Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-Graduação em Informática

Execução Eficiente do Padrão de Propagação de Ondas Irregulares na Arquitetura Many Integrated Core

Jeremias Moreira Gomes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. George Luiz Medeiros Teodoro

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-graduação em Informática

Coordenadora: Prof.ª Dr.ª Célia Ghedini Ralha

Banca examinadora composta por:

Prof. Dr. George Luiz Medeiros Teodoro (Orientador) — CIC/UnB
Prof. Dr.ª Aletéia Patrícia Favacho de Araújo — CIC/UnB
Prof. Dr. Eduardo Alves do Valle Junior — DCA/UNICAMP

CIP — Catalogação Internacional na Publicação

Moreira Gomes, Jeremias.

Execução Eficiente do Padrão de Propagação de Ondas Irregulares na Arquitetura Many Integrated Core / Jeremias Moreira Gomes. Brasília : UnB, 2016.
80 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2016.

1. *Irregular Wavefront Propagation Pattern*, 2. Intel® Xeon Phi™,
3. *Many Integrated Core*.

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-Graducação em Informática

Execução Eficiente do Padrão de Propagação de Ondas Irregulares na Arquitetura Many Integrated Core

Jeremias Moreira Gomes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. George Luiz Medeiros Teodoro (Orientador)
CIC/UnB

Prof. Dr.ª Aletéia Patrícia Favacho de Araújo
CIC/UnB

Prof. Dr. Eduardo Alves do Valle Junior
DCA/UNICAMP

Prof.ª Dr.ª Célia Ghedini Ralha
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 29 de janeiro de 2016

Agradecimentos

Agradeço a toda a minha família, principalmente ao meu pai Evaldo e minha mãe Solange pelo apoio incondicional em todas as escolhas da minha vida. Mesmo a distância, saibam que vocês são o meu exemplo e incentivo a tudo que faço. Estendo essa conquista inclusive a minha irmã Iolanda, na qual constitui o meu suporte nessa jornada distante de todos os meus familiares.

Ao meu orientador, George Luiz Medeiros Teodoro, por todas as oportunidades e tempo dedicado à minha pessoa. Agradeço verdadeiramente pelas orientações tão objetivas e pelo inestimável compartilhamento de experiências que me ajudaram a evoluir na área científica. Sua dedicação, entusiasmo com a pesquisa acadêmica, visão pragmática e paciência inigualável em direcionar minhas ideias possibilitaram tornar esta dissertação de mestrado uma realidade.

Aos amigos que fiz ao longo tempo em Brasília. Aqueles que conheci por meio do Exército e que desde a época das diversas missões sempre acreditaram em mim, e me deram assistência em todas as minhas necessidades, principalmente Morato, Leandro, Aline, Hermínio, Letícia Gentil, Arrais de Souza, Luzia, Campos de Souza, Bonato, Fábio Reis, Mateus, Nascimento, Letícia Pontes, Sigis, Elvis, Júlio Ramos e diversos outros. Os amigos que conheci por meio da internet, Micael (Mocs), Gil (Fish), Rafael (Snake), Lucas (Luvas), Pedro (Đead), Gesse (Sephir), Amanda (Amy), Raquel (ainda não sei o apelido), Talyana (Tatá) e Analice (Didi), que hoje são o contato diário indispensável para qualquer conversa ou risada, inclusive de madrugada. Aqueles que conheci na Universidade de Brasília, sejam por assistirmos aulas juntos (Nilson, Paula, Léia, Daniel, etc), por contato através dos diversos laboratórios (Ariane, Saad, Lucas Angelo, Thiago, Gustavo, André, Waldeyr, Breno, Fay, Gabriel, Wosley, Jonathan etc), por compartilhar uma caminhada de ideias loucas e risadas até o RU, ou pelos auxílios prestados em meus momentos de desalento. Aos companheiros de laboratório que sempre foram presentes e prestativos comigo, principalmente Alexandre e Luis que nunca me deixaram desmotivar e estavam sempre presentes nas alegrias, nas tristezas, nas dúvidas, nas conversas sobre a vida, nos objetivos, nas piadas ruins, no cumprimento de prazos e nos desesperos.

Agradeço também aos amigos de longa data Ivair, Natália, Wellington, Altamir e Derick que, mesmo distantes, torcemos uns pelo sucesso e felicidade dos outros.

Ao Departamento de Ciência da Computação da Universidade de Brasília, principalmente ao corpo docente pelos ensinamentos e participação em minha formação, ao corpo administrativo (Carol, Maria Helena, Rafael, Luis e Paula) pelas amizades, conversas e auxílios prestados, e aos amigos responsáveis pela segurança das instalações (Vera, Waldecyr, Sebastião e Juranir) pela incentivo diário e auxílios dos mais diversos. Não poderia deixar de destacar a professora Alba por todo o auxílio e acolhimento desde o início das minhas atividades na UnB, e a professora Célia, coordenadora do PPGInf, sempre amigável e disposta a cooperar e auxiliar em todas as minhas necessidades.

Por fim, agradeço a todos os amigos que fiz ao longo da vida pelo simples fato de terem existido na minha vida. Todos vocês contribuíram, ainda que indiretamente, para que este trabalhado fosse concretizado.

Resumo

A execução eficiente de algoritmos de processamento de imagens é uma área ativa da Bioinformática. Uma das classes de algoritmos em processamento de imagens ou de padrão de computação comum nessa área é a *Irregular Wavefront Propagation Pattern* (IWPP). Nessa classe, elementos propagam informações para seus vizinhos em forma de ondas de propagação. Esse padrão de propagação resulta em acessos a dados e expansões irregulares. Por essa característica irregular, implementações paralelas atuais dessa classe de algoritmos necessitam de operações atômicas, o que acaba sendo muito custoso e também inviabiliza a implementação por meio de instruções *Single Instruction, Multiple Data* (SIMD) na arquitetura *Many Integrated Core* (MIC), que são fundamentais para atingir alto desempenho nessa arquitetura. O objetivo deste trabalho é reprojetar o algoritmo *Irregular Wavefront Propagation Pattern*, de forma a possibilitar sua eficiente execução em processadores com arquitetura *Many Integrated Core* que utilizem instruções SIMD. Neste trabalho, utilizando o Intel® Xeon Phi™, foram implementadas uma versão vetorizada, apresentando ganhos de até 5.63× em relação à versão não-vetorizada; uma versão paralela utilizando fila *First In, First Out* (FIFO) cuja escalabilidade demonstrou-se boa com *speedups* em torno de 55× em relação à um núcleo do coprocessador; uma versão utilizando fila de prioridades cuja velocidade foi de 1.62× mais veloz que a versão mais rápida em GPU conhecida na literatura, e uma versão cooperativa entre processadores heterogêneos que permitem processar imagens que ultrapassem a capacidade de memória do Intel® Xeon Phi™, e também possibilita a utilização de múltiplos dispositivos na execução do algoritmo.

Palavras-chave: *Irregular Wavefront Propagation Pattern*, Intel® Xeon Phi™, *Many Integrated Core*.

Abstract

The efficient execution of image processing algorithms is an active area of Bioinformatics. In image processing, one of the classes of algorithms or computing pattern that works with irregular data structures is the Irregular Wavefront Propagation Pattern (IWPP). In this class, elements propagate information to neighbors in the form of wave propagation. This propagation results in irregular access to data and expansions. Due to this irregularity, current implementations of this class of algorithms requires atomic operations, which is very costly and also restrains implementations with Single Instruction, Multiple Data (SIMD) instructions in Many Integrated Core (MIC) architectures, which are critical to attain high performance on this processor. The objective of this study is to redesign the Irregular Wavefront Propagation Pattern algorithm in order to enable the efficient execution on processors with Many Integrated Core architecture using SIMD instructions. In this work, using the Intel® Xeon Phi™ coprocessor, we have implemented a vector version of IWPP with up to 5.63× gains on non-vectored version, a parallel version using First In, First Out (FIFO) queue that attained speedup up to 55× as compared to the single core version on the coprocessor, a version using priority queue whose performance was 1.62× better than the fastest version of GPU based implementation available in the literature, and a cooperative version between heterogeneous processors that allow to process images bigger than the Intel® Xeon Phi™ memory and also provides a way to utilize all the available devices in the computation.

Keywords: *Irregular Wavefront Propagation Pattern*, Intel® Xeon Phi™, *Many Integrated Core.*

Sumário

1 Introdução **1**

1.1 Problema . . . 3

1.2 Objetivo . . . 3

1.3 Objetivos Específicos . . . 3

1.4 Organização do Texto . . . 3

2 Referencial Teórico **5**

2.1 Algoritmos Morfológicos . . . 5

2.1.1 Notações em Algoritmos Morfológicos . . . 6

2.1.2 Exemplos de Algoritmos Morfológicos . . . 6

2.1.3 Estratégias para Implementação de Algoritmos Morfológicos . . . 12

2.2 Arquiteturas Paralelas . . . 16

2.2.1 Classificação de Arquiteturas Paralelas . . . 17

2.2.2 Processadores Vetoriais . . . 19

2.2.3 Intel® Xeon Phi™ . . . 20

2.2.4 *Data Race* e Condição de Corrida . . . 23

2.3 Sumário . . . 24

3 *Irregular Wavefront Propagation Pattern* (IWPP) **25**

3.1 Reconstrução Morfológica . . . 26

3.2 Transformada de Distância Euclidiana . . . 29

3.3 *Fill Holes* . . . 31

3.4 Trabalhos Relacionados . . . 32

3.4.1 IWPP . . . 32

3.4.2 Reconstrução Morfológica . . . 33

3.4.3 Transformada de Distância Euclidiana . . . 34

3.5 Sumário . . . 35

4 IWPP Paralelo Eficiente na Arquitetura *Many Integrated Core* **36**

4.1 Algoritmo Proposto . . . 36

4.2 Implementação Vetorial do Algoritmo Proposto 37
4.2.1 Identificação de Elementos Recebedores de Propagação 38
4.2.2 Propagação 43
4.3 Implementação Paralela do Algoritmo Proposto 43
4.4 Uso de Diferentes Estruturas de Dados 46
4.5 Execução Cooperativa em Processadores Heterogêneos 48
4.6 Sumário 50

5 Análise dos Resultados **51**
5.1 Ambiente de Desenvolvimento e Testes 51
5.2 Configuração dos Experimentos 52
5.3 Resultados Experimentais 53
5.3.1 Impacto da Vetorização para o Desempenho 54
5.3.2 Resultados da Paralelização do Algoritmo Vetorizado 55
5.3.3 Impacto do Percentual de Tecido de Cobertura 55
5.3.4 Impacto do Tamanho da Imagem 56
5.3.5 Avaliação de Diferentes Coprocessadores e Estrutura de Dados . . . 56
5.3.6 Avaliação da Execução Cooperativa em Processadores Heterogêneos 58

6 Conclusão **61**
6.1 Trabalhos Futuros 62

Referências **64**

Lista de Figuras

2.1 Representação de um Conjunto de Vizinhos 4-Conectado. 6
2.2 Representação de um Conjunto de Vizinhos 8-Conectado. 6
2.3 Representação Tridimensional de uma Imagem em Tons de Cinza. 7
2.4 Início da Inundação. 7
2.5 Avanço da Inundação. 7
2.6 Vista Superior da Segmentação. 8
2.7 Imagem Original. 9
2.8 Esqueleto Morfológico. 9
2.9 Visualização do Resultado. 9
2.10 Reconstrução Morfológica. Adaptado de [64]. 10
2.11 Exemplo Numérico da Transformada de Distância *City-block*. 12
2.12 Varredura *Raster*. 13
2.13 Varredura *Anti-raster*. 13
2.14 Pixels Utilizados nas Varreduras. 14
2.15 Taxonomia de Duncan. Adaptado de [13]. 18
2.16 Diagrama de Bloco do Intel® Xeon Phi™. Adaptado de [8]. 21
2.17 Abordagens de Programação Utilizando o Intel® Xeon Phi™. 22

3.1 Reconstrução Morfológica. 27
3.2 Imagem de Tecido Original 28
3.3 Imagem Máscara 28
3.4 Imagem Marcadora 28
3.5 Imagem Reconstruída 28
3.6 Imagem Original. 30
3.7 TDE. 30
3.8 Função *Imfill* Aplicada em Tons de Cinza. 31
3.9 Visão Global da IWPP em GPU. Adaptado de [60]. 33

4.1 Exemplo de Propagação Paralela Oriunda de Elementos Diferentes. 38
4.2 Exemplo de Propagação Paralela Utilizando a Abordagem Proposta. 38

4.3 Cálculo de Endereços e Operação *Gather* de um Pixel p. 39
4.4 Verificação da Condição de Propagação Vetorial. 40
4.5 Exemplo de Distribuição de Máscara em Registrador Vetorial. 40
4.6 Permutação 01. 41
4.7 Soma Vetorial 01. 41
4.8 Permutação 02. 41
4.9 Soma Vetorial 02. 41
4.10 Permutação 03. 41
4.11 Soma Vetorial 03. 41
4.12 Registrador Retornado como Resposta. 42
4.13 *Scatter* e Atualização da Quantidade de Elementos. 42
4.14 Divisão de Elementos Ativos entre *Threads*. 44
4.15 Exemplo de Identificação de Elementos Iguais por *Threads* Diferentes. . . . 45
4.16 Cenário 1: Estabilidade Alcançada na Iteração i. 45
4.17 Cenário 2: Estabilidade Alcançada na Iteração $i + 1$. 46
4.18 Exemplo de Propagação Utilizando Fila FIFO. 47
4.19 Exemplo de Propagação Utilizando Fila de Prioridades. 48
4.20 Divisão de Uma Imagem em Duas Tarefas. 48
4.21 Reconstrução Morfológica Aplicada em Cada Tarefa. 48
4.22 Reconstrução Morfológica Após a União das Imagens com a Área de Fronteira Corrigida. 49
4.23 Área Processada pela Correção de Bordas. 49

5.1 Tipos Diferentes de Cobertura. 53
5.2 Ganho de Vetorização Variando a Porcentagem de Cobertura. 54
5.3 Análise de Escalabilidade Variando a Quantidade de *Threads*. 55
5.4 Variação do Percentual de Tecido de Cobertura. 56
5.5 Variação do Tamanho das Imagens de Entrada para a Reconstrução Morfológica. 57
5.6 Avaliação de Múltiplos Coprocessadores e Uso de Diferentes Tipos de Estrutura de Dados. 57
5.7 Avaliação da Execução Cooperativa em Processadores Heterogêneos. 59

Lista de Tabelas

5.1 Características dos Processadores. 52
5.2 Porcentagem de Divisão das Tarefas por Dispositivo. 59

Capítulo 1

Introdução

A Ciência da Computação tem sido amplamente utilizada nos diversos níveis de cuidado a saúde, incluindo aqueles ligados a Patologia Clínica. A Patologia é a área da Medicina que estuda os desvios de um organismo em relação ao que é considerado normal [12]. Essa área tem como objetivo analisar e diagnosticar doenças, estabelecendo estágios e verificando fatores de risco para a saúde. Nos dias atuais essa análise advém, consideravelmente, do processamento de imagens e de sistemas de apoio a decisão.

O processamento de imagens é uma das áreas da computação que tem concebido numerosos trabalhos em conjunto com a Medicina [24, 43, 52, 53]. Tal fato tem ganhado destaque e, em parte, é consequência da constante evolução dos dispositivos de captura de imagens médicas, vinculados a formação de grandes bases de dados públicas. Imagens capturadas por *scanners* modernos chegam a ter resoluções de até 100K x 100K [26, 54, 58, 60]. Em decorrência do uso dessas tecnologias, há um significativo crescimento no volume dessas bases de dados, e a utilização das mesmas cria uma grande demanda por tarefas de armazenamento, de análise, de visualização e de compressão desses dados. Uma maneira de atender aos requisitos computacionais necessários para se extrair informação desses dados é a utilização de processamento paralelo, principalmente pelo uso de aceleradores.

Aceleradores, em especial, vem ganhando atenção da comunidade devido seu alto poder de processamento e bom custo benefício. Um dos processadores dessa classe de arquiteturas promissoras é o Intel® Xeon Phi™ [39]. O Intel® Xeon Phi™ é baseado na microarquitetura Larrabee [44] e é o primeiro produto a utilizar a arquitetura Intel® *Many Integrated Core* (MIC). Essa arquitetura suporta vários núcleos em um único processador altamente escalável [9], com características semelhantes às unidades de processamento gráfico e compatibilidade com a arquitetura x86 [20].

Por outro lado, as aplicações de análise de imagens médicas são construídas utilizando um conjunto de operações básicas [23, 25, 29], que são combinadas de formas diferentes conforme a análise desejada. Grande parte dessas operações consistem de transforma-

ções morfológicas dos dados, tais como Reconstrução Morfológica [65], Transformada de Distância [63], Transformação Watershed [66], entre outros. Assim, é importante implementar essas operações de forma otimizada nesses novos processadores (aceleradores) buscando beneficiar aplicações que utilizam as mesmas operações.

Muitos algoritmos morfológicos populares podem ser implementados eficientemente por meio de um padrão de computação, chamado Propagação de Ondas Irregulares - do inglês *Irregular Wavefront Propagation Pattern* (IWPP) [60]. IWPP é um padrão de computação no qual um conjunto de elementos formam ondas iniciais que irão se expandir de forma irregular a partir das suas frentes de onda. Essas ondas são dinâmicas, possuem dependência de dados e são computadas ao longo de suas expansões. Cada elemento da frente de onda, dada uma condição de propagação, pode propagar informação ao seu conjunto de elementos vizinhos.

Como apenas os elementos na frente de onda contribuem para o resultado das operações, o IWPP pode ser implementado eficientemente utilizando alguma estrutura de dados auxiliar, que identifique esses elementos, tal como uma fila.

Além dos algoritmos morfológicos previamente citados, diversos outros métodos em processamento de imagens tem seu funcionamento baseado no IWPP, como Esqueletos Euclidianos [32], Esqueletos por Zona de Influência [27], ou a Transformação Watershed [65].

Devido ao seu extenso uso aplicado a grandes volumes de dados e ao seu variado emprego, a construção de uma implementação eficiente para algoritmos IWPP pode beneficiar diversas ferramentas ou aplicações que são amplamente utilizadas em Bioinformática. Uma das formas de se alcançar a eficiência desses algoritmos é por meio do paralelismo, usando arquiteturas especializadas em *Central Processing Unit* (CPU) ou coprocessadores como *Graphic Processing Unit* (GPU) [59] e *Many Integrated Core* (MIC) [20].

Uma particularidade sobre essa classe de algoritmos é que implementações atuais exploram o uso de paralelismo com apoio em operações atômicas, para garantir o resultado final do processamento. Essas operações atômicas geram um custo adicional para as aplicações, e não permitem uma execução eficiente em arquiteturas cuja caraterística principal se baseia no uso do conjunto de instruções *Single Instruciton, Multipe Data* (SIMD). Dessa maneira, esta dissertação busca reprojetar o algoritmo IWPP de forma a possibilitar sua eficiente execução na arquitetura *Many Integrated Core* utilizando instruções *Single Instruction, Multiple Data*.

1.1 Problema

Algoritmos e implementações atuais do IWPP executam ineficientemente no Intel® Xeon Phi™ [56], pois requerem instruções atômicas que não são suportadas nesse copro-

cessador ao utilizar o conjunto de instruções *Single Instruction, Multiple Data* (SIMD). O uso desse conjunto de instruções é fundamental para o desempenho do Intel Phi, onde o tamanho do registrador é uma das características fundamentais dessa arquitetura.

1.2 Objetivo

O objetivo deste trabalho é reprojetar o algoritmo IWPP de forma a possibilitar sua eficiente execução na arquitetura *Many Integrated Core* utilizando instruções *Single Instruction, Multiple Data* (SIMD).

1.3 Objetivos Específicos

1. Reprojetar o algoritmo IWPP;
2. Implementar uma versão vetorial eficiente do algoritmo reprojetado;
3. Paralelizar em nível de *threads* o algoritmo vetorizado;
4. Implementar casos de uso do algoritmo;
5. Implementar uma versão cooperativa para ambientes com processadores

Os primeiros 4 itens dos objetivos já foram publicados [16].

1.4 Organização do Texto

O restante deste trabalho segue a seguinte organização:

Capítulo 2. [Referencial Teórico] Faz uma revisão teórica dos conceitos, métodos e técnicas apresentadas em trabalhos anteriores envolvendo Arquiteturas Paralelas e Algoritmos Morfológicos.

Capítulo 3. [*Irregular Wavefront Propagation Pattern* (IWPP)] Detalha algoritmos da classe IWPP e discute trabalhos relacionados.

Capítulo 4. [IWPP Paralelo Eficiente Utilizando MIC] Apresenta uma implementação eficiente para o algoritmo IWPP detalhando as estratégias utilizadas.

Capítulo 5. [Análise de Resultados] Descreve os resultados a partir do emprego das estratégias apresentadas no capítulo anterior.

Capítulo 6. [Conclusão] Apresenta as conclusões da dissertação por meio de uma discussão sobre objetivos alcançados, e indica sugestões de continuação do trabalho.

Capítulo 2

Referencial Teórico

Neste capítulo serão abordados Algoritmos Morfológicos apresentando notações, exemplos e estratégias para implementação desses algoritmos. Também serão detalhados alguns conceitos básicos sobre arquiteturas paralelas, processadores vetoriais, uma exposição do coprocessador Intel® Xeon Phi™ e definições e classificações envolvendo *data races* e condições de corrida.

2.1 Algoritmos Morfológicos

A Morfologia Matemática é uma abordagem não linear para análise espacial de estruturas. A base da morfologia consiste em extrair de uma imagem desconhecida a sua geometria através de transformações em uma outra imagem completamente definida, ou seja, extrair informações associadas à geometria dos objetos e a topologia de um conjunto desconhecido pela transformação através de outro conjunto bem definido, chamado elemento estruturante [68]. Inicialmente, ela foi concebida para manipular imagens binárias, sendo estendida, mais tarde, para imagens em tons de cinza utilizando teoria de reticulados [45] [48].

Transformações morfológicas são funções complexas que compartilham características simples e intuitivas. Utilizar uma aplicação de análise de imagens envolve, na maioria das vezes, a concatenação de diversas transformações de baixo nível. Por esta razão, a implementação de cada uma dessas transformações deve ser a mais otimizada possível.

Os algoritmos oriundos dessas transformações podem ser classificados em quatro famílias, que serão detalhadas na Seção 2.1.3: Algoritmos Paralelos, Algoritmos Sequenciais, Algoritmos Baseados em Filas de Pixels e Algoritmos Híbridos [64].

2.1.1 Notações em Algoritmos Morfológicos

Consideram-se imagens binárias ou em tons de cinza I o mapeamento de um domínio retangular $D_I \subset \mathbb{Z}^2$ em $\mathbb{Z}$ [64]. Em uma imagem, cada pixel p pode assumir o valor 0 ou 1 para imagens binárias, e valores de um conjunto $\{0, \ldots, L-1\}$ para imagens em tons de cinza, onde cada valor de L denota a tonalidade de cinza. Diversos algoritmos podem estender suas funcionalidades para espaços n-dimensionais, mas para efeitos deste trabalho, serão descritas, unicamente, imagens bidimensionais.

Uma grade subjacente $G \subset \mathbb{Z} \times \mathbb{Z}$ define uma relação de vizinhança entre pixels. G diz respeito a conectividade, e define o número de vizinhos acessíveis a um determinado pixel. A conectividade pode ser dada por uma grade quadrada ou hexagonal. $N_G(p)$ representa o conjunto de vizinhos q de um pixel $p \in \mathbb{Z}^2$, conforme uma grade G, de acordo com o apresentado na Equação 2.1.

$$N_G(p) = \{q \in \mathbb{Z}^2 \mid (p, q) \in G\} \tag{2.1}$$

Um pixel p é vizinho de um pixel q se $(p, q) \in G$. Se G é 4-conectado, então os pixels acessíveis a p são os quatro pixels adjacentes por borda (Figura 2.1), e se G é 8-conectado, p acessa pixels que são adjacentes por borda e por vértice (Figura 2.2).

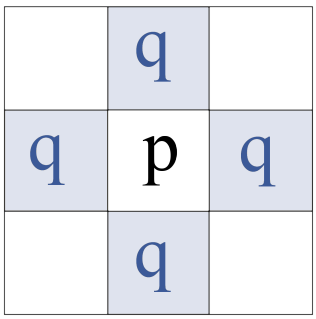

Figura 2.1: Representação de um Conjunto de Vizinhos 4-Conectado.

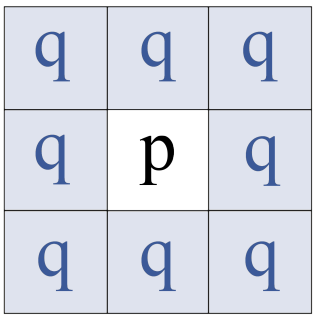

Figura 2.2: Representação de um Conjunto de Vizinhos 8-Conectado.

2.1.2 Exemplos de Algoritmos Morfológicos

Esta seção apresenta alguns exemplos de algoritmos morfológicos conhecidos da literatura.

Watershed

Watershed é um método para segmentação de uma imagem em tons de cinza em regiões nas quais, de um ponto de vista tridimensional, cada pixel corresponde a uma posição e os níveis de cinza determinam a altitude (Figura 2.3).

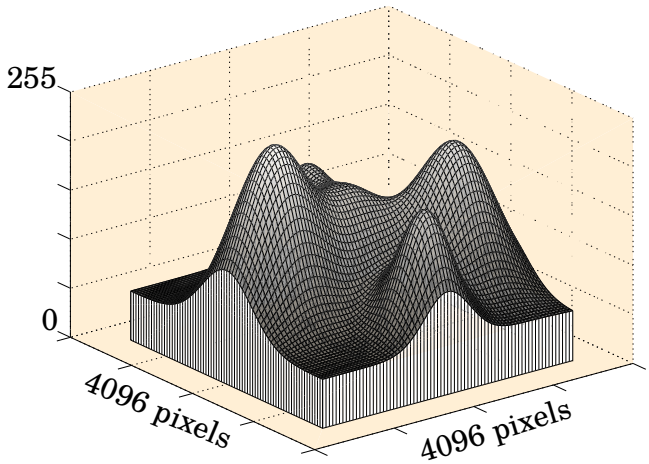

Figura 2.3: Representação Tridimensional de uma Imagem em Tons de Cinza.

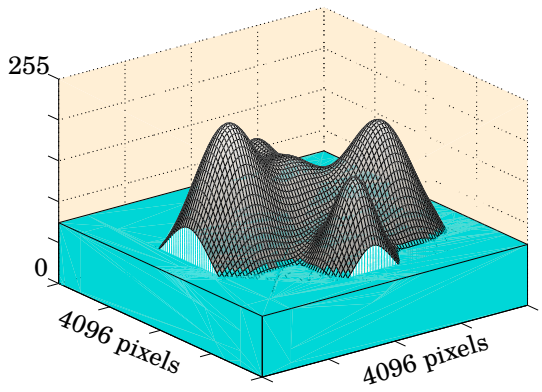

Figura 2.4: Início da Inundação.

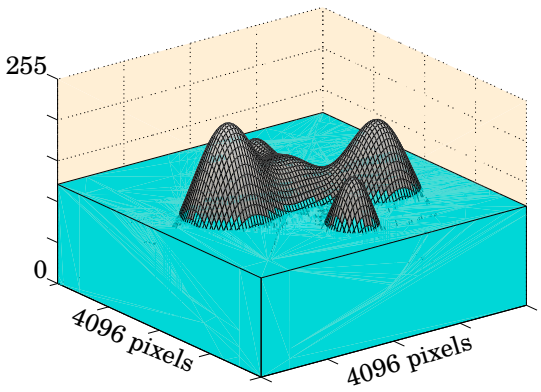

Figura 2.5: Avanço da Inundação.

Esse algoritmo utiliza dois passos: ordenação dos pixels e inundação do relevo. A ordenação determina a distribuição cumulativa por níveis de cinza e guarda os endereços únicos de acesso de cada um dos pixels. A inundação é o relacionamento da estrutura topográfica formada, com a inserção progressiva de água em suas regiões mais baixas. Essas inserções progressivas provocam o surgimento de bacias de captação. Com o avanço da inundação e uma vez que bacias de captação estão para se misturar, são formadas

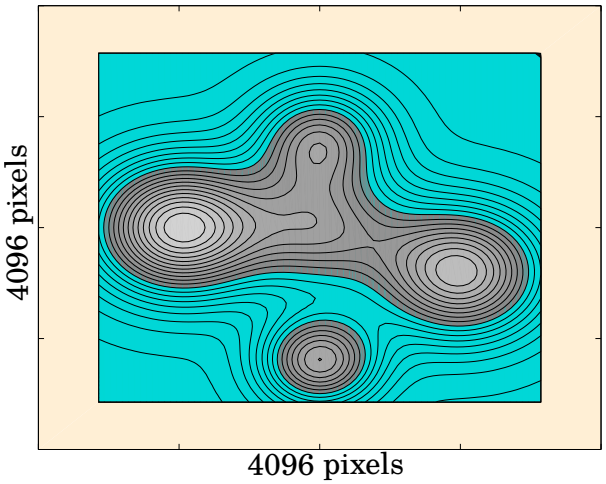

Figura 2.6: Vista Superior da Segmentação.

regiões de barragem que consistirão na imagem segmentada da Figura 2.6 em regiões com formatos descontínuos [17].

Para a inundação do relevo utiliza-se uma fila FIFO e busca em largura onde, para cada nível, a inundação continua em seus novos mínimos locais ou nas bacias de captação mais baixas. Todos os novos mínimos do nível analisados são, então, descobertos e tratados separadamente.

Esqueleto Morfológico

Na extração das características de um objeto, o esqueleto consiste no afinamento do referido, com intenção de formar um conjunto de retas ou linhas que sintetizem a informação do objeto original preservando a sua homotopia[1] (Figuras 2.7 e 2.8). Assim, em um conjunto $X \subset \mathbb{Z}^2$, o esqueleto $S(X)$ é o conjunto de pixels nos quais diferentes afinamentos, chamados de *wavefronts*, se encontram [4]. Pela necessidade da propriedade de homotopia e para sua utilidade prática, esse processo precisa preservar o número de componentes conectados e o número de buracos no conjunto original, assim como as relações entre esses componentes e os buracos.

Em uma definição mais formal [67], o esqueleto $S(X)$ de um conjunto $X \subset \mathbb{Z}^2$ é o conjunto de centros de *maximal balls* B dentro da forma dada pela Equação 2.2:

$$S(X) = \{p \in X, \exists r \geq 0, B(p,r) \textit{ maximal ball de } X\} \tag{2.2}$$

[1]Funções contínuas que transformam um espaço topológico em outro se uma puder ser "continuamente deformada" na outra.

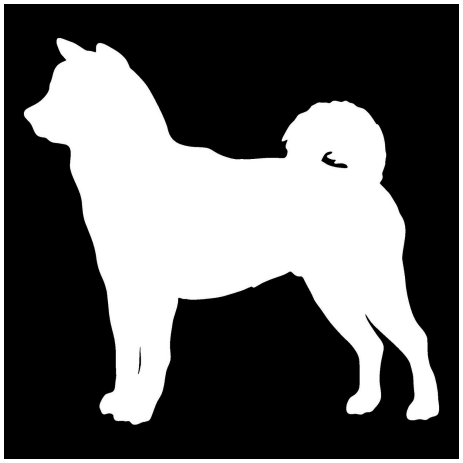

Figura 2.7: Imagem Original.

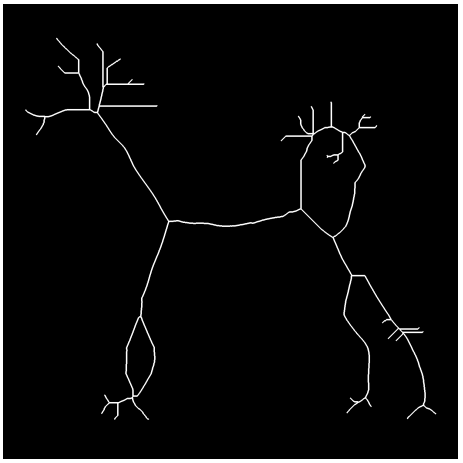

Figura 2.8: Esqueleto Morfológico.

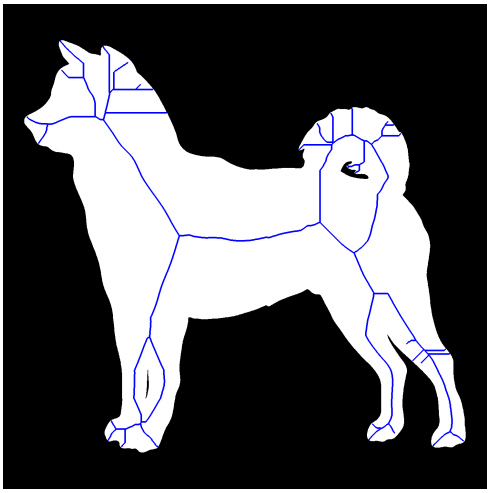

Figura 2.9: Visualização do Resultado.

A primeira proposta de solução para este método apresentada na literatura [4] realizava sucessivas remoções até que a estabilidade fosse alcançada. Propostas posteriores utilizaram uma abordagem relacionada a uma sequência definida de pixels [41]. Essas sequências são chamadas de varredura. Utilizando um número fixo de varreduras, chegou-se a uma abordagem precisa, mais eficiente e que poderia ser estendida a outros tipos de algoritmos. Contudo, esses algoritmos requeriam custosa análise de vizinhança e possuíam baixa flexibilidade. Dessa forma, outros métodos surgiram na literatura [48] [61], porém os mesmos apresentavam alto grau de complexidade e especificidade.

Baseada nas obervações anteriores, o algoritmo de esqueleto morfológico apresentado em 1991 [67] utilizava remoções homotópicas, *crest points* e contornos. Esse algoritmo inicia-se pelas fronteiras de X, onde são realizadas remoções até que a estabilidade seja alcançada. Tais remoções são implementadas utilizando-se filas de pixels. A cada passo, o pixel p pode ser removido se, e somente se, uma das seguintes condições forem atendidas:

p não pertence ao esqueleto de *maximal balls*, ou p não modifica a localidade homotópica. A primeira condição garante a precisão do algoritmo, e a segunda a propriedade de homotopia. Assim como outros algoritmos que utilizam filas FIFO, é um método eficiente uma vez que utiliza somente aqueles pixels ativos do passo de propagação.

Reconstrução Morfológica

Na reconstrução, dadas duas imagens (binárias ou em tons de cinza) I e J onde $J \leq I$ (ou seja, para cada pixel p, no domínio das imagens $J(p) \leq I(p)$), a reconstrução $R_I(J)$ de I em J é obtida por sucessivas dilatações em J utilizando I como limite, até que a estabilidade seja alcançada (Figura 2.10). I é chamada de imagem máscara e J é chamada imagem marcadora. As conectividades elementares utilizadas na reconstrução são a hexagonal (conectividade-6), quadrada S_1 (conectividade-4 com 5 pixels) ou quadrada S_2 (conectividade-8 com 9 pixels). Sejam δ_G as dilatações com respeito a G, e $\wedge$ a operação de comparação mínima pixel-a-pixel.

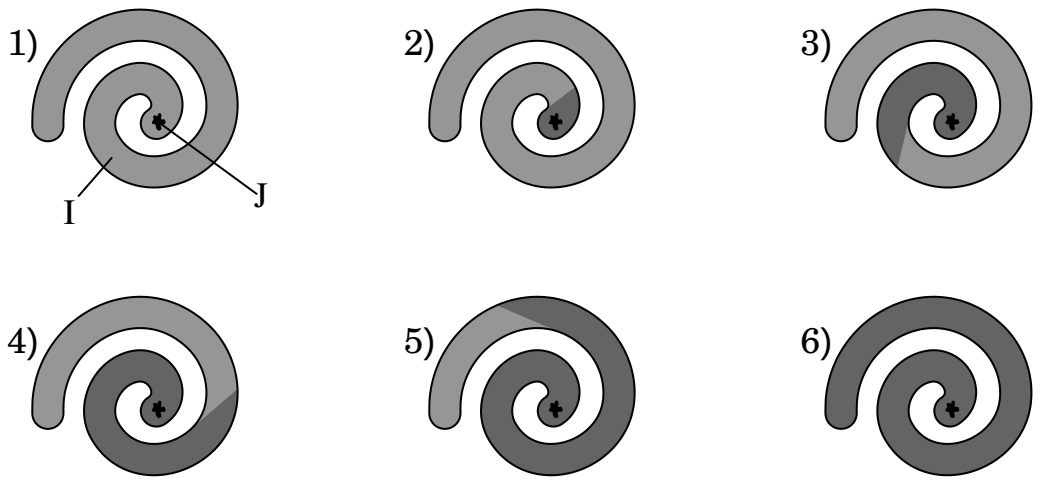

Figura 2.10: Reconstrução Morfológica. Adaptado de [64].

A reconstrução de I em J é obtida aplicando, até que a estabilidade seja alcançada, a operação apresentada pela Equação 2.3:

$$J \longleftarrow \delta_G(J) \wedge I. \tag{2.3}$$

Observando a Equação 2.3 é possível obter o Algoritmo 1 para a reconstrução. Esse algoritmo realiza sucessivas propagações verificando a vizinhança de cada pixel p (linhas

3 e 4), e esse procedimento continua até que a imagem reconstruída J não seja alterada por toda uma iteração do laço (linhas 1 a 6). Esse algoritmo é simples em sua construção, porém é ineficiente devido a irregularidade de suas atualizações.

Algoritmo 1: Reconstrução Morfológica

Entrada: I, imagem binária ou em tons de cinza
Entrada: J, imagem binária ou em tons de cinza
// $\forall p \in D_I, J \leq I$
Saída: J, imagem binária ou em tons de cinza reconstruída

```
1 repita
2 |   Copia J para J_temp
3 |   para cada pixel p ∈ D_{J_temp} faça
4 |   |   J(p) ← max{J_temp(p), N_G(p)} ∧ I(p)
5 |   fim
6 até alcançar a estabilidade;
```

Recorrendo a um algoritmo que realize alterações na própria imagem, é possível realizar as atualizações por meio de varreduras *raster* e *anti-raster* [64], porém tais varreduras precisam ser repetidas diversas vezes até que a estabilidade seja alcançada.

A reconstrução é uma ferramenta morfológica muito expressiva e possui diversas aplicações, principalmente, no caso de imagens em tons de cinza. Como exemplo, pode-se citar a extração das áreas de maior interesse (picos) que determinam objetos relevantes na imagem.

Transformada de Distância

A operação Transformada de Distância (DT) calcula o mapa de distâncias M de uma imagem binária de entrada I, onde para cada pixel $p \in I$, que faz parte do objeto (*foreground*), seu valor em M é a menor distância a partir de p até o pixel de fundo (*background*) mais próximo da imagem. A Figura 2.11 apresenta um exemplo de execução da Transformada de Distância. Inicialmente é dada uma imagem binária como entrada para o algoritmo no qual o fundo da imagem possui valor 0 (zero) e os objetos, cuja distância será calculada, possuem valor inicial de 1 (um). O algoritmo irá identificar para cada elemento ou pixel do objeto qual é o elemento do fundo mais próximo a ele, seguindo alguma métrica. Após a identificação, o valor da distância é calculado e inserido na posição do elemento do objeto. Após o cálculo da distância de todos elementos dos objetos, a imagem constituirá um mapa com a distância de cada um dos objetos da imagem binária.

A Transformada de Distância possui diversas aplicações, dentre elas podem ser citadas a base para separação de objetos sobrepostos por meio da Transformação *Watershed*, a navegação robótica para escolha do menor caminho, a computação de representações geométricas como diagramas de Voronói e triangulações de Delaunay, análise comparativa da forma de objetos, e análise de interação entre estruturas biológicas [14].

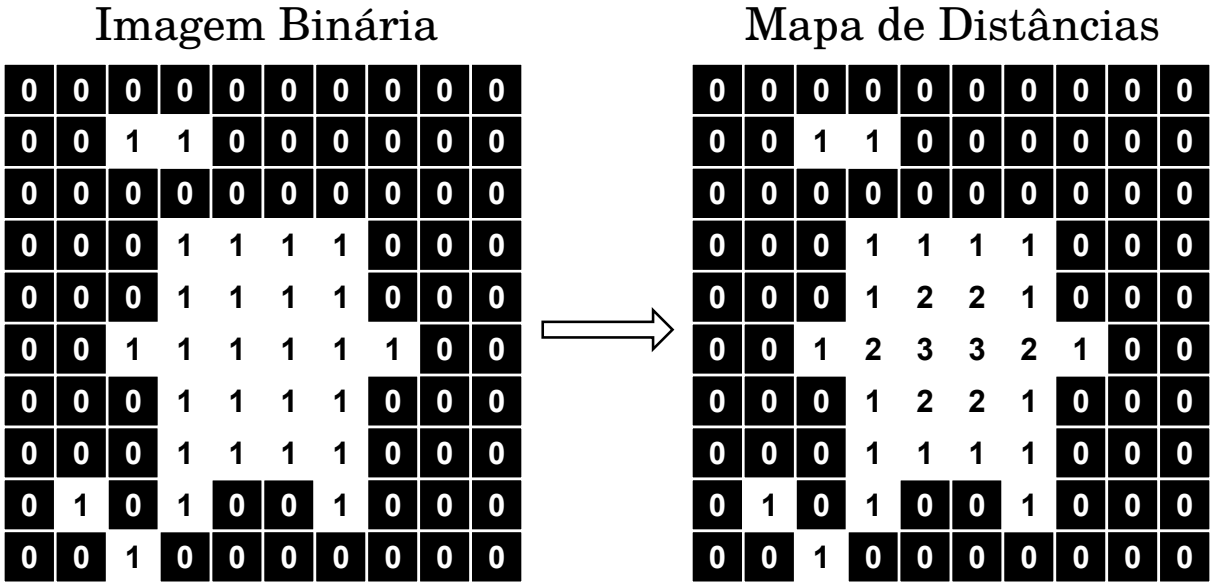

Figura 2.11: Exemplo Numérico da Transformada de Distância *City-block*.

Implementações para a Transformada de Distância podem ser classificadas de diversas formas: pela complexidade, pela eficiência, pela ordem de varredura ou pela métrica de distância utilizada. Quanto a métrica, a Transformada de Distância pode ser chanfrada como nas *city-block*, *chess-board* ou *octogonal* [6]; euclidiana ou euclidiana aproximada [11] devido as dificuldades de construções eficientes para a sua forma exata.

2.1.3 Estratégias para Implementação de Algoritmos Morfológicos

Esta seção apresenta estratégias de implementação para algoritmos morfológicos que podem ser classificadas em quatro grandes grupos: Algoritmos Paralelos, Algoritmos Sequenciais, Algoritmos Baseados em Filas de Pixels e Algoritmos Híbridos.

Algoritmos Paralelos

São os algoritmos clássicos e mais comuns no campo da morfologia. Eles funcionam da seguinte forma: dada uma imagem I como entrada, todos os pixels são escaneados e o novo valor de cada pixel p é determinado a partir da vizinhança $N(p)$. O êxito deste algoritmo reside na restrição de que a atualização dos pixels deverá ser realizada em uma imagem J diferente de I. J é então copiada para I e é realizada mais uma rodada de escaneamento dos pixels em I. Tais passos são executados até que nenhuma modificação em J seja feita a partir dos escaneamentos de I. Os passos descritos podem ser observados na Transformada de Distância Paralelo, demonstrados no Algoritmo 2.

Algoritmo 2: Transformada de Distância Paralelo

Entrada: I, imagem binária
Saída: J, imagem em tons de cinza $D_I; J \neq I$

```
1 repita
2     para cada pixel p ∈ D_I faça
3         se I(p) = 1 então
4             J(p) ← min{I(q), q ∈ N_G(p)} + 1
5         fim
6     fim
7     Copia J para I
8 até alcançar a estabilidade;
```

Como I é diferente de J, vários pixels p de uma rodada de escaneamento podem ser processados em paralelo. O número de escaneamentos deste algoritmo é proporcional à maior distância calculada. Devido a essa característica, algoritmos paralelos requerem grande número de varreduras na imagem completa, o que o torna inadequado para computadores convencionais, mesmo em algoritmos paralelos mais básicos.

Algoritmos Sequenciais

São algoritmos que foram desenvolvidos como forma de evitar o alto número de escaneamentos. Essa classe de algoritmos remete aos seguintes preceitos: os pixels devem ser escaneados em uma ordem pré-definida (chamado de varredura); e o valor atualizado do pixel, determinado a partir da computação com seus valores vizinhos, é escrito na própria imagem I analisada [64]. De maneira oposta aos algoritmos paralelos, a ordem de execução dos algoritmos sequencias é essencial ao resultado.

Para calcular a Transformada de Distância de uma imagem I, os escaneamentos são suficientemente realizados na forma de duas varreduras: *raster* que é realizada do início para o fim da imagem (Figura 2.12); e uma *anti-raster* que realiza a varredura na imagem do fim para o início (Figura 2.13). Cada uma dessas varreduras irá utilizar um seguimento

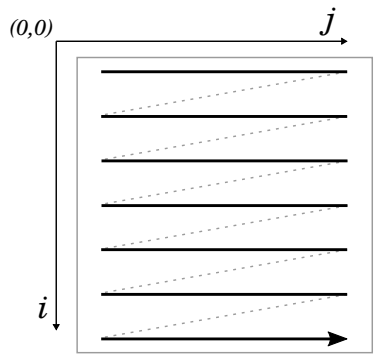

Figura 2.12: Varredura *Raster*.

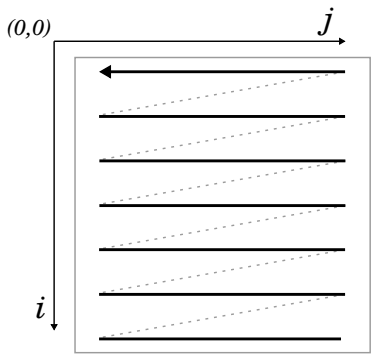

Figura 2.13: Varredura *Anti-raster*.

específico do seu conjunto de vizinhos u de um pixel $p \in \mathbb{Z}^2$ para realizar a respectiva atualização (Figura 2.14). Tais procedimentos podem ser observados no Algoritmo 3.

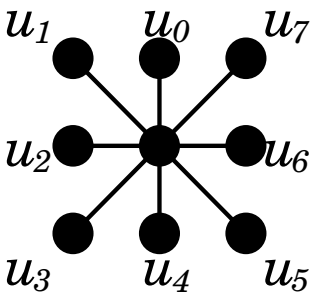

Figura 2.14: Pixels Utilizados nas Varreduras.

Algoritmo 3: Transformada de Distância Sequencial

Entrada: I, imagem binária
Saída: I, imagem em tons de cinza

1 **varredura** D_I **na ordem** *raster* **faça**
2 | Sendo p o pixel atual **se** $I(p) \neq 0$ **então**
3 | | $I(p) \leftarrow min\{I(p+u_0)+1, I(p+u_1)+1, I(p+u_2)+1, I(p+u_3)+1\}$
4 | **fim**
5 **fim**
6 **varredura** D_I **na ordem** *anti-raster* **faça**
7 | Sendo p o pixel atual **se** $I(p) \neq 0$ **então**
8 | | $I(p) \leftarrow min\{I(p), I(p+u_4)+1, I(p+u_5)+1, I(p+u_6)+1, I(p+u_7)+1\}$
9 | **fim**
10 **fim**

A varredura *raster* (linhas 1 a 5 do Algoritmo 3) gera uma imagem em tons de cinza intermediária, onde os maiores valores estão concentrados abaixo e a esquerda da imagem I. Cada pixel p possuirá o menor caminho P entre p e o *background* da imagem, seguindo a regra: cada caminho xy de P tem um componente vertical estritamente positivo ou um componente vertical zero, e um componente positivo à esquerda. Dessa forma, a varredura do tipo *anti-raster* (linhas 6 a 9 do Algoritmo 3), após a varredura *raster*, irá realizar os ajustes de distância a partir dos componentes inferiores e a direita [64].

Como duas varreduras são suficientes na execução do algoritmo [41], há uma evidente melhoria em relação ao algoritmo paralelo. Assim, algoritmos sequenciais constituem uma das melhores alternativas em termos de algoritmos morfológicos e, também, podem ser facilmente expandidos para trabalhar com n-dimensões [5].

Algoritmos Baseados em Filas de Pixels

Esta classe considera toda a imagem como um grafo, onde cada pixel é um vértice e as arestas são as conexões entre os mesmos, admitidas pela grade (conectividade). Os limites

de dilatação são estruturas isotrópicas[2], e é utilizada uma fila de pixels que executa uma busca em largura no grafo.

Algoritmo 4: Transformada de Distância com Filas de Pixels

Entrada: I, imagem binária
Saída: I, imagem em tons de cinza

```
1  para cada pixel p ∈ D_I faça
2      se I(p) = 1  E  ∃p' ∈ N_G(p), I(p') = 0 então
3          fila.insere(p);
4          I(p) ← 2
5      fim
6  fim
7  enquanto fila.vazia() = falso faça
8      p ← fila.retira()
9      para cada p' ∈ N_G(p) faça
10         se I(p') = 1 então
11             I(p') ← I(p) + 1
12             fila.insere(p')
13         fim
14     fim
15 fim
```

A estrutura de dados utilizada é a fila *First In, First Out* (FIFO), na qual os primeiros pixels inseridos, serão os primeiros a serem retirados para processamento. Dessa forma, cada pixel representa um vértice e a fila constitui-se de uma determinada quantidade de ponteiros que assinalam o endereço de um determinado pixel. As operações utilizadas na fila são *insere*, na qual é colocado um pixel p no final da fila; *retira* que realiza a leitura do pixel que se encontra no início da fila (leitura do seu endereço) e o retira da mesma; e *vazia* que retorna verdadeiro se a fila estiver vazia ou falso caso contrário. Dadas as operações do parágrafo anterior, a Transformada de Distância é apresentada no Algoritmo 4.

Algoritmos baseados em filas de pixels são extremamente eficientes e simples. Eles são, também, mais adequados para transformações mais complexas como a reconstrução em tons de cinza [59], esqueleto morfológico e a transformação *watershed*.

Algoritmos Híbridos

Algoritmos Híbridos são algoritmos projetados para utilizar segmentos das outras técnicas já citadas. Tal abordagem busca prover um algoritmo cuja combinação de técnicas seja mais eficiente do que a utilização individual de qualquer uma dessas.

A construção de um algoritmo híbrido é feita pela identificação de segmentos que podem ser aproveitados de forma isolada. A divisão mais comum separa algoritmos morfológicos nas fases de Inicialização e Processamento. Essa separação pode ser observada

[2]Estruturas que apresentam as mesmas propriedades em todas as direções

no Algoritmo 5. Na fase de Inicialização podem ser realizadas varreduras sequenciais ou paralelas, como aquelas já descritas nas seções anteriores. Essa fase tem como objetivo realizar um processamento mais grosseiro, com a possibilidade de inicializar uma estrutura de dados que terá continuidade na fase de Processamento. A fase de Processamento executa tarefas que processam em elementos específicos dentro da imagem, auxiliados por alguma estrutura de dados que indicará o ponto de partida e a continuação para tal execução.

Algoritmo 5: Transformada de Distância Híbrida

Entrada: I, imagem binária
Saída: I, imagem em tons de cinza

```
1  varredura D_I na ordem raster faça
2      Sendo p o pixel atual se I(p) ≠ 0  E  ∃p' ∈ N_G(p), I(p') = 0 então
3          fila.insere(p);
4          I(p) ← 2
5      fim
6  fim
7  enquanto fila.vazia() = falso faça
8      p ← fila.retira()
9      para cada p' ∈ N_G(p) faça
10         se I(p') = 1 então
11             I(p') ← I(p) + 1
12             fila.insere(p')
13         fim
14     fim
15 fim
```

Uma implementação possível híbrida da Transformada de Distância utiliza varreduras do método sequencial (apresentado no Algoritmo 3), em conjunto com uma estrutura de dados que irá receber aqueles elementos identificados na varredura inicial. Utilizando a estratégia baseada em filas de pixels, a estrutura de dados consome e insere elementos a medida que seus valores são verificados e atualizados, conforme condição de propagação.

2.2 Arquiteturas Paralelas

Arquitetura paralela é uma forma de fornecer estruturas explícitas e de alto nível para o desenvolvimento de soluções utilizando processamento paralelo, dado pelo uso de múltiplos processadores cooperando para resolver problemas através de execução concorrente [13]. Dessa forma, busca-se quebrar problemas em partes independentes de forma que cada um dos processadores disponíveis seja responsável por uma parcela específica do processamento.

O paralelismo pode ser realizado de diversas formas (bit, instrução, dado ou tarefa), e computadores paralelos são classificados de acordo com o suporte de paralelismo dado pelo hardware.

2.2.1 Classificação de Arquiteturas Paralelas

Diversas arquiteturas paralelas já foram sugeridas e implementadas na literatura, e também diversas classificações foram propostas. Dentre as mais conhecidas é possível citar a proposta por Flynn [15] e a proposta por Duncan [13].

Taxonomia de Flynn

A classificação proposta por Flynn [15] agrupa as arquiteturas segundo os seus fluxos de instruções e de dados. A classificação é a seguinte:

- *Single Instruction, Single Data* (SISD) - Um único fluxo de instrução para um único fluxo de dados. São as máquinas não paralelas da arquitetura de von Neumann [37];
- *Múltiple Instruction, Single Data* (MISD) - Múltiplos fluxos de instrução para um único fluxo de dados;
- *Single Instruction, Multiple Data* (SIMD) - Um único fluxo de instrução para múltiplos fluxos de dados, sendo ideal para a paralelização de laços simples que operam sobre grandes vetores de dados [37]. Nesta classe podem ser citados, como exemplo, os processadores vetoriais; e
- *Multiple Instruction, Multiple Data* (MIMD) - Múltiplos fluxos de instrução para múltiplos fluxos de dados. Nesta classe, encontram-se, por exemplo, sistemas multicomputadores (*clusters*) e multiprocessadores com memória compartilhada.

Taxonomia de Duncan

A Taxonomia de Duncan [13] surgiu de maneira a classificar de forma detalhada e hierárquica as arquiteturas paralelas. Essa classificação usa como base as principais arquiteturas paralelas e pode ser visualizada na Figura 2.15. A classificação é descrita como:

- Arquiteturas Assíncronas: Possuem controle descentralizado com unidades de processamento independentes que operam em conjuntos diferentes de dados. Elas são divididas em MIMD e paradigma MIMD.

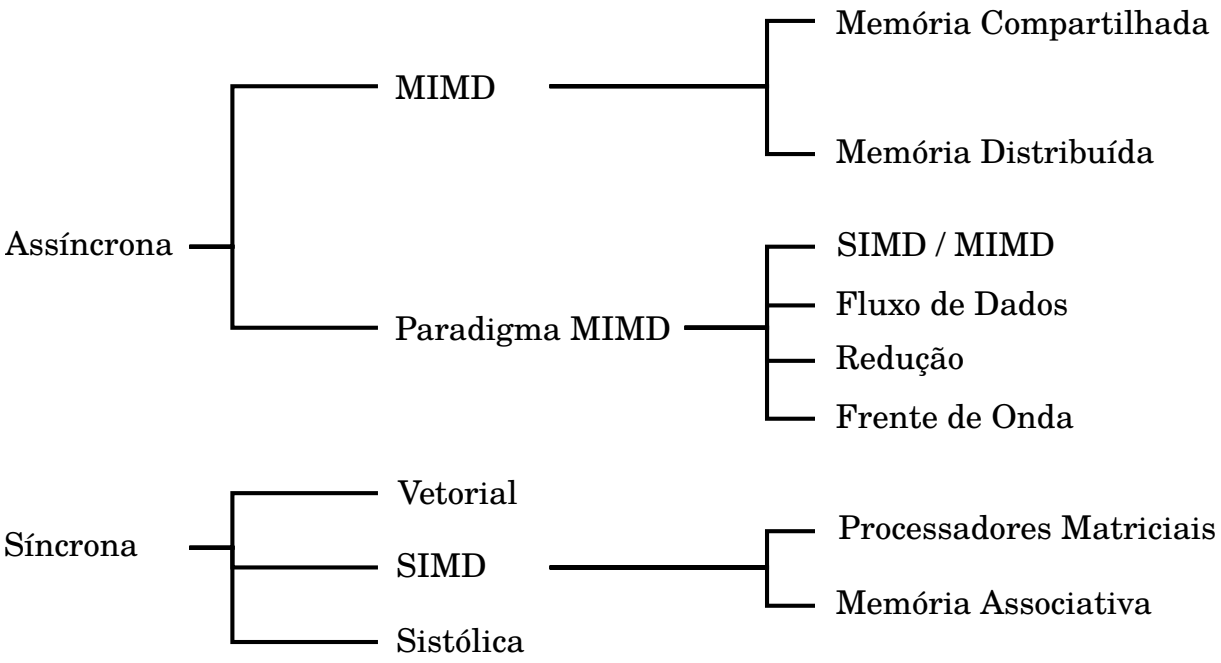

Figura 2.15: Taxonomia de Duncan. Adaptado de [13].

- MIMD: São arquiteturas formadas por múltiplas unidades de processamento independentes, executando conjuntos de instruções diferentes sobre conjuntos de dados distintos. São divididas em memória compartilhada e memória distribuída.
  - Memória Compartilhada: Possui uma única memória global utilizada para comunicação e sincronização entre as unidades de processamento.
  - Memória Distribuída: Cada unidade de processamento possui sua própria memória local e sua comunicação ocorre por meio de troca de mensagens.
- Paradigma MIMD: São arquiteturas com características peculiares além daquelas que as classificam como MIMD.
  - SIMD / MIMD: Possui segmentos de arquitetura MIMD que funcionam sob a arquitetura SIMD.
  - Fluxo de Dados: Instruções são executadas de acordo com a disponibilidade dos operandos utilizados.
  - Redução: Instruções são executadas somente quando seus respectivos resultados são requeridos como operandos de outra instrução em execução.
  - Frente de Onda: Combinam vários processadores em *pipeline*, nos quais apenas alguns realizam comunicação com a memória, trabalhando de acordo com a disponibilidade dos operandos utilizados por cada instrução.

- Arquiteturas Síncronas: Possuem controle centralizado dado pelo *clock*, unidade de controle central ou controlador de unidade vetorial.
  - Vetorial: Executam operações em vetores de dados.
  - SIMD: Um único fluxo de instrução para múltiplos fluxos de dados, sendo dividido em Arranjo de Processadores e Memória Associativa.
    * Processadores Matriciais: Nesta arquitetura as estruturas de processamento são arranjadas na forma de matrizes de dados para operação.
    * Memória Associativa: É a arquitetura com acesso a memória vinculado ao seu conteúdo, ao invés do uso de endereço.
  - Sistólica: Esta arquitetura busca suprir necessidades de computação intensiva em operações de Entrada e Saída (E/S), combinando vários processadores em pipeline, onde apenas alguns desses realizam comunicação com a memória.

A Taxonomia de Duncan detalha Arquiteturas Síncronas do tipo Vetorial no qual uma mesma operação é executada em todo um vetor de uma só vez. Essa mesma característica se apresenta na Taxonomia de Flynn em um contexto mais genérico como SIMD, observando um fluxo de instrução único para diversos fluxos de dados. O objeto de pesquisa deste trabalho é o coprocessador Intel® Xeon Phi™ que possui as características citadas, por ser um processador com instruções do tipo vetorial. Dessa forma, as seções seguintes irão detalhar tanto processadores vetoriais (Seção 2.2.2) quanto o coprocessador Intel® Xeon Phi™ (Seção 2.2.3).

2.2.2 Processadores Vetoriais

Processador Vetorial é um dos tipos de sistema SIMD que operam diretamente em matrizes ou vetores de dados, enquanto processadores convencionais operam em elementos de dados individuais ou escalares. Processadores Vetoriais possuem as seguintes características [37]:

1. Registradores Vetoriais - Registradores capazes de guardar um vetor de operandos e processar, simultaneamente, múltiplos operandos nesse vetor.
2. Unidades Funcionais Vetorizadas - As operações são aplicadas a cada elemento do vetor ou, então, a cada conjunto de elementos.
3. Instruções Vetoriais - Instruções que operam em todo o vetor ao invés da forma escalar. Por exemplo, em um processador vetorial onde a largura do vetor é *vector_lenght*, um simples laço como

```
for (i = 0; i < n; i++) {
    x[i] += y[i];
}
```

requer uma instrução de *load*, uma de *add* e uma de *store* para cada bloco de *vector_lenght* elementos, enquanto um sistema convencional (SISD) requer um *load*, um *add* e um *store* para cada elemento.

4. Memória Intercalada - A memória é dividida em bancos e, após o acesso a determinado local, é necessária uma espera antes de acessar esse local novamente. Então, os elementos de um vetor são distribuídos em vários bancos como forma de mitigação de atraso das instruções *load* e *store*.

5. Acesso espaçado a memória e *scatter/gather* - O acesso espaçado a memória permite acessar elementos de um vetor que possuam intervalos fixos. Já o *scatter/gather* escreve ou lê elementos localizados em posições irregulares da memória em um registrador vetorial.

Em aplicações regulares que manipulam matrizes, processadores vetoriais são fáceis e rápidos de se utilizar. Além disso, compiladores auxiliam na identificação de códigos que podem ser vetorizados e também fornecem informações úteis quando isso não é possível [37]. A próxima seção discutirá o Intel® Xeon Phi™ que é um exemplo específico de processador vetorial, além de ser o foco desta dissertação.

2.2.3 Intel® Xeon Phi™

O coprocessador Intel® Xeon Phi™ foi lançado em novembro de 2012 e é baseado na arquitetura *Intel Many Integrated Core* (Intel® MIC), suportando execução simultânea de até 244 *threads* em seus 61 núcleos para atingir um desempenho teórico de até 1.2 *teraflops* com baixo consumo energético [20][3].

Ele funciona acoplado a um *host*, porém possui espaço de memória independente com comunicação através do barramento *Peripheral Component Interconnect Express* 3.0 (*PCI-Express*). Comparado com CPUs convencionais, o Intel® Xeon Phi™ possui características singulares que são essenciais para se atingir alto desempenho. As principais características são:

- A largura dos vetores utilizados pela Unidade de Processamento Vetorial (VPU) é de 512 bits. Sendo assim, uma unidade pode processar até 8 elementos de dados de precisão dupla ou 16 elementos de precisão simples. Comparado com CPUs

[3]Informações baseadas no modelo 7120P.

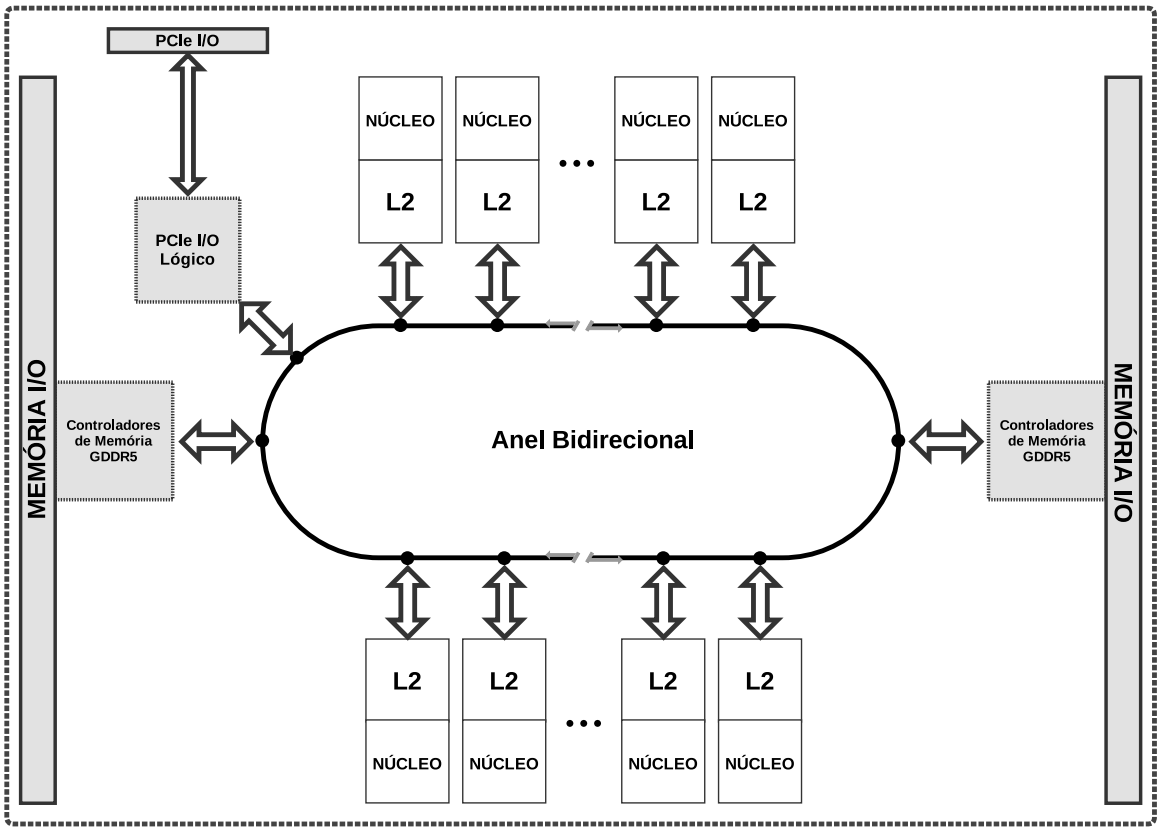

Figura 2.16: Diagrama de Bloco do Intel® Xeon Phi™. Adaptado de [8].

mais recentes como o Intel® Core™ i7-6700K ou o AMD Opteron™ 6386 SE, seus registradores possuem, pelo menos, o dobro da largura. Assim, a utilização das VPUs de forma efetiva é um do elementos essenciais para se obter alta performance nesse processador. As VPUs podem ser exploradas por implementações manuais, utilizando instruções SIMD, ou por autovetorização através do compilador Intel®. A autovetorização é um módulo que tenta identificar *loops* ou segmentos que podem ser vetorizados para utilizar VPUs SIMD em tempo de compilação.

- Cada núcleo do Intel® Xeon Phi™ suporta até quatro *threads* de hardware, totalizando 244 *threads*. Isso possibilita uma melhor utilização do processador em casos nos quais uma *thread* não é capaz de explorar completamente cada núcleo.

- Cada núcleo possui uma cache L2 unificada de 512 KB, que é coerente nos 61 núcleos. Tais caches são interconectados por um barramento em anel de largura de 512 bits (veja a Figura 2.16). Se ocorrer um *chache miss* em alguma cache L2, através de *Distributed Tag Directories* (DTD), as solicitações são encaminhadas para os outros núcleos através da rede em anel.

Plataformas MIC combinam características de CPUs de uso geral e aceleradores, como

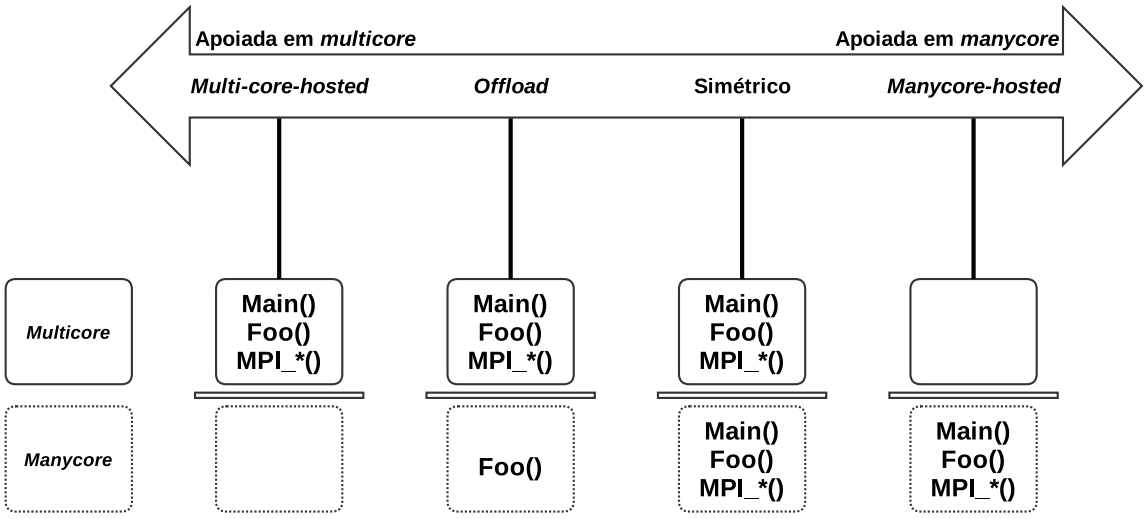

Figura 2.17: Abordagens de Programação Utilizando o Intel® Xeon Phi™.

GPUs. O Intel® Xeon Phi™ é baseado no conjunto de instruções x86 e suporta modelos de programação em memória compartilhada para a comunicação entre núcleos. Essas características minimizam o esforço de programação para o caso da portabilidade de código CPU para o Xeon Phi, em particular porque linguagens de programação como C/C++ e Fortran, e modelos de programação como *Open Multi-Processing* (OpenMP), *Posix Threads* (PThreads) e *Message Passing Interface* (MPI) podem ser utilizados sem modificação de código [39].

Além disso, o Intel® Xeon Phi™ possui 8GB de memória *Graphics Double Data Rate* (GDDR) local com 320Gb/s de transmissão, possui 32KB de memória cache L1 e 512KB de cache L2, executa um *kernel* do Linux em um de seus núcleos e cada núcleo possui sua própria Unidade de Processamento Vetorial (VPU). Assim, a partir das suas características, o Intel® Xeon Phi™ permite as seguintes abordagens de execução (veja a Figura 2.17):

- Modo "Nativo" (*Many-core-hosted*): Dado que o Intel® Xeon Phi™ executa seu próprio sistema operacional, é possível compilar uma aplicação de forma a permitir sua transferência e a execução do arquivo binário direto no coprocessador.
- Modo *Offload*: Nesse modo uma aplicação inicia sua execução no *host* e pode transferir partes da computação para o Intel® Xeon Phi™.
- Modo Simétrico: Modo onde um aplicativo está executando tanto no *host* quanto no Intel® Xeon Phi™, e processos específicos podem executar em diferentes dispositivos.

2.2.4 *Data Race* e Condição de Corrida

Em arquiteturas paralelas, diversos erros ou *bugs* de programação estão diretamente ligados a *data races* e condições de corrida [34].

Data race ocorre quando dois acessos a mesma posição de memória entram em conflito e pelo menos um desses acessos é uma escrita [2]. Já condição de corrida é definida como uma situação na qual múltiplas *threads* realizam leitura e escrita em uma área compartilhada de memória e o resultado final depende da ordem de execução [7].

Em uma análise detalhada desse contexto, *data races* podem ou não levar a condições de corrida, e diversos mecanismos tem sido desenvolvidos para localizar automaticamente *data races* e classificá-los de acordo com suas potencialidades dentro do código. *Data races* podem ser potencialmente prejudiciais ou potencialmente benignos [34]. *Data races* potencialmente prejudiciais afetam a corretude de um algoritmo e necessitam ser corrigidos. Já *data races* potencialmente benignos possuem comportamentos que dificultam a sua identificação em análises estáticas, porém eles não afetam a corretude do algoritmo, mas precisam de uma análise dinâmica para facilitar a sua identificação. Alguns motivos para a ocorrência de *data races* potencialmente benignos são:

- Sincronização Construída pelo Usuário: São primitivas de sincronização construídas pelo próprio usuário sem a utilização de barreiras ou operações atômicas providas pelo conjunto de instruções da arquitetura.
- Dupla Checagem: Utiliza mais de uma checagem em determinada condição, com a finalidade de otimizar a necessidade de sincronização.
- Ambos os Valores Válidos: Nesse caso, independente de leitura ou escrita para valores diferentes, a execução do algoritmo continua correta.
- Escritas Redundantes: Caso onde uma operação de escrita apenas reescreve o valor já contido na posição de memória.
- Manipulação Disjunta de Bits: Nesse caso, apesar de *threads* escreverem valores diferentes em uma mesma posição de memória compartilhada, cada *thread* modifica apenas bits específicos nessa posição.

A correta identificação e classificação de *data races* em um código permite direcionar o modelo de construção de algoritmos paralelos. Dessa forma, a existência *data races* benignos não exige tratamento específico para garantir a corretude do algoritmo, o que possibilita a aplicação de estratégias, principalmente, vetoriais que serão apresentadas no Capítulo 4.

2.3 Sumário

Neste capítulo foram apresentados conceitos básicos sobre Algoritmos Morfológicos e Arquiteturas Paralelas. A Seção 2.1 tratou sobre Algoritmos Morfológicos apresentando notações que serão utilizadas ao longo desta dissertação. Também foram mostrados alguns exemplos de algoritmos morfológicos buscando evidenciar a utilidade e a capacidade da Morfologia Matemática, e foram discutidas as principais estratégias de implementação para algoritmos morfológicos: Algoritmos Paralelos, Algoritmos Sequenciais, Algoritmos Baseados em Filas de Pixels e Algoritmos Híbridos.

Em seguida, a Seção 2.2 argumentou sobre Arquiteturas Paralelas, abordando as classificações propostas por Flynn e Duncan, detalhando as características de processadores vetoriais, uma visão acurada do Intel® Xeon Phi™ e uma categorização de *data races* e condições de corrida.

O próximo capítulo irá apresentar o *Irregular Wavefront Propagation Pattern* que generaliza as implementações atuais para Algoritmos Morfológicos, apresentando casos de uso utilizando esse padrão.

Capítulo 3

Irregular Wavefront Propagation Pattern (IWPP)

Irregular Wavefront Propagation Pattern [60] é um padrão de computação encontrado em diversos algoritmos morfológicos, como descritos anteriormente na Seção 2.1.2. IWPP são estruturas que, dados espaços multidimensionais $D_I \in \mathbb{Z}^n$, e um conjunto de pixels ativos S, propagam os valores de pixels ativos a elementos em *wavefront* de forma irregular, seguindo uma condição de propagação. Esses conjuntos de pixels ativos S são denominados ondas e, durante a execução, se expandem de forma irregular ao longo do espaço D. As principais características dessas ondas são:

- Dinamismo - Não é possível antever as direções de suas expansões;
- Dependência - Cada expansão depende diretamente da anterior e indiretamente da expansão de outras ondas.

O IWPP divide-se em duas fases: Identificação das Frentes de Onda e Propagação Irregular. A Identificação das Frentes de Onda tem por finalidade identificar as frentes de onda iniciais e alimentar a estrutura de dados que permitirá a execução da fase de Propagação Irregular. Já na tarefa de Propagação Irregular, cada elemento ativo constitui uma frente de onda que se expande enquanto a estrutura de dados contiver elementos e a condição de propagação for verdadeira. Essas duas tarefas podem ser observadas no Algoritmo 6.

Nesse algoritmo, a Fase de Inicialização identifica os elementos ativos de D que formarão o subconjunto de elementos ativos S (linhas 1 a 3). A partir desse subconjunto, na Fase de Propagação Irregular, enquanto ele não estiver vazio (linha 5), é extraído um elemento e_i de S (linha 6). Desse elemento é gerado um subconjunto Q com a vizinhança $N_G(e_i)$ de e_i (linha 7) e, para cada elemento e_j de Q, é verificado se condição de propagação é satisfeita para esse elemento (linhas 9 e 10). Quando a condição de propagação for

Algoritmo 6: Irregular Wavefront Propagation Pattern (IWPP)

```
   Entrada: D: Conjunto de elementos de um espaço multidimensional
   Saída: D: Conjunto estável com todas as propagações realizadas
 1 D ← conjunto de elementos de um espaço multidimensional
 2 {Fase de Inicialização}
 3 S ← subconjunto de elementos ativos de D
 4 {Fase de Propagação Irregular}
 5 enquanto S ≠ ∅ faça
 6 |   Extrai e_i de S
 7 |   Q ← N_G(e_i)
 8 |   enquanto Q ≠ ∅ faça
 9 |   |   Extrai e_j de Q
10 |   |   se PropagationCondition(D(e_i),D(e_j)) = verdadeiro então
11 |   |   |   D(e_j) ← max/min(D(e_i), D(e_j))
12 |   |   |   Insere e_j em S
13 |   |   fim
14 |   fim
15 fim
```

satisfeita, $D(e_j)$ recebe a propagação (linha 11), e e_j passa a fazer parte do subconjunto de elementos ativos S que podem propagar suas características aos seus elementos vizinhos. Então, e_j é inserido em S. A Fase de Propagação Irregular continua até que não hajam mais elementos no subconjunto S, indicando a estabilidade de D.

As seções a seguir exemplificam e detalham casos de uso de algoritmos morfológicos utilizando o padrão IWPP: a Reconstrução Morfológica (Seção 3.1), a Transformada de Distância Euclidiana (Seção 3.2) e o Algoritmo *Fill Holes* (Seção 3.3).

3.1 Reconstrução Morfológica

Reconstrução é uma transformação que utiliza operações morfológicas envolvendo duas imagens e um elemento estruturante. As operações morfológicas utilizadas na reconstrução são operações básicas adotadas em um conjunto amplo de algoritmos de processamento. Essas operações são aplicadas a pixels individuais e são processadas baseadas no valor atual do pixel e nos valores dos pixels em sua respectiva vizinhança.

No processo de reconstrução, as imagens são identificadas por marcadora (J), utilizada como ponto de partida para a transformação; e máscara (I), que representa o limite da transformação. O elemento estruturante utilizado define a conectividade do pixel p com sua vizinhança $N_G(p)$. A reconstrução morfológica $\rho(J)$ de uma máscara I, a partir de uma imagem marcadora J, é feita através de dilatações elementares de tamanho 1 em J por G. Uma dilatação elementar de um pixel p corresponde a propagação de p à sua vizinhança G.

O algoritmo básico executa suas dilatações elementares sucessivamente ao longo de toda a imagem J, atualizando cada pixel com uma comparação pixel-a-pixel mínima do resultado de suas dilatações e o pixel correspondente em I, dado por $J(p) \leftarrow (max\{J(q), q \in$

$N_G(p) \cup \{p\}\}) \wedge I(p)$, até que a estabilidade seja alcançada, quando não há mais modificações de valores em qualquer pixel (conforme mostrado na Figura 3.1).

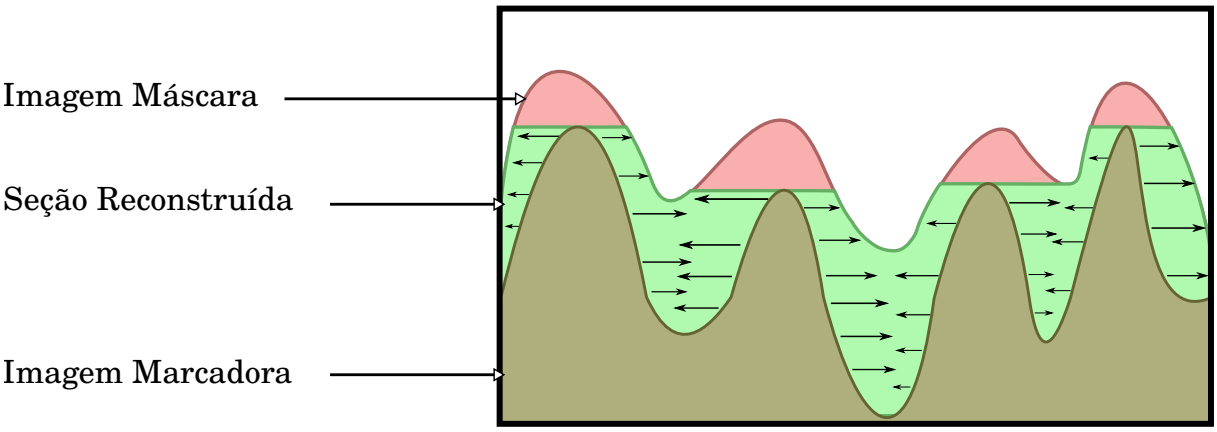

Figura 3.1: Reconstrução Morfológica.

Baseado nessa técnica, o Algoritmo 7 demonstra a reconstrução morfológica utilizando o padrão IWPP para imagens em tons de cinza. Na Fase de Varreduras desse algoritmo (linhas 3 a 14), o pixel é propagado na imagem marcadora alternando varreduras *raster* e *anti-raster*. A varredura *raster* inicia a partir do pixel $(0, 0)$ e desloca-se até o final da imagem $(i-1, j-1)$ (Figura 2.12), enquanto a varredura *anti-raster* realiza o deslocamento no caminho contrário, do final $(i-1, j-1)$ para o início da imagem $(0, 0)$ (Figura 2.13). Em cada varredura, a metade da vizinhança dos pixels vizinhos - superiores a esquerda ou inferiores a direita - são propagados quando a condição de propagação é satisfeita. Na passagem *anti-raster*, além da varredura, é utilizada uma fila de pixels *First In, First Out* (FIFO) que dará continuidade a execução do algoritmo na Fase de Propagação Irregular. A fila é inicializada com pixels que satisfazem a condição de propagação e a computação continua na Fase de Propagação Irregular removendo elementos da fila, varrendo seus elementos vizinhos, identificando aqueles pixels cuja condição de propagação ocorre, e inserindo na fila os modificados pela propagação. Assim, o algoritmo termina quando a fila está vazia, indicando que a estabilidade foi alcançada.

No contexto de Bioinformática, uma das aplicações da Reconstrução Morfológica é destacar setores para possibilitar a identificação e segmentação de células. As Figuras 3.2, 3.3, 3.4 e 3.5, exemplificam essa utilização da Reconstrução Morfológica. A Figura 3.2 representa a imagem escaneada com a região do tecido a ser segmentado. A partir dessa imagem escaneada, é gerada uma versão em tons de cinza para ser utilizada como máscara para a reconstrução (Figura 3.3). O ponto de partida para a identificação de células, a serem segmentadas na região do tecido, utiliza uma versão da Imagem Máscara (Figura 3.3) com sua intensidade reduzida (Figura 3.4). Essa redução de intensidade permite realizar a identificação de picos (ou sementes) que podem ser objetos ou núcleos

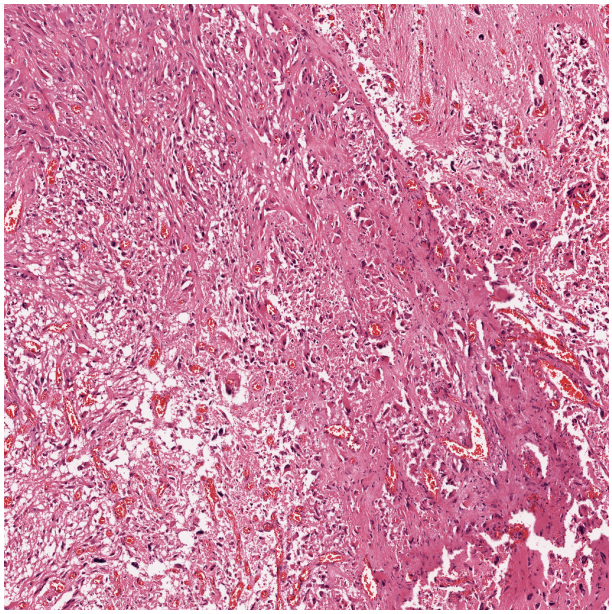

Figura 3.2: Imagem de Tecido Original

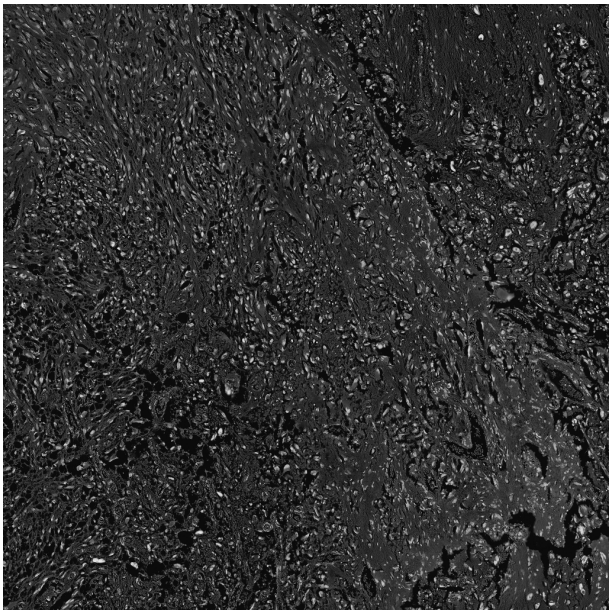

Figura 3.3: Imagem Máscara

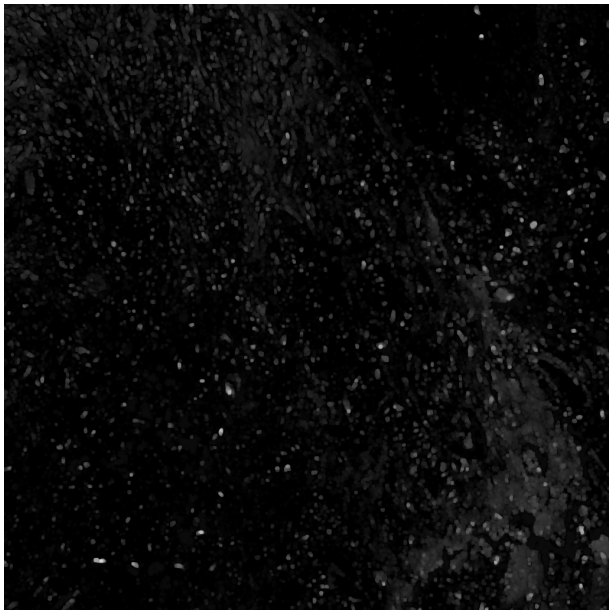

Figura 3.4: Imagem Marcadora

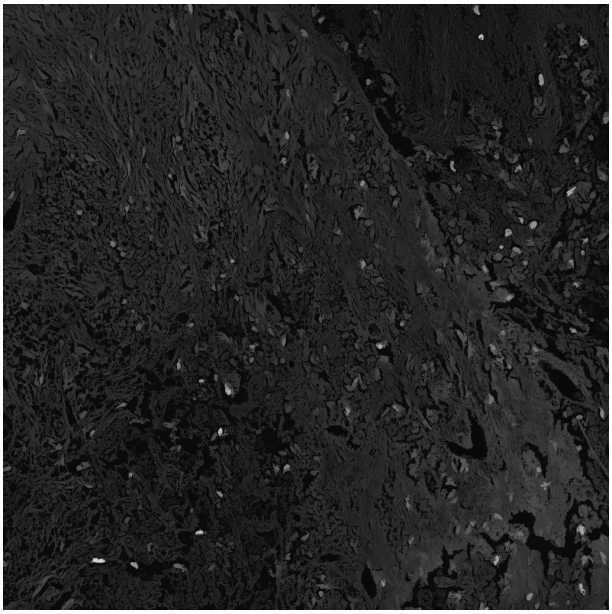

Figura 3.5: Imagem Reconstruída

no tecido. Dessa forma, a Reconstrução Morfológica irá expandir as as características dos picos da Imagem Marcadora até os limites permitidos pela Imagem Máscara gerando, assim, a Imagem Reconstruída (Figura 3.5).

Algoritmo 7: Reconstrução Morfológica - IWPP

```
   Entrada: I, Imagem Máscara
   Entrada: J, Imagem Marcadora
   Saída: J, Imagem Reconstruída
 1 {Fase de Inicialização}
 2 ...
 3 {Fase de Varreduras}
 4 varredura D_I em I e J na ordem raster faça
 5 |   Sendo p o pixel atual
 6 |   J(p) ← (max{J(q), q ∈ N_G^+(p) ∪ {p}}) ∧ I(p)
 7 fim
 8 varredura D_I em I e J na ordem anti-raster faça
 9 |   Sendo p o pixel atual
10 |   J(p) ← (max{J(q), q ∈ N_G^-(p) ∪ {p}}) ∧ I(p)
11 |   se ∃q ∈ N_G^-(p) | J(q) < J(p) e J(q) < I(q) então
12 |   |   fila.insere(p)
13 |   fim
14 fim
15 {Fase de Propagação Irregular}
16 enquanto fila.vazia == falso faça
17 |   p ← fila.retira()
18 |   para todo q ∈ N_G(p) faça
19 |   |   se J(q) < J(p) e I(q) ≠ J(q) então
20 |   |   |   J(q) ← min{J(p), I(q)}
21 |   |   |   fila.insere(q)
22 |   |   fim
23 |   fim
24 fim
```

3.2 Transformada de Distância Euclidiana

A operação Transformada de Distância calcula o mapa de distâncias M de uma imagem binária de entrada I, onde para cada pixel $p \in I$, que faz parte do objeto (*foreground*), seu valor em M é a menor distância a partir de p até o pixel de fundo (*background*) mais próximo da imagem.

A métrica de distância mais usual em processamento de imagens é a distância Euclidiana cuja operação é a Transformada de Distâncias Euclidiana (TDE) (Figuras 3.6 e 3.7). Essa operação é amplamente utilizada em diversos outros algoritmos como diagramas de Voronoi [40], triangulações de Delaunay [38] ou *watershed* [66], por exemplo. A TDE possui alto custo de execução, apesar da simplicidade de seus conceitos. Com isso, diversas implementações já foram apresentadas utilizando várias estratégias para a sua execução, que podem ser classificadas, principalmente, como exatas e não exatas. A implementação de Danielsson em 1980 [11] foi a primeira proposta de algoritmo TDE, e, apesar de não exato, as diferenças podem ser rapidamente detectadas e corrigidas com pós-processamento.

Na implementação da TDE, vista no Algoritmo 8, é utilizada uma fila de pixels IWPP para calcular a distância aproximada de cada vizinhança. Na Fase de Inicialização, todos os pixels p que fazem parte dos objetos na imagem (*foreground*) recebem valor infinito (linha 8). A partir desse ponto, são inseridos na fila pixels p que pertençam ao fundo da

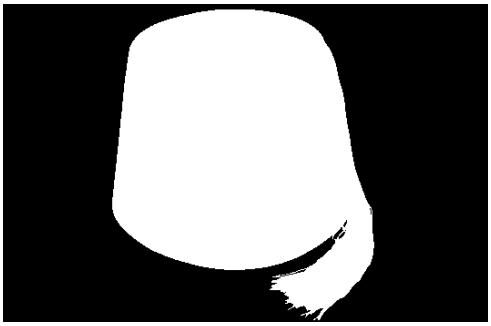

Figura 3.6: Imagem Original.

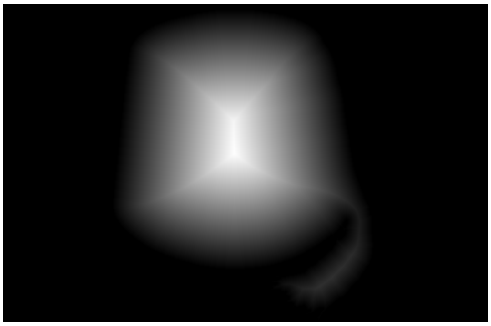

Figura 3.7: TDE.

Algoritmo 8: Transformada de Distância Euclidiana - IWPP

```
   Entrada: I, Imagem Máscara
   Saída: M, Mapa de distâncias
 1 FG: Foreground {Objeto na imagem}
 2 BG: Background {Fundo na imagem}
 3 {Fase de Inicialização}
 4 para todo p ∈ D_I faça
 5 |   se p == BG então
 6 |   |   VR(p) ← p
 7 |   senão
 8 |   |   VR(p) ← ∞
 9 |   fim
10 |   se I(p) == BG e (∃q ∈ N_G(p) | I(q) == FG) então
11 |   |   fila.insere(p)
12 |   fim
13 fim
14 {Fase de Propagação Irregular}
15 enquanto fila.vazia() == falso faça
16 |   p ← fila.retira(p)
17 |   para todo q ∈ N_G(p) faça
18 |   |   se DIST(q, VR(p)) < DIST(q, VR(q)) então
19 |   |   |   VR(q) = VR(p)
20 |   |   |   fila.insere(q)
21 |   |   fim
22 |   fim
23 fim
24 para todo p ∈ D_I faça
25 |   M(p) = DIST(p, VR(p))
26 fim
```

imagem e possuam algum vizinho q que faça parte dos objetos na imagem I (linhas 10 e 11). Esses pixels formarão a *wavefront* inicial. Na fase *wavefront propagation*, um pixel é retirado da fila (linha 16) e, para cada vizinho q de $N_g(p)$ em que a distância dele para o fundo seja maior que a distância de p para o fundo, a propagação irá ocorrer (linhas 17 a 19). Com a propagação, q será incluído na fila e as iterações continuam até que a fila esvazie. Alcançada a estabilidade (fila vazia), o mapa de distâncias M é, a partir do diagrama de Voronoi, calculado pelo algoritmo durante as propagações (linhas 18 e 25).

3.3 *Fill Holes*

Fill Holes ou *Imfill* é uma técnica utilizada para realizar o preenchimento de buracos ou áreas em uma imagem binária ou em tons de cinza [47]. Um buraco pode ser definido como uma região de fundo da imagem rodeada por uma borda completamente contornada pelo objeto da imagem (ou *foreground*) [17].

Em uma imagem binária são realizadas mudanças nos valores dos elementos do fundo da imagem (*background*) conectados aos valores do objeto com uma operação de preenchimento, ou inundação (*flood-fill*), até que os limites do objeto sejam atingidos. Já imagens em tons de cinza é utilizada a mesma ideia de inundação que modifica a intensidade de áreas escuras contornadas por áreas mais claras pelas próprias intensidades do contorno.

Uma das estratégias de funcionamento do algoritmo utiliza a Reconstrução Morfológica como base, onde a entrada é a imagem a ser preenchida. A máscara a ser utilizada é uma inversão (ou complemento) da própria imagem que irá limitar o resultado dentro da região de interesse a ser preenchida.

Esse preenchimento é utilizado no processamento de imagens para homogeneizar imagens buscando evitar falsa segmentação no uso de outros algoritmos, remover diferentes intensidades de luminância, ou facilitar o processamento de outros algoritmos morfológicos de qualquer natureza. A Figura 3.8 exemplifica com um objeto com diferentes tonalidades de sombra cuja execução do algoritmo gera uma imagem mais homogênea do mesmo.

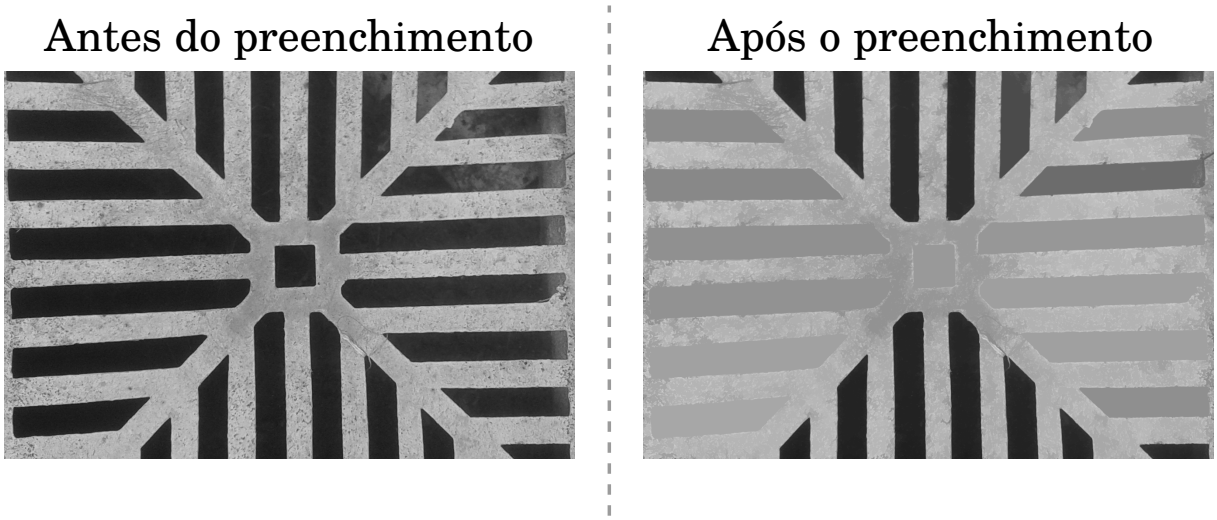

Figura 3.8: Função *Imfill* Aplicada em Tons de Cinza.

3.4 Trabalhos Relacionados

As seções subsequentes destacam aspectos e trabalhos relevantes relacionados ao IWPP (Seção 3.4.1), da Reconstrução Morfológica (Seção 3.4.2) e da Transformada de Distância Euclidiana (Seção 3.4.3).

3.4.1 IWPP

Breadth-First Search Eficiente

Algoritmos IWPP podem ser vistos como algoritmos de varredura em grafos com suporte a múltiplos pontos de origem. Alguns trabalhos envolvendo implementações eficientes do *Breadth-First Search* são apresentados em [19] e [49].

O trabalho apresentado em [19] tem como objetivo desenvolver técnicas para evitar o efeito de desbalanceamento de carga devido a existência de vértices com número irregular de arestas. Tais técnicas não são aplicáveis ao contexto do IWPP, uma vez que o número de vértices é constante por ser baseado no elemento estruturante.

Já em [49] são apresentadas, de maneira isolada, diversas técnicas para acelerar o BFS utilizando MIC. São utilizadas operações de leitura de arestas aproveitando a largura do vetor com instruções SIMD. No paralelismo em nível de *threads* são utilizados mapas de bits em conjunto com operações atômicas na exploração da vizinhança dos vértices para manipular uma fila compartilhada. O uso dessa estratégia busca garantir a não expansão de vértices iguais por *threads* diferentes, permitindo um maior controle, e mostrando ser satisfatório a sua implementação.

Esse trabalho também trás abordagens análogas e adaptáveis para o IWPP envolvendo tanto a execução no modo "nativo", por meio do uso de relaxamento de dependência durante a expansão de vértices, quanto na execução *offload*, por meio da computação em tarefas, cujas técnicas mostraram-se factíveis ao contexto deste trabalho.

IWPP em *Graphics Processing Units*

Os trabalhos [50, 51, 59, 60] apresentaram uma implementação paralela eficiente para GPUs. A implementação da fase de Propagação Irregular pode ser acompanhada no Algoritmo 9 e a estratégia de implementação completa pode ser visualizada na Figura 3.9, na qual utiliza-se a seguinte estratégia:

1. Durante a inicialização, identifica os elementos ativos e separa-os em um conjunto S de sementes;
2. Divide o conjunto S em Z partições $(P_1, \ldots, P_Z)$, onde cada partição é mapeada para uma *thread* t na GPU;

3. Cria-se uma instância independente da fila hierárquica para a *thread* t_i que é inicializada com os elementos da partição P_i;

4. Utilizam-se filas distintas de entrada e saída de pixels, como forma de melhorar os tempos de leitura e sincronização.

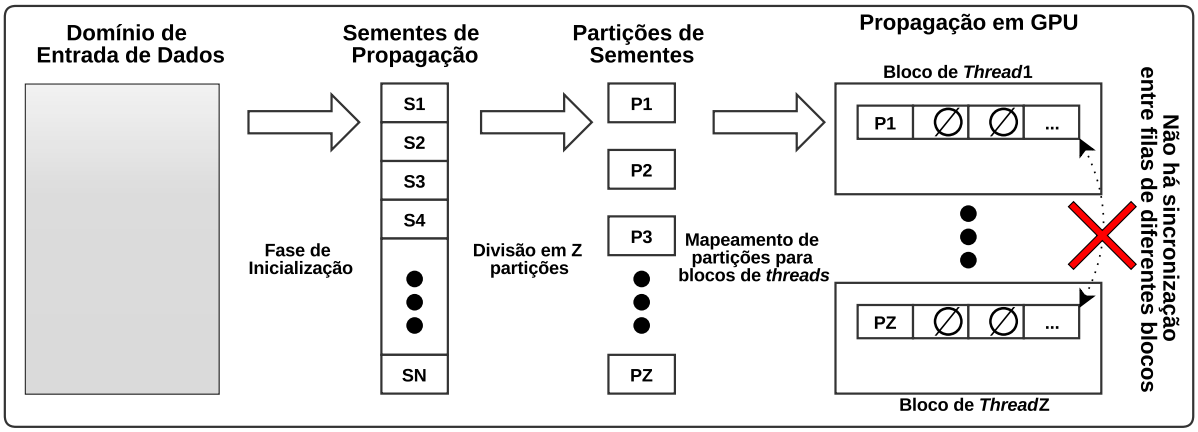

Figura 3.9: Visão Global da IWPP em GPU. Adaptado de [60].

Como cada *thread* possui sua própria fila de elementos a serem propagados de forma independente, necessita-se que a atualização do pixel vizinho, quando ocorrer, seja feita de forma atômica para evitar a ocorrência de *data races*, quando *threads* tentam atualizar o mesmo pixel com valores diferentes, o que poderia causar resultados não determinísticos ao algoritmo.

Como a ordem de processamento não pode ser determinada, operações de propagação necessitam ser comutativas e atômicas. Para garantir a atomicidade das atualizações nos pixels vizinhos, é então necessário custo extra de sincronização, dado pela operação *atomicCAS* (*atomic compare and swap*) disponível nas placas de vídeo CUDA [69] (linha 7 do Algoritmo 9).

3.4.2 Reconstrução Morfológica

Recentemente, a Reconstrução Morfológica vem sendo amplamente empregada nos mais diversos tipos de aplicações. Das principais publicações nesse campo, pode-se citar o trabalho [64] que definiu formalmente a Reconstrução Morfológica aplicada a imagens em tons de cinza. Ele apresentou diversas aplicações para a reconstrução, outorgando a utilização da reconstrução baseada em filas de pixels *First In, First Out* (FIFO).

Outra publicação de relevância foi a [28] que utilizou uma arquitetura em *cluster* com memória distribuída. A ideia dessa arquitetura é explorar paralelismo entre pixels

Algoritmo 9: Propagação Irregular em GPU

```
1  enquanto fila[idBloco] != ∅ faça
2      enquanto ( p = fila[idBloco].desenfileira(...)) != ∅ faça
3          para todo q ∈ N_G(p) faça
4              repita
5                  valorCorrenteQ = I(q)
6                  se CondiçãoDePropagação(I(p),valorCorrenteQ) então
7                      valorAntigo = atomicCAS(&I(q),valorCorrenteQ,op(I(p)))
8                      se valorAntigo == valorCorrenteQ então
9                          fila[idBloco].enfileira(q)
10                         terminaLaco
11                     fim
12                     senão
13                         terminaLaco
14                     fim
15                 fim
16             até Verdadeiro;
17         fim
18     fim
19     fila[idBloco].troca_entrada_saida
20 fim
```

próximos durante a reconstrução, através de estruturas de dados hierárquicas em uma abordagem assíncrona em imagens de tamanho pequeno.

Em [22] e [21], apesar do foco em arquiteturas distintas, [21] em *Field-Programmable Gate Arrays* (FPGAs) e [22] em GPUs, ambos utilizaram algoritmos paralelos cujos ganhos com paralelização não foram significativas quando comparadas com as versões mais eficientes das implementações sequenciais. Isso aconteceu devido a ambos os trabalhos terem paralelizado uma versão menos eficiente da reconstrução morfológica.

Em [1, 55–57] foram realizadas análises mensurando o uso de diversos algoritmos morfológicos em CPUs, GPUs e MICs. Nesses estudos foram constatados o bom desempenho da arquitetura *Many Integrated Core* em acessos regulares de dados, porém com o uso apenas de autovetorização por meio de anotações de diretivas no código (`#pragma simd`, por exemplo).

3.4.3 Transformada de Distância Euclidiana

A Transformada de Distância é um instrumento de suma importância no processamento de imagens, pois é utilizada em um extensivo número de aplicações. A Transformada de Distância Euclidiana possui diversas abordagens devido ao seu alto custo computacional e a possibilidade de busca pela eficiência através de soluções não determinísticas.

Uma das primeiras implementações apresentadas, equivocadamente tratada como exata, foi em [11]. Essa implementação gera um mapa com os valores absolutos das coordenadas com o fundo da imagem mais próximo, ao invés de um mapa de distâncias propriamente

dito. Essa diferença difere do caso exato devido a não conexidade[1] do diagrama de Voronoi no caso discreto [71]. Apesar dessa inexatidão, destacou-se por demonstrar a eficiência de algoritmos sequenciais.

Posteriormente, diversas abordagens foram apresentadas, como em [10] que propõe uma Transformada de Distância exata utilizando uma lista ordenada. Essa implementação realiza uma aproximação rápida do resultado e utiliza vizinhanças maiores para executar possíveis correções.

Outro algoritmo exato utiliza o método de Meijster [31] que destaca-se por ser ligeiramente mais eficiente que outras abordagens, e por possuir, também, uma implementação simples.

Implementações mais recentes em [60] mesclam o uso de GPUs e de CPUs, por meio de uma estratégia utilizando paralelizações em nível de tarefas, para a execução mútua nesses dispositivos. Também foram utilizadas estruturas de dados multiníveis (filas), buscando otimizar o uso de memórias mais rápidas. Esse estudo mostrou significativas performances, principalmente, para GPUs, alcançando *speedups* notáveis em sua execução. O uso extensivo desses algoritmos também incluem o uso em Diagramas de Voronoi [70], Esqueletos Morfológicos [46] e *Watersheds* [66].

3.5 Sumário

Neste capítulo foi apresentado o padrão de computação IWPP em algoritmos morfológicos, o qual pode ser observado em diversas soluções na literatura. Esses algoritmos são divididos em duas fases: Identificação das Frentes de Onda e Propagação Irregular.

Em seguida, foram apresentados casos de uso dos algoritmos de Reconstrução Morfológica, Transformada de Distância Euclidiana e *Fill Holes* buscando evidenciar o seu funcionamento e as características de IWPP.

Por fim, foram apresentados trabalhos relacionados elencando aspectos e resultados a serem considerados no desenvolvimento da solução e, também, utilizados como parâmetro de comparação na apresentação dos resultados.

O próximo capítulo abordará uma implementação eficiente para o IWPP na arquitetura *Many Integrated Core*, com foco no Intel® Xeon Phi™ [39], explorando as principais características dessa arquitetura.

[1] Espaço topológico que não pode ser representado como a união de dois ou mais conjuntos abertos disjuntos e não-vazios.

Capítulo 4

IWPP Paralelo Eficiente na Arquitetura *Many Integrated Core*

Este trabalho tem como objetivo a apresentação de uma abordagem eficiente para algoritmos da classe *Irregular Wavefront Propagation Pattern* na arquitetura MIC. Essa nova abordagem visa a obtenção de um algoritmo IWPP vetorizável e paralelizável utilizando instruções SIMD. A consequência de um algoritmo com as duas características citadas é uma implementação sem o uso de operações atômicas, que são um elemento limitador na eficiência de algoritmos no Intel® Xeon Phi™, pois o mesmo não suporta instruções SIMD atômicas [39].

4.1 Algoritmo Proposto

A partir do algoritmo IWPP (mostrado no Algoritmo 6), que não é vetorizável, a solução apresentada nesta seção propõe um algoritmo com *data races* benignas [35] no momento da propagação, cuja ocorrência não altera o resultado final. Tal abordagem se dá nos seguintes passos:

1. A fase de inicialização é similar a apresentada no algoritmo da Reconstrução Morfológica (Algoritmo 7), que na varredura *anti-raster* os elementos são inseridos em uma fila chamada *ondaAtual*;
2. Na fase de identificação de elementos recebedores de propagação, ao varrer os pixels p contidos em *ondaAtual*, o algoritmo irá inserir aqueles pixels $q \in N_G(p)$, cuja condição de propagação é verdadeira, em *proximaOnda* (linhas 6 a 11 do Algoritmo 10);
3. Na fase de propagação será utilizada a fila *proximaOnda*, que está populada com elementos que receberão alguma atualização, onde o pixel p irá receber a respectiva atualização de algum pixel $q \in N_G(p)$ (linhas 14 a 18 do Algoritmo 10);

4. Os elementos que se encontram em *proximaOnda* constituem o conjunto de elementos ativos das frentes de onda, então as filas *proximaOnda* e *ondaAtual* são trocadas (linhas 19 e 20 do Algoritmo 10) e o algoritmo continua sendo executado até que a mesma se encontre vazia.

Algoritmo 10: Novo IWPP proposto não-vetorizado

```
   Entrada: D: Conjunto de elementos em um espaço multidimensional
   Saída: D: Conjunto estável com todas as propagações realizadas
 1 {Inicialização}
 2 ...
 3 {Fase de Propagação Irregular}
 4 enquanto ondaAtual.vazia() = falso faça
 5 |   {Identificação de Elementos Recebedores de Propagação}
 6 |   para todo p ∈ ondaAtual faça
 7 |   |   para todo q ∈ N_G(p) faça
 8 |   |   |   se condioDePropagao(D(q), D(p)) = verdadeiro então
 9 |   |   |   |   proximaOnda.insere(q)
10 |   |   |   fim
11 |   |   fim
12 |   fim
13 |   {Propagação}
14 |   para todo p ∈ proximaOnda faça
15 |   |   para todo q ∈ N_G(p) faça
16 |   |   |   D(p) ← Max/Min(D(q), D(p))
17 |   |   fim
18 |   fim
19 |   ondaAtual ← proximaOnda
20 |   proximaOnda ← ∅
21 fim
```

As modificações propostas nesse algoritmo objetivam que elementos alterem unicamente seus próprios valores na Fase de Propagação Irregular. Na implementação apresentada no Algoritmo 7, um elemento pode modificar a sua vizinhança, e a versão paralela desse algoritmo exige o uso de operações atômicas durante a verificação da condição de propagação e a propagação, pois *threads* com elementos de posições diferentes e valores diferentes podem tentar modificar uma mesma posição (vizinho) ao mesmo tempo, ocasionando uma condição de corrida (ou *data race* prejudicial). Um exemplo dessa possibilidade de modificação de uma determinada posição por elementos diferentes, inseridos na estrutura de dados de *threads* diferentes pode ser visualizado na Figura 4.1. Já a abordagem proposta neste trabalho visa eliminar esse caso de escrita de valores diferentes em uma mesma posição, fazendo com que cada elemento modifique apenas o seu próprio conteúdo, como pode ser visualizado no exemplo da Figura 4.2.

4.2 Implementação Vetorial do Algoritmo Proposto

Utilizando a abordagem apresentada na seção anterior, uma variante vetorial pode ser implementada (conforme apresentado no Algoritmo 11). As fases desse algoritmo,

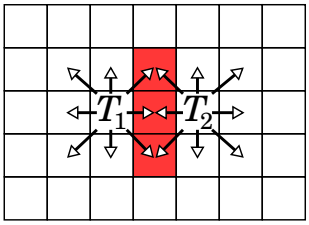

Figura 4.1: Exemplo de Propagação Paralela Oriunda de Elementos Diferentes.

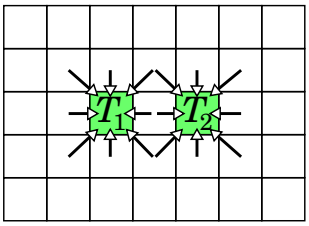

Figura 4.2: Exemplo de Propagação Paralela Utilizando a Abordagem Proposta.

identificação de elementos recebedores de propagação e propagação, são detalhadas nas próximas seções.

Algoritmo 11: IWPP vetorizado

```
   Entrada: D: Conjunto de elementos em um espaço multidimensional
   Saída: D: Conjunto estável com todas as propagações realizadas
 1 vet_shift ← Constante de distâncias de endereços da vizinhança
 2 {Inicialização}
 3 ...
 4 {Escaneamento}
 5 ...
 6 {Fase wavefront propagation}
 7 enquanto ondaAtual.vazia() == falso faça
 8 |   {Identificação de Elementos Recebedores de Propagação}
 9 |   para todo p ∈ ondaAtual faça
10 |   |   vet_p ← Extração de elementos ativos
11 |   |   vet_enderecos ← vetAdd(vet_p, vet_shift)
12 |   |   vet_vizinhos ← gather(d, vet_enderecos)
13 |   |   mascara_condicao ← vetorCondicaoDePropagacao(vet_p, vet_vizinhos)
14 |   |   vet_prefixsum ← prefixSum(mascara_condicao)
15 |   |   ondaproxima.insere(vet_enderecos, mascara_condicao, vet_prefixSum)
16 |   fim
17 |   {Propagação}
18 |   para todo q ∈ proximaOnda faça
19 |   |   vet_q ← Extração de elementos
20 |   |   vet_enderecos ← vetAdd(vet_q, vet_shift)
21 |   |   vet_vizinhos ← gather(d, vet_enderecos)
22 |   |   d(q) ← (Max/Min)Reducao(vet_q, vet_vizinhos)
23 |   fim
24 |   ondaAtual ← proximaOnda
25 |   proximaOnda ← ∅
26 fim
```

4.2.1 Identificação de Elementos Recebedores de Propagação

Esta fase é dividida em três passos principais, os quais são: Leitura de Pixels Vizinhos (linhas 10 a 12), Contagem de Elementos e *Prefix sum* (linha 13), e Inserção Vetorial na Fila (linhas 14 e 15).

Passo 1: Leitura de Pixels Vizinhos

Para um pixel p retirado da fila de entrada, é realizado a verificação da condição de propagação com cada um dos pixels de sua vizinhança. A leitura dos pixels vizinhos q

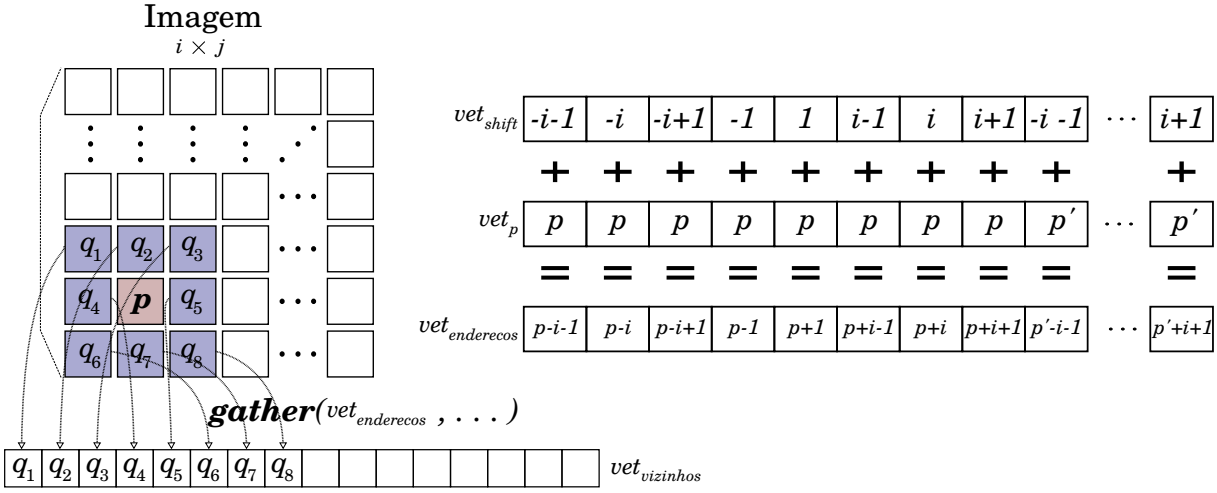

Figura 4.3: Cálculo de Endereços e Operação *Gather* de um Pixel p.

vetorial é feita pelo seguinte raciocínio:

- Dado o endereço na imagem da posição do pixel p, retirado da fila de entrada, as distâncias entre os pixels $q \in N_G(p)$ em relação a p são constantes. Com isso, é possível calcular a posição da vizinhança de p realizando a soma de seu endereço (vet_p) com um vetor[1] vet_{shift} de deslocamentos referentes à distância de q e seus vizinhos (linha 11). O resultado nos dá um vetor $vet_{enderecos}$ com os endereços dos pixels vizinhos a serem lidos.
- De posse do vetor $vet_{enderecos}$ contendo os endereços a serem buscados na imagem, o Intel® Xeon Phi™ dispõe de uma instrução de leitura chamada *gather* que carrega até 16 (dezesseis) inteiros de posições irregulares, como pode ser observado na Figura 4.3. Dessa forma, é possível carregar eficientemente os vizinhos de p da imagem máscara I e da imagem marcadora J utilizando essa instrução.

A partir dos pixels carregados para o vetor, são feitas as respectivas verificações acerca da condição de propagação. Cabe ressaltar nesta seção que operações de comparação vetoriais geram máscaras binárias com valor 1 (um) para posições onde a condição de propagação é verdadeira (Figura 4.4), que são utilizadas na contagem de elementos no passo de inserção na fila.

Passo 2: Contagem de Elementos e *Prefix Sum*

Após descobertos os elementos que deverão ser inseridos na fila, por meio da verificação da condição de propagação e a máscara de 16 bits gerada como resposta, necessita-se inserir na fila somente os elementos que possuem o valor 1 na sua posição da máscara de

[1]Vetor indica um registrador vetorial de 512 bits, para fins de simplificação.

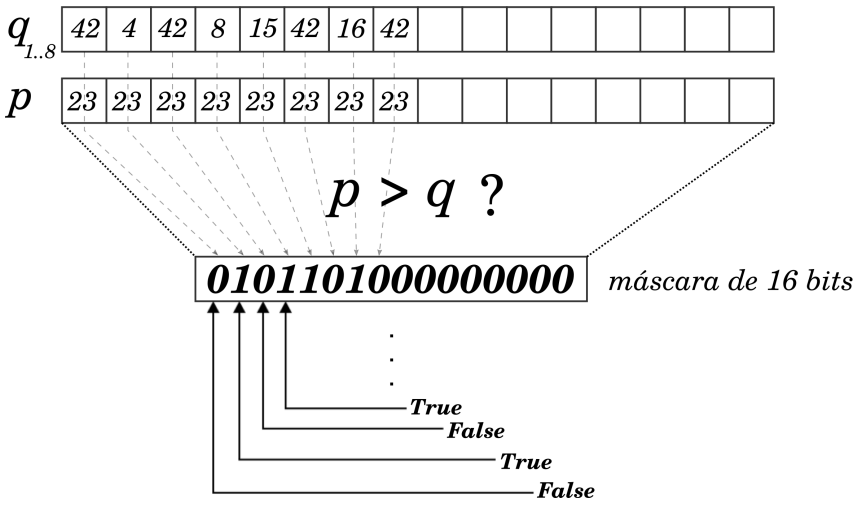

Figura 4.4: Verificação da Condição de Propagação Vetorial.

resposta. Esta etapa é executada em dois passos: a contagem dos elementos e o cálculo do vetor *prefix sum*.

Para a contagem de elementos, utiliza-se uma instrução *popcnt* [39] em que, dado um valor inteiro como entrada (a máscara de resposta da verificação da condição de propagação), ela retorna a quantidade de bits cujo valor é 1 em sua representação binária.

Já o *Prefix sum* é a soma cumulativa dos prefixos $\{y_0, y_1, y_2, \dots\}$ de uma determinada entrada $\{x_0, x_1, x_2, \dots\}$ [3]. Por meio desse vetor de prefixos e a máscara de respostas gerada pela verificação da condição de propagação é possível identificar e inserir esses elementos na fila.

A implementação utiliza um vetor de tamanho fixo (512 bits), que é suficiente para operar cada um dos bits da máscara recebida. A estratégia é executada nos seguintes passos:

1. Dado uma máscara de 16 bits, cada bit dessa máscara é distribuído para cada uma das 16 posições do registrador vetorial vet_{r1} de 512 bits (Figura 4.5).

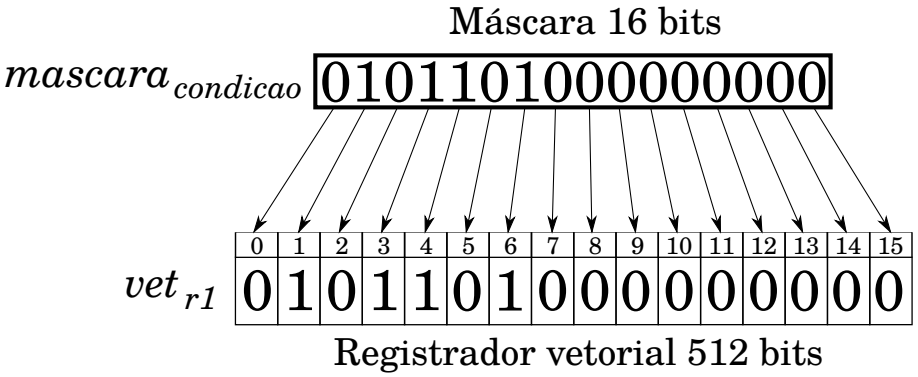

Figura 4.5: Exemplo de Distribuição de Máscara em Registrador Vetorial.

2. O registrador vet_{r2} recebe uma permutação dos valores de vet_{r1}, deslocando todos os elementos em uma posição à direita (Figura 4.6). É então realizada a soma de vet_{r1} e vet_{r2} (Figura 4.7). O resultado dessa soma elemento-a-elemento é salvo no próprio vet_{r1}.

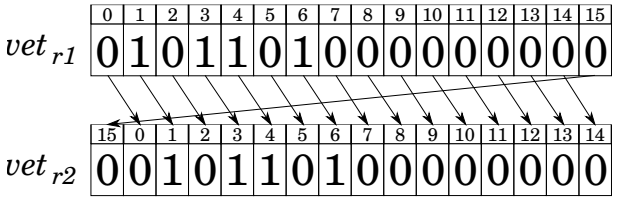

Figura 4.6: Permutação 01.

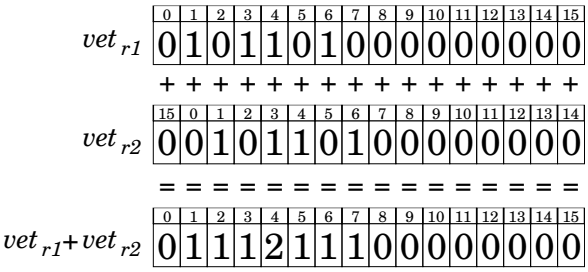

Figura 4.7: Soma Vetorial 01.

3. Para completar o cálculo do *prefix sum* nas oito primeiras posições do vetor, são aplicadas mais duas vezes o passo anterior, como única ressalva de modificação das permutações dos valores. Em cada passo serão permutadas duas (Figuras 4.8 e 4.9) e quatro posições (Figuras 4.10 e 4.11) à direita.

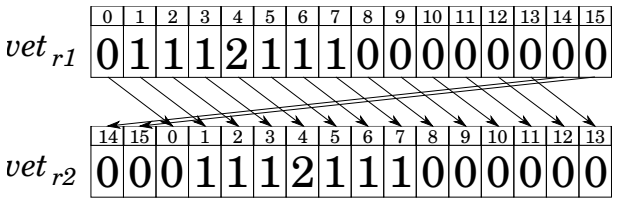

Figura 4.8: Permutação 02.

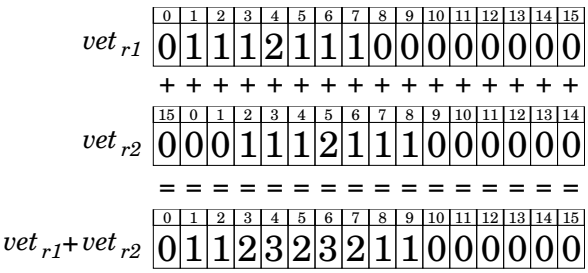

Figura 4.9: Soma Vetorial 02.

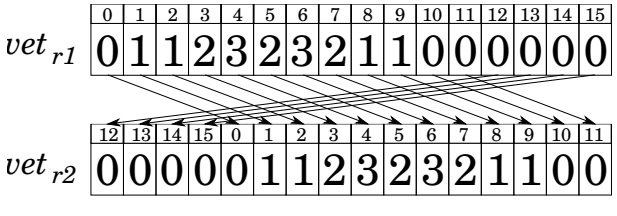

Figura 4.10: Permutação 03.

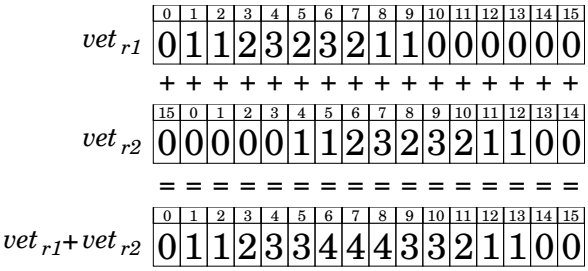

Figura 4.11: Soma Vetorial 03.

4. Os valores contidos na primeira metade do registrador vet_{r1} são o *prefix sum*, referente a máscara de entrada, e seu conteúdo é retornado como resultado, conforme apresentado na Figura 4.12.

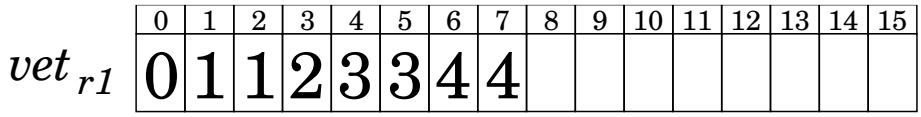

Figura 4.12: Registrador Retornado como Resposta.

Há, na literatura [18], implementações do *prefix sum* que exploram eficientemente o processamento dessa operação para grandes entradas de dados. Porém, como o tamanho da entrada é um valor fixo (512 bits), essas versões *work-efficient*, que foram concebidas para grandes entradas de dados, não melhoram o desempenho nesse caso. Dessa forma, é utilizada uma adaptação da versão paralela [18] para funcionar de forma vetorial e operando com os pixels vizinhos G de um pixel.

Passo 3: Inserção Vetorial na Fila

O último passo da Identificação de Elementos Recebedores de Propagação integra na função *scatter* a utilização do vetor de endereços calculados, a máscara de valores a serem inseridos na fila de saída e o vetor contendo o *prefix sum* das posições a serem inseridas na fila (linha 15 do Algoritmo 11). Para cada posição da máscara cujo valor é 1, o elemento correspondente a mesma posição no vetor $vet_{enderecos}$ é inserido ao final da fila, na posição apontada pelo vetor $vet_{prefixSum}$, como pode ser visualizado na Figura 4.13.

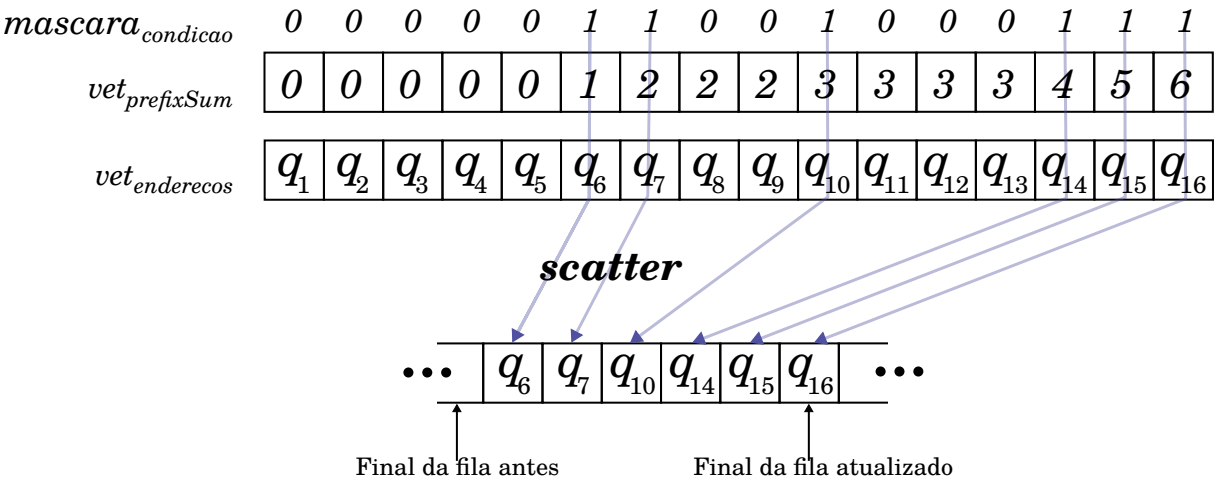

Figura 4.13: *Scatter* e Atualização da Quantidade de Elementos.

Após a inserção, a variável responsável por controlar o final da fila é, então, acrescida da quantidade de elementos adicionados.

4.2.2 Propagação

Durante a fase de Propagação, o pixel q verifica qual pixel $r \in N_G(q)$ ele irá receber a atualização. Nessa estratégia, a função verifica pela atualização de dois pixels simultaneamente, como forma de aproveitar a largura do vetor disponível. Sua execução ocorre nos seguintes passos:

- São consumidos da fila de saída dois pixels p_1 e p_2;
- Utilizando o mesmo método apresentado na Seção 4.2.1, os vizinhos r_{pn}, onde n varia de $\{1..8\}$ para cada um dos elementos de uma determinada vizinhança, são carregados para um registrador vetorial (*gather*), de forma que a vizinhança de cada um dos pixels ocupe metade do vetor com a seguinte organização:

$$[r_{11} \quad r_{12} \quad r_{13} \quad r_{14} \quad r_{15} \quad r_{16} \quad r_{17} \quad r_{18} \quad r_{21} \quad r_{22} \quad r_{23} \quad r_{24} \quad r_{25} \quad r_{26} \quad r_{27} \quad r_{28}]$$

- São geradas máscaras para cada uma das vizinhanças q de um pixel p, para os pixels q cuja condição de propagação é verdadeira;
- A partir dessas máscaras, é calculado o valor de atualização a partir da vizinhança, sendo ele o pixel r que irá propagar a q. As atualizações são feitas individualmente para cada um dos pixels;
- O procedimento continua até que toda a fila de saída seja consumida.

Durante a execução da Propagação, existe a possibilidade de um mesmo elemento q ser inserido diversas vezes em uma mesma iteração da fila *proximaOnda*. Quando isso acontece, é preciso avaliar um cenário específico no qual o mesmo pixel é processado concorrentemente, ou seja p_1 é igual a p_2, ocasionando um *data race* no momento da atualização em $D(q)$. Contudo, para o processamento desses dois elementos - que são a réplica de um único elemento - ocorrer, as vizinhanças desses dois elementos serão lidas por uma única instrução *gather*, fazendo com que as duas vizinhanças possuam sempre os mesmos elementos lidos. Dessa forma, este cenário gera um *data race* benigno e não afeta o resultado da aplicação [34].

4.3 Implementação Paralela do Algoritmo Proposto

A partir da versão vetorizada (apresentado no Algoritmo 11), a estratégia de paralelização se aplica diretamente no processamento independente dos elementos ativos que serão inseridos na fila, após as varreduras *raster* e *anti-raster*.

A última das varreduras realiza a distribuição das iterações, a partir das linhas da imagem, entre as *threads* disponíveis. Cada uma das *threads* irá percorrer um determinado segmento da imagem, identificando elementos ativos que formarão as ondas de propagação de cada *thread*, como pode ser visto na Figura 4.14.

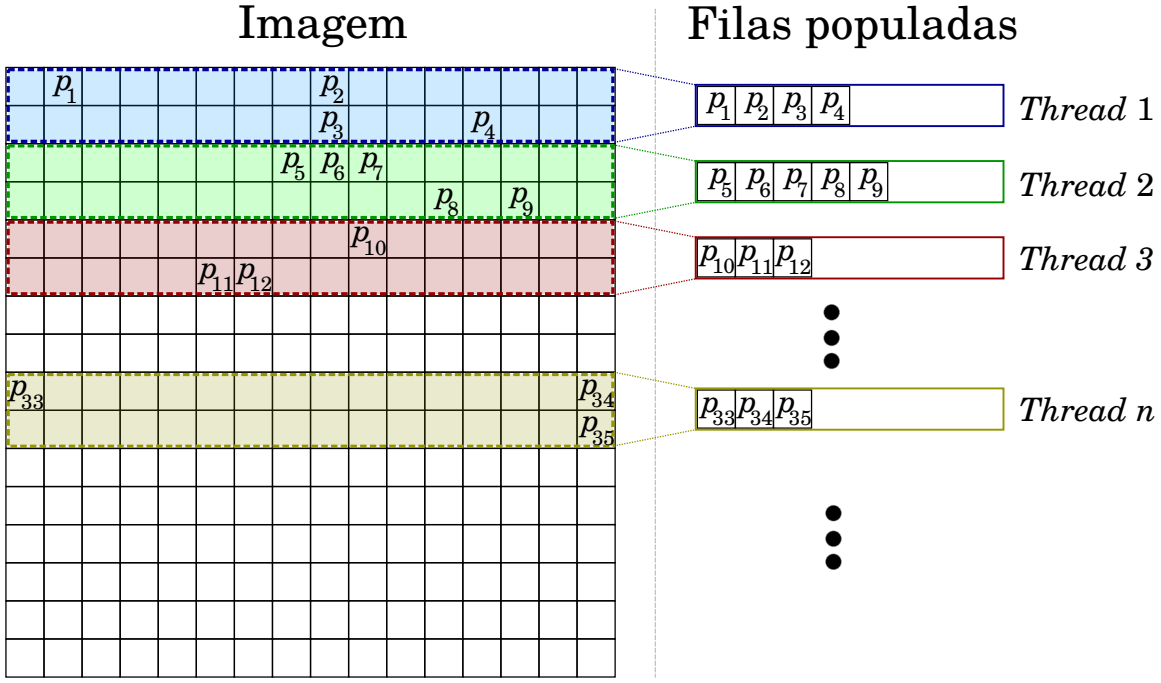

Figura 4.14: Divisão de Elementos Ativos entre *Threads*.

Assim, nessa modificação, cada *thread* passa a possuir suas próprias filas (ondaAtual e ondaProxima) a serem utilizadas na fase de Propagação com um conjunto específico de frentes de onda (sementes) e, a partir deles, serão realizadas as propagações de forma independente entre as *threads*, utilizando as estratégias já descritas na Seção 4.2.

Apesar das ondas de propagação iniciais de cada *thread* serem independentes e iniciarem em posições distintas, durante o processamento (Fase de Propagação Irregular) as frentes de onda de diferentes *threads* podem cruzar umas com as outras. Como consequência, é possível que diferentes *threads* insiram um mesmo elemento q em seus conjuntos distintos de elementos ativos. Neste caso, um *data race* pode ocorrer na atualização de $D(q)$ por múltiplas *threads*.

A consequência do caso descrito no parágrafo anterior é exemplificado na Figura 4.15 para a Reconstrução Morfológica. Neste exemplo, a *Thread* 01 possui o elemento e_{x-1} em sua fila *ondaAtual* na iteração i, enquanto a *Thread* 02 possui os elementos e_{x+2} e e_{x+1}. É notável que na iteração i ambas as *threads* possuem elementos que satisfazem a condição de propagação e irão inserir o elemento e_x e suas respectivas filas *proximaOnda*, na Fase de Identificação. Dada essa possibilidade, durante a Fase de Propagação, a leitura da vizinhança de e_x pode desencadear dois cenários ($C1$ e $C2$) distintos.

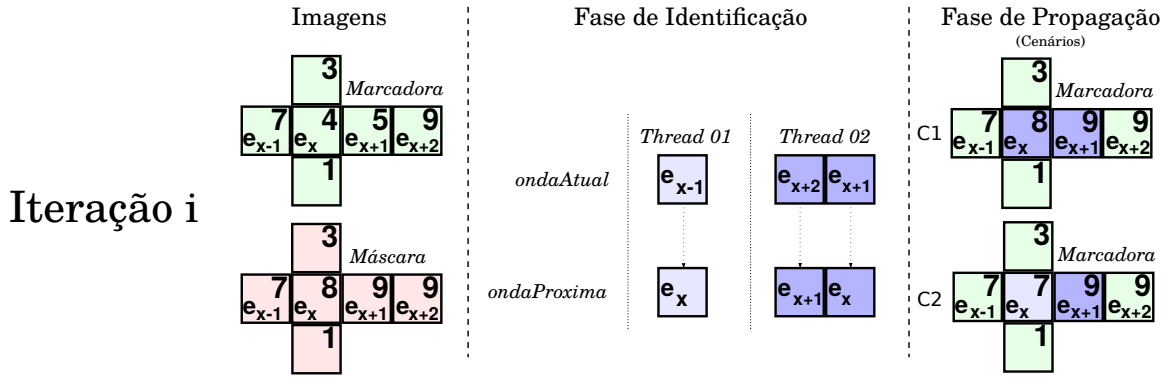

Figura 4.15: Exemplo de Identificação de Elementos Iguais por *Threads* Diferentes.

No primeiro cenário ($C1$), a *thread* 01 verifica a vizinhança de e_x somente após a *thread* 02 escrever a propagação do elemento e_{x+1} e, nesse caso, a estabilidade será alcançada já na iteração i onde os valores dos elementos serão finais (Figura 4.16).

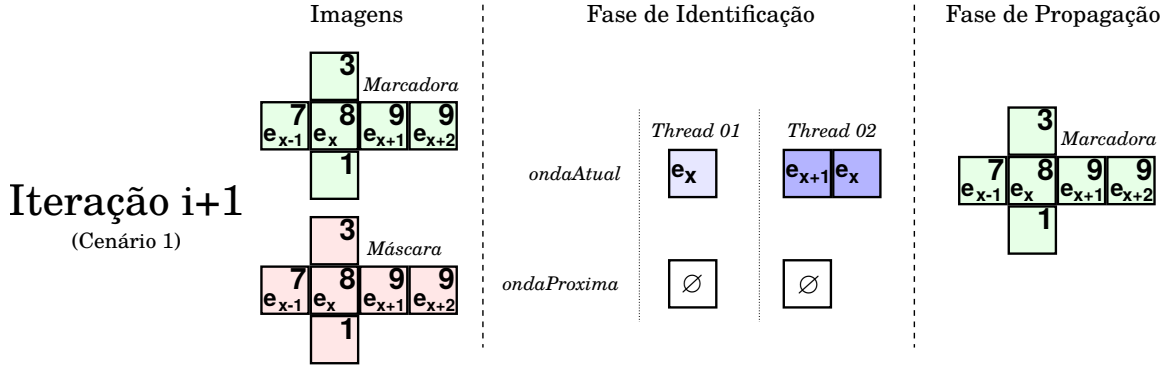

Figura 4.16: Cenário 1: Estabilidade Alcançada na Iteração i.

No segundo cenário ($C2$), a execução ocorre na seguinte ordem:

- A *thread* 01 realiza a leitura da vizinhança de e_x;
- A *thread* 02 processa leitura da vizinhança e propagação dos elementos e_{x+1} e e_x;
- A *thread* 01 realiza a propagação de e_x.

Neste cenário, o maior valor da vizinhança lido pela *thread* 01 (7) é menor do que o maior valor da vizinhança lido pela *thread* 02 (9)[2]. Dessa forma, mesmo possuindo um valor maior após a propagação executada pela *thread* 02, e_x irá terminar a iteração com um valor menor oriundo da leitura da *thread* 01.

Apesar dos cenários diferenciados, como na iteração i a *thread* 01 inseriu o elemento e_x e a *thread* 02 inseriu os elementos e_x e e_{x+1}, na iteração $i+1$ esses elementos constituirão as

[2]O valor é atualizado para 8 devido aos limites de reconstrução dados pela Máscara.

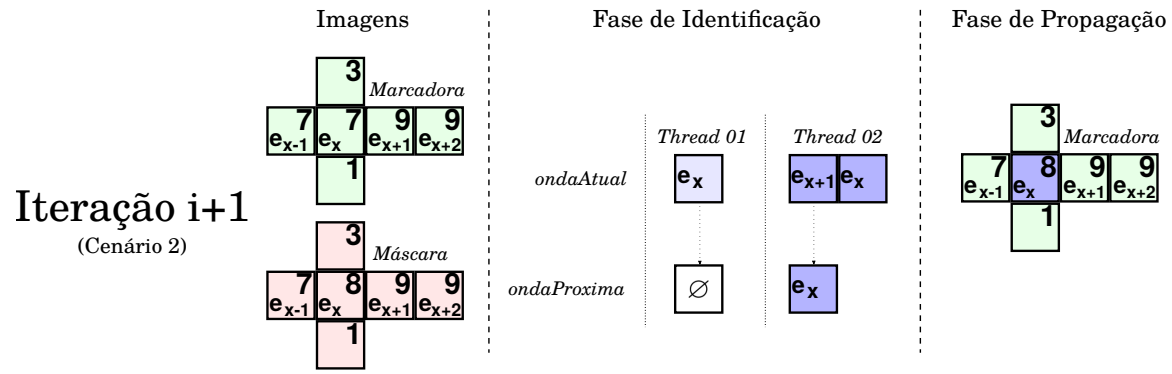

Figura 4.17: Cenário 2: Estabilidade Alcançada na Iteração $i + 1$.

frentes de onda e estarão na fila *ondaAtual*. Dessa forma, durante a Fase de Identificação a *thread* 02 irá inserir e_x na fila *proximaOnda*, e a propagação resultará em uma imagem igual aquela gerada pelo Cenário 1, quando a propagação alcançou a estabilidade.

Por consequência dessa rechecagem de estados, observada no Cenário 2, o *data race* identificado na implementação paralela proposta é benigno e classificado como dupla checagem [34], não influenciando no resultado final.

4.4 Uso de Diferentes Estruturas de Dados

Como já apontado pela Seção 4.3, na implementação paralela do algoritmo proposto, apesar das *threads* serem disparadas em diferentes segmentos, elas trabalham simultaneamente em toda a imagem onde ondas de propagação podem atravessar e influenciar umas as outras. Dessa forma, além da ocorrência de *data races* durante a Fase de Propagação, outra consequência do uso de paralelismo é o crescimento da quantidade de elementos a serem propagados a cada iteração, ou a quantidade de elementos inseridos na fila para processamento.

Esse fato ocorre em razão dos elementos poderem ser inseridos na estrutura de dados mais de uma vez ao longo do tempo, devido a origem das ondas de propagação serem díspares e ainda serem oriundos de *threads* diferentes. Os reprocessamentos provenientes desses casos citados tendem a aumentar, à medida em que se aumenta ou o nível de paralelismo explorado ou o número de *threads*. Assim, o aumento do *throughput* (elementos processados por unidade de tempo) utilizando paralelismo não reflete na mesma proporção no tempo de execução por causa desse reprocessamento.

A Figura 4.18 mostra um exemplo desses reprocessamentos. Nesse caso, a estrutura de dados é uma fila FIFO que possui três valores inseridos em ordem crescente. Da forma como estão inseridos, inicialmente, o elemento $x + 5$ propaga o seu valor. Em seguida, o elemento $x + 3$ realiza a propagação em uma camada superior a do valor $x + 5$, e por

último, o elemento $x+1$ também realiza a propagação superior aos outros dois elementos. É possível observar que a ordem de inserção dos elementos, por causa da fila FIFO, provoca uma propagação gradativa das ondas de propagação até que seja alcançado o valor final dessas propagações.

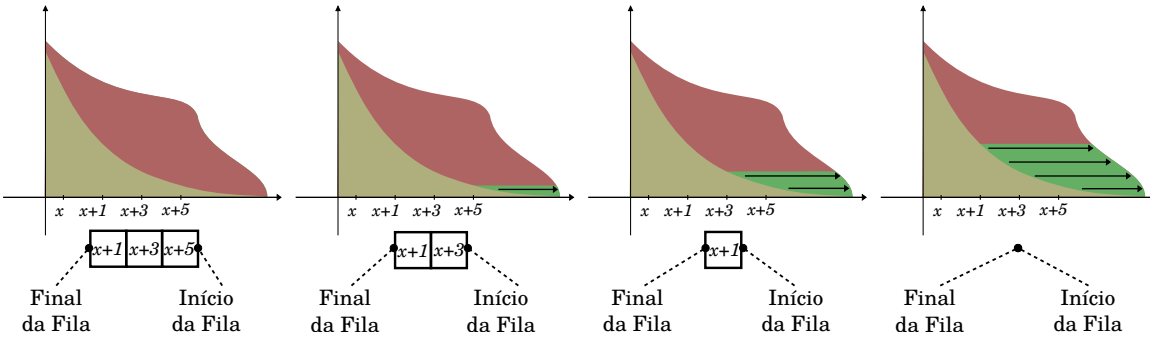

Figura 4.18: Exemplo de Propagação Utilizando Fila FIFO.

Uma forma de lidar com este problema é processar os elementos na Fase de Propagação utilizando uma ordem diferente daquela dada pela Fase de Identificação e inserção na fila, na tentativa de reduzir o número de operações executadas.

Uma vez que a operação almejada por algoritmos IWPP é a maximização ou a minimização de valores em determinada posição, executar primeiro as propagações cujos valores estão mais próximos do valor final, possibilita a redução do processamento, já que este cenário pode evitar que ondas de propagação com valores mais distantes do final sejam executadas e sobrepostas em seguida por ondas com valores mais próximos do almejado.

Uma das estruturas de dados que se adequam ao IWPP, que já utiliza inicialmente a fila FIFO, agregando a ordenação e a extração dos elementos de acordo com seu conteúdo, é a fila de prioridades [62]. Essa fila utiliza uma estrutura *heap*[33] que mantém as ondas de propagação ordenadas de acordo com seus respectivos valores, permitindo a redução de reprocessamento ocasionado pelos eventos citados anteriormente.

A Figura 4.19 demonstra o funcionamento da fila de prioridades. A estrutura possui seus elementos ordenados utilizando uma *heap*, e a extração retira da fila o elemento com maior valor de dentro da estrutura para realizar a propagação (elemento $x+1$). Assim, quando os elementos com valores inferiores são retirados da estrutura (elementos $x+3$ e $x+5$), nenhuma propagação é realizada economizando em processamento.

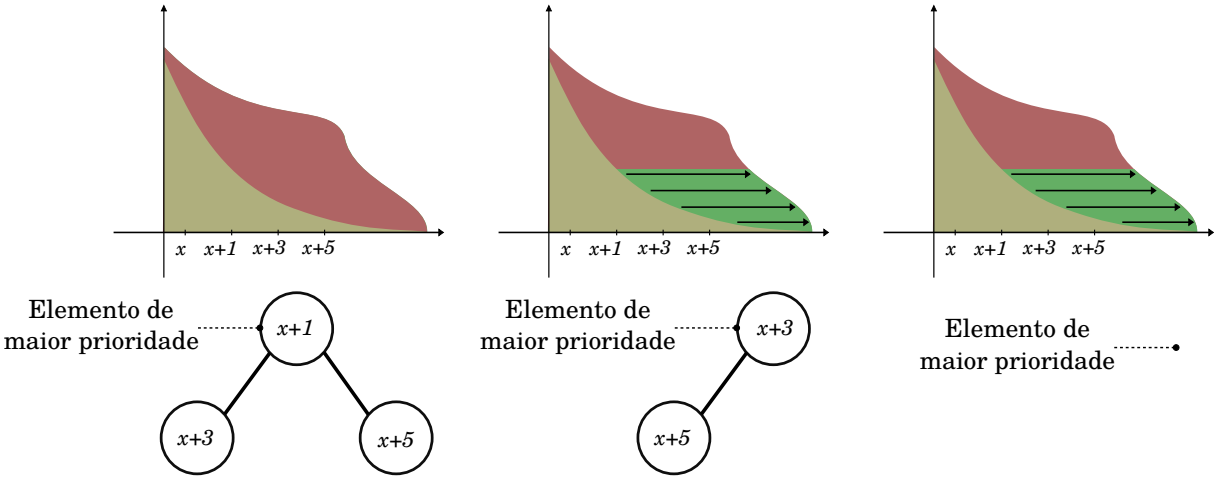

Figura 4.19: Exemplo de Propagação Utilizando Fila de Prioridades.

4.5 Execução Cooperativa em Processadores Heterogêneos

Como forma de aproveitar os recursos disponíveis, e pela possibilidade das imagens utilizadas neste estudo poderem possuir um tamanho superior a memória disponível no Intel® Xeon Phi™, uma estratégia para execução cooperativa em processadores heterogêneos é utilizada.

Em um computador, além da própria CPU, podem existir mais de um Intel® Xeon Phi™ e até GPUs que podem ser assinaladas para execução paralela entre os processadores disponíveis para o processamento de uma imagem. Dessa forma, é possível empregar uma estratégia que divida a entrada a ser processada em tarefas. A Figura 4.20 demonstra um exemplo de divisão de uma imagem em duas tarefas.

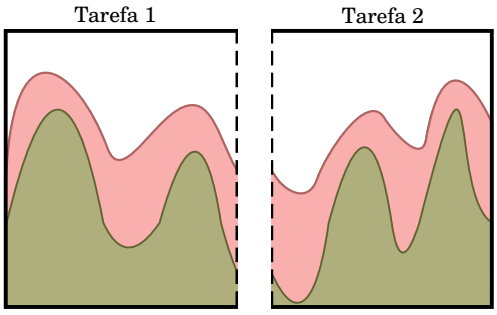

Figura 4.20: Divisão de Uma Imagem em Duas Tarefas.

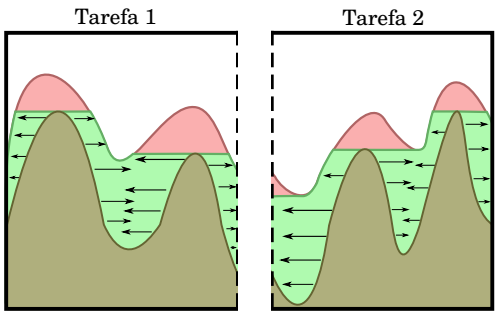

Figura 4.21: Reconstrução Morfológica Aplicada em Cada Tarefa.

Utilizando a Reconstrução Morfológica como exemplo, em cada uma das tarefas é executada individualmente esse algoritmo, gerando uma versão reconstruída que pode ser visualizada na Figura 4.21.

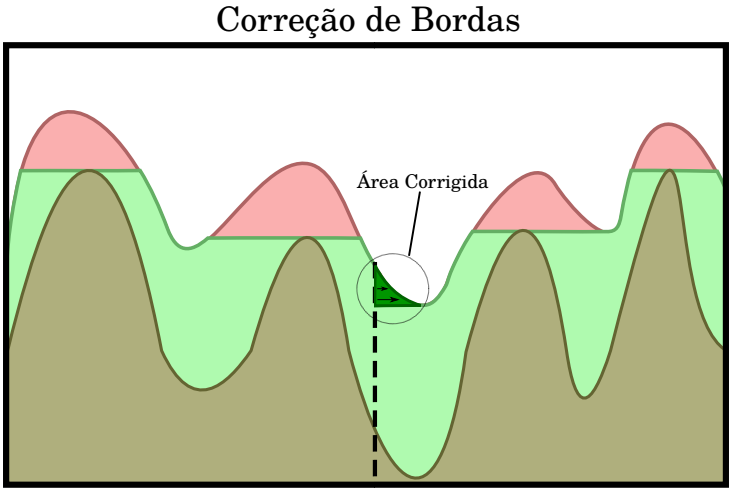

Figura 4.22: Reconstrução Morfológica Após a União das Imagens com a Área de Fronteira Corrigida.

Após o processamento das tarefas de maneira independente, existem propagações que podem migrar das fronteiras de uma imagem (tarefa) para outra. Dessa forma, após a união das imagens processadas individualmente como tarefas, um processamento para a correção da propagação dessas fronteiras é realizado. A região corrigida ou reprocessada para o exemplo da Reconstrução Morfológica pode ser visualizada na Figura 4.22.

A implementação dessa estratégia pode ser observada na Figura 4.23 que constitui-se de quatro estágios: Divisão da Imagem, Processamento de Tarefas, União da Imagem e Correção das Fronteiras. A Divisão da Imagem consiste na criação das tarefas a serem processadas, dada uma imagem como entrada. Essas tarefas passam a fazer parte de uma fila de tarefas a serem processadas na fase de Processamento de Tarefas. Nessa fase, os dispositivos disponíveis para executar algum processamento (GPUs, MICs e CPU) irão consumir, de forma independente, tarefas da fila até que a mesma se encontre vazia. Na fase de União da Imagem todas as tarefas já foram processadas e esses pedaços da imagem assinados para cada tarefa são montados em sua posição original. E por último, na fase Correção de Fronteiras, é executado um processamento em CPU identificando elementos ativos nas fronteiras dos pedaços das imagens, agora unidas, e a propagação das ondas relacionadas a essas fronteiras.

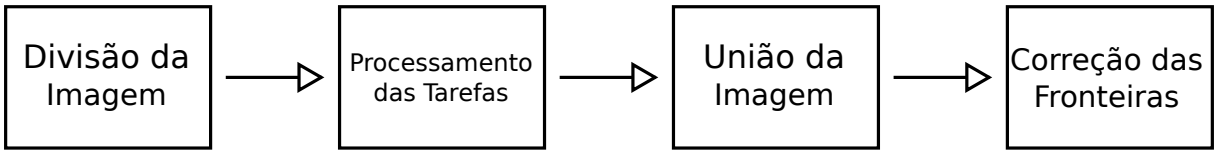

Figura 4.23: Área Processada pela Correção de Bordas.

4.6 Sumário

Neste capítulo foi apresentada uma abordagem eficiente para algoritmos da classe *Irregular Wavefront Propagation Pattern*. Inicialmente, para permitir a aplicação de estratégias de alto desempenho, foram propostas modificações no algoritmo IWPP separando a propagação em duas fases: Fase de Identificação e Fase de Propagação.

A partir desse novo algoritmo, foram aplicadas estratégias de vetorização na identificação dos elementos recebedores de propagação, realizando leitura da vizinhança de um elemento e a inserção de elementos na fila utilizada na fase de Propagação do algoritmo. Para a realização da inserção vetorial, também foi apresentada a função *prefix sum* vetorial que permite identificar a posição dos elementos a serem inseridos na estrutura de dados (fila) utilizada.

Posteriormente, foi empregada uma estratégia paralela na versão vetorizada do algoritmo. Essa estratégia permite o uso de várias *threads* por meio da distribuição de segmentos da imagem durante a fase de Inicialização. Após a Inicialização, as *threads* executam a fase de Propagação e trabalham por toda a imagem realizando modificações de forma independente entre as *threads*.

Em seguida, foi relatado o uso de uma fila de prioridades na implementação paralela. A finalidade do seu uso é ordenar os elementos inseridos na Fase de Identificação pelo seu conteúdo, buscando realizar primeiro a propagação de elementos que estão mais próximos de seu valor final.

Por fim, foi apresentada uma estratégia para a execução cooperativa em processadores heterogêneos. Essa estratégia permite processar imagens que ultrapassem o tamanho da memória disponível no Intel® Xeon Phi™, realizando a divisão dela em tarefas que podem ser processadas de forma independente, a partir de uma fila de tarefas e uso de correção das fronteiras dos pedaços dessas imagens.

Capítulo 5

Análise dos Resultados

Neste capítulo será apresentada uma análise dos resultados alcançados pelas estratégias desenvolvidas no Capítulo 4. As seções a seguir detalham os ambientes de desenvolvimento e testes (Seção 5.1), as configurações dos experimentos realizados (Seção 5.2) e os resultados alcançados (Seção 5.3).

5.1 Ambiente de Desenvolvimento e Testes

Os testes e análises foram realizados em três computadores com as seguintes configurações:

1. Um SGI C1104 com Sistema Operacional CentOS 6.5, e as seguintes configurações:
   - 02 (dois) processadores Intel® Xeon® 8-*Core* 64-bit E5-processors com 2.6GHz de frequência;
   - 20MB de memória cache L3;
   - 64GB de memória RAM DDR3-1866MHz;
   - 01 (um) coprocessador Intel® Xeon Phi™ 7120P 1.33GHz;
   - HD SATA de 1TB, 10000 RPM.
2. Um computador Dell C8220z com Sistema Operacional CentOS 6.3, e as seguintes configurações:
   - 02 (dois) processadores Intel® Xeon® 8-*Core* 64-bit E5-processors com 2.7GHz de frequência;
   - 20MB de memória cache L3;
   - 256KB de memória cache L2;

- 32GB de memória RAM DDR3-1600MHz;
- 01 (um) coprocessador Intel® Xeon Phi™ SE10P 1.1GHz;
- 01 (uma) placa gráfica NVIDIA Tesla K20 (GK110);
- HD SATA de 250GB, 7500 RPM.

3. Um computador com Sistema Operacional CentOS 6.5, e as seguintes configurações:

   - 02 (dois) processadores Intel® Xeon® 8-*Core* 64-bit E5-processors com 2.7GHz de frequência;
   - 20MB de memória cache L3;
   - 256KB de memória cache L2;
   - 32GB de memória RAM DDR3-1600MHz;
   - 02(dois) coprocessadores Intel® Xeon Phi™ SE10P 1.1GHz.

Os algoritmos desenvolvidos foram comparados com outras implementações para CPU e GPU. Para isso, foram utilizadas as versões mais eficientes conhecidas na literatura para CPU [65] e GPU [60] [59].

A GPU utilizada foi a NVIDIA Tesla K20, e cada um dos dispositivos utilizados nos testes tiveram sua largura de banda testada de acordo com o tipo de acesso aos dados (regular ou randômico). No acesso regular, a largura de banda foi mensurada utilizando o *STREAM benchmark* [30]. Para o acesso randômico foi desenvolvido um *benchmark* simplificado com acesso randômico paralelo de 10 milhões de elementos em uma imagem $4K \times 4K$. Os resultados podem ser visualizados na Tabela 5.1 que contém os *benchmarks* realizados e as informações relevantes para os dispositivos testados.

Tabela 5.1: Características dos Processadores.

	K20 GPU	SE10P	7120P
Número de núcleos	2496	61	61
Clock por núcleo (MHz)	706	1100	1238
Largura de banda - Acesso regular (GB/s)	148	160	177
Largura de banda - Acesso randômico (MB/s)	895	399	438

5.2 Configuração dos Experimentos

O desempenho da implementação do IWPP foi avaliado utilizando a Reconstrução Morfológica e a função *Fill Holes*. Para ambos os algoritmos, suas configurações foram

variadas em termos de conectividade (4 e 8), tamanho da imagem de entrada e o tipo ou grau de cobertura. Os testes foram executados em imagens coletadas para pesquisa do *In Silico Brain Tumor Research Center* (ISBTRC)[42] da *Emory University*. Essas imagens foram variadas quanto ao tamanho de entrada de $4K \times 4K$ até $32K \times 32K$, e foram diferidas em tipo de cobertura e percentual aproximado de tecido na imagem (Figura 5.1).

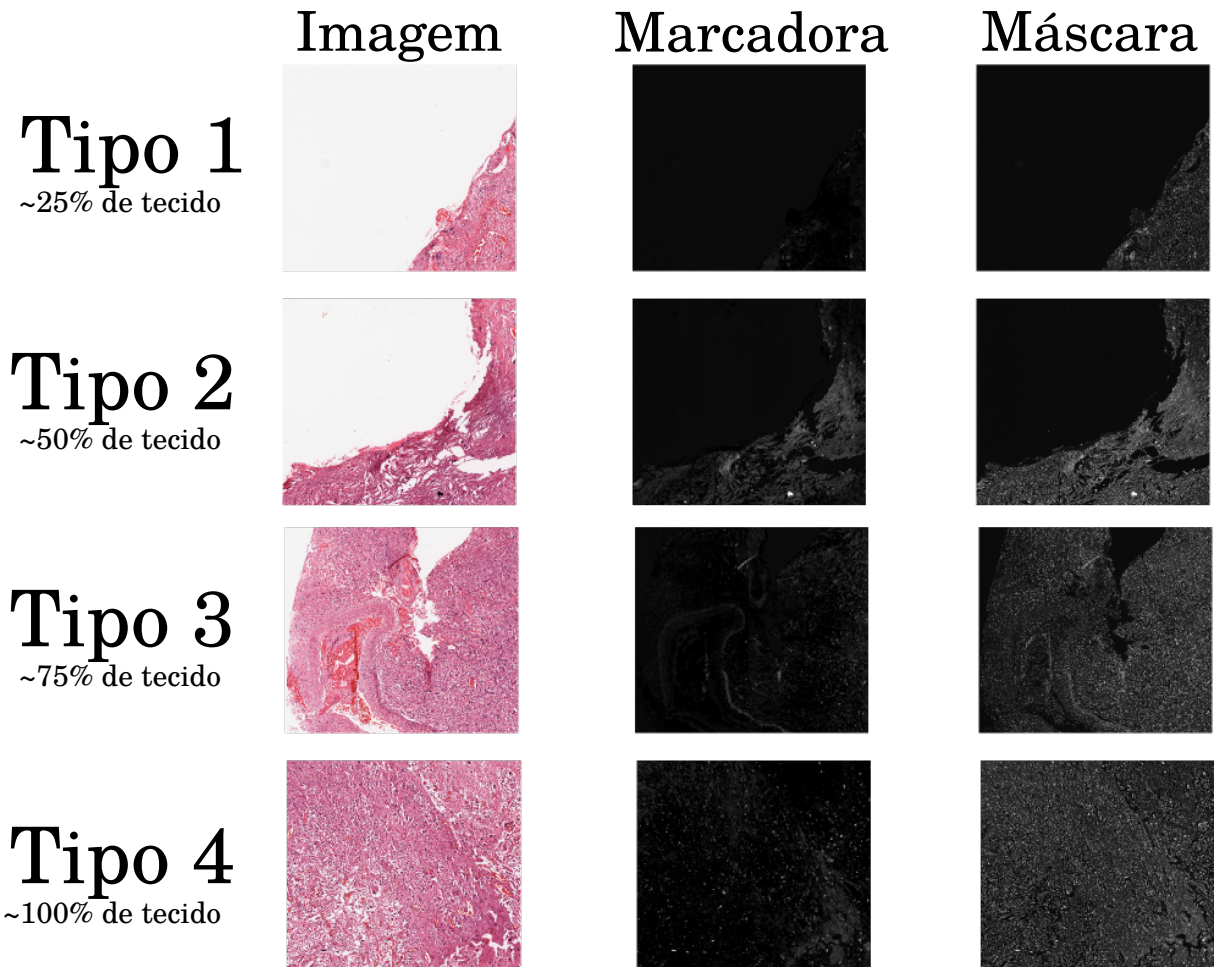

Figura 5.1: Tipos Diferentes de Cobertura.

Em todas as amostras, o algoritmo foi executado dez vezes para cada uma das configurações com cada tipo de cobertura. Para essas execuções, o coeficiente de variação não foi superior a 1%.

5.3 Resultados Experimentais

Esta seção apresenta os resultados obtidos da execução dos testes e uma discussão acerca da análise dos mesmos.

5.3.1 Impacto da Vetorização para o Desempenho

Nesta seção foi avaliado o ganho de vetorização em relação a versão não-vetorizada do algoritmo. Os experimentos foram executados no Intel® Xeon Phi™ SE10P com a Reconstrução Morfológica e o *Imfill* utilizando fila FIFO e as imagens de entrada com tamanho $4K \times 4K$ variando seu percentual de cobertura.

A Figura 5.2 mostra o gráfico de *speedups* do algoritmo vetorizado em relação ao algoritmo sem vetorização.

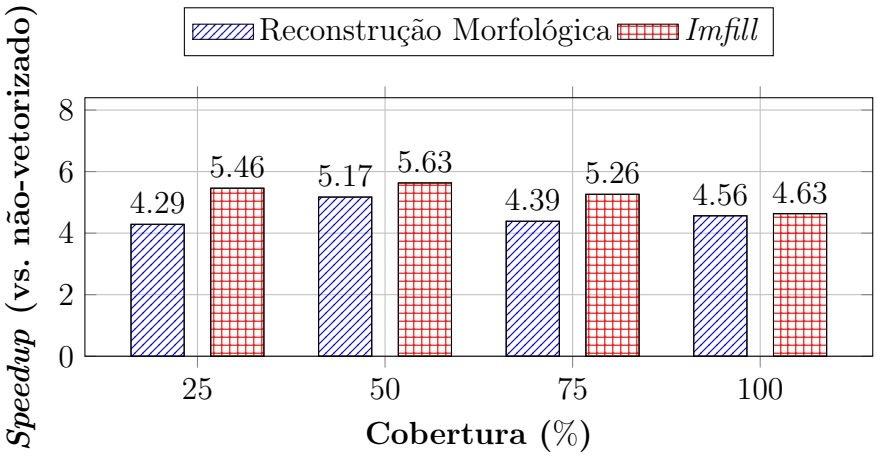

Figura 5.2: Ganho de Vetorização Variando a Porcentagem de Cobertura.

Como pode ser visto na Figura 5.2, os algoritmos alcançaram significativos *speedups* para todas as configurações de entrada e o *Imfill* obteve melhores resultados em termos de vetorização do que a Reconstrução Morfológica. A melhor configuração obteve *speedup* de 5.63× para imagens com 50% de cobertura. A cobertura da imagem é um indicativo da quantidade de trabalho a ser processado durante a execução do algoritmo, porém, devido a irregularidade do IWPP, imagens com coberturas menores podem executar menos ou mais trabalho que imagens de mesmo tamanho com coberturas maiores, ocasionando essa oscilação nos *speedups*, que podem ser observadas no desempenho de imagens com 75% e 100% de cobertura, que obtiveram ganhos inferiores aos de 50%.

Dadas as instruções vetoriais disponíveis e utilizadas no Intel® Xeon Phi™, a estratégia mostrou-se eficiente. Contudo, os ganhos em desempenho são menores que a proporção de aumento da quantidade de elementos que são processados em um vetor de 512 bits, quando comparado com a versão não vetorizada. Na implementação são utilizados inteiros de 32 bits para manipular os dados e as instruções SIMD podem processar até 16 elementos por vez. Isso ocorre porque o código vetorizado não é uma tradução direta do código não-vetorizado, e sua versão demanda mais instruções que o código original, em virtude da sua natureza irregular, como pode ser visto nos detalhes da implementação. A inserção

vetorial na fila, por exemplo, requer o uso da função *prefix sum* que inexiste na versão não-vetorizada.

5.3.2 Resultados da Paralelização do Algoritmo Vetorizado

A Figura 5.3 apresenta o gráfico de *speedups* alcançados ao variar a quantidade de *threads* utilizadas para a paralelização apresentada na Seção 4.2. Os *speedups* são calculados tomando como referência a execução do respectivo algoritmo no MIC utilizando 1 *thread*.

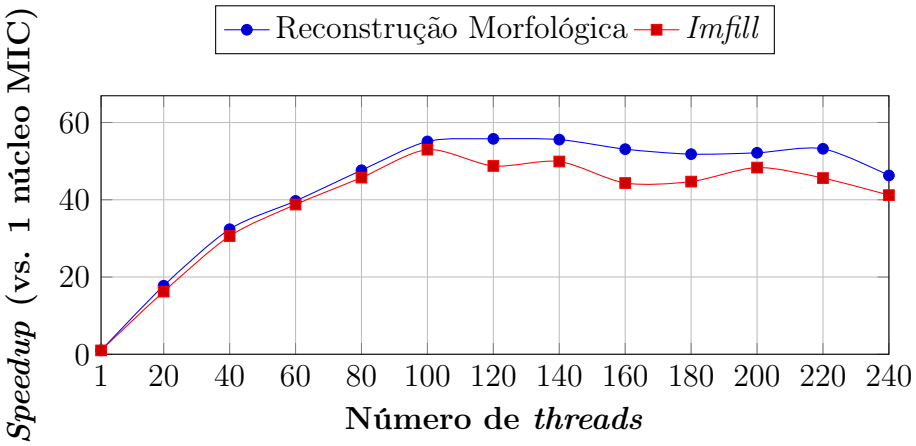

Figura 5.3: Análise de Escalabilidade Variando a Quantidade de *Threads*.

Nessa implementação, o algoritmo atingiu *speedups* de até 55.7× para a configuração utilizada (imagens de $8K \times 8K$). Apesar de o coprocessador suportar 240 *threads* de hardware, o ganho acima do número de núcleos do coprocessador (60) é limitado. Além disso, não é possível perceber ganhos significativos com o aumento de *threads* para valores superiores a 120.

5.3.3 Impacto do Percentual de Tecido de Cobertura

A Figura 5.4 contém o gráfico do impacto do tipo de cobertura para o algoritmo da Reconstrução Morfológica e do algoritmos *Fill Holes*. Neste experimento foram utilizadas imagens com $4K \times 4K$ onde a execução do algoritmo é comparada com a execução sequencial em CPU.

Ambos os algoritmos apresentaram melhorias quando comparados com a versão sequencial de CPU. Como pode ser observado no gráfico, maior quantidade de tecido a ser processado resulta em maior *speedup*. Isso ocorre devido ao aumento da quantidade de

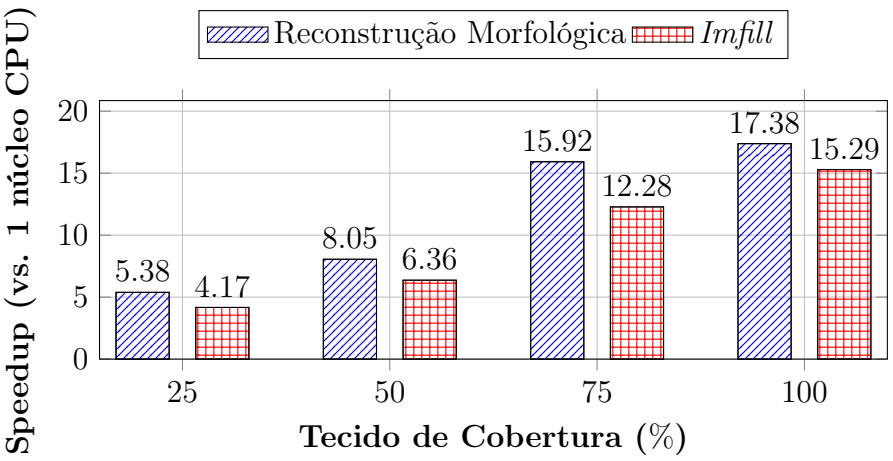

Figura 5.4: Variação do Percentual de Tecido de Cobertura.

processamento e redução da porcentagem de *overhead* necessário ao início do processamento.

Outro fato observado são os *speedups* maiores por parte da Reconstrução Morfológica. Isso ocorre porque a função *Imfill* utiliza a própria inversa da imagem a ser processada como máscara, resultando em um número muito maior de propagações. Dessa forma, é requerido um número maior de escaneamentos na inicialização do algoritmo, e o ganho na Fase de Propagação Irregular em relação à execução do algoritmo é reduzida.

5.3.4 Impacto do Tamanho da Imagem

Esta seção avalia o impacto do tamanho da entrada para a execução do IWPP. Para esses testes foi escolhido o algoritmo da Reconstrução Morfológica utilizando a fila FIFO.

Os resultados apresentados na Figura 5.5 demonstraram o aumento do *speedup* ao passo em que a imagem de entrada também aumenta. Essa melhoria no *speedup* é de 1.45× quando as imagens são aumentadas de $4K \times 4K$ para $16K \times 16K$. A melhoria apresentada por esse experimento é resultado de uma maior utilização do paralelismo que permite um melhor uso do coprocessador e reduz o custo de computação e transferência de dados entre CPU e o coprocessador.

5.3.5 Avaliação de Diferentes Coprocessadores e Estrutura de Dados

Esta seção compara a performance do IWPP no Intel® Xeon Phi™ SE10P, 7120P, a GPU NVIDIA K20 e a CPU Intel® Xeon® 8-*Core* E5. Ambos os coprocessadores possuem a mesma quantidade de núcleos (61), porém os mesmos se diferem no *clock*,

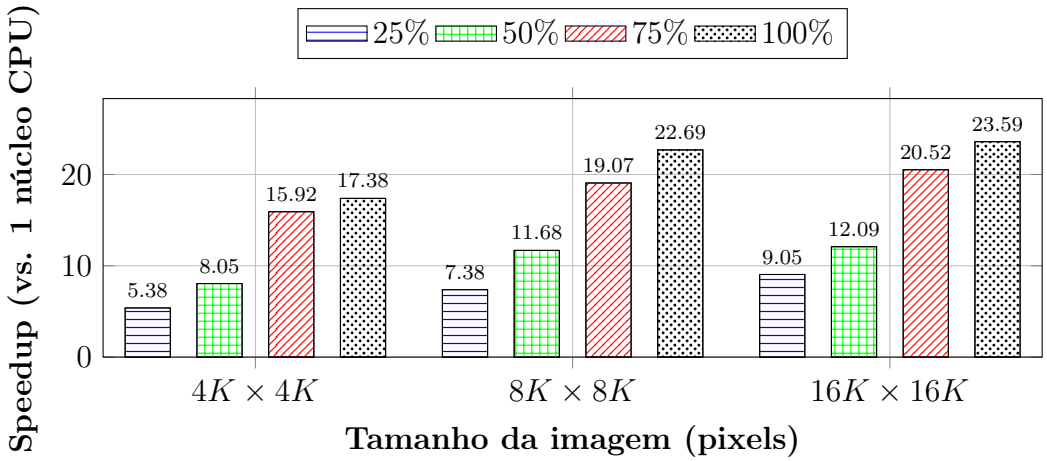

Figura 5.5: Variação do Tamanho das Imagens de Entrada para a Reconstrução Morfológica.

como já apresentado na Tabela 5.1 (1.1GHz e 1.33GHz). Para ambos os processadores, também foram avaliados o uso da fila de prioridades que ordena as ondas de propagação de acordo com a intensidade dos elementos inseridos na Estrutura de Dados.

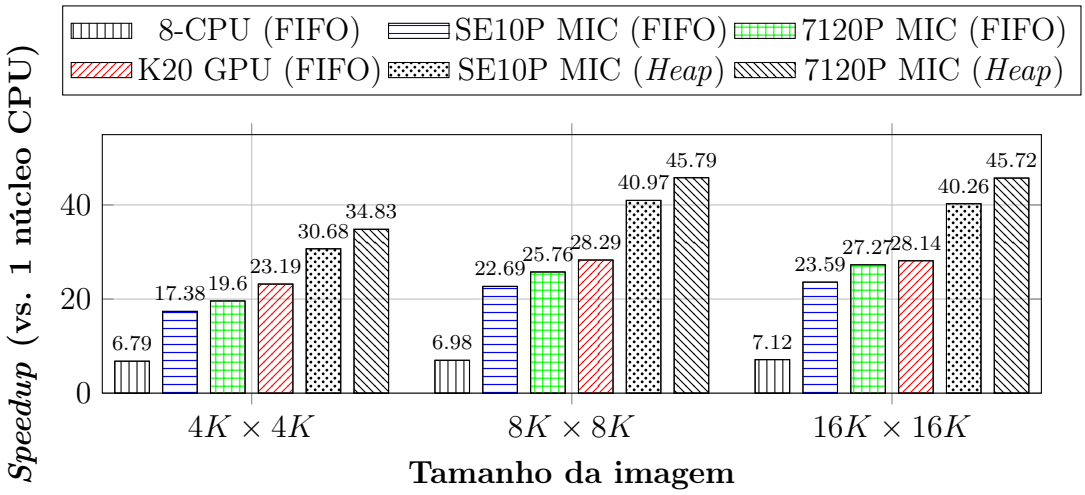

Figura 5.6: Avaliação de Múltiplos Coprocessadores e Uso de Diferentes Tipos de Estrutura de Dados.

Utilizando a execução sequencial em CPU como base, a Figura 5.6 apresenta o gráfico comparativo de desempenho. Como pode ser observado, todas as versões avaliadas apresentaram ganho em relação a versão utilizada como base.

A execução do Intel Phi 7120P apresentou ganho de 1.14× em relação ao modelo SE10P, o que pode ser explicado pela diferença entre o *clock* de cada núcleo dos coprocessadores.

Já a versão de GPU apresentou melhor *speedup* do que as melhores execuções no Intel Phi utilizando filas FIFO, devido a diferença de largura de banda entre ela e os coprocessadores (28.29×). Algoritmos da classe IWPP possuem acesso intensivo de dados de forma regular na Fase de Inicialização, e irregular na Fase de Propagação Irregular. Como pode ser observado na Tabela 5.1, o Intel Phi é mais eficiente que a GPU K20 no acesso regular, porém a GPU possui largura de banda 2.04× mais rápida quando o acesso é irregular.

Por fim, os testes utilizando uma fila de prioridades, onde os elementos inseridos estão ordenados pela intensidade de seus valores, resultaram na mais significativa melhoria deste experimento. A execução no 7120P, utilizando imagens de tamanho $16K \times 16K$, alcançou *speedup* 1.62× mais rápido do que a melhor execução em GPU. Ainda essa execução, utilizando fila de prioridades, resultou em um ganho decorrente da redução de 20× menos elementos processados durante a Fase de Propagação Irregular do que a versão com fila FIFO do mesmo modelo de coprocessador. Isso se deve a ordenação propiciada pela fila de prioridades que ocasiona em propagações de elementos cujos valores estão mais próximos de seu valor final, como já explicado na Seção 4.4.

5.3.6 Avaliação da Execução Cooperativa em Processadores Heterogêneos

Esta seção retrata o uso da execução cooperativa em processadores heterogêneos, de acordo com a estratégia apresentada na Seção 4.5. Os testes foram realizados utilizando o algoritmo da Reconstrução Morfológica, e com as seguintes configurações:

- Versão serial em 1 *core* de CPU utilizada como base;
- Versão CPU *multithread* executada em 16 *cores*;
- 1 MIC;
- 1 MIC + versão CPU *multithread*;
- 2 MICs;
- 2 MICs + versão CPU *multithread*.

Visando um melhor aproveitamento dos recursos de processamento disponíveis, aquelas configurações que combinam o uso de CPUs e Intel® Xeon Phi™ utilizaram uma estratégia de divisão do tamanho da imagem (ou tarefa) a ser processada. Essa divisão aproveita informações anteriores dos *speedups* individuais de cada um dos dispositivos, e se dá da seguinte forma: Dado o *speedup* S_{CPU} da execução *multithread* e o *speedup* S_{MIC} da

execução no Intel® Xeon Phi™, a porcentagem do tamanho da imagem P_{CPU} que a versão *multithread* deverá processar é a seguinte:

$$P_{CPU} = \frac{100 * S_{CPU}}{S_{CPU} + S_{MIC}} \tag{5.1}$$

e a porcentagem do tamanho da imagem P_{MIC} que o Intel® Xeon Phi™ deverá processar é:

$$P_{MIC} = \frac{100 * S_{MIC}}{S_{CPU} + S_{MIC}} \tag{5.2}$$

Dessa forma, a Tabela 5.2 apresenta as porcentagens empregadas em cada um dos dispositivos utilizados nos testes de uso combinado de MICs e CPUs.

Tabela 5.2: Porcentagem de Divisão das Tarefas por Dispositivo.

Dispositivos	1 MIC + CPU (*multithread*)		2 MIC + CPU (*multithread*)	
Tamanho	MIC (%)	CPU (%)	MIC (%)	CPU (%)
$16K \times 16K$	68	32	82	18
$24K \times 24K$	71	29	84	16
$32K \times 32K$	70	30	82	18

Assim, a Figura 5.7 analisa as execuções das configurações citadas anteriormente.

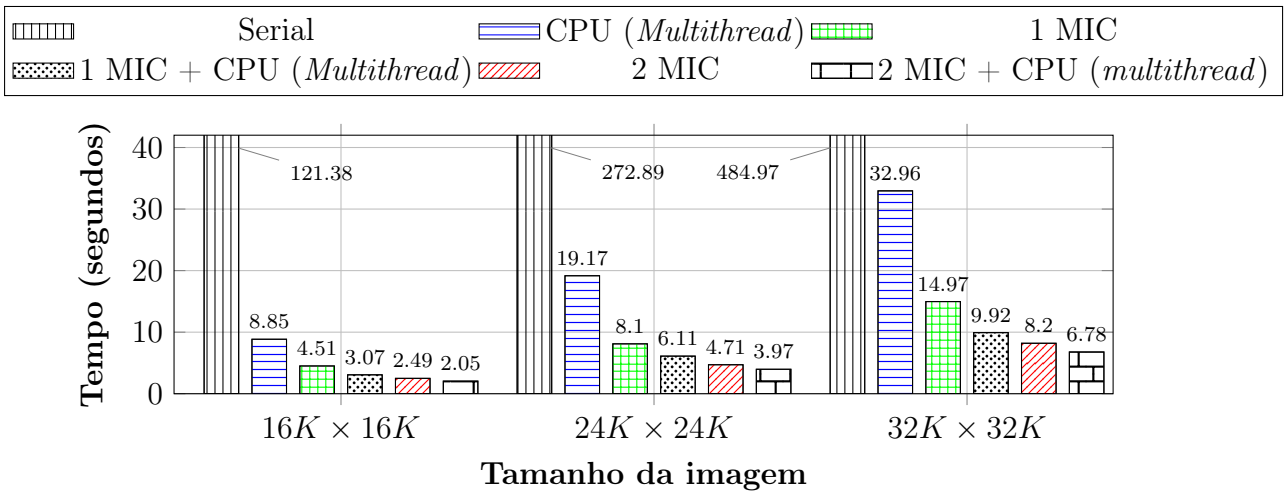

Figura 5.7: Avaliação da Execução Cooperativa em Processadores Heterogêneos.

Como pode ser notado, o uso combinado de dispositivos se demonstrou proveitoso em todas as configurações do experimento, alcançando uma redução média de até 50% dos tempos de execução dos dispositivos individuais.

A execução utilizando 2 Intel® Xeon Phi™ alcançou *speedups* próximos a 1.8×, comparado a execução utilizando apenas um desses dispositivos. A justificativa para esse valor de redução do tempo abaixo de linear remete ao tempo de transferência dessas imagens (*offload transfer*) para os dispositivos que, mesmo divididas a uma fração de seu tamanho, continuam possuindo um custo alto de inicialização para cada um dos dispositivos. Na arquitetura MIC, o uso de *offload* inclui um *overhead* de inicialização, gerência e transferência de dados, e chamada de função na aplicação [36].

Por último, o uso combinado de CPU e 2 MICs resultou em um decrescimento de tempo médio próximo a 18% em relação ao uso somente de 2 MICs. Esse valor representa a proporção das tarefas entregues ao dispositivo mais lento (CPUs), quando comparados em relação ao tempo de execução da configuração de maior *speedup* utilizando um mesmo tipo de dispositivo (execução em 2 MICs).

Capítulo 6

Conclusão

Nesta dissertação foi proposta e avaliada uma solução eficiente para o *Irregular Wavefront Propagation Pattern* na arquitetura *Many Integrated Core*. Esta solução explorou o uso de instruções *Single Instruction, Multiple Data* SIMD que permitiu alcançar bons resultados para a solução.

Inicialmente, o algoritmo do IWPP foi reprojetado para permitir uma implementação eficiente sem o uso de operações atômicas. Essa versão separa a fase *Wavefront Propagation* em duas fases (Identificação de Elementos Recebedores de Propagação e Propagação), fazendo com que elementos passem a modificar apenas suas próprias posições.

Por meio dessa versão reprojetada, a implementação vetorial explora a largura do vetor de 512 bits dos Intel® Xeon Phi™ durante a leitura e a propagação de elementos vizinhos, a execução de uma função *prefix sum* com permutação dos valores dentro do próprio registrador vetorial e a inserção vetorial de elementos na estrutura de dados utilizada durante a propagação. Essa estratégia alcançou *speedups* de até 5.63× para ambos os algoritmos implementados (Reconstrução Morfológica e *Imfill*), demonstrando a viabilidade do uso de instruções vetoriais ao se desenvolver aplicações para o Intel® Xeon Phi™.

A versão paralela do algoritmo vetorizado realiza uma segmentação do trabalho inicial das *threads* entre as linhas da imagem. Apesar dessa segmentação inicial, durante a fase de propagação, as *threads* podem realizar propagações por toda a imagem, gerando *data races* comprovados benignos devido a rechecagem de estados utilizado no algoritmo reprojetado. Testes nessa versão paralelizada demonstraram boa escalabilidade, atingindo *speedups* de 55.7× quando comparados a execução de um núcleo do Intel® Xeon Phi™. Ao variar o percentual de tecido de cobertura, aumenta-se a quantidade de processamento distribuído a cada *thread* que reduz o *overhead* vinculado ao início do processamento. Os testes para essa variação resultaram em *speedups* que variaram de 5.38× em imagens com 25% de tecido de cobertura a 17.38× em imagens com 100% de tecido de cobertura na Reconstrução Morfológica, e de 4.17× em imagens com 25% de tecido de cobertura a

15.29× em imagens com 100% de tecido de cobertura na *Imfill*. Avaliações referentes ao tamanho da entrada demonstraram um aumento de 1.45× quando imagens são aumentadas de $4K \times 4K$ para $16K \times 16K$, sendo resultado de uma maior utilização do paralelismo do coprocessador.

Como consequência do uso da estratégia paralela e pela possibilidade de elementos recomputados aumentarem de acordo com o nível de paralelismo, foi empregada uma fila de prioridades que mantiveram as ondas de propagação ordenadas de acordos com seus respectivos valores, ocasionando a redução da quantidade desse reprocessamento. Além disso, foram feitos testes relacionados ao uso dessa estrutura comparando dois tipos de estruturas de dados (fila FIFO e fila de prioridades) e dois modelos de coprocessador com a implementação mais rápida conhecida na literatura em GPU e uma versão *multicore* para CPU. Os resultados mostraram que as versões do Intel® Xeon Phi™ utilizando fila FIFO alcançaram bons *speedups* em relação à versão *multicore* em CPU (cerca de 2.72× mais rápido), porém não mais que a implementação em GPU utilizando fila FIFO que foi 1.06× mais rápido. Já o uso de fila de prioridades alcançou *speedups* de 1.62× mais rápido do que a implementação em GPU, confirmando a redução de elementos recomputados devido ao uso de paralelismo ocasionada pela propagação de elementos cujos valores estão mais próximos de seu valor final.

Por último, para possibilitar o processamento de imagens que superem a capacidade de memória do Intel® Xeon Phi™, e como forma de aproveitar os dispositivos disponíveis no computador, foi implementada uma estratégia para execução cooperativa em processadores heterogêneos. Essa estratégia realiza a divisão do processamento em estágios. Nesses estágios, a imagem é divida em tarefas que são entregues aos dispositivos para processamento. Após esse processamento, a imagem é novamente unida e, então, é executada uma correção das fronteiras da imagem que não foram totalmente processadas em virtude da divisão. Essa estratégia demonstrou-se exequível com os tempos de execução, utilizando 2 Intel® Xeon Phi™ mais a implementação *multithread* em CPU, reduzindo o tempo médio de execução para 56% do tempo de processamento utilizando apenas 1 Intel® Xeon Phi™.

6.1 Trabalhos Futuros

As implementações desenvolvidas neste trabalho utilizam apenas o paradigma de programação de memória compartilhada, não aproveitando a possibilidade de uso em sistemas distribuídos para uma mesma imagem utilizada como entrada. Como trabalho futuro, deseja-se propor uma solução em memória distribuída que aproveite de maneira eficiente os recursos disponíveis em tempo de execução.

Além disso, é possível aproveitar as estratégias apresentadas nesse algoritmos do IWPP reprojetado e implementar outros algoritmos que também tenham seus princípios de funcionamento baseados nele (Transformação Watershed, Esqueletos por Zona de Influência, etc).

Por fim, o barramento do Intel® Xeon Phi™ não permite um controle detalhado do barramento como forma de melhorar a coerência de cache, permitindo um controle da memória interna de cada núcleo do coprocessador. Assim, é válido um estudo detalhado para aumentar a coerência de memória, através de outras estratégias como *prefetching* de instruções [20].

Referências

[1] G. Andrade, R. Ferreira, G. Teodoro, L. Rocha, J.H. Saltz, and T. Kurc. Efficient Execution of Microscopy Image Analysis on CPU, GPU, and MIC Equipped Cluster Systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 89–96, Oct 2014.

[2] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer L Welch. Shared Memory Consistency Conditions for Non-sequential Execution: Definitions and Programming Strategies. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 241–250. ACM, 1993.

[3] Guy E Blelloch. Prefix Sums and their Applications. 1990.

[4] Harry Blum et al. A Transformation for Extracting New Descriptors of Shape. *Models for the perception of speech and visual form*, 19(5):362–380, 1967.

[5] Gunilla Borgefors. Distance Transformations in Arbitrary Dimensions. *Computer vision, graphics, and image processing*, 27(3):321–345, 1984.

[6] Heinz Breu, Joseph Gil, David Kirkpatrick, and Michael Werman. Linear Time Euclidean Distance Transform Algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(5):529–533, 1995.

[7] Steve Carr, Jean Mayo, and Ching-Kuang Shene. Race Conditions: A Case Study. *Journal of Computing Sciences in Colleges*, 17(1):90–105, 2001.

[8] Intel Corporation. Even Higher Efficiency for Parallel Processing. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-block-diagram.html`. Accessed: 2015-07-07.

[9] Intel Corporation. Intel (TM) Many Integrated Core Architecture (Intel (R) MIC Architecture) – Advanced. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`. Accessed: 2014-12-12.

[10] Olivier Cuisenaire and Benoît Macq. Fast Euclidean Distance Transformation by Propagation Using Multiple Neighborhoods. *Computer Vision and Image Understanding*, 76(2):163–172, 1999.

[11] Per-Erik Danielsson. Euclidean Distance Mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[12] Oxford English Dictionary. Oxford: Oxford University Press, 1989.

[13] Ralph Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, 1990.

[14] Ricardo Fabbri, Luciano Da F Costa, Julio C Torelli, and Odemir M Bruno. 2D Euclidean Distance Transform Algorithms: A Comparative Survey. *ACM Computing Surveys (CSUR)*, 40(1):2, 2008.

[15] Michael Flynn. Some Computer Organizations and their Effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[16] Jeremias M Gomes, George Teodoro, Alba de Melo, Jun Kong, Tahsin Kurc, and Joel H Saltz. Efficient Irregular Wavefront Propagation Algorithms on Intel (R) Xeon Phi (TM). In *Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on*, pages 25–32. IEEE, 2015.

[17] Rafael C Gonzalez and Richard E Woods. Digital Image Processing, 2002.

[18] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel Prefix Sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[19] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *ACM SIGPLAN Notices*, volume 46, pages 267–276. ACM, 2011.

[20] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[21] Ioan Jivet, Alin Brindusescu, and Ivan Bogdanov. Image Contrast Enhancement Using Morphological Decomposition by Reconstruction. *WSEAS Trans. Cir. and Sys*, 7(8):822–831, 2008.

[22] Pavel Karas. Efficient Computation of Morphological Greyscale Reconstruction. In *OASIcs-OpenAccess Series in Informatics*, volume 16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[23] Jun Kong, Lee A. D. Cooper, Fusheng Wang, Jingjing Gao, George Teodoro, Lisa Scarpace, Tom Mikkelsen, Matthew J. Schniederjan, Carlos S. Moreno, Joel H. Saltz, and Daniel J. Brat. Machine-Based Morphologic Analysis of Glioblastoma Using Whole-Slide Pathology Images Uncovers Clinically Relevant Molecular Correlates. *PLoS ONE*, 8(11):e81049, 11 2013.

[24] Jun Kong, Fusheng Wang, G. Teodoro, L. Cooper, C.S. Moreno, T. Kurc, T. Pan, J. Saltz, and D. Brat. High-performance Computational Analysis of Glioblastoma Pathology Images with Database Support Identifies Molecular and Survival Correlates. In *2013 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 229–236, Dec 2013.

[25] Jun Kong, Fusheng Wang, George Teodoro, Yanhui Liang, Yangyang Zhu, Carol Tucker-Burden, and Daniel J. Brat. Automated Cell Segmentation with 3D Fluorescence Microscopy Images. In *12th IEEE International Symposium on Biomedical Imaging, ISBI 2015, Brooklyn, NY, USA, April 16-19, 2015*, pages 1212–1215, 2015.

[26] Tahsin Kurc, Xin Qi, Daihou Wang, Fusheng Wang, George Teodoro, Lee Cooper, Michael Nalisnik, Lin Yang, Joel Saltz, and David J. Foran. Scalable Analysis of Big Pathology Image Data Cohorts Using Efficient Methods and High-Performance Computing Strategies. *BMC Bioinformatics*, 16(1):1–21, 2015.

[27] Christian Lantuejoul. Skeletonization in Quantitative Metallography. *Issues of Digital Image Processing*, 34(107-135):109, 1980.

[28] Christophe Laurent and Jean Roman. Parallel Implementation of Morphological Connected Operators Based on Irregular Data Structures. In José M. Laginha M. Palma, Jack Dongarra, and Vicente Hernández, editors, *VECPAR*, volume 1573 of *Lecture Notes in Computer Science*, pages 579–592. Springer, 1998.

[29] Yanhui Liang, Fusheng Wang, Darren Treanor, Derek R. Magee, George Teodoro, Yangyang Zhu, and Jun Kong. A 3D Primary Vessel Reconstruction Framework with Serial Microscopy Images. In *18th International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, pages 251–259, 2015.

[30] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 19–25, December 1995.

[31] Arnold Meijster, Jos BTM Roerdink, and Wim H Hesselink. A General Algorithm for Computing Distance Transforms in Linear Time. In *Mathematical Morphology and its applications to image and signal processing*, pages 331–340. Springer, 2000.

[32] Fernand Meyer. Digital Euclidean Skeletons. In *Lausanne-DL tentative*, pages 251–262. International Society for Optics and Photonics, 1990.

[33] David R. Musser, Gilmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., 2001.

[34] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *ACM SIGPLAN Notices*, volume 42, pages 22–31. ACM, 2007.

[35] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free Irregular Computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 96–107. ACM, 2013.

[36] Chris J Newburn, Serguei Dmitriev, Renukaprasad Narayanaswamy, John Wiegert, Rohan Murty, Francisco Chinchilla, Rajiv Deodhar, and Robert McGuire. Offload

Compiler Runtime for the Intel (R) Xeon Phi Coprocessor. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1213–1225. IEEE, 2013.

[37] Peter Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011.

[38] Franco P Preparatat and Michael Ian Shamos. Computational Geometry: an Introduction. 1985.

[39] Rezaur Rahman. Xeon Phi Vector Architecture and Instruction Set. In *Intel (R) Xeon Phi (TM) Coprocessor Architecture and Tools*, pages 31–48. Springer, 2013.

[40] Guodong Rong and Tiow-Seng Tan. Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams. In *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on*, pages 176–181. IEEE, 2007.

[41] Azriel Rosenfeld and John L Pfaltz. Sequential Operations in Digital Picture Processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966.

[42] JH Saltz, Tahsin Kurc, Sharath Cholleti, Jun Kong, Carlos Moreno, Ashish Sharma, Tony Pan, EV Meir, T Mikkelsen, A Flanders, et al. Multi-scale, Integrative Study of Brain Tumor: In Silico Brain Tumor Research Center. *Annual Symposium of American Medical Informatics Association 2010 Summit on Translational Bioinformatics (AMIA-TBI 2010)*, 2010.

[43] Joel H. Saltz, George Teodoro, Tony Pan, Lee A.D. Cooper, Jun Kong, Scott Klasky, and Tahsin M. Kurc. Feature-based Analysis of Large-scale Spatio-temporal Sensor Data on Hybrid Architectures. *International Journal of High Performance Computing Applications*, 27(3):263–272, 2013.

[44] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a Many-core x86 Architecture for Visual Computing. *ACM Transactions on Graphics (TOG)*, 27(3):18, 2008.

[45] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., 1983.

[46] Frank Y Shih and Christopher C Pu. A Skeletonization Algorithm by Maxima Tracking on Euclidean Distance Transform. *Pattern Recognition*, 28(3):331–341, 1995.

[47] Pierre Soille. *Morphological Image Analysis: Principles and Applications*. Springer Science & Business Media, 2013.

[48] Stanley R Sternberg. Grayscale Morphology. *Computer vision, graphics, and image processing*, 35(3):333–355, 1986.

[49] Gao Tao, Lu Yutong, and Suo Guang. Using MIC to Accelerate a Typical Data-intensive Application: the Breadth-first Search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1117–1125. IEEE, 2013.

[50] G. Teodoro, T.M. Kurc, T. Pan, L.A.D. Cooper, Jun Kong, P. Widener, and J.H. Saltz. Accelerating Large Scale Image Analyses on Parallel, CPU-GPU Equipped Systems. In *2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 1093–1104, May 2012.

[51] G. Teodoro, T. Pan, T.M. Kurc, Jun Kong, L.A.D. Cooper, N. Podhorszki, S. Klasky, and J.H. Saltz. High-throughput Analysis of Large Microscopy Image Datasets on CPU-GPU Cluster Platforms. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 103–114, May 2013.

[52] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of GPU and CPU for Improving Performance of Compute Intensive Applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, Aug 2009.

[53] George Teodoro, Timothy D. R. Hartley, Umit Catalyurek, and Renato Ferreira. Run-time Optimizations for Replicated Dataflows on Heterogeneous Environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 13–24, New York, NY, USA, 2010. ACM.

[54] George Teodoro, Timothy D. R. Hartley, Ümit V. Çatalyürek, and Renato Ferreira. Optimizing Dataflow Applications on Heterogeneous Environments. *Cluster Computing*, 15(2):125–144, 2012.

[55] George Teodoro, Tahsin Kurc, Guilherme Andrade, Jun Kong, Renato Ferreira, and Joel Saltz. Application Performance Analysis and Efficient Execution on Systems with Multi-core CPUs, GPUs and MICs: A Case Study with Microscopy Image Analysis. *International Journal of High Performance Computing Applications*, 2015.

[56] George Teodoro, Tahsin Kurc, Guilherme Andrade, Jun Kong, Renato Ferreira, and Joel Saltz. Performance Analysis and Efficient Execution on Systems with multi-core CPUs, GPUs and MICs. *arXiv preprint arXiv:1505.03819*, 2015.

[57] George Teodoro, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz. Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1063–1072. IEEE, 2014.

[58] George Teodoro, Tony Pan, Tahsin Kurc, Jun Kong, Lee Cooper, Scott Klasky, and Joel Saltz. Region templates: Data representation and management for high-throughput image analysis. *Parallel Computing*, 40(10):589 – 610, 2014.

[59] George Teodoro, Tony Pan, Tahsin M Kurc, Lee Cooper, Jun Kong, and Joel H Saltz. A Fast Parallel Implementation of Queue-based Morphological Reconstruction Using GPUs. *Emory University, Center for Comprehensive Informatics Technical Report CCI-TR-2012-2*, 2012.

[60] George Teodoro, Tony Pan, Tahsin M Kurc, Jun Kong, Lee AD Cooper, and Joel H Saltz. Efficient Irregular Wavefront Propagation Algorithms on Hybrid CPU–GPU Machines. *Parallel computing*, 39(4):189–211, 2013.

[61] Panagiotis Tzionas, Ph Tsalides, and Adonios Thanailakis. A Parallel Skeletonization Algorithm Based on Two-dimensional Dellular Automata and its VLSI Implementation. *Real-Time Imaging*, 1(2):105–117, 1995.

[62] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[63] Luc Vincent. Exact Euclidean Distance Function by Chain Propagations. In *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on*, pages 520–525. IEEE, 1991.

[64] Luc Vincent. Morphological Algorithms. *OPTICAL ENGINEERING-NEW YORK-MARCEL DEKKER INCORPORATED-*, 34:255–255, 1992.

[65] Luc Vincent. Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms. *Image Processing, IEEE Transactions on*, 2(2):176–201, 1993.

[66] Luc Vincent and Pierre Soille. Watersheds in Digital Spaces: an Efficient Algorithm Based on Immersion Simulations. *IEEE transactions on pattern analysis and machine intelligence*, 13(6):583–598, 1991.

[67] Luc M Vincent. Efficient Computation of Various types of Skeletons. In *Medical Imaging V: Image Processing*, pages 297–311. International Society for Optics and Photonics, 1991.

[68] A von WANGENHEIM. Seminário Introdução à Visão Computacional. *Visão Computacional–Aldon von Wangenheim's HomePage*, 2001.

[69] Nicholas Wilt. *The Cuda Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[70] Q-Z Ye. The Signed Euclidean Distance Transform and its Applications. In *Pattern Recognition, 1988., 9th International Conference on*, pages 495–499. IEEE, 1988.

[71] FA Zampirolli. Transformada de Distância por Morfologia Matemática. *Doutor, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computaçao, Campinas, SP, Brasil*, 2003.