem como objetivo identificar elementos, tomados individualmente ou formando tuplas de diferentes cardinalidades, que são essenciais (críticos) para que a rede seja observável como um todo, o que possibilita a realização do processo de EE. Sendo assim, é de interesse que esta análise seja realizada de forma rápida e eficiente, pois as condições de operação do sistema mudam constantemente devido esquemas de manutenção preventiva/corretiva e interrupções fortuitas [37].

Contudo, a busca por combinações de medidas que venham a se tornar indisponíveis constitui um problema difícil, de custo computacional elevado, com um número fatorial de possíveis casos a analisar. Além disto, a adição de novos equipamentos ao sistema, UMFs e DEIs, torna todos os processos e modelos envolvidos mais complexos e computacionalmente custosos [26]. Especificamente para a análise de criticalidades , conforme o sistema se expande e mais medidas são adicionadas, o número de combinações a serem avaliadas aumenta significantemente, tornado a avaliação mais custosa, inclusive para cardinalidades mais baixas.

Neste cenário, análises de criticalidades para redes elétricas de maior porte encontradas em sistemas reais podem ser realizadas utilizando-se computação paralela que se insere no campo da computação de alto desempenho (CAD). Dentre as tecnologias deste campo, as unidades de processamento gráfico (mais conhecida pela terminologia inglesa, *Graphical Processing Units*-GPUs) vem se destacando devido a seu baixo custo e potencial para processamento paralelo massivo [27]. Como será apresentado a seguir, GPUs podem ser empregadas para aprimorar o desempenho de implementações em sistemas de potência.

A presente Dissertação apresenta uma abordagem via programação paralela em GPUs para tornar mais eficiente a busca por criticalidades, basicamente fazendo com que as combinações de medidas sejam avaliadas simultaneamente.

1.2 Aplicações de GPUs em Sistemas de Potência

As redes de energia elétrica vêm evoluindo rapidamente com a adição de novas tecnologias que possibilitaram a agregação de fontes renováveis, resultando no que se entende por geração distribuída, o que demanda a utilização de unidades de medição inteligentes. Apesar desta evolução apontar para sistemas mais robustos para atender à crescente necessidade de energia elétrica, a complexidade dos modelos utilizados é incrementada, requerendo respostas e análises em tempo real. Isto posto, busca-se aqui o ganho de desempenho para realizar de forma mais eficiente tarefas de operação de sistemas elétricos de potência (SEP) através das tecnologias da área de CAD.

Dentre as novas tecnologias, os SEP vêm se beneficiando da programação paralela em GPU, pois boa parte das aplicações consistem em algoritmos altamente paralelizáveis os quais utilizam matrizes de grandes dimensões ou grandes amostras de dados independentes [26].

Alguns trabalhos que fizeram uso da GPU são mencionados a seguir. Em [29] encontra-se uma simulação de estabilidade transitória ágil/precisa em sistemas de grande escala. A referência [23] investiga a implementação paralela em unidades de processamento gráfico para a análise de contingência baseada em fluxo de potência CC. Já em [51], utiliza-se a programação paralela heterogênea (CPU/GPU) como forma de agilizar a análise simultânea de múltiplas instâncias de uma rede para ao processo de fluxo de potência.

Das aplicações mais próximas ao tema desenvolvido nesta Dissertação, podem-se destacar os trabalhos a seguir referidos. Em [10], há uma demonstração dos benefícios para a operação de redes de energia referentes ao aumento de desempenho, através da programação paralela, das funções de EE. As referências [50, 32] utilizam GPUs para acelerar o processo de EE. O primeiro trabalho visa melhorar o desempenho do método de Gauss-Newton utilizando implementações em CUDA. Já o segundo, utiliza a biblioteca de álgebra linear cuBLAS (CUDA *Basic Linear Algebra Subprograms*) para os processamentos matriciais de várias etapas da EE. Por fim, em [33] encontra-se uma implementação referente a uma EE dinâmica de alto desempenho com o auxílio de GPU para detectar ataques cibernéticos.

Como descrito, as GPUs são tecnologias já presentes na busca por aperfeiçoamento das técnicas e procedimentos aplicados aos SEP. Contudo, não existem aplicações referentes às análises de observabilidade e criticalidade presentes na EE. Sendo assim, o objetivo primordial desta Dissertação será introduzir a programação de GPUs na análise da criticalidade de elementos necessários à EE, servindo como base para futuras pesquisas na área.

1.3 Objetivos

A identificação das criticalidades de elementos que integrem um sistema de medição que atenda à EE permite avaliar de forma quantitativa a capacidade de tal sistema atender aos requisitos necessários a processos de estimação efetivos, isto é, informa a proximidade de uma possível perda de observabilidade e em que medida está apto para tratar EGs. Contudo, a natureza combinatória da análise de criticalidades requer um tempo computacional elevado, o que vem limitando de forma expressiva a extensão de estudos nesta área.

Assim sendo, o presente trabalho objetiva agilizar o processo de análise de criticalidades com a utilização da programação paralela, em que um sistema de medição é submetido a indisponibilidade de medidas, tomadas individualmente ou formando grupos. Para este fim, devido ao requisito de processamento paralelo massivo da solução proposta via matriz de covariância de resíduos, será introduzido a programação de GPUs na análise de criticalidades. O algoritmo desenvolvido apresenta-se suficientemente flexível para adaptações que visem aprimorar seu desempenho. Como o intuito do trabalho é avaliar a implementação da solução do problema com o emprego de novas tecnologias, uma enumeração não sofisticada, por *força bruta*, foi utilizada para comparar o desempenho da implementação da análise de criticalidades com o uso de CPUs e de GPUs.

1.4 Descrição dos Capítulos

A presente Dissertação está organizada da seguinte forma:

O Capítulo 1 introduz o problema da EE contextualizando sua aplicação na operação de sistemas de potência, bem como apresenta a análise de criticalidades, justificando a necessidade de sua rápida e eficiente execução.

O Capítulo 2 aborda os principais conceitos envolvidos no tema da Dissertação e realiza uma revisão bibliográfica das análises de observabilidade e de criticalidades.

O Capítulo 3 introduz conceitos da programação paralela, evidenciando as peculiaridades da utilização de GPU.

O Capítulo 4 descreve a análise de criticalidades pela matriz de covariância de resíduos, evidenciando as características desta abordagem que justificam sua implementação em GPUs. Posteriormente, são apresentadas adaptações realizadas no algoritmo para possibilitar a execução concorrente do método.

O Capítulo 5 sintetiza os resultados numéricos obtidos a partir de simulações nos sistemas para teste do IEEE de 14-, 30- e 118-barras, apresentando também otimizações realizadas na implementação visando aprimorar o desempenho.

O Capítulo 6 conclui o trabalho, discutindo os resultados e apresentando sugestões de implementações para trabalhos futuros.

O Apêndice A apresenta as demonstrações das propriedades relacionadas à análise de criticalidades.

O Apêndice B expõe os resultados obtidos de forma detalhada, em que são exibidos os tempos despendidos por cardinalidade analisada.

1.5 Publicações

Este trabalho de pesquisa originou a seguinte publicação durante seu desenvolvimento [15], onde foi realizada uma adaptação parcial do método para a arquitetura da GPU.

Capítulo 2

Análise de Criticalidades

Neste capítulo são abordados alguns conceitos relativos à EE, suas principais etapas e o modelo linear adotado, necessários para o desenvolvimento das análises de observabilidade e de criticalidades. Apresenta-se também uma revisão dos trabalhos encontrados na literatura especializada.

2.1 Estimação de Estado

Como descrito anteriormente, a EE realiza a tarefa de filtrar medidas cruas coletadas da rede elétrica para estimar o estado mais provável de operação do sistema, a ser utilizado por aplicativos computacionais que integram o SGE. O trabalho de Schweppe [52] foi o primeiro a abordar o problema da EE em sistemas de potência. Os principais conceitos utilizados neste capítulo encontram-se detalhados em Abur [1] e Monticelli [38].

2.1.1 Etapas

Convencionalmente, a EE inclui quatro etapas: Pré-processamento; configuração da rede; análise de observabilidade; filtragem e processamento dos erros grosseiros. A seguir, tais etapas são brevemente descritas:

- **Pré-processamento:** Avalia-se, de acordo com limites plausíveis, os valores das medições e o estado dos equipamentos de chaveamento, com a finalidade de eliminar medidas flagrantemente espúrias e corrigir eventuais erros de configuração de rede (e.g., com erros que violam limites físicos de equipamentos) e corrigir eventuais erros de configuração de rede.

- **Configurador de rede:** Processam-se dados dos *status* de chaves e disjuntores para construir o modelo barra-ramo representativo da rede.
- **Análise de Observabilidade:** Verifica-se a existência de medidas disponibilizadas pelo sistema de medição em quantidade e posicionamento que permitam a obtenção da solução da EE que se estenda por toda a rede supervisionada. Identificam-se também ramos não observáveis e ilhas observáveis no sistema, caso existam.
- **Filtragem:** Considerada como núcleo ou principal etapa do processo de EE, aqui filtram-se erros estatisticamente consistentes com o processo de medição, determinando-se uma estimativa otimizada para o estado mais provável da rede elétrica, usualmente através do método do mínimos quadrados ponderados (MQP).
- **Processamento de Erros Grosseiros:** Confrontam-se as medidas estimadas pela etapa anterior com aquelas oriundas do sistema de medição, definindo-se os chamados resíduos da estimação, com o objetivo de detectar/identificar a presença de erros grosseiros (EGs). Destaca-se que esta etapa requer uma redundância suficiente e adequado posicionamento de medidas para que possa produzir resultados confiáveis

A Figura 2.1 ilustra o encadeamento das etapas do processo de EE descritas anteriormente. Alguns aspectos básicos da EE convencional, com ênfase na análise de observabilidade e criticalidade, serão apresentados em sequência.

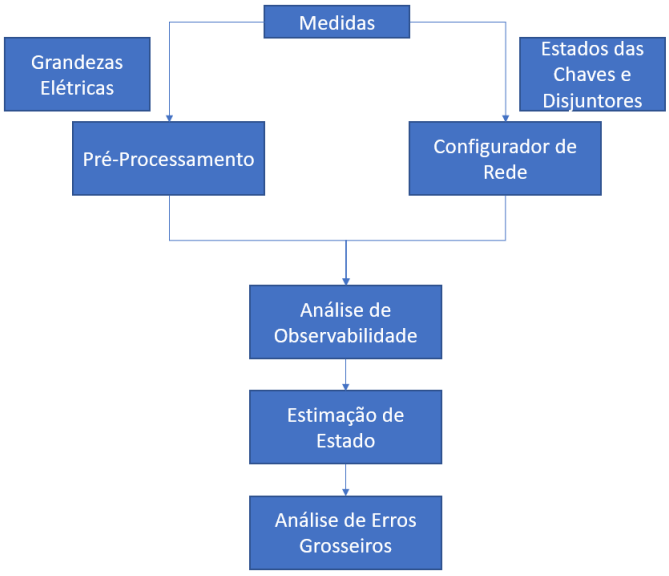

Figura 2.1: Etapas da EE.

2.1.2 Conceitos Básicos

Para uma dada configuração de rede, o estado de operação e as medidas coletadas do sistema são modeladas de acordo com:

$$\boldsymbol{z} = \boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{e} \tag{2.1}$$

onde: $\boldsymbol{z}[m \times 1]$ — é o vetor de medidas coletadas, $\boldsymbol{h}$ — é vetor de funções que correlaciona o estado às medidas, $\boldsymbol{x}[n \times 1]$ — é o vetor de estado , $\boldsymbol{e}$ — é o vetor de erros das medidas, supondo apresentar uma distribuição Gaussiana com média zero e matriz de covariância $\boldsymbol{R}$ e n — é o número de barras da rede elétrica.

O vetor $\boldsymbol{x}$ contém os ângulos e magnitudes das tensões nas barras, enquanto o $\boldsymbol{z}$ os valores medidos de magnitudes de tensão, fluxos e injeções de potências ativas/reativas. Caso haja no esquema de medição unidades de medição fasorial (UMFs), valores de magnitude/ângulo das tensões nodais e das correntes nos ramos nelas incidentes também estarão presentes em $\boldsymbol{z}$.

Utilizando (2.1) e o processo de estimação MQP, o estado da rede elétrica pode ser determinado pela solução do seguinte problema de otimização que minimiza a função objetivo $\boldsymbol{J(x)}$:

$$min_x \boldsymbol{J}(x) = [\boldsymbol{z} - \boldsymbol{h}(x)]^t \boldsymbol{R}^{-1} [\boldsymbol{z} - \boldsymbol{h}(x)] \tag{2.2}$$

A solução de (2.2) pode ser obtida através seguinte sistema de equações não lineares:

$$\boldsymbol{H}^t \boldsymbol{R}^{-1} [\boldsymbol{z} - \boldsymbol{h}(x)] = 0 \tag{2.3}$$

Utilizando o método de Newthon-Raphson, (2.3) é solucionada pelo seguinte processo iterativo:

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} + [\boldsymbol{H^t R^{-1} H}]^{-1} \boldsymbol{H^t R} - \boldsymbol{1}[\boldsymbol{z} - \boldsymbol{h(x)}] \tag{2.4}$$

sendo $\boldsymbol{H} = \dfrac{\boldsymbol{\partial h(x_k)}}{\boldsymbol{\partial x}}$, a matriz Jacobiana do sistema e $\boldsymbol{G} = \boldsymbol{H^t R^{-1} H}$ a matriz de Ganho.

Utilizando os conceitos apresentados anteriormente, o Algoritmo 1 descreve o processo adotado para solucionar a EE.

Algoritmo 1: Algoritmo iterativo para solucionar a EE.

`Contador de iterações`

$k = 0$;

`Inicializa vetor de estado` x_k

para $i \leftarrow 1$ ***até*** n **faça**

$\lfloor\ x_k[i] = 1\angle 0$;

`Processo iterativo até que a tolerância (`tol`) seja alcançada`

faça

$\boldsymbol{H = \delta h(x_k)/\delta x}$

$\boldsymbol{G = H^t R^{-1} H}$

$\boldsymbol{\Delta x_k = G(x_k)^{-1} H(x_k)^t R^{-1}(z - h(x_k))}$

$\boldsymbol{x_{k+1} = x_k + \Delta x_k}$

enquanto $max|\Delta x_k| > tol$;

No processo de avaliação dos resultados obtidos na estimação, o vetor de resíduos $\boldsymbol{r}$, contendo a diferença entre $\boldsymbol{z}$ e seus respectivos valores estimados $\boldsymbol{\hat{z}}$, é normalizado e submetido a seguinte validação:

$$\boldsymbol{r_N(i) = \frac{|r(i)|}{\sigma_{\Omega(i)}} \leq \lambda} \tag{2.5}$$

sendo $\boldsymbol{\sigma_{\Omega(i)} = \sqrt{\Omega(i,i)}}$ o desvio padrão do i-ésimo componente do vetor de resíduos obtido através da matriz de covariância de resíduos $\boldsymbol{\Omega}$, obtida por:

$$\boldsymbol{\Omega = I - HG^{-1}H^t} \tag{2.6}$$

A violação do limite pré-estabelecido ($\boldsymbol{\lambda}$) aponta inconsistências entre os valores estimados e medidos, provavelmente pela presença de medidas corrompidas. A identificação e eliminação de medidas portadoras de EGs requer um grau de redundância compatível com a quantidade de medidas espúrias [12]. Resíduos de estimação possuem propriedades estatísticas úteis para as análises de observabilidade e criticalidades[18] que serão abordadas posteriormente.

2.1.3 Análise de Observabilidade

Uma rede elétrica é definida como observável se existir um número suficiente de medidas, diversificadas e bem distribuídas pela rede para que a filtragem do estado seja possível em toda a rede. Matematicamente, a análise de observabilidade destina-se à verificação da possibilidade de solução do sistema de equações estabelecido em (2.2). Em outras palavras, se o processo iterativo construído em (2.4) convergir para uma solução única, a partir das condições iniciais, classifica-se a rede como numericamente observável. Alternativamente, considera-se o sistema topologicamente observável se for possível determinar unicamente todos os fluxos de potência na rede partindo-se das medidas disponibilizadas [38].

2.1.4 Modelo Linear

Adota-se aqui a análise de observabilidade por meio de um modelo linear, com desacoplamento $P\theta$ (potência ativa-ângulo de fase). Supõe-se que as medidas de potência de fluxo e injeção, ativa e reativa, sejam tomadas aos pares, como também, que ao menos uma medida de ângulo do fasor de tensão esteja presente. Sendo assim, o seguinte modelo é utilizado:

$$\boldsymbol{H_a\theta} + \boldsymbol{e_a} = \boldsymbol{z_a} \tag{2.7}$$

onde $\boldsymbol{H_a}[m \times n]$ — é a matriz Jacobiana das medidas ativas; $\boldsymbol{\theta}[n \times 1]$ — é o vetor de ângulos das tensões das barras; $\boldsymbol{e_a}[m \times 1]$ — é o vetor de medidas ativas; m — é o total de medidas ativas;

Em um sistema topologicamente observável a matriz $\boldsymbol{H_a}$ deve apresentar posto completo, com pelo menos uma medida de ângulo presente. Sendo assim, a determinação da observabilidade depende apenas da topologia da rede e da configuração das medidas do sistema (número, tipo e localização das medidas) [25]. Portanto, costuma-se assumir que todas as medidas do esquema possuam igual peso de participação no processo de estimação ($R_a = I$, matriz identidade) e que os ramos da rede estejam representados somente por susceptâncias-série unitárias.

A partir destas considerações, os elementos de $\boldsymbol{H_a}$, associados a uma dada medida l, passam a ser definidos da seguinte maneira [25]:

- Fluxo de Potência Ativa no ramo i-k:

$$\boldsymbol{H}_a(l,i) = 1;\ \boldsymbol{H}_a(l,k) = -1 \tag{2.8}$$

- Injeção de Potência Ativa na barra i:

$$\boldsymbol{H}_a(l,i) = nb_i;\ \boldsymbol{H}_a(l,k) = -1 \tag{2.9}$$

  onde nb_i corresponde ao número de barras conectadas à barra i.

Caso UMFs estejam presentes no esquema de medição, a matriz Jacobiano incluirá medidas ângulo e de fluxo de corrente representadas da seguinte forma [16]:

- A medida de ângulo da tensão na barra i:

$$\boldsymbol{\theta}_i \rightarrow \boldsymbol{H}_a(l,i) = 1 \tag{2.10}$$

- A parte real da corrente entre as barras i e k:

$$I_{ik} \rightarrow \boldsymbol{H}_a(l,i) = 1;\ \boldsymbol{H}_a(l,k) = -1 \tag{2.11}$$

A verificação da observabilidade pode ser realizada através da fatoração da matriz de ganho ($\boldsymbol{G_a}$), construída utilizando o modelo linear de forma similar à apresentada previamente. O algoritmo descrito em [3] e [25] apresenta detalhadamente esta verificação. Uma rede é considerada observável se $\boldsymbol{G_a}$, e consequentemente $\boldsymbol{H_a}$, possuir posto completo, ou seja, caso não ocorram pivôs nulos durante a fatoração da matriz $\boldsymbol{G_a}$.

Adicionalmente, caso exista exatamente um pivô nulo ao longo processo, o posto de $\boldsymbol{G_a}$ e $\boldsymbol{H_a}$ será $n-1$. Nesta situação, se o esquema de medição não possuir medidas de ângulo, a inserção de uma única pseudo-medida angular (análogo a se adotar um ângulo de referência) garantirá a determinação de todas as medidas de fluxo na rede. Dessa forma, a rede é considerada observável.

2.2 Análise de Criticalidades

Durante a operação da rede, podem ocorrer indisponibilidades de medidas, levando a redundância do esquema de medição a níveis críticos, que representam risco de perda iminente de observabilidade e comprometimento da confiabilidade da EE.

Cabe então a análise de criticalidades revelar as vulnerabilidades do esquema de medição, identificando elementos, tomados individualmente ou em grupos que são essenciais para que o sistema seja observado com um todo. Sendo assim, decorrem as seguintes definições:

- Medida Crítica (C_{meds}): quando ocorre a indisponibilidade destas, o sistema torna-se inobservável.
- $k-$Tupla Crítica (C_k): grupo de k medidas cuja indisponibilidade simultânea leva o sistema à inobservabilidade.

A determinação de k-tuplas críticas é um problema difícil de natureza combinatória, relativamente pouco explorado até o momento [3] e [55], exceto pela determinação de criticalidades de baixa ordem (k até 3). O trabalho apresentado em [18] apresenta um procedimento simples para identificação C_{meds} e conjuntos críticos (C_{conjs}), definidos da seguinte forma:

- Conjunto Crítico (C_{conj}): Conjunto de duas ou mais medidas, onde a indisponibilidade de qualquer uma delas faz com que as demais tornem-se C_{meds}, e por consequência, a remoção de qualquer uma destas remanescentes torne a rede inobservável.

Algumas propriedades de C_{meds} e C_{conjs} podem ser destacadas utilizando o vetor de resíduos normalizados ($\boldsymbol{r_a}$) e a matriz de covariância de resíduos ($\boldsymbol{\Omega}_a$) do modelo linear:

- C_{meds} apresentam sempre valores nulos de resíduos e de elementos correspondentes em $\boldsymbol{\Omega}_a$ (demonstração no Apêndice A.1). Devido ao fato de C_{meds} não se correlacionarem com nenhuma outra medida, não se alcançam qualquer benefício do processo de EE, pois o valor medido iguala-se ao estimado, o que torna impossível a identificação de EGs pela análise residual.
- Medidas que compõem C_{conjs} possuem uma forte correlação entre si, e por esta razão, apresentam valores em $\boldsymbol{r_a}$ numericamente idênticos. Sendo assim, EGs presentes em elementos de C_{conjs} podem ser detectados, porém não identificados.

Como a identificação de C_{meds} e C_{conjs} depende apenas da correlação entre as medidas e não necessariamente dos valores coletados, o procedimento descrito em [18] ressalta que vetores de valor unitários para as medidas $\boldsymbol{z_a}$ podem ser adotados.

Desta forma, uma medida i pode ser considerada uma C_{med} se:

$$\boldsymbol{r}_a[i] = 0 \tag{2.12}$$

$$\sigma_{\boldsymbol{E}_{a_i}=\sqrt{\boldsymbol{E}_a[i,i]}} = 0 \tag{2.13}$$

Para o caso de C_{conjs}, a identificação pode ser realizada a partir do denominado coeficiente de correlação:

$$\gamma_{ij} = \frac{\boldsymbol{E}_a(i,j)}{\sqrt{\boldsymbol{E}_a(i,i)}\sqrt{\boldsymbol{E}_a(i,j)}} \tag{2.14}$$

Caso duas medidas i e j possuírem $\boldsymbol{r_a[i] = r_a[j]}$ e $\boldsymbol{\gamma_{ij} = 0}$, então o par de medidas i e j fazem parte de um C_{conj}.

2.2.1 Tuplas Críticas de Medidas

Como mencionado anteriormente, as propriedades numéricas das criticalidades que envolvam até dois elementos se encontram bem estabelecidas. Contudo, uma análise mais generalizada diz respeito a identificação de tuplas críticas de medidas de cardinalidades superiores. Entende-se por C_k-tupla de medidas (C_{ks}) o grupo de k medidas em que a indisponibilidade simultânea de todo o conjunto resulta na inobservabilidade da rede. A partir desta definição, a cardinalidade (k) de uma tupla crítica corresponde ao número de elementos que a compõem, sendo assim, C_{meds} representam uma C_1-tupla e os pares de medidas que formam um C_{conj} são, portanto, C_2-tuplas.

Conforme mencionado anteriormente, a presença de C_{ks} em um esquema de medição compromete a credibilidade da EE em relação a sua capacidade de detecção e identificação de dados espúrios. De forma generalizada, se $k-2$ medidas contendo erros grosseiros pertencem a uma C_k tais erros podem ser detectados e identificados, porém se $k-1$ ou k medidas espúrias estiverem presentes em uma C_k, esses só podem ser detectados [16].

Conforme descrito em [3], uma k-tupla crítica de medidas apresenta as seguintes propriedades:

1. Uma C_k não pode conter uma C_j, onde $k > j$;

   Esta propriedade está diretamente relacionada à definição de C_k. Apenas a remoção de todas as k medidas do conjunto levarão a inobservabilidade, logo este subconjunto de j elementos não será crítico.

2. Caso $j < k$ medidas de uma C_k se tornem indisponíveis, as medidas restantes formarão uma $C_{(k-j)}$.

   Com base na propriedade 1, a retirada $j < k$ medidas de uma C_k não provoca inobservabilidade da rede, porém, uma diminuição da redundância. Neste novo cenário, a inobservabilidade será alcançada se as $k - j$ medidas restantes, as quais formam uma $C_{(k-j)}$ se tornarem indisponíveis.

3. Dado um plano de medição contendo m medidas, a cardinalidade máxima (teórica) de uma C_k é definida por:

   $$k_{max} = m - n + 1 \tag{2.15}$$

   onde n é o número de elementos do vetor de estado do processo de estimação.

   Demonstração desta propriedade encontra-se no Apêndice A.2

4. As colunas de $\boldsymbol{\Omega_a}$, relacionadas às medidas de uma dada C_k, formam um conjunto linearmente dependente.

   A comprovação desta propriedade encontra-se no Apêndice A.3.

Estas propriedades são fundamentais no contexto da análise de criticalidades pela matriz $\boldsymbol{\Omega_a}$. A partir da Propriedade 4 é possível determinar se um conjunto de k medidas, selecionadas de um esquema de medição de tamanho, possui criticalidades em sua

composição. Ou seja, caso uma submatriz $\tilde{\boldsymbol{\Omega}}_{\boldsymbol{a}}$, formada pelas linhas e colunas de $\boldsymbol{\Omega}_{\boldsymbol{a}}$ referentes às medidas desse conjunto, apresente colunas linearmente dependentes, em outras palavras, se $\tilde{\boldsymbol{\Omega}}_{\boldsymbol{a}}$ for singular o conjunto apresenta uma C_j, onde $j \leq k$. A confirmação da criticalidade do conjunto como um todo pode ser realizada pela propriedade 1. Ou seja, se nenhum subconjunto de medidas apresentar criticalidades, de fato essa tupla é crítica. O procedimento adotado neste trabalho para identificar C_{ks} em um esquema de medição será apresentado detalhadamente no Capítulo 4, referente à metodologia proposta.

2.3 Trabalhos Relacionados

Busca-se aqui referenciar os principais trabalhos associados ao tema abordado nesta Dissertação.

Inicialmente, no que diz respeito à análise de observabilidade para a EE, destacam-se os trabalhos apresentados em [1] e [38].

Existem duas abordagens principais para o problema de análise de observabilidade: A topológica que se fundamenta na teoria de grafos para realizar uma análise puramente estrutural, e a numérica que se baseia nas propriedades de matrizes do processo de estimação.

Dentre as abordagens topológicas, os principais trabalhos foram desenvolvidos por Clements, Krumpholz e Davis [35, 11, 13, 14]. Particularmente no primeiro trabalho [35] foi estabelecido a análise residual do processo de estimação, a qual é relacionada com a topologia da rede. Outras análises baseadas puramente na teoria de grafos para avaliação da observabilidade podem ser encontradas no trabalho de Quintana [49], e notadamente no trabalho de Mori [42], nos quais adotou-se o conceito de árvores geradoras mínimas (*minimum spanning trees*). Por fim, o trabalho de Nucera e Gilles em [44] faz uso de conceitos de otimização combinatória e de sequencias ampliadas para a determinação de observabilidade de uma rede.

Contudo, como a abordagem topológica apresenta uma elevada complexidade computacional e requerem um embasamento teórico mais complexo, trabalhos voltados à análise de observabilidade numérica foram desenvolvidos alternativamente. O trabalho de Monticelli e Wu [39, 57] aborda os conceitos de ilha observável e de ramo não observável, explorando também, algoritmos para a avaliação da observabilidade pela fatoração triangular da matriz de Ganho. O mal condicionamento numérico desta matriz, motivou o desenvolvimento de uma nova abordagem baseada na transformação ortogonal da matriz

Jacobiana [40]. A partir de propriedades de alocação das medidas de fluxo e injeção e a topologia da rede, o algoritmo de [9] manipula apenas números inteiros na matriz Jacobiana de medidas, o que torna o processo de análise de observabilidade mais simples e ágil. Bei Gou apresenta em seu trabalho [25] uma análise algébrica que utiliza de fatores triangulares da matriz de Ganho, tanto para a determinação de ilhas observáveis, quanto para a determinação de quais pseudomedidas devem ser inseridas para restaurar a observabilidade da rede. Castillo, em seu trabalho [8], apresenta uma técnica eficiente que se baseia no cálculo do espaço nulo da matriz Jacobiana. Já Almeida em [2] utiliza da matriz de Gram, calculada a partir da Jacobiana, para realizar análise e verificar redundância. Bei Gou, em [24], realiza a análise a partir do método da eliminação de Gauss para tornar o processo mais simples e eficiente. Por fim, Solares em [54] também utiliza do método de Gauss, porém aplicando a aritmética binária para evitar erros de arredondamento.

A capacidade de observação de um sistema de medição e a depuração de erros grosseiros estão intimamente relacionadas à presença de criticalidades. O primeiro trabalho relacionado a identificação generalizada de k-tuplas críticas foi proposto por Clements em [12], que apresenta a definição de C_{ks} de medidas e suas principais propriedades, como também, propõe um método geométrico baseado em geometria analítica para identificação das criticalidades.

Existem dois trabalhos desenvolvidos por Clements que utilizavam abordagem topológica para a identificação de medidas críticas. O primeiro trabalho [11] utiliza o conceito de arvore geradora, para identificar medidas críticas e regiões de propagação de erro. O segundo, apresentado em [14], dá continuidade ao trabalho anterior e avalia a inclusão de pseudo-medidas no conjunto de medição. Simões em [53] estendeu o trabalho de Quintana [49], e propôs a utilização de matróides e de árvores geradoras para a determinação de medidas críticas e conjuntos críticos.

Dentre as abordagens numéricas, [7] utiliza a análise dos resíduos normalizados e dos elementos da matriz de covariância para identificar C_{conjs}. O trabalho de Exposito em [20] apresenta um método generalizado de observabilidade que também é capaz de classificar de C_{meds} e medidas redundantes. Coutto, em [17, 18], desenvolveu um método que utiliza propriedades dos resíduos normalizados das medidas para a determinação de C_{meds} e C_{conjs}. Outro trabalho relevante no sentido de determinação de tuplas de baixas cardinalidades, foi realizado por Coutto em [16], onde foi apresentado uma abordagem da análise de criticalidades considerando a situação mais provável, onde ocorre a indisponibilidade de uma unidade medição contendo várias medidas.

Além das análises topológicas e numéricas, existem também as híbridas que combinam a teoria dos grafos e métodos numéricos. O algoritmo numérico-simbólico de classificação de C_{meds} e C_{conjs} apresentado por Korres em [34] emprega os conceitos de ilhas de fluxo, áreas de propagação residual e modelo reduzido da rede, para isolar erros em menores regiões do sistema, possibilitando uma redução na busca por EGs.

Conforme destacado intensamente, devido à natureza combinatória do problema em estudo a quantidade de trabalhos relacionados a análise de criticalidades em cardinalidades elevadas ($k > 3$) é relativamente pequena. Contudo, diversos equipamentos da rede elétrica (medidores inteligentes) começaram a utilizar sistemas de comunicação digital de dados, tornando sistemas vulneráveis à ataques cibernéticos [28]. O primeiro caso de um ataque desta natureza e bem sucedido em redes de potência pode ser encontrado em [58]. A possibilidade de invasores corromperem de forma intencional medidas coletadas para a EE serve de estímulo para o estudo de cardinalidades mais altas.

Considerando o perigo de ataques cibernéticos e a vulnerabilidade do sistema de medição pela presença de C_k-tuplas, [55] propôs um método de programação linear inteira mista para identificação de C_{ks} de medidas (as mais esparsas, i.e., de menor k possível) nas quais uma medida pré-estabelecida pertença. Posteriormente, em [5], desenvolveu-se uma estratégia para identificar C_{ks} para $k > 3$, a partir dos resultados obtidos das cardinalidades inferiores. Em sequência, a referência [3] contém o desenvolvimento de uma abordagem pela heurística de *branch-and-bound* (B&B) para enumerar de forma eficiente as combinações de medidas e identificar C_{ks}, para qualquer valor de k, seja de medidas, unidades de medição ou ramos da rede. Dando continuidade a este trabalho, a publicação [4], aprofundou a abordagem de B&B para criticalidades de UMs.

Finalmente, como pode-se depreender desta seção, o problema que envolve a determinação de criticalidades, principalmente às de cardinalidades mais elevadas, foi pouco explorado e ainda há espaço para pesquisas nessa área, buscando-se conhecer melhor o problema e estabelecer estratégias e limites interessantes para tais estudos.

Capítulo 3

Programação em GPUs

Neste capítulo será apresentado inicialmente uma introdução à programação paralela em GPUs, seguida da introdução de conceitos da programação em CUDA que foram utilizados no desenvolvimento desta Dissertação.

3.1 Introdução

Ao longo dos anos, simulações científicas têm requerido resultados numéricos cada vez mais extensos e confiáveis, capazes de retratar problemas do mundo real. Nesse contexto, a computação de alto desempenho (CAD) é uma das principais frentes de pesquisa por novas técnicas e tecnologias que visam otimizar, ou até mesmo viabilizar, o processamento de simulações numéricas de difícil execução.

3.1.1 Revolução *Multi-Core*

A componente chave para computação de alto desempenho é a unidade central de processamento (*central processing unit* - CPU), comumente denominada de núcleo [31]. Tradicionalmente, o ganho de desempenho era alcançado pelo desenvolvimento de CPUs com velocidades de processamento cada vez maiores. De acordo com a lei de Moore em 1965, o número de transistores nos circuitos internos de processadores teria um aumento de 100%, dobrando sua capacidade de processamento pelo mesmo custo, a cada período de 18 meses [41]. Contudo, esta projeção chegou ao fim devido às limitações físicas na fabricação de circuitos integrados. Restrições de consumo e calor, bem como um limite no tamanho dos transistores, inviabilizou o ganho de desempenho através do aumento de velocidade de processamento.

Alternativamente, processadores contendo múltiplos núcleos (*multi-core*) tornaram-se a nova tendência dos fabricantes. A utilização de duas ou mais CPUs em um mesmo circuito, permite a execução simultânea de instruções, o que resulta em um aumento de desempenho sem a necessidade de um aumento da velocidade de processamento das unidades. Devido a este fato, a programação paralela tornou-se uma nova realidade, que passa a ser responsabilidade do programador dividir um processo em tarefas menores que serão distribuídas pelos núcleos para serem realizadas de forma concorrente.

Sendo assim, o aumento da quantidade de núcleos tornou-se a principal via para ganho de desempenho, e com o tempo, foram desenvolvidas arquiteturas contendo cada vez mais núcleos para suprir a necessidade de maior processamento. O termo *many-core* foi criado para descrever arquiteturas *multi-core* com dez a cem núcleos. Devido a este fato, desenvolvedores voltaram sua atenção para unidades de processamento gráfico (*graphical processing units* - GPU), que são arquiteturas *many-core* voltadas para processamento gráfico.

3.1.2 Programação em GPUs

GPUs no início dos anos 2000 eram essencialmente projetadas para gerar imagens em uma tela [30]. Para realizar esta tarefa, a GPU utiliza unidades aritméticas programáveis, conhecidas como *shaders*, para combinar vários parâmetros de entrada e produzir uma cor final em um ponto *(x,y)* da tela. Tais parâmetros podem ser cores, coordenadas de textura ou outros atributos que seriam passados ao *shader* durante sua execução.

A forma como a GPU processava concorrentemente milhares destes pontos para formar imagens chamou a atenção de desenvolvedores, e então foi observado que as "cores" de entrada poderiam ser substituídas por quaisquer outros tipos de dados. Portanto, se estes parâmetros fossem substituídos por dados numéricos, que significavam algo diferente de uma cor, os desenvolvedores poderiam programar os *shader* para realizar cálculos sobre esses dados e a "cor" final retornada pela GPU representaria o resultado. Em essência, a GPU é *enganada* para realizar tarefas diferentes de renderização de imagens.

Apesar dos resultados promissores, o modelo de programação de GPUs ainda era muito restritivo para se popularizar entre os desenvolvedores científicos. O fato de os parâmetros de entrada serem necessariamente representados por cores e texturas, como também, pela GPU não tratar bem variáveis de ponto flutuante, significava que boa parte dos programas científicos não poderiam ser adaptados para GPU. Além disso, desenvolvedores deveriam utilizar linguagens de programação gráficas como *Open GL* e *DirectX*

para poder interagir com a GPU.

3.1.3 Arquitetura CUDA

Em novembro de 2006, a NVIDIA trouxe a GeForce 8800, a primeira GPU construída com a arquitetura unificada de dispositivos (*compute unified device architecture* - CUDA). Esta nova arquitetura visava aliviar muitas das limitações da programação de processadores gráficos para uso geral, além de permitir um bom desempenho para tarefas gráficas tradicionais. Para isto, as novas unidades lógicas e aritméticas (*arithmetic logic unit* -ALU) foram construídas para cumprir os requisitos aritméticos de ponto flutuante do IEEE. Permitindo assim, um conjunto de instruções feito para computação geral em vez de especificamente para gráficos.

Além das inovações de *hardware*, alguns meses depois, a NVIDIA tornou público o novo compilador para a linguagem CUDA C. Esta utilizava do padrão C para atingir o maior número de desenvolvedores possível e adicionou algumas palavras-chaves para aproveitar os recursos da nova arquitetura CUDA. Tornando-se assim, a primeira linguagem especificamente projetada para facilitar a programação de GPUs para o uso geral.

3.2 Diferenças entre Unidades de Processamento

Apesar dos avanços tecnológicos possibilitarem uma programação cada vez mais ampla em GPUs, ela não visa substituir a programação CPUs. Ambos dispositivos não possuem uma origem comum e existem diferenças tanto em suas composições, quanto em suas funcionalidades.

CPUs são dispositivos que visam diminuir a latência, i.e., tempo para uma tarefa ser realizada. Como pode ser visto na Figura 3.1, uma CPU é composta por unidades de controle, lógicas, aritméticas e memória cache mais robustas, capazes de realizar paralelamente algumas dezenas de cálculos complexos, visando diminuir o tempo execução. Em contrapartida, GPUs objetivam otimizar o que se conhece por *throughput* (tarefas realizadas por segundo). Devido ao fato de as GPUs serem compostas por versões mais simples destas unidades, porém em maior quantidade, elas são mais eficientes para realizar milhares de cálculos leves de forma concorrente.

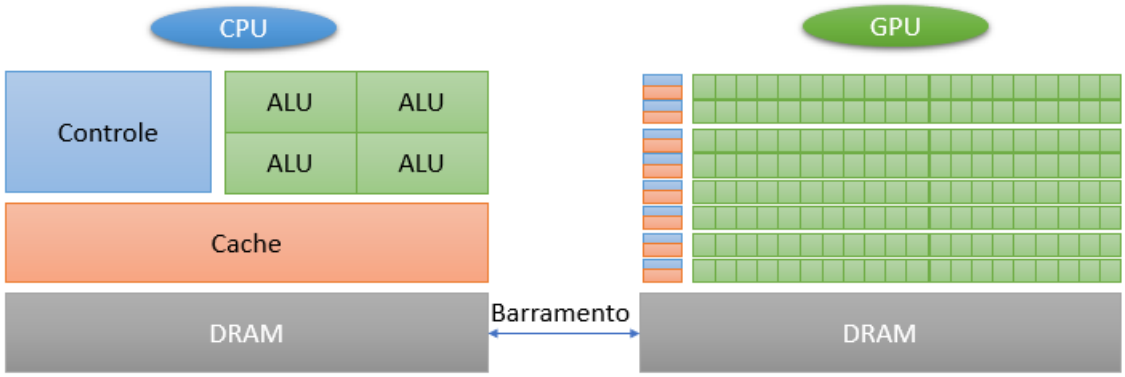

Figura 3.1: Distribuição de Recursos em CPU e GPU.

Além da questão das distinções de aplicabilidade, GPUs não são plataformas autônomas e atuam como um co-processador em conjunto à uma CPU hospedeira. Devido a este fato, em termos de programação de GPUs, CPUs são intituladas *host* (hospedeiro) e GPUs *device* (dispositivo).

Diz-se que um programa em CUDA é heterogêneo, ou seja, seu código particiona-se em etapas que serão executadas no *host* e outras no *device*, conforme ilustra Figura 3.2. A fração do código realizada na CPU é responsável por inicializar o programa, bem como, por gerenciar o código e os dados que serão executados na GPU.

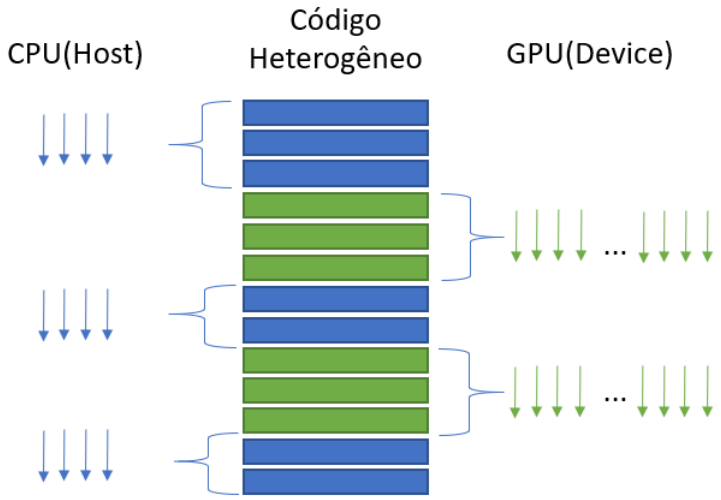

Figura 3.2: Particionamento de um código em que etapas são realizadas no *Host* e *Device*.

Em geral, uma aplicação possui etapas sequencias e paralelas. Implementações que utilizam a programação heterogênea em CPU/GPU são bem sucedidas, pois aproveitam as características complementares das arquiteturas para maximizar o desempenho geral da aplicação; utilizando do processamento em CPUs, em etapas constituídas de cálculos complexos que requerem um baixo nível de paralelismo, e GPUs nas que requerem um alto nível de paralelismo [46].

3.3 Modelo de Programação

CUDA é uma plataforma de computação paralela de uso geral e um modelo de programação que possibilita a utilização de recursos de GPUs da NVIDIA para resolver problemas computacionais complexos de maneira mais eficiente [46]. Utilizando CUDA é possível realizar programação em GPUs de forma análoga à programação em CPUs, utilizando de extensões de linguagens de programação, incluindo C, C++, Fortran e Python. Esta Dissertação fez uso exclusivo da CUDA C.

Nesta seção, serão apresentadas algumas peculiaridades da programação paralela heterogênea em CUDA que, diferentemente da programação sequencial, unicamente em CPUs, requer um certo nível de conhecimento da arquitetura da GPU, assim como, um novo paradigma de programação que consiste em dividir e organizar um processo maior em tarefas menores (*threads*) concorrentes.

3.3.1 Organização de *Threads*

Uma *Thread* pode ser definida como uma linha de execução realizada em cada núcleo. Para exemplificar, em uma adição de vetores realizada em programação sequencial (*single-thread*), uma única *thread* é responsável por adicionar ordenadamente cada elemento dos vetores. Por outro lado, na programação paralela, múltiplas *threads* são invocadas para adicionar simultaneamente cada elemento do vetor, como se vê na Figura 3.3.

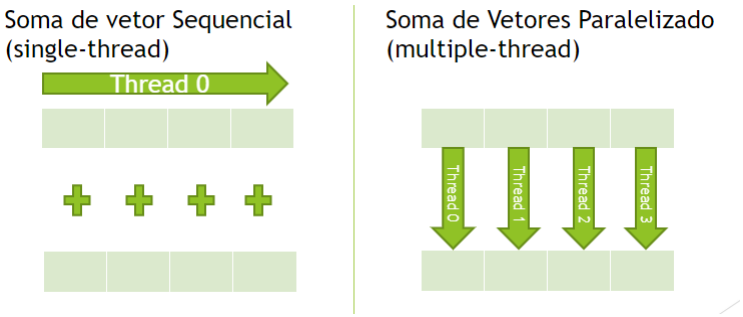

Figura 3.3: Soma de vetores de forma sequencial e paralela.

Particularmente na programação em GPUs, as denominadas *CUDA Threads* são invocadas a partir da declaração de *Kernels* (funções executadas pelo *device*). A extensão CUDA C permite que este tipo de função seja definido em C. Quando um *kernel* é declarado em um código do *host*, a execução é movida para o *device* onde um grande número de *threads* é gerado e cada uma delas executa os comandos especificados pela função.

Para ilustrar o funcionamento das *threads* na GPU, a Figura 3.4 representa a declaração de um *kernel* no *host* que invoca n *threads* no *device*, para somar concorrentemente os n elementos dos vetores $\boldsymbol{A}$ e $\boldsymbol{B}$. Cada *thread* possui seu próprio *ThreadID*, que as distingue entre si e identifica a porção de dados específicos que deverá ser processada ($thread_N$ realiza a operação $\boldsymbol{A[n]} + \boldsymbol{B[n]}$).

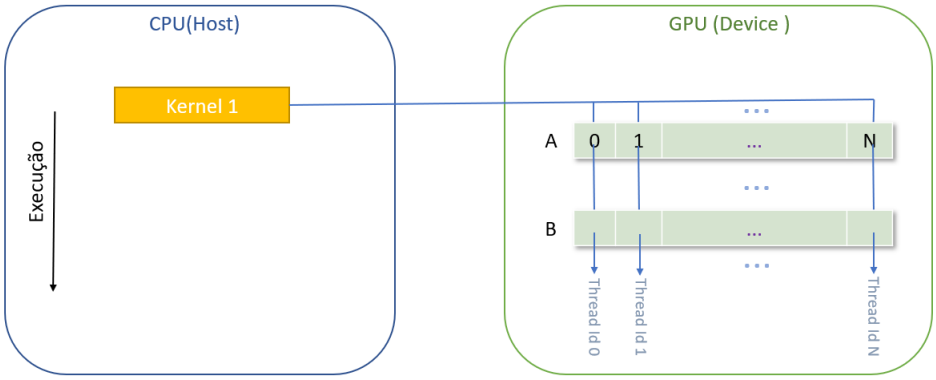

Figura 3.4: Adição de Vetor em CUDA C.

Para uma melhor organização das *threads*, cada *threadId* pode possuir até três coordenadas (x, y, z), o que torna mais intuitiva a representação de estruturas multi-dimensionais como matrizes. Para exemplificar, a Figura 3.5 ilustra a invocação de *threads* que utilizam *threadsIDs* de duas dimensões para acessar elementos das matrizes $\boldsymbol{A}$ e $\boldsymbol{B}$ de forma mais clara.

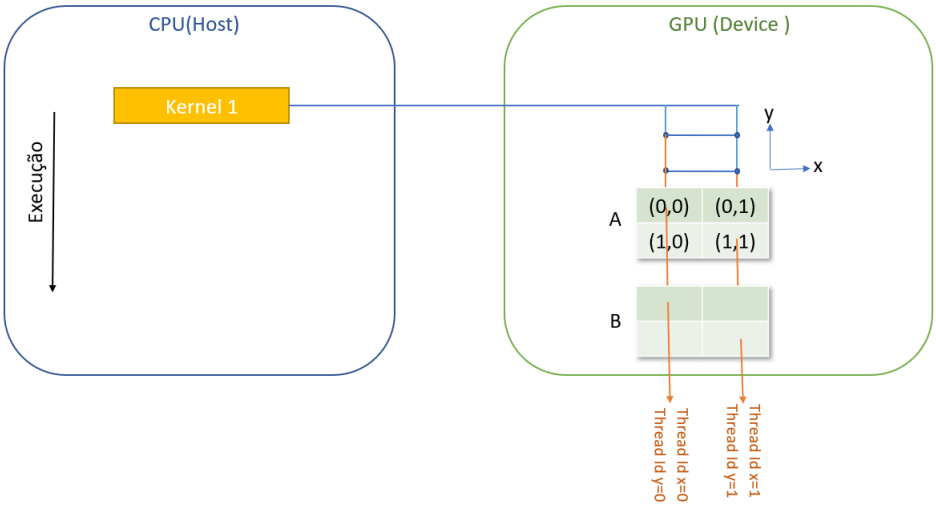

Figura 3.5: Adição de vetor em CUDA C.

Para auxiliar o gerenciamento de milhares de *threads*, estas são agrupadas em blocos (*thread blocks*) de também até três dimensões. E de forma similar, blocos são agrupados em uma rede (*Grid*) e são identificados por um *BlockId* de até três coordenadas. Blocos da mesma rede devem conter o mesmo número de *threads*.

Considere uma rede composta por nb blocos contendo nt *threads* por bloco. O número total de *threads* invocadas na rede *(ntr)* pode ser obtido por:

$$ntr = nb \times nt \tag{3.1}$$

Para auxiliar na ilustração da hierarquia de *threads*, a Figura 3.6 apresenta um modelo de execução de um programa em CUDA, onde um *kernel* invoca uma rede $[2 \times 3]$ de blocos $[3 \times 3]$. Utilizando (3.1) é possível contabilizar $6 \times 9 = 54$ *threads*.

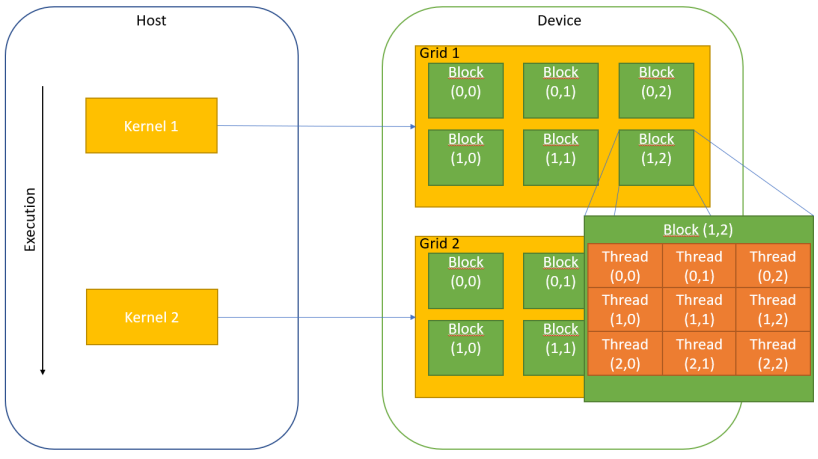

Figura 3.6: Modelo de execução e hierarquia de *threads* em CUDA.

A hierarquia de *threads* é uma abstração do ponto de vista do *software*. Visando um ganho de desempenho, existem alguns conceitos relacionados ao *hardware* da GPU que são relevantes para a organização das *threads*.

Do ponto de vista do *hardware*, a arquitetura de uma GPU é constituída de diversos *streaming multiprocessores* (SMs). Quando um *kernel* é lançado, os blocos de *threads* da rede são distribuídos entre os SMs disponíveis para a execução. Uma vez que o bloco é designado, as *threads* que o compõem são executadas de forma concorrente no SM. Múltiplos blocos podem ser designados a um determinado SM ao mesmo tempo, e são coordenados de acordo com a disponibilidade dos recursos. A Figura 3.7 ilustra a relação de execução ponto de vista do programador e *hardware*.

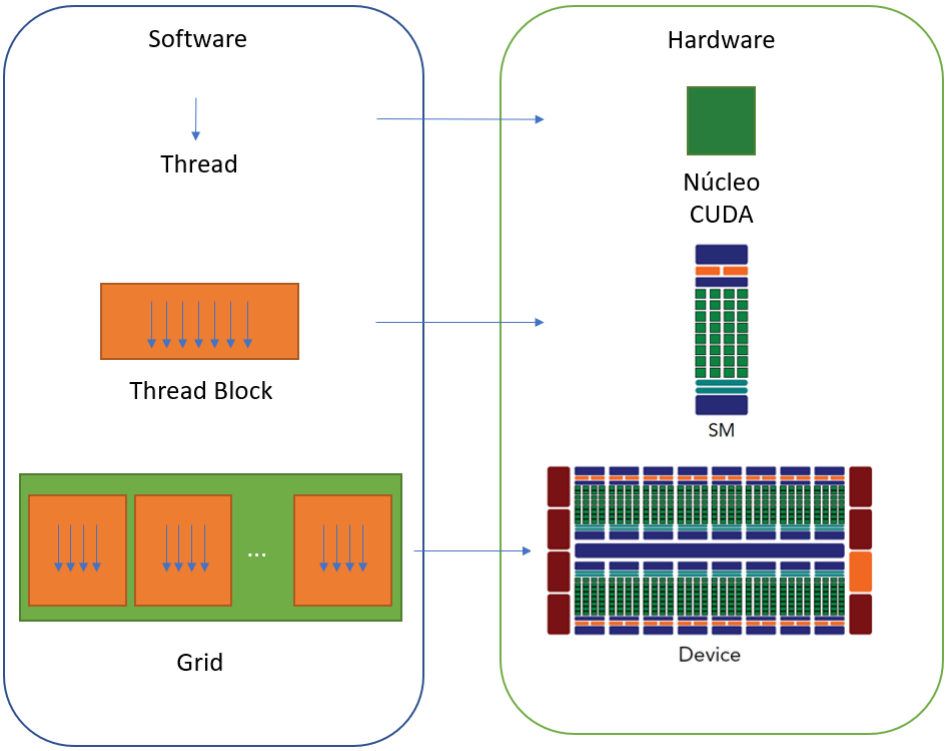

Figura 3.7: Relação da hierarquia de *threads* e onde são executadas [46].

Os SMs executam as *threads* em grupos de 32 denominados *warps*. Sendo assim, blocos organizados com um número de *threads* múltiplo de 32 utilizam de forma mais eficiente os recursos disponíveis de um SM, produzindo um ganho expressivo de performance.

Devido ao fato de os SMs possuírem recursos de memória limitados que devem ser distribuídos entre as *threads* executadas simultaneamente, existe também um número máximo de *threads* em um mesmo bloco (1024 em GPUs atuais). Contudo, aplicações que utilizam o processamento em GPU comumente requisitam um alto paralelismo, e

consequentemente, um elevado número de *threads*. Nestas situações, redes de blocos são utilizados para gerenciar um grande número de *threads* entre os SMs. Sendo assim, cada *thread* i contida em um bloco j assumirá o seguinte Id, que é único em toda rede:

$$Id = (blockId[j] \times nT) + threadId[i] \tag{3.2}$$

A Figura 3.8 apresenta um exemplo em menor escala para ilustrar o particionamento da soma dos vetores A e B de tamanho 8 em dois blocos contendo 4 *threads*. São destacados os Id de algumas *threads*, obtidos em (3.2).

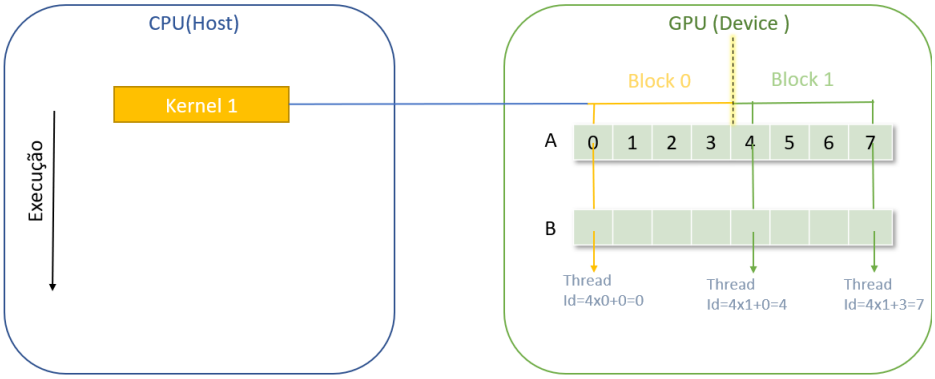

Figura 3.8: Adição de vetores particionada em blocos de *threads*.

3.4 Hierarquia de Memória

O gerenciamento do acesso à memória é um ponto importante para a computação de alto desempenho. Como o desempenho de uma implementação é limitado pela velocidade de armazenamento e coleta de dados, uma memória com baixo tempo de acesso e alta taxa de transferência é sempre desejada no aprimoramento de desempenho [31]. Contudo, uma memória com grande capacidade e alto desempenho nem sempre é economicamente viável. Em vez disso, lança-se mão da hierarquia de memória para utilizar da melhor maneira possível os diferentes tipos de memória disponíveis, o que significa, armazenar em memórias de menor capacidade e menor tempo de acesso, dados que são requisitados com maior frequência. O modelo de programação CUDA expõe totalmente a memória da GPU para possibilitar um melhor controle e armazenamento de dados para melhorar o desempenho. Como pode ser visto na Figura 3.9, *threads* têm acesso diferenciado a múltiplas categorias de memória durante sua execução. No nível fundamental, cada *thread* possui sua própria memória local e um grupo de registradores. *Threads* de um

mesmo bloco possuem um acesso coletivo a memória compartilhada que é local e possui a mesma vida útil do bloco. Todas as *threads* possuem acesso a mesma memória global que é a memória principal da GPU com maior espaço porém maior tempo de acesso.

Existem também outras duas memórias de leitura que todas as *threads* possuem acesso comum. A memória constante possui um acesso mais rápido que a memória global, porém um menor espaço de armazenamento. Esta memória é otimizada para casos onde *threads* precisam ler um mesmo espaço de memória. A memória de textura geralmente é utilizada para fins gráficos e otimizada para casos em que o acesso a memória segue padrões de localidade espacial (se um item é referenciado, itens cujos endereços são próximos a este tenderão a serem referenciados também).

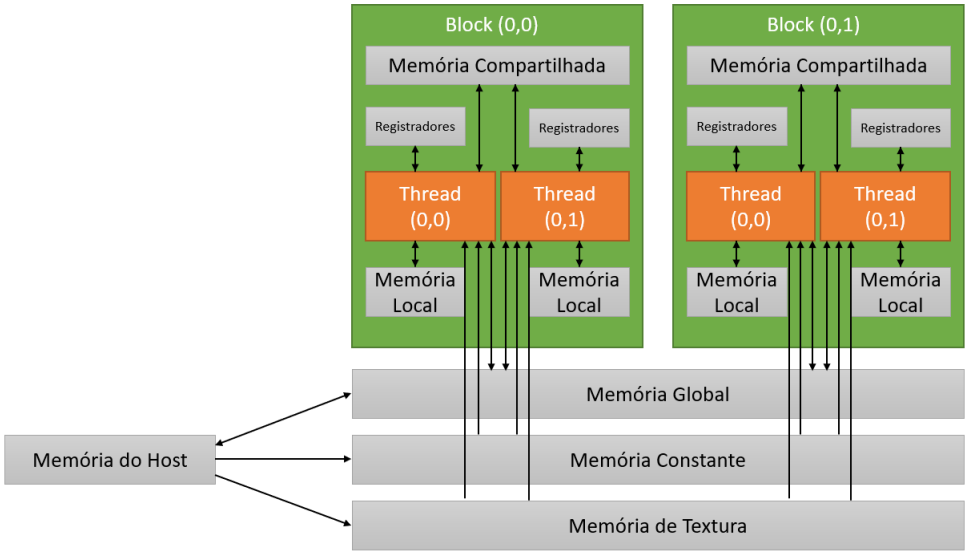

Figura 3.9: Abstração da hierarquia de memória da GPU [33].

3.5 Fluxo de Dados

Um código em CUDA possui um modelo de programação heterogêneo, em que as *threads* invocadas pelos *kernels* são executadas na GPU como um coprocessador fisicamente separado da CPU. Também é assumido que ambas as arquiteturas possuem suas próprios memórias de armazenamento. Sendo assim, faz parte da programação em CUDA o gerenciamento de dados entre as estruturas.

Como um código em CUDA é inicializado no *host*, os dados que serão utilizados pelo *device* devem ser transferidos, previamente a sua manipulação, e coletados ao fim do processo. E por este motivo, um típico fluxo de dados em um programa em CUDA se

estabelece através dos seguintes passos:

1. Copiar dados da memória da CPU para memória da GPU.
2. Invocar *kernels* para operar os dados armazenados na memória da GPU.
3. Copiar os dados novamente da memória da GPU para CPU.

Existem recursos, como a memória unificada, que permitem "mascarar" a alocação e transferência de dados entre arquiteturas, visando facilitar a portabilidade de aplicações para a programação heterogênea em GPUs. Contundo, visando uma ótima implementação e um ganho de desempenho, o tempo investido na transição de dados deve ser considerado.

Como a taxa de transferência de dados (*bandwidth*) entre de dispositivos distintos é significativamente menor do que entre qualquer memória da GPU e seus processadores, a transferência de dados entre *host* e *device* torna-se um dos principais gargalos de desempenho. Sendo assim, é desejado minimizar, sempre que possível, a transferência de dados entre *Host* e *Device*. Uma forma de se alcançar este objetivo é levar etapas com baixo nível de paralelismo, bem implementáveis na CPU, para que sejam executadas na GPU. A perda de desempenho do tempo de execução da etapa, muitas vezes é compensada pela menor transferência de dados entre dispositivos [45].

Capítulo 4

Metodologia Proposta

Neste capítulo será apresentado o método descrito em [6] que utiliza a matriz de covariância de resíduos para identificação de $k-$tuplas críticas. Em seguida, são apresentadas as motivações que justificam a adaptação deste método para GPU. E por fim, vêm as adaptações realizadas no algoritmo original sequencial para a programação paralela, com enfoque nas peculiaridades de GPUs.

4.1 Método da Matriz de Covariância de Resíduos

Conforme visto no Capítulo 2, a análise das criticalidades pode ser realizada tanto pela matriz de ganho ($\boldsymbol{G}$), quanto pela matriz de resíduos da estimação $\boldsymbol{\Omega}$, sabendo-se que a construção de ambas depende apenas da distribuição e do tipo das medidas presentes no sistema, e não necessariamente do valor coletado pelos medidores.

Contudo, como pôde ser visto em [21], a abordagem por $\boldsymbol{G}$ é computacionalmente mais custosa, pois requer a reestruturação da matriz Jacobiana ($\boldsymbol{H}$) e uma subsequente inversão de $\boldsymbol{G}$, a cada nova combinação de medidas a ser analisada. Partindo dessa premissa, como $\boldsymbol{G}$ possui uma dimensão $[n \times n]$, onde n representa o número de variáveis de estado, as matrizes avaliadas neste método serão, em geral, maiores que as submatrizes avaliadas no método apresentado na sequência. Isto posto, o processo baseado em $\boldsymbol{G}$ necessita de um maior esforço computacional, o que é indesejável para processos combinatórios.

Em relação à matriz $\boldsymbol{\Omega}$, essa possui dimensão $[m \times m]$, sendo m o número de medidas presentes em um esquema de medição, e seus elementos representam o grau de interação entre as medidas. Devido ao fato de C_{meds} não se correlacionarem com nenhum elemento do esquema de medição, a identificação de criticalidades de cardinalidade $k = 1$ pode ser realizada de forma direta pelas linhas e colunas nulas presentes em $\boldsymbol{\Omega}$.

Para descrever em termos gerais a utilização de $\boldsymbol{\Omega}$ no processo de identificação de C_{ks}, presentes em esquemas de medição, as seguintes propriedades das $k-$tuplas críticas são destacadas:

1. As colunas de $\boldsymbol{\Omega}$ associadas a medidas que formam uma C_k são linearmente dependentes.

2. Uma C_k não contem uma C_j, $\forall k > j$.

Caso ambas propriedades sejam cumpridas para um conjunto de k medidas, este pode ser considerado uma C_k. A avaliação da primeira propriedade pode ser realizada a partir da construção de uma matriz auxiliar $\tilde{\boldsymbol{\Omega}}$, utilizando as linhas e colunas de $\boldsymbol{\Omega}$ associadas a este conjunto de medidas (Algoritmo 2). Em seguida, a verificação de colunas linearmente dependentes em $\tilde{\boldsymbol{\Omega}}$ pode ser realizada a partir da Eliminação de Gauss [6]. Adicionalmente, a segunda propriedade requer a comparação do conjunto de medidas com todas as criticalidades de cardinalidade inferior.

Algoritmo 2: Construção da matriz auxiliar $\tilde{\boldsymbol{\Omega}}$

```
medida[k]; Conjunto de k medidas de interesse
para i ← 1 até k faça
    para j ← 1 até k faça
        Ω̃_a[i, j] = Ω_a[medida[i], medida[j]]
```

Sintetizando, os seguintes passos definem o processo de identificação de C_{ks}:

1. Selecionar um grupo de k medidas de interesse, $k \leq m$;

2. *Verificação da propriedade 1*: Formar a matriz simétrica $\tilde{\boldsymbol{\Omega}}$; iniciar o processo de Eliminação de Gauss em $\tilde{\boldsymbol{\Omega}}$ para verificar a existência de colunas linearmente dependentes;

3. *Verificação da propriedade 2*: Caso exista uma criticalidade de ordem j contida no grupo de k medidas selecionadas $\forall j < k$.

4. Caso ambas as propriedades sejam satisfeitas, o conjunto forma uma C_k.

Este procedimento deve ser realizado considerando todas as $(\frac{m!}{k!(m-k)!})$ combinações de medidas possíveis, o que demonstra a natureza combinatória do problema. Em casos comuns, são utilizados esquemas de medição com cerca de $m = 200$ medidas, o que significa, para a cardinalidade $k = 5$, uma ordem de 1 bilhão combinações a serem analisadas. Para ilustrar melhor a aplicação deste procedimento e facilitar o entendimento do processo, na sequência um exemplo numérico (de pequena escala) será apresentado.

4.1.1 Exemplo Numérico

Considere o sistema de 6 barras, representado pela Figura 4.1, contendo cinco medidas de fluxo e quatro medidas de injeção de potência. Todas as C_{ks} presentes serão identificadas.

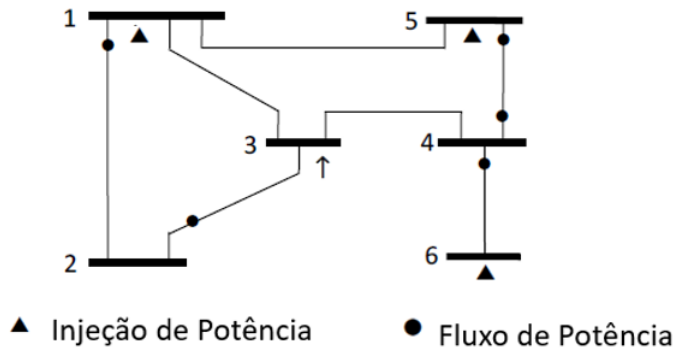

Figura 4.1: Sistema de potência 6 barras e seu esquema de medição contendo 9 medidas.

A matriz simétrica $\boldsymbol{\Omega}$, obtida do modelo desacoplado do sistema de teste e usualmente adotada na análise de observabilidade clássica [1], é apresentada na Tabela (4.1).

Tabela 4.1: Matriz de Covariância Ω, obtida do sistema de 6 barras ilustrado na Figura 4.1.

Medidas #	1 P_{1-2}	2 P_{2-3}	3 P_{4-5}	4 P_{4-6}	5 P_{5-4}	6 P_1	7 P_3	8 P_5	9 P_6
P_{1-2}	0.57	0.38	-0.07	0.00	0.07	-0.23	0.05	-0.19	0.00
P_{2-3}		0.53	0.16	0.00	-0.16	-0.02	0.17	0.15	0.00
P_{4-5}			0.66	0.00	0.34	0.13	0.10	0.23	0.00
P_{4-6}				0.50	0.00	0.00	0.00	0.00	0.50
P_{5-4}					0.66	-0.13	-0.10	-0.23	0.00
P_1						0.17	0.05	0.21	0.00
P_3							0.07	0.12	0.00
P_5								0.34	0.00
P_6									0.50

Como pode ser visto, não há nenhuma coluna/linha nula em $\boldsymbol{\Omega}$, sendo assim, o esquema de medição é livre de qualquer C_{meas}. Para a cardinalidade dois, uma C_2 é identificada envolvendo as medidas #4 e #9, ou seja, no ramo $4-6$ o fluxo de potência ($P_{4,6}$), e a medida de injeção na barra 6 (P_6). A submatriz $\tilde{\boldsymbol{\Omega}}_{4-9}$ é construída a partir das respectivas linhas e colunas na matriz $\boldsymbol{\Omega}$. A Eliminação de Gauss confirma a existência de linhas linearmente dependentes e pela Propriedade 1 é possível confirmar a presença de uma C_2. Avaliando desta forma todas as combinações remanescentes, é confirmado que não há outra C_2 presente no esquema de medição.

$$(\tilde{\Omega}_{4,9}) = \begin{bmatrix} 0.50 & 0.50 \\ 0.50 & 0.50 \end{bmatrix} \tag{4.1}$$

$$(\tilde{\Omega}_{4,9}) = \begin{bmatrix} 0.50 & 0.50 \\ 0.00 & 0.00 \end{bmatrix} \tag{4.2}$$

Para a cardinalidade 3, todas as submatrizes $\tilde{\boldsymbol{\Omega}}$ referentes a combinações de três medidas que contiverem P_{4-6} e P_6 possuem colunas linearmente dependentes. Contudo, devido a propriedade 2, estas não podem ser consideradas C_3. Desconsiderando estas ocorrências, por exemplo, linhas $\#2, \#7$ e #8, que são associadas às medidas (P_{2-3}, P_3, P_5), formam uma C_3. A submatriz $\Omega_{2,7,8}$ em (4.3) e sua forma triangular equivalente (4.4), suportam esta conclusão.

$$(\tilde{\Omega}_{2,7,8}) = \begin{bmatrix} 0.53 & 0.17 & 0.15 \\ 0.17 & 0.07 & 0.12 \\ 0.15 & 0.12 & 0.33 \end{bmatrix} \tag{4.3}$$

$$(\tilde{\Omega}_{2,7,8}) = \begin{bmatrix} 0.53 & 0.06 & 0.15 \\ 0.00 & -0.05 & 0.22 \\ 0.00 & 0.00 & 0.00 \end{bmatrix} \tag{4.4}$$

Junto a (P_{2-3}, P_3, P_5), nove outras C_3 foram identificadas a saber: (P_{1-2}, P_{2-3}, P_1), (P_{1-2}, P_{2-3}, P_3), (P_{1-2}, P_{2-3}, P_5), (P_{1-2}, P_1, P_3), (P_{1-2}, P_1, P_5), (P_{1-2}, P_3, P_5), (P_{2-3}, P_1, P_3), (P_{2-3}, P_1, P_5), (P_1, P_3, P_5). A Tabela 4.2 apresenta todas as C_{ks} encontradas no processo. Existe um limite superior (k_{lim}) para a cardinalidade máxima de C_{ks}, dependente da diferença entre m (número de medidas disponíveis) e n (número de variáveis de estado), dado por: $k_{lim} = m - n + 1$. Sendo assim, para o sistema em análise: $k_{lim} = 9 - 6 + 1 = 4$.

Tabela 4.2: Número de combinações visitadas e C_{ks} identificadas no sistema de 6 barras apresentado na Figura 4.1.

Cardinalidade k	Nº de combinações de medidas visitadas	Nº de C_{ks} identificadas
1	9	0
2	36	1
3	84	10
4	126	10
5	126	0
6	84	0
7	36	0
8	9	0
9	1	0
Total	511	21

4.2 Motivações para Implementação em GPU

A natureza combinatória do problema de identificação de criticalidades tem limitado as pesquisas relacionadas ao tema. Assim, o principal objetivo deste trabalho é adaptar o método de avaliação de criticalidades para GPUs de forma que a análise possa ser realizada em um tempo adequado e que mais estudos sobre o tema possam ser realizados na área.

Contudo, existem certas condições para que uma aplicação apresente um bom desempenho em GPUs. Como visto no Capítulo 3, os núcleos de GPUs são projetados para se sobressair em aplicações que realizam concorrentemente milhares de operações (*threads*) simples. Partindo desse princípio, o método da análise de criticalidades via matriz de covariância de resíduos é bem implementável em GPUs, pois existe um número fatorial de combinações que podem ser avaliadas simultaneamente. Além disso, estas avaliações são compostas de operações simples, teste de singularidade de matrizes pequenas. Sendo assim, como se vê na Figura 4.2, a implementação do método em GPUs consiste essencialmente em designar milhares de *threads* para realizar concorrentemente a eliminação de Gauss em sub-matrizes de pequena dimensão.

Além do requisito da implementação ser realizada paralelamente em múltiplas *threads*, também é necessário salientar a questão da transferência de dados entre arquiteturas. Como foi descrito, GPUs atuam como um co-processador (agilizador de *hardware*) para a CPU e que ambas as arquiteturas são equipamentos com memórias principais independentes e fisicamente separadas.

Um código em CUDA inicializa sua execução na CPU e quando uma etapa da aplicação requisita um processamento paralelo massivo, a execução é movida para a GPU. De forma similar, os dados que serão manuseados durante o processo precisam ser previamente transferidos coletados após o término da execução. Estas transferências são custosas em termos de desempenho e devem ser minimizadas, um programa eficiente deve investir mais tempo realizando cálculos que manejando dados.

Considerando também este ponto, o método da matriz de covariância se adapta bem a estas condições, pois depende da transferência de $\boldsymbol{\Omega}$ para realizar a análise de todas as combinações e do retorno das criticalidades encontradas à CPU no fim do processo, como ilustrada na Figura 4.2.

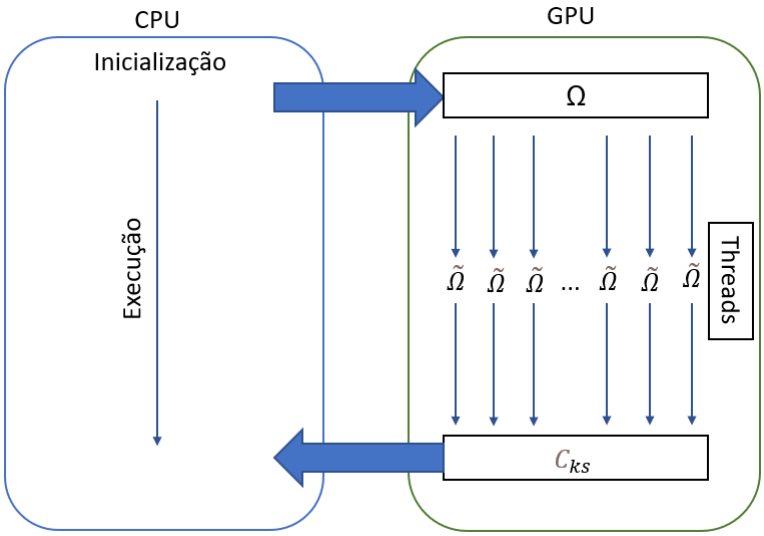

Figura 4.2: Representação do fluxo de execução e dados para a aplicação proposta em GPUs.

4.3 Algoritmo Proposto

A presente seção descreve o algoritmo computacional proposto, baseado no método da matriz de covariância de resíduos, que identifica todas as C_{ks} presentes num esquema de medição, apresentando adaptações realizadas para programação paralela em CPUs e GPUs. Para realizar esta tarefa, são fornecidos como parâmetros de entrada: $\boldsymbol{\Omega}$, o número de medidas disponíveis (m) e a cardinalidade máxima que será avaliada (k_{max}). Por fim, o algoritmo retorna um conjunto-solução ($SolSet$) contendo todas as C_{ks} identificadas.

4.3.1 Etapas

As etapas estabelecidas baseiam-se na estratégia de projeto apresentada pelos guias de programação em CUDA da NVIDIA [45]. O primeiro passo para se iniciar uma adaptação de um código para programação paralela é particionar a aplicação computacional em etapas principais. Esta prática permite identificar as porções do código que podem se beneficiar melhor da adaptação proposta, permitindo uma paralelização parcial do código onde os ganhos de desempenho são obtidos de forma gradativa.

Seguindo esta estratégia, o processo de análise de criticalidade baseia-se em quatro etapas principais, sendo realizadas a cada iteração até o valor k atingir um k_{max} preestabelecido, como apresentado no Algoritmo 3. Na etapa 1 (Enumeração) todas as combinações pretendidas de k medidas são listadas. Na etapa 2 (Avaliação) são demarcadas todas as combinações de k medidas que tornam o sistema inobservável, quando indisponíveis simultaneamente. Na etapa 3 (Confirmação), como uma C_k não pode incluir uma C_j de cardinalidade inferior ($j < k$), são excluídas da solução final todas as combinações que não atendem esta propriedade. Por fim, na etapa 4 (Atualização), as combinações confirmadas na etapa anterior são adicionadas ao $SolSet$.

Algoritmo 3: Entrada, Saída e Etapas envolvidas na análise de criticalidades.

Entrada: Ω, m, k_{max}
Saída : $solSet$
para $k \leftarrow 1$ ***até*** k_{max} **faça**
  Etapa 1: Enumeração
  Etapa 2: Avaliação
  Etapa 3: Confirmação
  Etapa 4: Atualização

Certas cardinalidades apresentaram um número extenso de combinações para ser armazenado em uma única estrutura. Nestes casos, a análise foi particionada de forma que 2^{20} combinações são avaliadas por vez. Este número permitiu um uso mais eficiente da arquitetura da GPU por ser múltiplo de 32, como foi apresentado no Capítulo 3. Sendo assim, como nesta implementação cada *thread* é responsável por tratar uma combinação e foram utilizadas 1024 *threads* por bloco, as declarações de *kernels* invocaram $\frac{2^{20}}{2^5} = 2^{15}$ blocos de *threads*.

Sendo assim, a seguir serão apresentadas justificativas para a computação envolvida em cada uma das etapas e as adaptações realizadas para executar tais etapas com o auxílio de arquiteturas paralelas.

4.4 Enumeração

Esta etapa do programa desenvolvido enumera e armazena em uma matriz denominada ***Combs*** todas as combinações de k medidas, removidas de um esquema de medição de tamanho m. A alocação das combinações em uma matriz, possibilita o acesso simultâneo nas etapas subsequentes. Cada combinação é expressa por um vetor binário de tamanho m onde os valores unitários representam medidas indisponíveis, conforme mos-

tra a Figura 4.3. O total de combinações é calculado por $\binom{k}{m}$. Sabendo disso, ***Combs*** é $[\binom{k}{m} \times m]$.

Tabela 4.3: Representação da remoção das medidas de C_{ks} $\{P_{4-6}; P_6\}$ (primeira linha) e $\{P_{2-3}; P_3\}$ (segunda linha).

$\boldsymbol{P_{1-2}}$	$\boldsymbol{P_{2-3}}$	$\boldsymbol{P_{4-5}}$	$\boldsymbol{P_{4-6}}$	$\boldsymbol{P_{5-4}}$	$\boldsymbol{P_1}$	$\boldsymbol{P_3}$	$\boldsymbol{P_5}$	$\boldsymbol{P_6}$
0	0	0	1	0	0	0	0	1
0	1	0	0	0	0	1	0	0

Na sequência, serão apresentados dois algoritmos a saber: o algoritmo denominado *coolex*, que é sequencial e foi usado em trabalhos anteriores [4, 21], e um segundo algoritmo que utilizado para gerar combinações de forma concorrente.

4.4.1 Enumeração Sequencial

O algoritmo *Coolex*, descrito em [22], é capaz de enumerar sequencialmente todas as combinações de k elementos em m espaços. A partir de um vetor binário inicial, a combinação seguinte é gerada pela "rotação" dos valores unitários da combinação anterior, como mostra a Figura 4.3. Apesar de eficiente, este método requer um resultado gerado anteriormente, o que torna inviável sua adaptação para programação em paralela na GPU. Devido a este fato, outro algoritmo precisou ser desenvolvido para possibilitar uma enumeração concorrente.

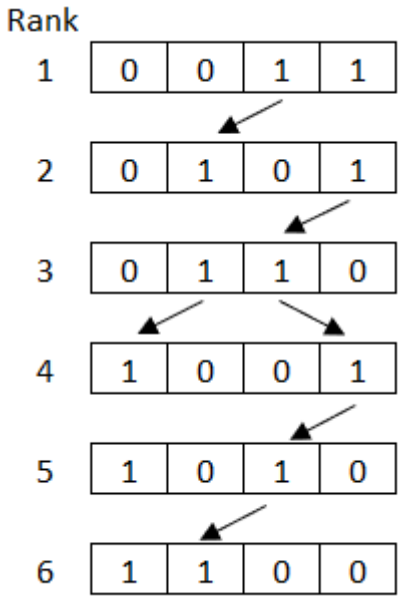

Figura 4.3: Enumeração de $m = 4$ para $k = 2$ realizada pelo algoritmo *Coolex*.

4.4.2 Enumeração Paralela

Algoritmos para enumeração paralela de combinações podem ser encontrados em trabalhos relacionados à listagem de anagramas (permutações de letras para formar palavras). Sendo assim, é necessário definir formalmente a ordem lexicográfica ou alfabética:

Considere $X = [x_1, x_2...x_r]$ e $Y = [y_1, y_2...y_s]$. $X < Y$ se e somente se existe um t não negativo onde:

1. $t \leq r$ e $t \leq s$;
2. Para todo inteiro positivo $i < t, x_i = y_i$;
3. $x_{t+1} < y_{t+1}$;

Utilizando este conceito, o algoritmo apresentado em [48] recebe como entrada um valor n e um conjunto de letras, retornando diretamente a n-ésima permutação destas letras, sem precisar consultar as permutações anteriores. Exemplificando: recebendo como entrada n=3 e "abbc", a saída será "acbb" pois é a terceira palavra seguindo a ordem lexicográfica:{abbc, abcb, acbb, babc, bacb, bbac, bbca, bcab, bcba, cabb, cbab, cbba}. Seguindo esta proposta, algoritmo apresentado em [56] usa este conceito e explora a programação em GPUs para realizar a enumeração simultânea de todas as palavras que contenham o conjunto de letras inicial.

Para aproveitar este algoritmo no problema da análise de criticalidades foram realizadas simplificações, pois são necessários apenas dois caracteres para representar indisponibilidades de medidas: "0"para representar disponíveis e "1"para representar indisponíveis. O tamanho das palavras pode ser expresso por m, que é o tamanho do esquema de medição, e quantidades de caracteres '1's pode ser obtida através de k que é a cardinalidade sendo analisada. Para exemplificar a nova implementação: caso sejam recebidos como entrada os valores n=3, m=3 e k=2, o a combinação retornada será {110}, pois a ordem lexicográfica será {011, 101, 110}.

A partir destas adaptações, o Algoritmo 4 é construído e expressa como pode ser gerada a n-ésima combinação. Inicialmente, o algoritmo contabiliza o número de $0's$ ($nZeros$) e de $1's$ ($nOnes$) presentes na combinação a partir dos valores de entrada m e k. Supondo que o primeiro elemento da n-ésima combinação seja zero, é possível calcular o número combinações de $(m-1)$ elementos com $nZeros-1$ por $\binom{m-1}{nZeros-1}$. Se este valor for maior que n, é confirmado que o primeiro elemento é 0, caso contrário, este é confirmado como 1, $nZeros$ é incrementado novamente e $nOnes$ é decrementado. Este processo será realizado m vezes até que a $n-$ésima combinação de medidas seja totalmente expressa e armazenada em $\boldsymbol{Combs}$.

Algoritmo 4: Algoritmo para enumerar paralelamente todos as combinações de m medidas contendo k indisponibilidades.

Entrada: n, m, k
Saída : $\boldsymbol{Combs}$
$nZeros = m - k$;
$nOnes = k$;
para $i \leftarrow 0$ ***até*** m **faça**
  $nZeros - -$;
  $zComb = \binom{m-1-i}{nZeros}$;
  se $zComb < n$ **então**
    $nOnes - -$;
    $nZeros + +$;
    $n = n - zComb$;
    $\boldsymbol{Combs[n][i] = 1}$;
  senão
    $\boldsymbol{Combs[n][i] = 0}$;
  fim
fim

Desconsiderando os cálculos fatoriais para obter a quantidade de combinações $\binom{m-1-i}{nZeros}$, este algoritmo possui complexidade linear O(m) [56]. Sabendo disso, com o intuito de aprimorar a desempenho do algoritmo, estes cálculos podem ser realizados e armazenados

previamente o que possibilita o acesso direto aos resultados.

A implementação paralela é realizada de forma direta, o valor de n passa a receber o *id* da *thread* responsável permitindo que todas as combinações sejam enumeradas e armazenadas, simultaneamente, na matriz $\boldsymbol{Combs}$. Especificamente na implementação na GPU, esta etapa requer a transferência prévia da matriz $\boldsymbol{\Omega}$ e dos resultados dos cálculos fatoriais para a memória global da GPU.

4.5 Avaliação

Esta etapa identifica quais combinações de medidas resultam na perda de observabilidade do sistema, quando removidas simultaneamente, de acordo com as propriedades de $\boldsymbol{\Omega}$. Para este fim, o algoritmo constrói uma submatriz auxiliar $\tilde{\boldsymbol{\Omega}}$ para cada combinação listada em $\boldsymbol{Combs}$. Em seguida, ele avalia a singularidade de cada $\tilde{\boldsymbol{\Omega}}$ por intermédio da Eliminação de Gauss.

Como pode ser evidenciado no Algoritmo 5, o vetor auxiliar binário $\boldsymbol{isCrit}$ é utilizado para demarcar as combinações de medidas que formam as C_{ks}, cada elemento deste vetor corresponde a uma linha de $\boldsymbol{Combs}$. Se o algoritmo encontra uma $\tilde{\boldsymbol{\Omega}}$ singular, ele atribui o valor 1 no seu índice correspondente em $\boldsymbol{isCrit}$ ou 0 caso contrário.

Algoritmo 5: Etapa 2: Avaliação.

```
para i ← 0 até (k m) faça
    Constrói Ω̃(Ω, k, Combs);
    se Ω̃ singular então
        isCrit[i]=1;
    senão
        isCrit[i]=0;
    fim
fim
```

A adaptação desta etapa para programação paralela realiza cada laço do algoritmo 5 simultaneamente. O vetor $\boldsymbol{isCrit}$ permite que *threads* demarquem simultaneamente quais linhas de $\boldsymbol{Combs}$ ocasionam em uma $\tilde{\boldsymbol{\Omega}}$ singular. Especificamente, a realização desta etapa na GPU requer o armazenamento prévio de $\tilde{\boldsymbol{\Omega}}$ na memória global antes do laço principal.

4.6 Confirmação

Esta etapa confirma as possíveis C_{ks} encontradas na etapa anterior, utilizando as propriedades de $\tilde{\boldsymbol{\Omega}}$. Para tal, considere P o conjunto das $C_j - Tuplas$ $(1 \leq j \leq k-1)$ anteriormente adicionadas em ***SolSet*** e já confirmadas. Dessa forma, cada nova $C_k - tupla$ candidata é confrontada com o conjunto P para validar a sua condição de criticalidade. Assim, caso uma linha em ***Combs*** possuir o seu valor em ***isCrit*** igual a 1, e contiver qualquer C_j, então a combinação de medidas encontrada no passo anterior não é uma C_k, consequentemente, o algoritmo atualiza o valor correspondente em ***isCrit*** para 0.

O Algoritmo 6 utiliza uma subtração elementar para averiguar se um vetor contém outro. Por exemplo, considerando $C_1 = \{0010\}$ uma criticalidade anteriormente armazenada em ***SolSet*** e $C_2 = \{0011\}$ uma possível C_k encontrada na segunda etapa. Como pode ser visto na Figura 4.4, subtraindo cada elemento deste vetor, $(C_2 - C_1)$, nenhum elemento resultante foi igual a -1, sendo assim, é possível concluir que $C_1 \subset C_2$. Então, $C_2 = \{m_3, m_4\}$ não pode ser considerada crítica e seu elemento correspondente em ***isCrit*** é atualizado para 0. O mesmo não acontece caso $C_2 = \{0101\}$. Nesta situação, a subtração retorna um valor -1, logo $C_1 \not\subset C_2$, o que confirma a $C_2 = \{m_3, m_4\}$ como crítica.

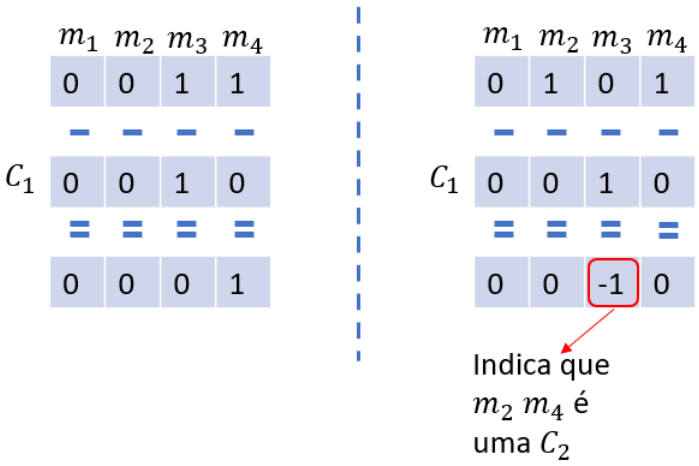

Figura 4.4: Exemplo da confirmação dos casos $C_2 = (m_3, m_4)$.

A adaptação desta etapa para programação paralela executa concorrentemente o laço interno do Algoritmo 6. Em outras palavras, cada solução já armazenada no conjunto solução é comparada, simultaneamente, com todas as possíveis combinações demarcadas na etapa anterior.

Algoritmo 6: Etapa 3: Confirmação.

```
para j em SolSet faça
    para i=0 até (k m) faça
        se isCrit[i]==1 então
            se SolSet[j] ⊂ Combs[i] então
                isCrit[i] == 0;
            fim
        fim
    fim
fim
```

4.7 Atualização

Nesta etapa as combinações que foram demarcadas como críticas são adicionadas ao conjunto solução. Como pode ser visto no Algoritmo 7, as combinações de medidas i que possuem o valor $\boldsymbol{isCrit}[i] = 1$ são adicionadas a $\boldsymbol{SolSet}$.

Algoritmo 7: Etapa 4: Atualização.

```
para i =0 até (k m) faça
    se isCrit[i]==1 então
        Adiciona(Combs[i], SolSet)
    fim
fim
```

Como inicialmente, o número de C_{ks} presentes no esquema de medição é desconhecido e muito menor que o total de combinações avaliadas, a estrutura de pilha se mostrou eficaz para representar $\boldsymbol{SolSet}$. Em contrapartida, a adição de elementos de forma paralela torna-se não-trivial com essa escolha. Para solucionar esse obstáculo, duas primitivas da programação paralela foram utilizadas. A primeira primitiva, denominada *Scan* ou *Cumulative Sum*, consiste em gerar uma sequência de números $y_0, y_1, y_2...$, a partir de outra $x_0, x_1, x_2...$, onde $y_i = \sum_{j=0}^{i-1} x_j$, o algoritmo utilizado nesta implementação que aproveita também a memória compartilhada da GPU para realizar esta tarefa, para garantir maior desempenho, pode ser visto em [43]. Já segunda primitiva, o *Compact*, utiliza da sequência gerada pelo *Scan* para comprimir dados de um vetor em outro de menor tamanho.

A Figura 4.5 exemplifica como estas primitivas foram aplicadas ao trabalho. Considere a matriz $\boldsymbol{Combs}$, onde as colunas 1 e 3 representam C_{2s}, evidentemente, seus elementos equivalentes no vetor $\boldsymbol{isCrit}$ possuem valor 1. Sabendo disso, o *Scan* é utilizado para gerar um vetor auxiliar, e em seguida, o *Compact* utiliza os elementos equivalentes às C_{ks} no vetor auxiliar para armazenar paralelamente as soluções em *solSet*.

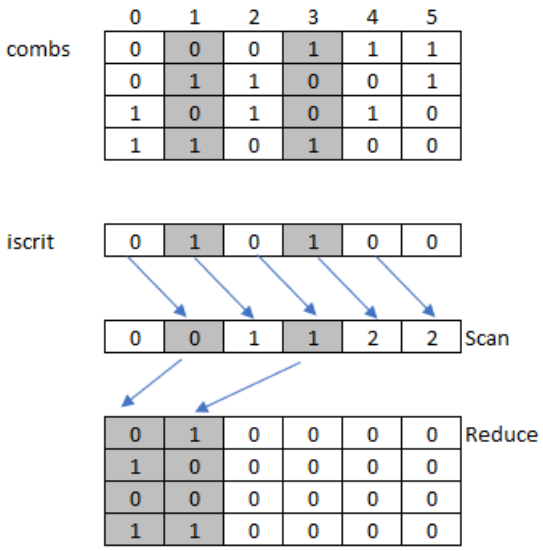

Figura 4.5: Exemplo do uso das primitivas *Scan* e *Compact* no armazenamento das C_2 1 e 3 em $solSet$.

4.8 Organização dos Kernels

Para auxiliar a reprodutibilidade do trabalho, o Algoritmo 8 apresenta a organização dos *kernels* invocados para serem realizados na GPU, assim como, os principais parâmetros requisitados em suas execuções. Foi utilizado um laço principal que incrementa o valor k até a cardinalidade máxima k_{max}. Devido a um número extenso de combinações presentes em certas cardinalidade, o laço interior particiona análise destes casos, possibilitando que $c_{max} = 2^{20}$ combinações sejam analisadas simultaneamente até que todas as $\binom{m}{k}$ combinações sejam analisadas. Também foi decidido arbitrativamente utilizar 1024 *threads* por bloco (tpb).

A Etapa 1 de Enumeração é realizada por apenas um *kernel* que requisita valor k analisado, o número de combinações já avaliadas ($cAnalidas$) e uma matriz contento os resultados dos cálculos fatoriais (C_n), previamente armazenados. Ao fim do processo, o *kernel* retorna a matriz $Combs$ contendo todas as combinações enumeradas. Sem seguida, a Etapa 2 de Avaliação requisita da matriz Ω, previamente armazenada na GPU, e de $Combs$, construída na etapa anterior. O vetor $isCrit$ que informa quais combinações são críticas é retornada ao fim da execução do *kernel*. Na Etapa 3 de Confirmação um *kernel* é invocado para cada combinação previamente armazenada no conjunto solução, permitindo assim, uma comparação simultânea. Esta função recebe como parâmetros de entrada o $i-$ésimo elemento do conjunto solução, $Combs$ e retorna ao fim o vetor

isCrit atualizado com as combinações confirmadas. Por fim, a Etapa 4 de Atualização realiza as duas primitivas *Scan* e *Compact*. Como apresentado em [43], são necessários três *kernels* para realizar o *Scan* em um vetor extenso de 2^{20} elementos. Em sequencia, utilizando o vetor auxiliar (*scan*) resultado do processo anterior, o último *kernel* realiza a primitiva *compact* compactando os elementos de combs, demarcados como críticos por *isCrit* em *SolSet*. Ao fim do laço principal, *SolSet* é retornado para CPU contendo todas as criticalidade do plano de medição.

Algoritmo 8: Organização dos Kernels

$tpb = 1024$
$c_{max} = 2^{20}$
para $k \leftarrow 1$ ***até*** k_{max} **faça**
  repita
    `Etapa 1`
    `enumerar`$<<< max_{Combs}/tbp, tpb >>>$`(`$cAnalisadas, k, C_n,\ Combs$`)`
    `Etapa 2`
    `avaliar`$<<< max_{Combs}/tbp, tpb >>>$`(`$k,\ \Omega,\ Combs,\ isCrit$`)`
    `Etapa 3`
    para i ***em*** $SolSet$ **faça**
      `confirmar`$<<< max_{Combs}/tbp, tpb >>>$`(`$SolSet[i],\ combs,\ isCrit$`)`
    fim
    `Etapa 4`
    `Scan:`
    `scan`$<<< c_{max}/tpb, tpb >>>$`(`$scan, isCrit, sums$`)`
    `scan`$<<< 1, tpb >>>$`(`$sums$`)`
    `scanAdd`$<<< c_{max}/tpb, tpb >>>$`(`$scan,\ sum,\ SolSet$`)`
    `Compact:`
    `compact`$<<< c_{max}/tpb, tpb >>>$`(`$Combs,\ isCrit,\ scan,\ SolSet$`)`
    $cAnalisadas = cAnalisadas + c_{max}$
  até $cAnalisadas >= \binom{m}{k}$;
fim

4.9 Visão Geral

Para sintetizar o funcionamento do algoritmo proposto, considere um esquema de medição contendo quatro medidas, considere também que o esquema contém uma medida crítica m_3, adicionada em ***SolSet*** na primeira iteração ($k = 1$).

A Figura 4.6 apresenta um exemplo dos processos realizados em cada etapa para o sistema de medição previamente dito, quando $k = 2$. A partir dos valores de entrada, a primeira etapa enumera simultaneamente as linhas da matriz de combinações ***Combs***. Em seguida, na segunda etapa, as submatrizes $\boldsymbol{\Omega'}$ são construídas de forma concorrente e as combinações de medidas que resultarem e matrizes singulares são demarcadas em ***iscrit*** com o valor 1. Na terceira etapa, m_3, que foi previamente alocada em ***SolSet***, é comparada com todas as combinações enumeradas, caso $m_3 \subset \boldsymbol{combs_i}$ o valor correspondente $iscrit[i]$ é atualizado para 0. Por fim, as combinações que apresentarem seu valor ***iscrit***$= 1$ são compactadas no conjunto solução ***SolSet*** utilizando as primitivas *Scan* e *Compact*.

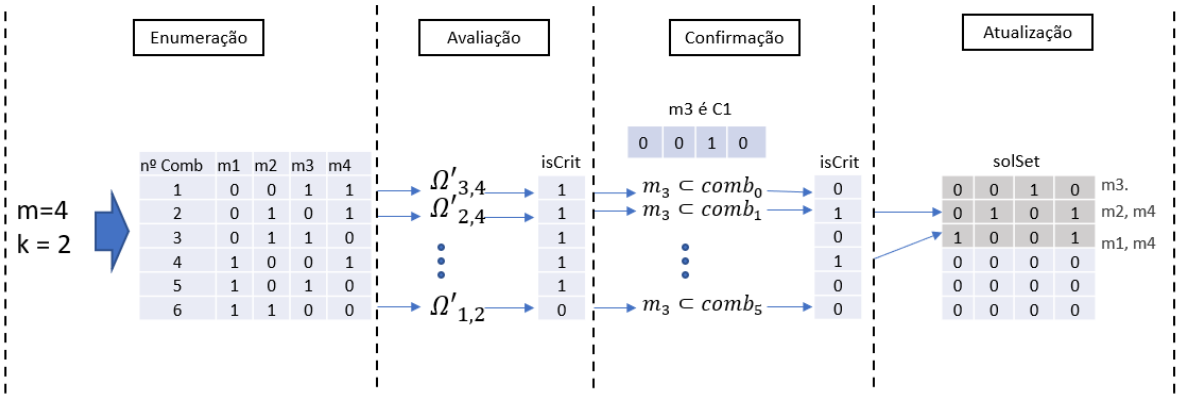

Figura 4.6: Exemplo da execução das etapas do Algoritmo proposto

Capítulo 5

Análise de Resultados

Este capítulo apresenta uma comparação dos resultados obtidos na implementação sequencial e suas adaptações para arquiteturas paralelas. Em síntese foram realizadas três implementações: CPU (*single-thread*), CPU (*multi-threads*), GPU (*many-threads*). A segunda implementação foi incluída buscando-se uma comparação justa, pois como foi ilustrado no Capítulo 3, algumas implementações não são bem aplicáveis em GPUs.

5.1 Descrição das Simulações

As simulações foram realizadas utilizando um computador com 8GB de RAM, processador Intel Core i5-9300H e placa de vídeo Nvidia GeForce GTX 1650. O código original foi implementado em uma única *thread* em C++. A versão do código para a arquitetura paralela de CPU foi implementada utilizando o auxílio de diretivas do *OpenMP* e utilizou 8 *threads*. A versão do código para a arquitetura da GPU foi implementada em CUDA e utilizou 1.024 *threads* invocadas por bloco em cada chamada de *kernel*. Por fim, as simulações foram realizadas incrementando o valor k até 10 ou até que um tempo cômodo de execução (30 minutos) fosse superado pela implementação sequencial.

5.1.1 Casos Estudados

Cada uma das três implementações foi avaliada nos sistemas do IEEE de 14, 30 e 118 barras, contendo os esquemas de medição utilizados em trabalhos encontrados na literatura. Esquemas com diversos níveis de redundância foram utilizados, como será visto a seguir, estando tais níveis relacionados ao tempo computacional necessário à execução da etapa de Confirmação.

Visando padronizar o cálculo da redundância do sistema de medição [19] foram consideradas apenas medidas convencionais de fluxo e injeção de potência, e para tal utilizou-se a seguinte equação:

$$\frac{m - nbar}{m_{max} - nbar} \tag{5.1}$$

onde:

- m representa o número total de medidas no esquema de medição.
- $nbar$ representa o número de barras do sistema.
- m_{max} o número máximo de medidas de fluxo e de injeção de potência que podem ser alocadas na rede elétrica 5.1.

5.1.1.1 Sistema de 14 Barras

Adotou-se o sistema teste de 14 barras do IEEE com o esquema de medição apresentado em [6], que possui a sua disposição 33 medidas distribuídas conforme consta na Figura 5.1. Este sistema possui uma redundância de medidas de 48%, conforme a equação (5.1).

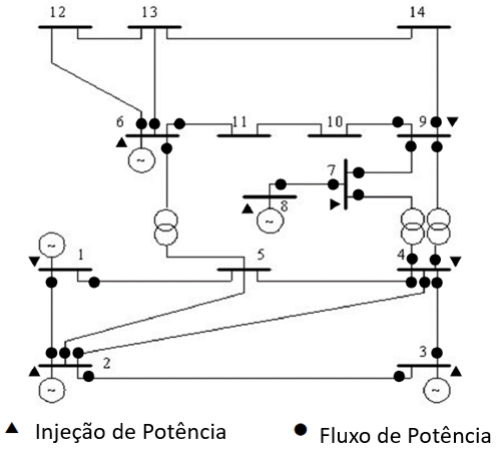

Figura 5.1: Sistema IEEE 14-barras e respectivo esquema de medição [6].

A Tabela 5.1 apresenta o número de combinações avaliadas durante a simulação. Como pode ser visto, este valor cresce de forma acentuada, demonstrando a complexidade do problema. Nota-se que o número de combinações a ser analisado depende unicamente do número de medidas presentes no sistema. A Tabela 5.1 também apresenta o número

de C_{ks} encontradas até a cardinalidade máxima estabelecida pela condição de parada da simulação ($k = 10$).

Tabela 5.1: Espaço de busca para o sistema IEEE 14-barras.

Cardinalidade k	**Nº combinações visitadas de medidas**	**Nº de** C_{ks} **identificadas**
1	33	0
2	528	13
3	5.456	0
4	40.920	1
5	237.336	1
6	1.107.568	1
7	4.272.048	13
8	13.884.156	9
9	38.567.100	13
10	92.561.039	11

5.1.1.2 Sistema de 30 Barras

O sistema IEEE de 30 barras utilizado nas simulações possui um esquema de medição contendo 70 medidas [21], sendo constituído também por medidas de ângulo e de corrente, distribuídas conforme a Figura 5.2. Este sistema possui uma redundância de 49%, segundo definição estabelecida.

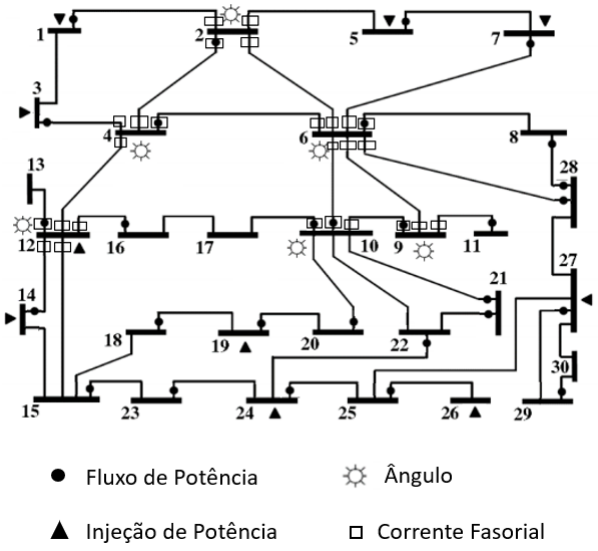

Figura 5.2: Sistema IEEE 30-barras e respectivo esquema de medição [21].

Como pode ser visto na Tabela 5.2, devido ao maior número de medidas presente no sistema, mais combinações precisaram ser visitadas. Como resultado disso a análise tornou-se mais custosa, inclusive na busca por criticalidades de cardinalidades mais baixas. Havendo a violação do limite de tempo estabelecido para a implementação original, as simulações realizadas neste sistema foram interrompidas para um $k = 7$.

Tabela 5.2: Espaço de busca para o sistema IEEE 30-barras.

Cardinalidade k	**Nº combinações visitadas de medidas**	**Nº de** C_{ks} **identificadas**
1	70	4
2	2.415	10
3	54.740	21
4	916.895	8
5	12.103.014	26
6	131.115.985	42
7	1.198.774.720	35

5.1.1.3 Sistema de 118 Barras

Por fim, testes também foram realizados no sistema IEEE de 118 barras contendo 176 medidas [4], representado na Figura 5.3. Este sistema é formado apenas por medidas de fluxo e possui uma redundância de 16%, que é menor que as apresentadas nos sistemas anteriores.

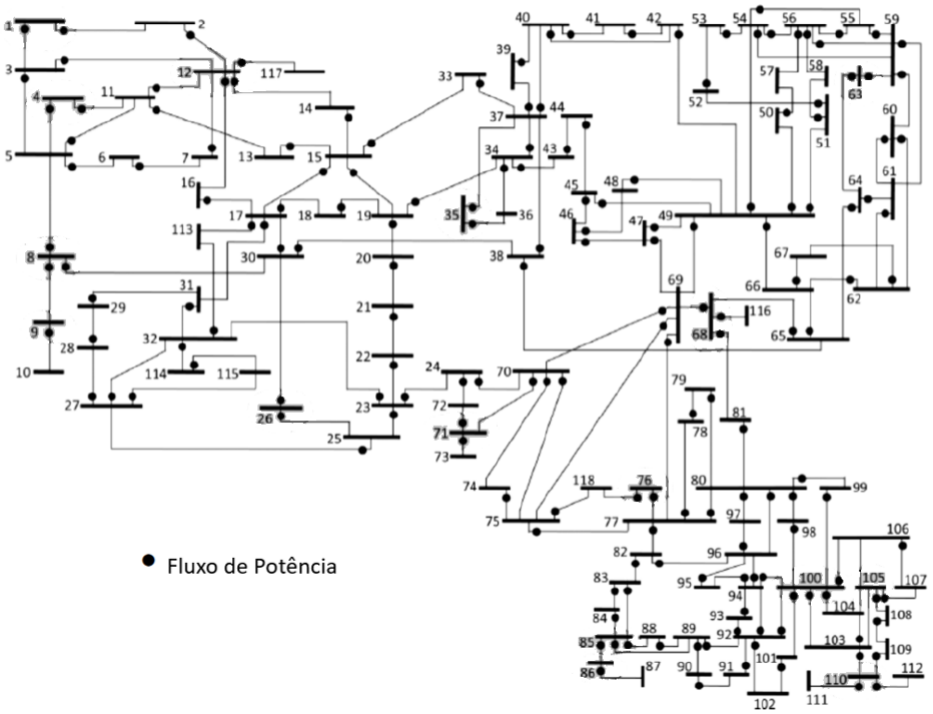

Figura 5.3: Sistema IEEE 118-barras e respectivo esquema de medição. [4].

Analisando a Tabela 5.3, conclui-se que, devido ao maior número de medidores presentes neste sistema, o número de combinações cresce de forma mais drástica do que observado para os sistemas anteriores. À medida que o valor da cardinalidade k é incrementado o número de combinações a serem analisadas aumenta substancialmente. Também é possível evidenciar que devido à menor redundância de medidas presentes no plano de medição, mais C_{ks} são encontradas, principalmente nas cardinalidades mais baixas.

Tabela 5.3: Espaço de Busca para o sistema IEEE 118-barras.

Cardinalidade k	**Nº combinações visitadas de medidas**	**Nº de** C_{ks} **identificadas**
1	176	9
2	15.400	91
3	893.200	56
4	38.630.900	118
5	1.328.902.960	289

5.1.2 Métrica de Desempenho

Para comparar os ganhos obtidos pela implementação paralela foi utilizada a métrica de *speed-up*, a qual representa o ganho de velocidade de processamento em relação à implementação sequencial. Quanto maior o *speed-up*, mais eficiente é o algoritmo do código paralelo. Este fator pode ser obtido a partir da seguinte equação:

$$S_n = \frac{T_p}{T_s}, \tag{5.2}$$

onde T_p — é o tempo paralelo após as adaptações e T_s — é o tempo serial original. Neste trabalho, foram calculados *speed-ups* para comparar o desempenho de ambas implementações que utilizaram arquiteturas paralelas (CPU e GPU) em relação a programação sequencial original.

5.2 Resultados

Reúnem-se aqui os resultados obtidos nas implementações do problema. Como apresentado no Capítulo 4, a segmentação da aplicação permite identificar as etapas do programa que demandam maior esforço computacional. Além disso, possibilita que a adaptação para a programação paralela seja realizada de forma gradativa. Isto posto, o processo da análise de criticalidades via matriz $\boldsymbol{\Omega}$ foi particionado em quatro etapas:

- Enumeração: Aqui todas as combinações de medidas são listadas.
- Avaliação: Nesta etapa, são demarcadas quais combinações de medidas causam perda de observabilidade.

- Confirmação: Todas as combinações que possuírem criticalidades de cardinalidade inferior são excluídas da solução final.

- Atualização: Etapa em que o conjunto solução é atualizado de acordo com as combinações demarcadas nas etapas anteriores.

Com estas etapas estabelecidas, a Tabela 5.4 apresenta o percentual do tempo computacional dispendido em cada estágio nos casos avaliados neste trabalho. Como pode ser visto, as etapas 2 e 3 são as mais custosas. Vale ressaltar que a etapa 2 é comumente a mais fastidiosa, contudo, isto não ocorre em sistemas que possuem redundâncias menores, conforme pode ser identificado no caso com 118 barras. Isto posto, aligeirar estas etapas, através de da arquitetura paralela da GPU, é o principal objetivo e contribuição deste trabalho.

Tabela 5.4: Percentual do tempo gasto nas etapas da análise de criticalidades.

Nº barras	**14**	**30**	**118**
Nº medidas	**33**	**70**	**176**
Enumeração	3%	8%	11%
Avaliação	87%	72%	39%
Confirmação	9%	19%	49%
Atualização	1%	1%	1%

5.2.1 Simulação 1: Adaptação das etapas mais custosas

Primeiramente, foram realizadas simulações em que apenas as etapas de Avaliação e Confirmação foram adaptadas para a programação paralela. Sob estas condições foram registrados, para a qualificação de desempenho, os tempos computacionais dispendidos em cada uma destas etapas.

As tabelas contendo os tempos particionados por cardinalidade podem ser encontradas no Apêndice B.1. Devido ao pequeno número de combinações a serem analisadas nas cardinalidades de ordem inferior, o tempo computacional da implementação sequencial se mostrou desprezível, o que torna desnecessária a adaptação para programação paralela quando: k<6 para o caso de 14 barras, k<4 para o caso de 30 barras e k<3 para o caso de 118 barras. Observando-se os resultados para tais valores de cardinalidade k as Tabelas 5.1, 5.2 e 5.3 pode-se identificar que o uso da programação paralela torna-se viável para cardinalidades que contenham uma quantidade de combinações em torno de 10^6.

A Tabela 5.5 apresenta o tempo total consumido pela etapa de Avaliação nas três implementações. Nas simulações envolvendo GPUs também foi contabilizado o tempo investido na transferência de dados entre CPU (*host*) e GPU (*device*). Também são apresentados na Tabela 5.5 os *speed-ups* obtidos pelas adaptações para programação paralela, em comparação com a implementação sequencial original. Por uma questão de simplicidade, os valores de *speed-up* serão apresentados como o número inteiro mais próximo, o que facilita a visualização dos resultados e não compromete a análise comparativa realizada. Assim sendo, nota-se que a implementação que utilizou GPU apresentou os melhores resultados, alcançando um *speed-up* de 14x para o sistema de 14 barras.

Tabela 5.5: Tempos para 2ª etapa (Avaliação)

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	458,53	126,79	4	31,65	14
30	2.170,32	613,09	4	218,47	10
118	2.317,46	710,60	3	369,51	6

De forma similar, a Tabela 5.6 apresenta os tempos computacionais requisitados e os *speed-ups* obtidos para a etapa de confirmação. De forma similar à etapa anterior, a GPU também apresentou os melhores *speed-ups*, atingindo o valor de 20x para o caso de 14-barras.

Tabela 5.6: Tempos para 3ª etapa (Confirmação)

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	49,29	22,98	2	2,52	20
30	584,71	271,92	2	35,57	16
118	3.003,56	1.729,54	2	184,41	16

Como pode ser visto, os *speed-ups* obtidos pela implementação paralela em CPUs foram menores que os obtidos na etapa anterior, em contrapartida, a GPU obteve resultados ainda melhores. Isso se justifica pelo fato de que as *threads* desta etapa realizam tarefas computacionalmente mais simples que a inversão de matrizes da etapa anterior.

Apesar dos altos *speed-ups* alcançados pelas implementações em GPUs, é possível notar uma queda nos valores de *speed-up* obtidos conforme o aumento do número de

medidas presentes em cada sistema de medição. Este fato demonstra uma adaptação não escalável da aplicação.

Utilizando a ferramenta NVIDIA profiler [47], percebe-se que a GPU demanda um tempo computacional considerável na transferência de dados. A Tabela 5.7 apresenta o diagnóstico desta ferramenta para as três implementações. Observa-se, para o sistema de 14 barras, que cerca de 14% do tempo total do processo é investido nesta movimentação. Este percentual se eleva à medida que o sistema aumenta, alcançando 40% do tempo da implementação da GPU para o sistema de 118 barras, se tornando a etapa mais custosa de todos os processos e demonstrando uma debilidade frente à escalabilidade do problema.

Tabela 5.7: Percentual de tempo investido pela GPU.

	Casos		
Atividade	**14**	**30**	**118**
Avaliar	82%	58%	30%
Enumerar	4%	7%	30%
Transferência	14%	35%	40%

Sendo assim, apesar das etapas Enumeração e de Atualização exigirem menor esforço computacional, a adaptação destas para a GPU também é necessária para reduzir o tempo investido na transferência de dados entre *host* e *device*. A Figura 5.4 apresenta o fluxo de dados necessários entre as unidades de processamento em ambos os cenários. Com a ausência de adaptação na etapa de Enumeração para a GPU, todas as combinações listadas precisam ser transferidas para o *device* a cada incremento de k. Analogamente, com a etapa de Atualização sendo realizada na CPU, a estrutura contendo todas as C_{ks}, encontradas pelas etapas de Avaliação e Confirmação, tem de ser retornada para o *host* a cada iteração. Com a adequação destas etapas para a GPU, nenhuma transferência de dados é feita durante as iterações. Neste cenário, apenas o armazenamento da matriz $\boldsymbol{\Omega}$ é realizado durante a inicialização do programa seguido do retorno das C_{ks} ao fim da execução.

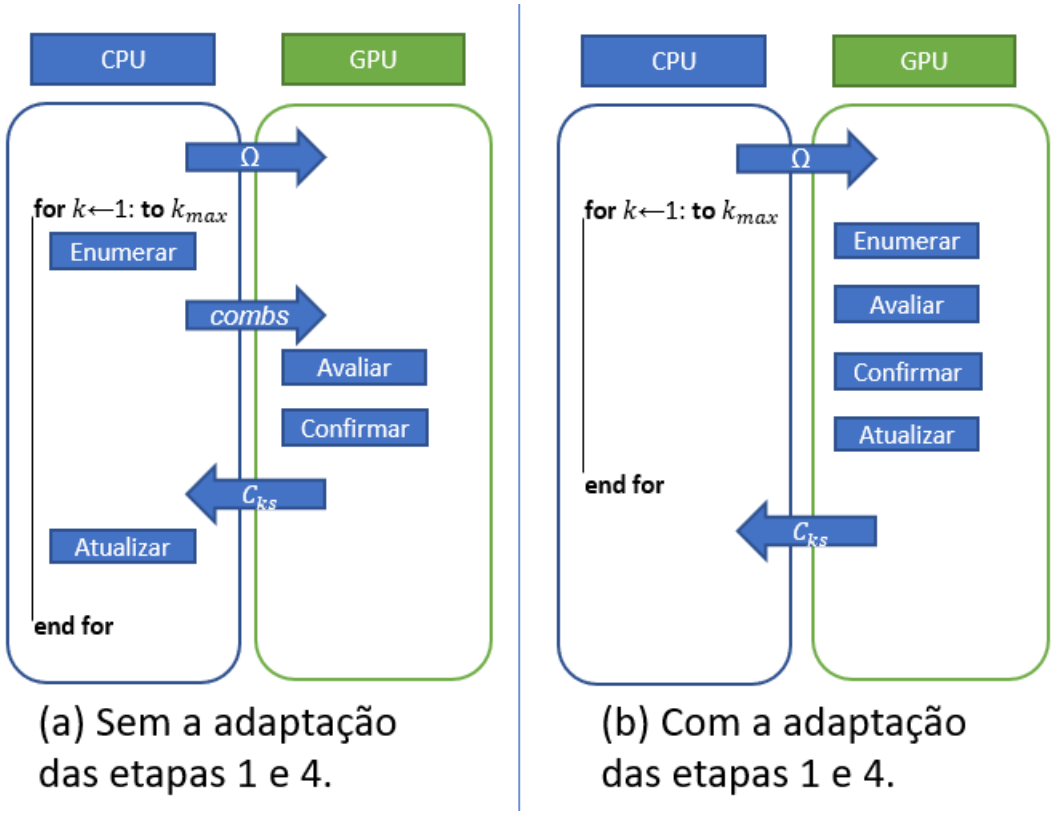

Figura 5.4: Fluxos de dados entre *device* e *host* nos cenários (a) e (b)

5.2.2 Simulação 2: Adaptação das Etapas Remanescentes

Os resultados mais detalhados da segunda simulação, que expõem os tempos registrados por cardinalidade, encontram-se no Apêndice B.2. A Tabela 5.8 apresentam os resultados obtidos para os *speed-ups* alcançados pela etapa de Enumeração. Nota-se que os valores registrados foram menores, pois conforme apresentado no Capítulo 4, algoritmos distintos foram utilizados nas implementações sequenciais e paralelas. O algoritmo 4 se demonstrou mais lento que algoritmo *coolex* utilizado originalmente na implementação original sequencial, porém se provou mais rápido com o auxílio da programação paralela.

Tabela 5.8: Tempos para 1ª etapa (Enumeração) — 2ª simulação

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	17,14	11,47	1	3,85	4
30	255,50	169,06	2	91,61	3
118	621,69	383,06	2	255,44	2

A Tabela 5.9 apresenta os valores registrados de tempo para a etapa de Avaliação, após a adaptação das etapas remanescentes para a GPU. Avaliando os resultados obtidos nesta etapa, pode-se comprovar os maiores speed-ups de 23x e 10x, para os sistemas de de 30 e 118 barras respectivamente, devido a eliminação total da transferência de dados.

Tabela 5.9: Tempos para 2ª etapa (Avaliação) — 2ª simulação

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	655,11	152,66	4	29,50	22
30	3425,13	814,57	4	147,68	23
118	3019,90	710,60	4	307,91	10

Na etapa de Confirmação os *speed-ups* se mantiveram elevados para as implementações em GPU, como pode ser visto na Tabela 5.10. Em relação à etapa anterior, poucos acréscimos nos *speed-ups* foram obtidos, isto se deve ao fato do conjunto de C_{ks} retornado ao *host* possuir dimensão inferior à matriz de combinações, o que significa uma menor movimentação de dados entre as arquiteturas.

Tabela 5.10: Tempos para 3ª etapa (Confirmação) — 2ª simulação

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	48,01	18,31	3	2,46	20
30	568,72	316,28	2	27,98	20
118	2835,15	1729,54	2	184,38	15

Apesar da adaptação da etapa de atualização ser mais complexa, esta continuou a representar cerca de 1% de todo o tempo computacional requisitado, tornando-se desnecessário comentar os *speed-ups* obtidos pela adaptação.

Não obstante da melhoria de desempenho obtida, a queda de *speed-up* para o caso de 118 barras continua a existir. Com isso, um último melhoramento foi proposto visando aprimorar a forma de representação das combinações de medidas que se tornam indisponíveis.

5.2.3 Simulação 3: Aprimoramento na Representação

A representação booleana descrita na Tabela 4.3 é fundamental para os algoritmos desenvolvidos em trabalhos anteriores [4, 21], pois a enumeração era realizada através de uma *movimentação* dos valores unitários. Uma desvantagem dessa representação é apresentada na Figura 5.5. Como pode ser visto, a dimensão do vetor que expressa as combinações de medidas indisponíveis aumenta juntamente com o tamanho do sistema de medição.

Como Algoritmo 4, proposto para a implementação paralela, mapeia diretamente as combinações utilizando de cálculos numéricos, a representação booleana deixa de ser mandatória. Com isso, foi utilizada uma nova concepção, onde são expressos os números das medidas que se tornaram indisponíveis, permitindo uma melhor escalabilidade do algoritmo. Esta estratégia torna-se expressivamente vantajosa, pois são tratadas quantidades elevadas de combinações, na ordem de 10^9. Em especial para a GPU, esta nova representação é mais adequada visto que são reduzidos o número de requisições de dados da memória global, que como pode ser visto no Capítulo 3, possui um tempo de acesso mais elevado que as outras memórias disponíveis.

Como pode ser visto na Figura 5.5, utilizando a nova representação, a dimensão do vetor de combinações não se altera, sendo a mesma tanto para um esquema de medição contendo 5 medidas, quanto para um contendo 176.

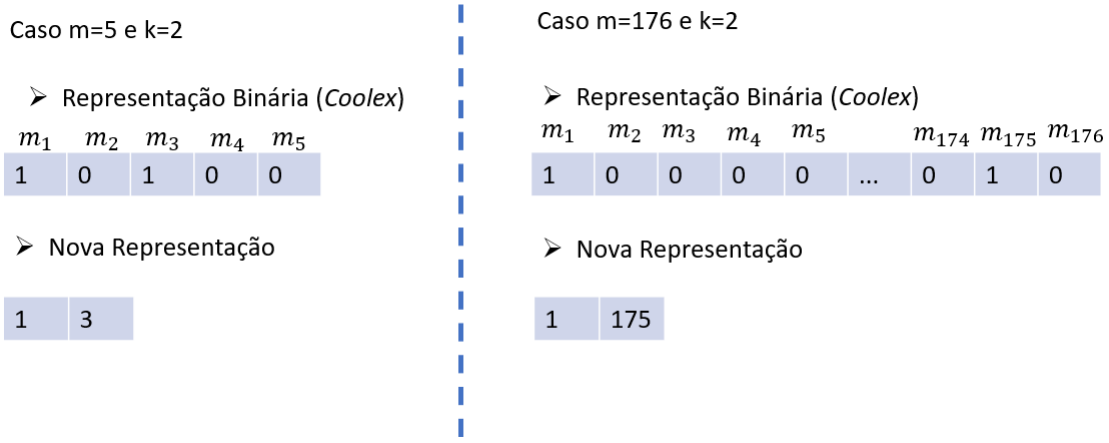

Figura 5.5: Comparação de representações para esquemas de medição com 5 e 176 medidas.

Com a nova representação houve um ganho de desempenho inclusive na adaptação sequencial do código. Os resultados mais detalhados da terceira simulação encontram-se no Apêndice B.3. Como pode ser visto na Tabela 5.11, a alteração na representação apresentou um aumento expressivo nos *speed-ups* da etapa de enumeração em todas as simulações *multi-threads* para todos os casos, alcançando *speed-ups* de até 400x.

Tabela 5.11: Tempos para 1ª etapa (Enumeração) — 3ª simulação

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	18,61	5,56	3	0,19	99
30	289,84	77,11	4	2,30	126
118	619,259	167,621	4	1,497	414

Na etapa de Avaliação, também houve um aumento expressivo nos *speed-ups* obtidos em todos os casos, Tabelas 5.12. Como pode ser visto, a GPU apresentou ganhos de desempenho expressivos tanto para o caso de 14 barras onde ocorrem eliminações de Gauss [10x10], quanto para o caso de 118-barras, onde existem um número maior de matrizes de ordem [5x5] a serem avaliadas. Este segundo caso apresentou *speed-ups* ainda maiores de 40x.

Tabela 5.12: Tempos para 2ª etapa (Avaliação) — 3ª simulação

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	613,31	153,43	4	26,35	23
30	2681,45	689,03	4	85,30	31
118	1.559,69	388,00	4	34,84	45

Os resultados obtidos na Etapa de confirmação podem ser vistos na Tabela 5.13, sendo possível perceber que os *speed-ups* obtidos foram mais homogêneos, com valores entre 40x e 50x.

Tabela 5.13: Tempos para 3ª etapa (Confirmação) — 3ª simulação

Sistema	**Sequencial**	**CPU *multi-thread***		**GPU**	
(barras)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
14	29,56	11,83	3	0,66	45
30	319,07	119,01	3	7,75	41
118	1.063,09	376,56	3	21,84	49

Sintetizando os resultados obtidos e apresentando as grandezas de tempo de forma mais expressiva, é possível perceber a grande aptidão da GPU para solucionar esse tipo de problema. Avaliações que demandavam cerca de uma hora para serem realizadas agora requerem um minuto, representando um *speed-up* de 39x.

Observa-se também que para o sistema de 30 barras o tempo de execução foi maior, o que a primeira vista aparenta ser contraditório, devido ao tamanho do sistema. Contudo, como pode ser visto nas Tabelas 5.2 e 5.3, devido a cardinalidade máxima alcançada $k_{max} = 7$, este caso realiza um maior número de avaliações de matrizes de ordem 7, o que demanda maior tempo computacional na etapa de Avaliação.

Tabela 5.14: *Speed-ups* finais.

	14 barras		**30 barras**		**118 barras**	
	Tempo	***Speed-up***	**Tempo**	***Speed-up***	**Tempo**	***Speed-up***
CPU (*single*)	11 min		54 min		54 min	
CPU (*mult*)	3 min	3x	14 min	3,7x	15min	3,5x
GPU	30 seg	23x	2 min	31x	1 min	39x

Conforme mencionado no Capítulo 2, a literatura especializada é limitada em estudos relacionados ao problema de busca por elementos críticos na estimação de estados. Em [4], podem ser encontrados resultados de um algoritmo *Branch and Bound* implementado de maneira sequencial para identificar C_{ks} de cardinalidades de até cinco no sistema IEEE de 30 barras. No entanto, não são fornecidas informações sobre o tempo estimado de computação para a realização dessa tarefa, o que poderia possibilitar estudos comparativos sobre o assunto. Neste ponto, é oportuno comentar que os resultados obtidos neste trabalho apontam para uma nova aplicação bem-sucedida de GPUs para realizar a tarefa de análise de criticidade em estudos relacionados à estimação de estado de sistemas de potência. Assim, a aplicação de GPUs proposta pode ser vista como uma implementação de referência, para demonstrar (prova de conceito) ser esta uma opção viável quando comparada com

suas contrapartes de CPU, servindo como ponto de partida para implementações mais elaboradas (otimizadas).

Capítulo 6

Conclusões e Trabalhos Futuros

Esta Dissertação abordou o problema da identificação de criticalidades presentes em planos de medição construídos para servir a Estimação de Estado em sistemas de potência. Destaca-se a relevância do presente estudo para a operação de redes elétricas, pois a presença de criticalidades em um sistema de medição indica riscos na obtenção de resultados confiáveis de processos clássicos de Estimação de Estado. Na análise de criticalidades, avalia-se a perda iminente de observabilidade da rede, que impacta também o processamento de possíveis medidas espúrias presentes entre os valores coletados. Trata-se de um problema complexo de natureza combinatória e que demanda um elevado tempo computacional, devido a um número fatorial de casos a analisar.

Para tornar análise de criticalidades mais eficiente, em termos de tempo de processamento, este trabalho apresentou uma nova abordagem para o problema em estudo, através do uso de programação paralela, em que combinações de medidas candidatas são avaliadas de forma concorrente. Visando identificar uma forma adequada para a execução da tarefa pretendida, a metodologia proposta foi levada a efeito em arquiteturas *multi-threads* de CPU e GPUs e consequentemente, os tempos computacionais registrados foram confrontados com os obtidos pela implementação sequencial original, apresentada em trabalhos publicados na área.

O uso da arquitetura da GPU mostrou-se o mais adequado para tratar a análise de criticalidades, superando os resultados obtidos pelas outras implementações de CPU em todas as etapas do processo. Isso foi condizente com as características do problema, onde milhares de tarefas simples são realizadas simultaneamente.

A partir dos resultados obtidos, foi possível desenvolver estratégias para aprimorar o desempenho da implementação realizada na GPU. Primeiramente, as adaptações das etapas referentes aos processos de enumeração das combinações e de atualização do conjunto solução, que inicialmente não se demonstravam paralelizáveis, resultaram em um ganho de desempenho, devido à uma redução do tempo de transferência de dados entre arquiteturas. Em sequência, foi possível constatar que a representação booleana das combinações de medidas, utilizada em trabalhos anteriores, não se revelou escalável com o porte do problema, pois o tamanho das estruturas utilizadas se acentua com o tamanho do sistema de medição. Esta forma de representação gera um grande número de acessos a memória global, em especial para a implementação em GPUs, e o que acarreta uma perda expressiva de desempenho. A representação compacta, utilizando o número referente as medidas, se provou mais eficiente, por aprimorar estas questões.

Utilizando a nova abordagem por GPUs e as otimizações propostas, o tempo computacional necessário à análise do caso de 118 barras contendo 176 medidores foi reduzida de aproximadamente uma hora para um minuto, alcançando *speed-ups* de até 39x. O desempenho alcançado por esta implementação se reflete em um aumento de confiabilidade dos resultados da Estimação de Estado, através da identificação de criticalidades de diversas cardinalidades. Os resultados também confirmaram a viabilidade do uso de GPUs para auxiliar a análise de criticalidades, desta forma, este trabalho pode vir a servir como ponto de referência para implementações futuras mais sofisticadas.

Dentre os possíveis aprimoramentos futuros, é desejável aperfeiçoar a etapa de enumeração de combinações utilizando a heurística de *Branch and Bound*, previamente utilizada em outros trabalhos [4, 3, 21]. A inclusão desta abordagem mais sofisticada, em relação à "força bruta" utilizada neste trabalho, é desejada, pois irá reduzir o número de combinações a serem enumeradas, e consequentemente, o tempo investido nas etapas subsequentes.

Futuramente, também deseja-se realizar uma análise de complexidade do problema para estimar o tempo ótimo da implementação. Juntamente, pretende-se realizar outros ensaios para identificar o melhor número de *threads* por bloco invocados pelos *kernels* e a ocupação da memória durante a execução do programa na GPU. Visando igualmente o ganho de desempenho, alveja-se avaliar o comportamento da implementação em GPUs mais robustas e em *Clusters* contendo múltiplas unidades.

Utilizar melhor as memórias disponíveis na GPU também é um objetivo futuro para otimizar desempenhos. Uma das possíveis estratégias sugeridas seria armazenar a ma-

triz de covariâncias de resíduos na memória constante, que requisita um menor tempo de acesso que a memória global, utilizada na implementação atual. Contudo, será necessário explorar propriedades como a simetria de covariância para possibilitar seu armazenamento em uma memória de menor capacidade. Utilizar a memória compartilhada na implementação também é ideal para minimizar o tempo de investido no acesso a memória, contudo, esta não é uma tarefa trivial, pois requer uma organização mais complexa dos blocos de *threads* devido ao escopo de acesso da memória. Uma possível estratégia de organização, é identificar um padrão de acesso à matriz de covariância, visando acomodar combinações que utilizam medidas em comum em um mesmo bloco, desta forma, matrizes auxiliares geradas na etapa de avaliação, podem ser armazenadas na memória compartilhada do mesmo.

Os bons resultados obtidos também indicam a possibilidade de se utilizar GPUs para uma análise de criticalidades que contempla os casos mais prováveis, por exemplo, aqueles em que ocorre a perda de unidades de medição [16]. Sob esta hipótese, matrizes maiores precisam ser avaliadas, pois cada unidade de medição concentra em geral várias medidas. Os resultados obtidos com o sistema de 14 barras, em que cardinalidades maiores foram alcançadas (k=10), demonstram que a abordagem proposta via GPUs é promissora. Isto é corroborado pelos *speed-ups* de 20x obtidos na etapa avaliação, onde ocorre de fato a eliminação, como também, pelos valores expressivos alcançados em outras etapas.

Referências

[1] Ali Abur, A. G. E. *Power System State Estimation: Theory and Implementation*. Marcel Decker, New York, N.J., 2004.

[2] Almeida, M. C. d., Asada, E. N., Garcia, A. V. On the Use of Gram Matrix in Observability Analysis. *IEEE Transactions on Power Systems 23*, 1 (2008), 249–251.

[3] Augusto, A. A. *Avaliação da Capacidade de Observação do Estado Operativo de Redes Elétricas*. Tese de Doutorado, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Fevereiro 2016.

[4] Augusto, A. A., Do Coutto Filho, M. B., Stacchini de Souza, J. C., Guimaraens, M. A. R. Branch-and-Bound Guided Search for Critical Elements in State Estimation. *IEEE Transactions on Power Systems 34*, 3 (maio de 2019), 2292–2301.

[5] Augusto, A. A., Do Coutto Filho, M. B., Stacchini de Souza, J. C. d. Low-cardinality critical k-tuples in measurement sets for state estimation. In *2013 IEEE Grenoble Conference* (Junho 2013), p. 1–6.

[6] Augusto, A. A., Guimaraens, M. A., Do Coutto Filho, M. B., Stacchini de Souza, J. C. Assessing strengths and weaknesses of measurement sets for state estimation. In *2017 IEEE Manchester PowerTech* (Manchester, junho de 2017), IEEE, p. 1–6.

[7] Ayres, M., Haley, P. H. Bad Data Groups in Power System State Estimation. *IEEE Transactions on Power Systems 1*, 3 (1986), 1–7.

[8] Castillo, E., Conejo, A. J., Pruneda, R. E., Solares, C. State estimation observability based on the null space of the measurement Jacobian matrix. *IEEE Transactions on Power Systems 20*, 3 (2005), 1656–1658.

[9] Chen, R. A fast integer algorithm for observability analysis using network topology. *IEEE Transactions on Power Systems 5*, 3 (1990), 1001–1009.

[10] Chen, Y., Huang, Z., Elizondo, M. Value of Faster Computation for Power Grid Operation. *IFAC Proceedings Volumes 45*, 21 (2012), 242–247.

[11] Clements, K., Krumpholz, G., Davis, P. Power System State Estimation Residual Analysis: An Algorithm Using Network Topology. *IEEE Transactions on Power Apparatus and Systems PAS-100*, 4 (1981), 1779–1787.

[12] Clements, K. A., Davis, P. W. Multiple bad data detectability and identifiability: A geometric approach. *IEEE Transactions on Power Delivery 1*, 3 (1986), 355–360.

[13] Clements, K. A., Krumpholz, G. R., Davis, P. W. State Estimator Measurement System Reliability Evaluation – An Efficient Algorithm Based on Topological Observability Theory. *IEEE Transactions on Power Apparatus and Systems PAS-101*, 4 (1982), 997–1004.

[14] Clements, K. A., Krutnpholz, G. R., Davis, P. W. Power System State Estimation with Measurement Deficiency: an Observability/Measurement Placement Algorithm. *IEEE Transactions on Power Apparatus and Systems PAS-102*, 7 (1983), 2012–2020.

[15] da Silva Junior, A. N., Clua, E. W. G., Do Coutto Filho, M. B., de Souza, J. C. S. GPU-Based Criticality Analysis Applied to Power System State Estimation. In *Computational Science and Its Applications – ICCSA 2020* (Cham, 2020), O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blečić, D. Taniar, B. O. Apduhan, A. M. A. C. Rocha, E. Tarantino, C. M. Torre, and Y. Karaca, Eds., Springer International Publishing, p. 121–133.

[16] Do Coutto Filho, M. B., Stacchini de Souza, J. C., Augusto, A. A. Critical measuring units for state estimation. In *2014 Power Systems Computation Conference* (Wrocław, Poland, agosto de 2014), IEEE, p. 1–7.

[17] Do Coutto Filho, M. B., Stacchini de Souza, J. C., Oliveira, F. M. F. d., Schilling, M. T. Identifying critical measurements & sets for power system state estimation. In *2001 IEEE Porto Power Tech Proceedings (Cat. No.01EX502)* (Porto, Pt, setembro de 2001), vol. 3, p. 6 pp.

[18] Do Coutto Filho, M. B., Stacchini de Souza, J. C., Schilling, M. T. Handling Critical Data and Observability. *Electric Power Components and Systems 35*, 5 (2007), 553–573. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/15325000601078187.

[19] Do Coutto Filho, M. B., Stacchini de Souza, J. C., Tafur, J. E. V. Quantifying Observability in State Estimation. *IEEE Transactions on Power Systems 28*, 3 (agosto de 2013), 2897–2906.

[20] Exposito, A. G., Abur, A. Generalized observability analysis and measurement classification. In *Proceedings of the 20th International Conference on Power Industry Computer Applications* (Maio 1997), p. 97–103.

[21] Flor, V. B. B. *Identificação de Criticalidades em Sistemas de Medição Através de Algoritmos Branch and Bound.* Tese de Doutorado, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Fevereiro 2020.

[22] Frank, R., Aaron, W. The coolest way to generate combinations. *Discrete Mathematics 309* (setembro de 2009), 5305–5320.

[23] Gopal, A., Niebur, D., Venkatasubramanian, S. DC Power Flow Based Contingency Analysis Using Graphics Processing Units. In *2007 IEEE Lausanne Power Tech* (julho de 2007), p. 731–736.

[24] Gou, B. Jacobian matrix-based observability analysis for state estimation. *IEEE Transactions on Power Systems 21*, 1 (2006), 348–356.

[25] Gou, B., Abur, A. A direct numerical method for observability analysis. *IEEE Transactions on Power Systems 15*, 2 (2000), 625–630.

[26] 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* (San Diego, CA, julho de 2011), IEEE, p. 1–8.

[27] Green, R. C., Wang, L., Alam, M. Applications and trends of high performance computing for electric power systems: Focusing on smart grid. *IEEE Transactions on Smart Grid 4*, 2 (junho de 2013), 922–931.

[28] He, H., Yan, J. Cyber-physical attacks and defences in the smart grid: a survey. *IET Cyber-Physical Systems: Theory & Applications 1*, 1 (dezembro de 2016), 13–27.

[29] Jalili-Marandi, V., Dinavahi, V. Large-scale transient stability simulation on graphics processing units. In *2009 IEEE Power Energy Society General Meeting* (julho de 2009), p. 1–6. ISSN: 1932-5517.

[30] Jason Sanders, E. K. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley/Pearson Education, Indianapolis, IN, 2010.

[31] John Cheng, Max Grossman, T. M. *Professional CUDA C programming*. John Wiley & Sons, Indianapolis, IN, 2014.

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

[33] Karimipour, H., Dinavahi, V. On false data injection attack against dynamic state estimation on smart power grids. In *2017 IEEE International Conference on Smart Energy Grid Engineering (SEGE)* (agosto de 2017), p. 388–393.

[34] Korres, G. N., Contaxis, G. C. Identification and updating of minimally dependent sets of measurements in state estimation. *IEEE Transactions on Power Systems 6*, 3 (1991), 999–1005.

[35] Krumpholz, G. R., Clements, K. A., Davis, P. W. Power System Observability: A Practical Algorithm Using Network Topology. *IEEE Transactions on Power Apparatus and Systems PAS-99*, 4 (1980), 1534–1542.

[36] London, J. B. A., Alberto, L. F. C., Bretas, N. G. Network observability: identification of the measurements redundancy level. In *PowerCon 2000. 2000 International Conference on Power System Technology. Proceedings (Cat. No.00EX409)* (2000), vol. 2, p. 577–582 vol.2.

[37] Momoh, J. A. *Smart Grid*. Wiley, HOboken, N.J., 2012.

[38] Monticelli, A. *State Estimation in Electric Power Systems: A Generalized Approach*. Power Electronics and Power Systems. Springer US, 2012.

[39] Monticelli, A., Wu, F. F. Network Observability: Identification of Observable Islands and Measurement Placement. *IEEE Transactions on Power Apparatus and Systems PAS-104*, 5 (1985), 1035–1041.

[40] Monticelli, A., Wu, F. F. Observability Analysis for Orthogonal Transformation Based State Estimation. *IEEE Power Engineering Review PER-6*, 2 (1986), 45–45.

[41] Moore, G. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE 86*, 1 (janeiro de 1998), 82–85.

[42] Mori, H., Tsuzuki, S. A fast method for topological observability analysis using a minimum spanning tree technique. *IEEE Transactions on Power Systems 6*, 2 (1991), 491–500.

[43] Nguyen, H. *GPU gems3*. Addison-Wesley, Upper Saddle River, N.J., 2008.

[44] Nucera, R. R., Gilles, M. L. Observability analysis: a new topological algorithm. *IEEE Transactions on Power Systems 6*, 2 (1991), 466–475.

[45] NVIDIA. CUDA C Best Pratices Guide, 2020. Disponível em `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.

[46] NVIDIA. CUDA C++ Programming Guide, 2020. Disponível em `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[47] NVIDIA. Profiler user's guide, 2020. Disponível em `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[48] Pradhan, S. Find n_{th} lexicographically permutation string, 2020.

[49] Quintana, V. H., Simoes-Costa, A., Mandel, A. Power System Topological Observability Using a Direct Graph-Theoretic Approach. *IEEE Transactions on Power Apparatus and Systems PAS-101*, 3 (1982), 617–626.

[50] Rahman, M. A., Venayagamoorthy, G. K. Dishonest Gauss Newton method based power system state estimation on a GPU. In *2016 Clemson University Power Systems Conference (PSC)* (mar 2016), p. 1–6.

[51] 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 (julho de 2017), 1639–1648.

[52] Schweppe, F. C., Wildes, J. Power System Static-State Estimation, Part I: Exact Model. *IEEE Transactions on Power Apparatus and Systems PAS-89*, 1 (janeiro de 1970), 120–125.

[53] Simoes Costa, A., Piazza, T., Mandel, A. Qualitative methods to solve qualitative problems in power system state estimation. *Power Systems, IEEE Transactions on 5* (09 1990), 941 – 949.

[54] Solares, C., Conejo, A. J., Castillo, E., Pruneda, R. E. Binary-arithmetic approach to observability checking in state estimation. *IET Generation, Transmission & Distribution 3*, 4 (2009), 336–345.

[55] Sou, K. C., Sandberg, H., Johansson, K. H. Computing Critical k-Tuples in Power Networks. *IEEE Transactions on Power Systems 27*, 3 (agosto de 2012), 1511–1520. Conference Name: IEEE Transactions on Power Systems.

[56] Wong, S. V. Permutations with cuda and opencl, 2016.

[57] Wu, F. F., Monticelli, A. Network Observability: Theory. *IEEE Transactions on Power Apparatus and Systems PAS-104*, 5 (1985), 1042–1048.

[58] Zetter, K. Insisde the cunning, unprecedent hack of ukraine's powergrid, 2016.

APÊNDICE A – Propriedades Relacionadas à Análise de Criticalidades

A.1 Elementos do Vetor de Resíduos e da Matriz de Covariância para o Caso de Medidas Críticas

Elementos nulos presentes no vetor de resíduos ($\boldsymbol{z}$) e a matriz de covariância de resíduos($\boldsymbol{\Omega}$) indicam a presença de medidas críticas (C_{meds}). A demonstração matemática apresentada em [18] será desenvolvida a seguir.

Suponha que exista uma rede onde haja um conjunto de medidas, as quais possuem redundância nula. Dessa maneira, o número de medidas será igual ao número de variáveis de estado, sendo assim, a remoção de qualquer medida torna a rede não observável e a EE não pode ser realizada, o que define todas as medidas como C_{meds}.

Nesta situação, as matrizes $\mathbf{H}$ e $\mathbf{R}$ são matrizes quadradas, além disso o resultado da expressão $\mathbf{H^t R^{-1}}$ também é uma matriz quadrada e não singular. Dessa forma, multiplicando a Equação 2.2 por $(\mathbf{H^t R^{-1}})^{-1}$ e considerando a Equação $\boldsymbol{z = Hx}$, que resulta-se na seguinte formulação do problema de EE:

$$(\mathbf{H^t R^{-1}})^{-1}\mathbf{H^t R^{-1}}[\mathbf{z - Hx}] = \mathbf{0} \rightarrow [\mathbf{z - Hx}] = \mathbf{0} \rightarrow \mathbf{r = z - \hat{z} = 0} \tag{A.1}$$

onde $\mathbf{x}$ é o vetor de estado, $\mathbf{z}$ é o vetor de medidas e $\hat{\mathbf{z}}$ é o vetor de medidas estimado.

A Equação A.1 indica de que os resíduos de C_{meds} são nulos. A inversa da matriz de ganho pode ser dada pela Equação A.2

$$\mathbf{G^{-1}} = \mathbf{H^{-1}R(H^t)^{-1}} \tag{A.2}$$

Com essa equação, pode-se obter a matriz de covariância dos resíduos relacionada às C_{meds}, conforme a Equação A.3:

$$\mathbf{\Omega} = \mathbf{R} - \mathbf{H}\mathbf{G}^{-1}\mathbf{H}^{\mathbf{t}} = \mathbf{0} \tag{A.3}$$

Dessa maneira, no caso de medidas críticas, a matriz de covariância dos resíduos também é nula.

A.2 Cardinalidade máxima Teórica de Tuplas Críticas

Dado um plano de medição contendo m medidas, a cardinalidade máxima (teórica) de uma C_k é definida por:

$$k_{max} = m - n + 1 \tag{A.4}$$

onde n é o número de elementos do vetor de estado do processo de estimação.

A demonstração desta propriedade pode ser vista em [3] ou pelo ponto de vista topológico em [36]. Considerando uma rede com n variáveis de estado. Este sistema é é observável, como um todo, se o posto da matriz $\boldsymbol{H}[m \times n]$ for completo. Em outras palavras, $\boldsymbol{H}$ deve conter, pelo menos n linhas linearmente independentes entre as $m \geq n$ [3]:

$$n = Posto(\mathbf{H}) \leq min(m, n) \tag{A.5}$$

Sabendo-se que a remoção de qualquer tupla de medidas, de tamanho $m-n+1$, resulta em uma H' de dimensão $(n-1 \times n)$. Pode-se dizer que para a remoção de $m-n+1$ medidas tem-se:

$$Posto(\mathbf{H}') \leq min(n-1, n) < n \tag{A.6}$$

Sendo assim, como a remoção de $m-n+1$ medidores necessariamente torna a rede inobservável, logo, levando em consideração também a Propriedade 1, a maior cardinalidade de C_k não pode ser superior a $k_{max} = m - n + 1$.

A.3 Elementos da Matriz de Covariância de Resíduos para o Caso de Tuplas Críticas

As colunas de Ω_a, relacionadas às medidas de uma dada C_k, formam um conjunto linearmente dependente. Esta propriedade pode ser demonstrada utilizando o lema de determinante de matrizes [3, 21]. De acordo com esse lema, se $\mathbf{M}(n \times n)$ é uma matriz não-singular, $\mathbf{U}$ e $\mathbf{V}$ matrizes $(n \times m)$ e $\mathbf{W}$ uma matriz quadrada $(m \times m)$. Sendo assim, pode-se representar o determinante da matriz $(M + UWV^t)$ da seguinte maneira:

$$\mathbf{det(M + UWV^t) = det(W^{-1} + V^t M^{-1} U) det(W) det(M)} \tag{A.7}$$

Retomando o foco às equações relacionadas a EE. A extração de um conjunto de medidas do plano de medição provoca a seguinte alteração na matriz de ganho $\boldsymbol{(G)}$:

$$\mathbf{\bar{G} = G - \bar{H}^t \bar{R}^{-1} \bar{H}} \tag{A.8}$$

Onde:

- $\mathbf{G}$ é a matriz de ganho original do processo de EE;
- $\mathbf{\bar{G}}$ é a matriz de ganho com as medidas removidas;
- $\mathbf{\bar{H}}$ é a matriz Jacobiano estabelecida apenas pelas medidas removidas
- $\mathbf{\bar{R}}$ é a matriz dos resíduos das medidas removidas.

Considerando $\mathbf{M = G}$, $\mathbf{U = V = \bar{H}^t}$ e $\mathbf{W = \bar{R}^{-1}}$, utilizando o lema do determinante de matriz, Equação A.7, o determinante de $\mathbf{\bar{G}}$ pode ser determinado da seguinte maneira:

$$\mathbf{det(\bar{G}) = det(\bar{R} + \bar{H} G^{-1} \bar{H}^t) det(\bar{R}^{-1}) det(G)} \tag{A.9}$$

Caso a remoção do conjunto de medidas torne o sistema não observável, então $\mathbf{\bar{G}}$ é uma matriz singular, e consequentemente o seu determinante é nulo, de forma que essa matriz também pode ser utilizada como função objetivo para a avaliação de tuplas críticas. Como o $\mathbf{det(\bar{R}) > 0}$ e $\mathbf{det(G)} > 0$, pois o sistema original é observável, então a avaliação da criticalidade do conjunto de medidas removido depende apenas se:

$$\mathbf{det(\bar{R} + \bar{H} G^{-1} \bar{H}^t) = 0} \tag{A.10}$$

Dessa forma, $\bar{\mathbf{\Omega}} = \bar{\mathbf{R}} + \bar{\mathbf{H}}\mathbf{G}^{-1}\bar{\mathbf{H}}^{\mathbf{t}}$ representa a partição da matriz de covariância dos resíduos completa $(\mathbf{\Omega})$, formada pela tupla de medidas retiradas. Sendo assim, pela Equação A.10 a matriz $\bar{\mathbf{E}}$ é singular, e dessa forma suas colunas são linearmente dependentes.

APÊNDICE B – Resultados de Simulações

B.1 Primeira Simulação

Neste apêndice se encontram os tempos registrados por cardinalidade em todas as simulações.

B.1.1 Avaliação

Tabela B.1: Tempos para 2ª etapa (Avaliação) — sistema de 14-barras.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	1,28	0,33	4	0,13	10
7	6,63	1,76	4	0,53	12
8	28,41	7,70	4	2,10	14
9	102,69	27,95	4	7,24	14
10	319,30	88,98	4	21,48	15
Total	458,53	126,79	4	31,65	14

Tabela B.2: Tempos para a 2ª etapa (Avaliação) — sistema de 30-barras.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,65	0,30	2	0,11	6
5	11,81	4,78	2	1,51	8
6	168,56	63,76	3	18,53	9
7	1.989,28	745,71	3	198,13	10
Total	2.170,32	613,09	4	218,47	10

Tabela B.3: Tempos para a 2ª etapa (Avaliação) — sistema de 118-barras.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,89	0,34	3	0,23	4
4	50,41	15,69	3	9,69	5
5	2.266,15	694,56	3	359,27	6
Total	2.317,46	710,60	3	369,51	6

B.1.2 Confirmação

Tabela B.4: Tempos para a 3ª etapa (Confirmação) — sistema de 14-barras.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	0,19	0,11	2	0,02	12
7	0,86	0,44	2	0,05	16
8	3,55	1,65	2	0,19	18
9	11,59	5,55	2	0,59	20
10	33,06	15,21	2	1,64	20
Total	49,29	22,98	2	2,52	20

Tabela B.5: Tempos para a 3ª etapa (Confirmação) — sistema de 30-barras.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,18	0,09	2	0,02	11
5	3,13	1,60	2	0,23	14
6	44,85	25,40	2	2,81	16
7	536,55	289,18	2	32,49	17
Total	584,71	271,92	2	35,57	16

Tabela B.6: Tempos para a 3ª etapa (Confirmação) — sistema de 118-barras.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,73	0,42	2	0,04	19
4	55,06	37,61	1	3,26	17
5	2.947,77	1.691,51	2	181,11	16
Total	3.003,56	1.729,54	2	184,41	16

B.2 Segunda Simulação

B.2.1 Enumeração

Tabela B.7: Tempos para a 1ª etapa (Enumeração) — sistema 14-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	0,12	0,08	1	0,03	4
7	0,48	0,37	1	0,11	4
8	1,57	1,04	2	0,36	4
9	4,39	2,84	2	0,99	4
10	10,55	7,11	1	2,37	4
Total	17,14	11,47	1	3,85	4

Tabela B.8: Tempos para a 1ª etapa (Enumeração) — sistema 30-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,22	0,11	2	0,06	3
5	2,37	1,56	2	0,82	3
6	25,04	15,98	2	8,91	3
7	227,85	151,40	2	81,81	3
Total	255,50	169,06	2	91,61	3

Tabela B.9: Tempos para a 1ª etapa (Enumeração) — sistema 118-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,52	0,30	2	0,17	3
4	17,95	10,27	2	7,16	3
5	603,22	372,47	2	248,11	2
Total	621,69	383,06	2	255,44	2

B.2.2 Avaliação

Tabela B.10: Tempos para a 2ª etapa (Avaliação) — sistema 14-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	2,24	0,49	5	0,07	32
7	10,77	2,70	4	0,41	26
8	43,99	9,86	4	1,81	24
9	152,16	35,05	4	6,67	23
10	445,47	104,47	4	20,53	22
Total	655,11	152,66	4	29,50	22

Tabela B.11: Tempos para a 2ª etapa (Avaliação) — sistema 30-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	1,12	0,30	4	0,05	22
5	18,60	4,78	4	0,81	23
6	266,98	63,76	4	11,37	23
7	3138,38	745,71	4	135,45	23
Total	3425,13	814,57	4	147,68	23

Tabela B.12: Tempos para a 2ª etapa (Avaliação) — sistema 30-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	1,25	0,34	4	0,12	10
4	65,75	15,69	4	7,31	9
5	2952,88	694,56	4	300,48	10
Total	3019,90	710,60	4	307,91	10

B.2.3 Confirmação

Tabela B.13: Tempos para a 3ª etapa (Confirmação) — sistema 14-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	0,18	0,10	2	0,01	26
7	0,88	0,41	2	0,03	26
8	3,59	1,44	2	0,16	22
9	11,75	4,37	3	0,58	20
10	31,56	11,97	3	1,67	19
Total	48,01	18,31	3	2,46	20

Tabela B.14: Tempos para a 3ª etapa (Confirmação) — sistema 30-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,15	0,09	2	0,01	19
5	2,83	1,60	2	0,13	22
6	44,49	25,40	2	2,02	22
7	521,25	289,18	2	25,82	20
Total	568,72	316,28	2	27,98	20

Tabela B.15: Tempos para a 3ª etapa (Confirmação) — sistema 118-barras — 2ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,70	0,42	2	0,05	15
4	51,85	37,61	1	3,34	16
5	2782,60	1691,51	2	181,00	15
Total	2835,15	1729,54	2	184,38	15

B.3 Terceira Simulação

B.3.1 Enumeração

Tabela B.16: Tempos para a 1ª etapa (Enumeração) — sistema 14-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	0,17	0,06	3	0,002	85
7	0,51	0,21	2	0,006	85
8	1,66	0,58	3	0,017	98
9	4,70	1,35	3	0,048	98
10	11,53	3,32	3	0,12	100
Total	18,61	5,56	3	0,19	99

Tabela B.17: Tempos para a 1a etapa (Enumeração) — sistema 30-barras — 3a simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,21	0,08	3	0,013	16
5	2,39	0,82	3	0,11	21
6	27,54	7,00	4	1,02	27
7	259,69	69,20	4	1,15	226
Total	289,84	77,11	3,76	2,30	126

Tabela B.18: Tempos para a 1ª etapa (Enumeração) — sistema 118-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,47	0,14	3	0,001	469
4	16,71	4,53	4	0,044	379
5	602,08	162,95	4	1,45	415
Total	619,259	167,621	4	1,497	414

B.3.2 Avaliação

Tabela B.19: Tempos para a 2ª etapa (Avaliação) — sistema de 14-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	2,06	0,59	3	0,05	41
7	9,68	2,92	3	0,32	30
8	40,48	10,11	4	1,53	26
9	141,99	35,08	4	5,86	24
10	418,71	104,60	4	18,58	23
Total	613,31	153,43	4	26,35	23

Tabela B.20: Tempos para a 2ª etapa (Avaliação) — sistema de 30-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,83	0,21	4	0,01	83
5	12,72	3,81	3	0,29	44
6	199,38	50,19	4	5,58	35
7	2468,49	634,81	4	79,42	31
Total	2681,45	689,03	4	85,30	31

Tabela B.21: Tempos para a 2ª etapa (Avaliação) — sistema de 118-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,53	0,12	4	0,01	53
4	28,65	7,36	4	0,51	56
5	1.530,50	380,52	4	34,32	45
Total	1.559,69	388,00	4	34,84	45

B.3.3 Confirmação

Tabela B.22: Tempos para a 3ª etapa (Confirmação) — sistema 14-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
6	0,08	0,05	2	0,003	27
7	0,43	0,26	2	0,012	36
8	1,94	0,94	2	0,045	43
9	6,94	2,71	3	0,15	46
10	20,15	7,85	3	0,44	45
Total	29,56	11,83	3	0,66	45

Tabela B.23: Tempos para a 3ª etapa (Confirmação) — sistema 30-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
4	0,07	0,076	1	0,003	24
5	1,24	0,59	2	0,039	31
6	23,14	8,21	3	0,54	42
7	294,62	110,14	3	7,17	41
Total	319,07	119,01	3	7,75	41

Tabela B.24: Tempos para a 3ª etapa (Confirmação) — sistema 118-barras — 3ª simulação.

Card.	**Sequencial**	**CPU *multi-thread***		**GPU**	
(*k*)	**Tempo(seg.)**	**Tempo(seg.)**	***Speed-up***	**Tempo(seg.)**	***Speed-up***
3	0,18	0,10	2	0,009	20
4	15,46	5,98	3	0,42	37
5	1.047,45	370,48	3	21,41	49
Total	1.063,09	376,56	3	21,84	49