3
Fundamentos teóricos

Este capítulo aborda os fundamentos teóricos importantes para o entendimento do algoritmo de segmentação desenvolvido nesta dissertação e exposto no capítulo 4. São apresentados conceitos sobre segmentação de imagens, o modelo de segmentação utilizado como base, uma breve introdução sobre computação paralela e a arquitetura básica de uma GPU.

3.1. Segmentação de imagens

A segmentação de imagens implica na divisão ou separação da imagem em regiões cujos *pixels* compartilham alguma propriedade. (Gonzalez & Woods, 2007). Pode-se afirmar também que a segmentação tem como objetivo gerar regiões que representem objetos de interesse (Russ, 1998). As técnicas de segmentação são em geral agrupadas em duas categorias principais: à primeira, pertencem as técnicas baseadas na detecção de descontinuidades e, à segunda, as baseadas em similaridade.

Técnicas baseadas em descontinuidades pressupõem a existência de uma variação brusca de intensidade nas proximidades da fronteira dos segmentos. Já as técnicas baseadas em similaridade admitem que pixels no interior dos segmentos são semelhantes, de acordo com algum critério, pré-estabelecido. A seguir, são descritas alguns exemplos de técnicas de cada uma destas classes.

3.1.1. Segmentação por descontinuidades

A segmentação baseada nas descontinuidades consiste em detectar alterações abruptas na intensidade dos *pixels*. As descontinuidades são detectadas calculando-se a derivada espacial em duas dimensões. Nas chamadas "arestas", onde se encontram as fronteiras dos segmentos, as derivadas costumam ter valores extremos (muito positivo ou muito negativo).

O cálculo das derivadas é realizado através do uso de filtros lineares, que utilizam máscaras para realizar a varredura da imagem. As abordagens mais comuns são:

- Utilização da 1ª derivada - Magnitude do gradiente elevada é evidência de aresta.
- Utilização da 2ª derivada - Cruzamento em zero é evidência de aresta.

3.1.2. Segmentação por similaridades

A segmentação por detecção de similaridades se baseia na análise do interior dos objetos. Presume-se que os *pixels* que formam um objeto têm determinadas propriedades semelhantes, enquanto *pixels* de objetos distintos possuem diferentes propriedades.

Nesta categoria estão presentes as técnicas de limiarização e as baseadas em crescimento de regiões, descritas a seguir.

Limiarização

A limiarização é uma das técnicas mais simples de segmentação e classifica os *pixels* de uma imagem de acordo com sua intensidade obedecendo a um ou mais limiares. A segmentação é realizada a partir do histograma da imagem, onde os *pixels* são agrupados conforme sua intensidade, obedecendo ao limiar ou limiares propostos.

Segmentação por regiões

Os métodos de segmentação por regiões visam formar grupos de *pixels* adjacentes, segundo um critério de similaridade, no intuito de gerar ao final do processo regiões homogêneas.

Desta forma, regiões são agrupadas ou divididas baseadas nas suas características de cor, textura ou forma de maneira a reunir *pixels* que possuam características semelhantes.

Dentre os métodos desta categoria destacam-se as técnicas de crescimento de regiões (*region growing)*, divisão e união de regiões (*split and merge*) e divisor de águas (*watershed*). Estas três técnicas são descritas sucintamente a seguir.

a) Crescimento de regiões (region growing)

Neste método, um conjunto de *pixels* chamados sementes é escolhido inicialmente. A partir deste, as regiões crescem através da agregação de outros *pixels* ou sub-regiões vizinhas que atendam a determinado critério de homogeneidade. As sementes podem ser selecionadas de maneira aleatória, determinística ou pelo usuário (Pedrini & Schwartz, 2008).

Algumas dificuldades encontradas neste método são a definição do critério de parada para o crescimento das regiões, a dependência dos resultados da escolha das sementes e a determinação das características apropriadas para utilização como critério de homogeneidade.

Uma das principais propostas referentes a esta solução pode ser encontrada em (Baatz & Schäpe, 2000). Esta técnica foi usada como base para deste trabalho e será discutida em detalhes mais adiante.

b) Divisão e união (*split and merge*)

Duas fases são bem definidas nesse tipo de abordagem: a primeira corresponde a uma divisão de regiões não homogêneas em partes comparativamente mais homogêneas; e a segunda a um processo iterativo de união de regiões adjacentes similares (Horowitz & Pavlidis, 1974).

A etapa de divisão se inicia dividindo a região em 4 quadrantes. Verifica-se se o interior de cada quadrante atende um determinado critério de homogeneidade. Em caso afirmativo a etapa de divisão cessa para aquele quadrante. Em caso negativo, cada quadrante é novamente dividido em quadrantes e o processo se repete de forma recursiva. Uma breve ilustração desta etapa pode ser verificada na figura 3.1.

O resultado desta etapa é a entrada para a etapa de união, que na realidade é praticamente um algoritmo de crescimento de regiões a menos de pequenas modificações. A figura 3.2 ilustra esta etapa.

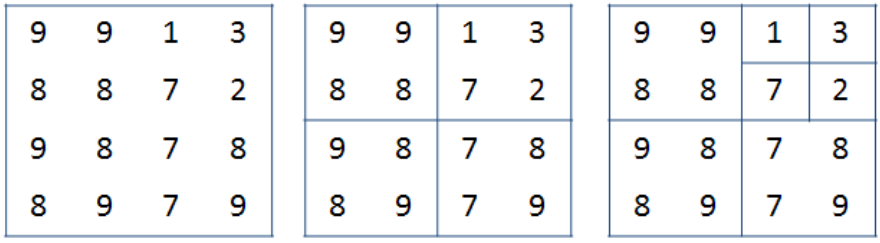

Figura 3.1 Etapa de divisão ao longo das iterações

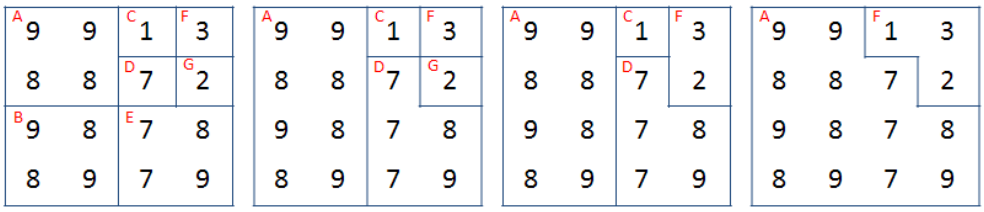

Figura 3.2. Etapa de união ao longo das iterações

c) Divisor de águas (*watershed*)

Nesse método, a imagem é vista como uma superfície topográfica composta de vales e montanhas onde o valor da magnitude do gradiente relativo à intensidade dos *pixels* corresponde à altitude dos pontos (Beucher & Meyer, 1993).

Um processo simula a inundação dessa superfície a partir dos mínimos locais, formando bacias ou poças. Quando as águas de duas bacias vizinhas estão na iminência de fazerem contato para transformarem-se em uma única bacia, uma linha de contenção é criada, formando os contornos e por fim as bordas de cada segmento resultante (Pedrini & Schwartz, 2008).

Assim, pode-se dizer que os segmentos são formados por regiões que, partindo de um mínimo local, formam uma bacia hidrográfica (como exibido na Figura 3.3).

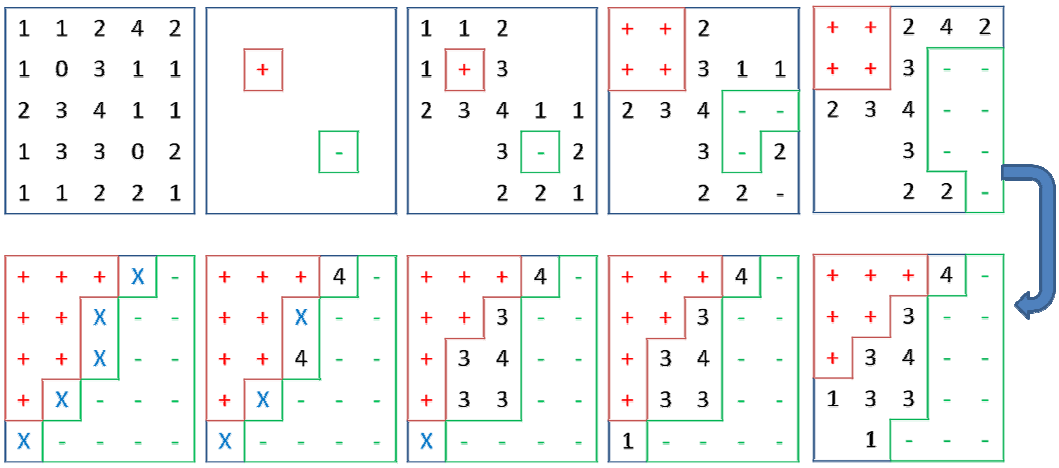

Figura 3.3. Esquema da segmentação por divisor de águas

3.2.Modelo de segmentação utilizado

A escolha do método de segmentação mais adequado a cada problema de interpretação é uma tarefa importante. Esta dissertação está voltada às aplicações de sensoriamento remoto, área em que os algoritmos de segmentação por crescimento de regiões são os mais usados entre todas as categorias referidas na seção anterior.

O algoritmo proposto nesta dissertação é uma versão paralela de um algoritmo de segmentação por crescimento de regiões amplamente utilizado pela comunidade de sensoriamento remoto, proposto inicialmente em (Baatz & Schäpe, 2000). Este método foi considerado como um dos algoritmos mais eficazes de segmentação em estudos recentes (Neubert et al., 2008). Além disso, algoritmos baseados neste método estão disponíveis como operadores na plataforma InterIMAGE (InterIMAGE, 2011) e no sistema Definiens (Definiens, 2008).

Este modelo consiste em um método iterativo, de otimização local, que visa minimizar a heterogeneidade média dos objetos resultantes da imagem. Todos os *pixels* da imagem são primeiramente considerados como sementes ou segmentos iniciais e a cada passo ou iteração calcula-se a diferença de heterogeneidade entre cada par de segmentos adjacentes e a homogeneidade do segmento que resultaria da sua fusão.

Esta diferença, chamada custo de fusão, deve ser inferior a certo limiar, chamado escala, para permitir a agregação destes segmentos. O processo se repete até que não seja possível realizar mais nenhuma fusão.

O custo de fusão (f) é definido pela soma ponderada de uma componente relativa à heterogeneidade espectral (h_{cor}) e outra relativa à heterogeneidade espacial (h_{forma}). A importância relativa destas componentes é definida por um peso (w_{cor}), conforme indicado na Equação 3.2.

$$f = w_{cor}.h_{cor} + (1 - w_{cor}).h_{forma} \tag{3.2}$$

A heterogeneidade espectral, por sua vez, é dada pela Equação 3.3, onde $obj1$ refere-se ao segmento selecionado para crescer, $obj2$ a um de seus segmentos adjacentes e $obj12$ ao segmento resultante da eventual fusão entre estes. Cada banda da imagem, indexada aqui pelo símbolo c, é levada em consideração, sendo multiplicada por um fator w_c que expressa a importância relativa de cada uma. A parcela referente a cada segmento na fórmula é baseada no produto de sua área (n) pelo desvio padrão dos valores de seus *pixels* em cada banda (σ_c .)

$$h_{cor} = \sum_c w_c \left(n_{obj12}.\sigma_c^{obj12} - \left(n_{obj1}.\sigma_c^{obj1} + n_{obj2}.\sigma_c^{obj2}\right)\right) \tag{3.3}$$

A heterogeneidade espacial é constituída por duas componentes de forma: uma referente à compacidade (h_{comp}), definida na Equação 3.4, e outra relativa à suavidade (h_{svd}), definida na Equação 3.5.

$$h_{comp} = n_{obj3}.Comp_{obj3} - \left(n_{obj1}.Comp_{obj1} + n_{obj2}.Comp_{obj2}\right) \tag{3.4}$$

$$h_{svd} = n_{obj3}.Svd_{obj3} - \left(n_{obj1}.Svd_{obj1} + n_{obj2}.Svd_{obj2}\right) \tag{3.5}$$

A medida de compacidade $Comp$ de um segmento (Equação 3.6) é dada pela razão entre o comprimento de borda b e a raiz quadrada da área n.

$$Comp = \frac{b}{\sqrt{n}} \tag{3.6}$$

Já a suavidade Svd (Equação 3.7) se refere à razão entre o comprimento de borda b do segmento e o comprimento de borda do seu retângulo envolvente mínimo $bbox$ (*bounding box*). Como pode ser visto na Equação 3.8, existe um peso w_{comp} que define o grau de importância relativa entre compacidade e suavidade.

$$Svd = \frac{b}{\sqrt{bbox}} \tag{3.7}$$

$$h_{forma} = w_{comp}.h_{comp} + \left(1 - w_{comp}\right).h_{svd} \tag{3.8}$$

O algoritmo possui, portanto, um critério de heterogeneidade ajustável. Parâmetros, como a relevância de cada banda espectral e a importância relativa entre forma e cor, e entre compacidade e suavidade podem ser regulados com a finalidade de se alcançar uma melhor segmentação. Além disso, a definição do custo máximo de fusão, denominado parâmetro de escala (s), influencia diretamente no tamanho dos segmentos gerados.

3.2.1. Cálculo dos atributos de forma

O algoritmo apresentado nesta seção utiliza dois atributos com a finalidade de mensurar a heterogeneidade espacial dos objetos. Cada um destes atributos utiliza diferentes características de forma do objeto. Nesta seção, será descrito o significado de cada uma destas características.

Entende-se por comprimento de borda (b) o número total de arestas que fazem parte da fronteira, ou seja, que estão no limite do objeto. A Figura 3.4 exibe um objeto em uma imagem. Os *pixels* em verde são *pixels* localizados no interior do objeto e os *pixels* em azul são *pixels* de borda. O contorno vermelho representa a fronteira do objeto. O valor escrito em cada *pixel* de borda é o número de arestas de fronteira que cada *pixel* de borda possui. A soma destes valores resulta no comprimento de borda do objeto. Vale ressaltar que alguns autores consideram esta medida como uma possível medida para o perímetro. (Russ, 1998).

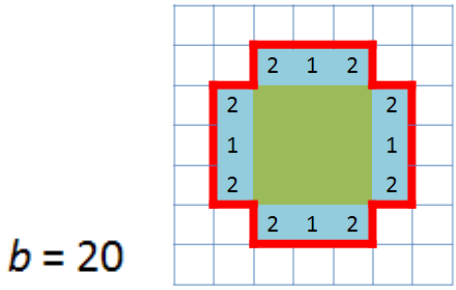

Figura 3.4. Um objeto com comprimento de borda igual a 20 representado em vermelho.

A área de um objeto (n) é dada pela quantidade de *pixels* que ele possui. A Figura 3.5 ilustra a área de um objeto. Os *pixels* em azul fazem parte do objeto e estão demarcados com o valor 1. Desta forma, chega-se ao valor da área contando o número de *pixels* do objeto ou totalizando os valores.

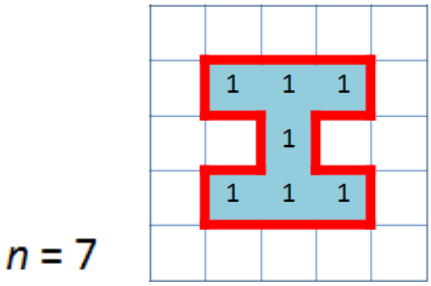

Figura 3.5. Um objeto com área igual a 7.

O retângulo envolvente mínimo de um objeto pode ser calculado de algumas maneiras. Nesta dissertação, este cálculo é feito a partir de um método de ajuste de elipse que se baseia nas estatísticas da distribuição espacial do objeto.

O ajuste de elipse pode ser efetuado por intermédio da matriz de covariância S. Esta matriz, definida pela Equação 3.9, é formada pelas variâncias $Var(X)$ e $Var(Y)$ (Equações 3.10 e 3.11) e pela covariância $Cov(X,Y)$ (Equação 3.12) das coordenadas dos *pixels* do objeto, a partir das seguintes definições:

- X é um vetor com as coordenadas x de todos os *pixels* do objeto;
- Y é um vetor com as coordenadas y de todos os *pixels* do objeto;
- $\bar{x}$ é a média das coordenadas x dos *pixels* do objeto; e
- $\bar{y}$ é a média das coordenadas y dos pixels do objeto.

$$S = \begin{bmatrix} Var(X) & Cov(X,Y) \\ Cov(X,Y) & Var(Y) \end{bmatrix} \tag{3.9}$$

$$Var(X) = \frac{1}{n}\sum_{X}(x - \bar{x})^2 \tag{3.10}$$

$$Var(Y) = \frac{1}{n}\sum_{y}(y - \bar{y})^2 \tag{3.11}$$

$$Cov(X,Y) = \frac{1}{n}\sum_{X}\sum_{Y}(x - \bar{x})\,(y - \bar{y}) \tag{3.12}$$

Através do cálculo dos autovalores da matriz de covariância é possível obter os eixos da elipse ajustada. Nesta dissertação, o eixo maior e o eixo menor serão chamados respectivamente diâmetro máximo $dmax$ e diâmetro mínimo $dmin$. As equações 3.13 e 3.14 definem estes valores.

$$dmax = \sqrt{8 \cdot (Var(X) + Var(Y) + \sqrt{(Var(X) - Var(Y))^2 + 4 \cdot Cov(X,Y)^2})} \tag{3.13}$$

$$dmin = \sqrt{8 \cdot (Var(X) + Var(Y) - \sqrt{(Var(X) - Var(Y))^2 + 4 \cdot Cov(X,Y)^2})} \tag{3.14}$$

O retângulo envolvente mínimo é então circunscrito na elipse ajustada e possui, portanto, seus lados equivalentes aos eixos desta elipse. O comprimento de borda do retângulo envolvente ($bbox$) é dado pela equação 3.15 e a área do retângulo envolvente ($nbox$) pela equação 3.16. A Figura 3.6 mostra um objeto na imagem (em azul) e seu retângulo envolvente mínimo (em preto) a partir da elipse ajustada (em verde).

$$bbox = 2.dmax + 2.dmin \tag{3.15}$$

$$nbox = dmax\,.\,dmin \tag{3.16}$$

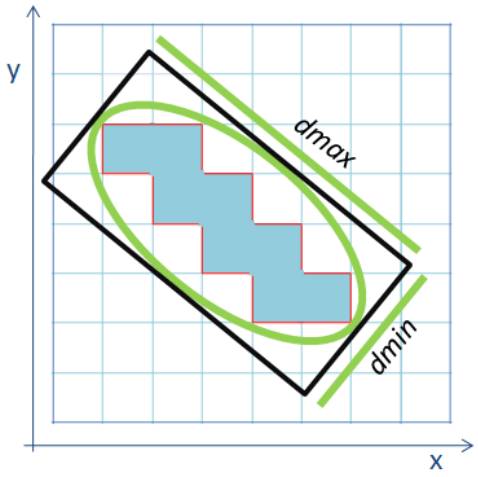

Figura 3.6. Definição do retângulo envolvente mínimo de um objeto através da elipse ajustada.

3.2.2. Heurísticas de decisão para fusão de segmentos

Após realizar o cálculo do custo de fusão entre um objeto e seus vizinhos, é necessária alguma heurística que defina o procedimento que une estes objetos. Quatro alternativas são apresentadas em (Baatz & Schäpe, 2000) e descritas a seguir:

- O procedimento mais simples, denominado neste trabalho de "vizinho aleatório" (*fitting*), admite a fusão de um segmento com qualquer um de seus segmentos vizinhos que atenda ao critério de heterogeneidade.
- O método do "melhor vizinho" (*best fitting*) estabelece que um segmento deve ser fundido com o segmento vizinho que, além de atender ao critério de heterogeneidade, resulta no menor aumento de heterogeneidade entre todos os vizinhos.
- A decisão pelo "melhor vizinho mútuo" (l*ocal mutual best fitting*) é atendida quando o melhor vizinho (segmento vizinho que resulta no menor aumento de heterogeneidade) de um dado segmento tem neste segmento também o seu melhor vizinho. Ou seja, dado um segmento A, cujo melhor vizinho seja o segmento B, é necessário que A também seja o melhor vizinho do segmento B para que A e B possam ser fundidos num novo e único segmento.

- A última heurística, denominada "melhor vizinho mútuo global" (g*lobal mutual best fitting*), determina que somente o par de melhores vizinhos mútuos que resulte no menor aumento de heterogeneidade poderá ser fundido.

Em todas as alternativas mencionadas, a fusão só se consuma se o aumento de heterogeneidade decorrente da fusão não exceder o valor atribuído ao parâmetro de escala (s).

3.3. Ajuste de Parâmetros

Uma vez selecionado o algoritmo de segmentação, faz-se necessário definir o valor dos seus parâmetros de acordo com as características da imagem e com as classes de objetos a serem reconhecidas na aplicação. Contudo, a correspondência entre os valores desses parâmetros e o resultado da segmentação nem sempre é óbvia.

Usualmente, a seleção dos valores ótimos dos parâmetros para uma determinada aplicação é realizada através de um trabalho exaustivo de tentativa e erro. Métodos alternativos têm sido empregados para este fim, como a utilização de algoritmos genéticos (Bhanu et al., 1995).

3.3.1. Algoritmos genéticos

Algoritmos genéticos são métodos estocásticos de busca de soluções para problemas de otimização. Inspirados na teoria da evolução das espécies de Charles Darwin (Darwin, 1859), estes algoritmos consistem em um processo evolucionário que visa maximizar ou minimizar determinada função de aptidão.

Este processo ocorre de maneira iterativa através das gerações de indivíduos, onde cada indivíduo pode ser descrito como uma possível solução para o problema de otimização.

Em cada geração, o conjunto de indivíduos existentes é denominado população e cada indivíduo é avaliado através do cálculo de uma função de aptidão. Esta função de aptidão é a responsável por indicar numericamente a capacidade de determinado indivíduo para resolver um dado problema (Michalewicz, 1996). Os indivíduos considerados como menos aptos são então

rejeitados, enquanto novos indivíduos são gerados por meio da reprodução dos mais aptos com o auxílio de operadores genéricos.

Neste caso específico, o objetivo é encontrar os valores ótimos dos parâmetros de segmentação. Desta forma, cada indivíduo representa um conjunto de valores dos parâmetros de segmentação e a função de aptidão é definida por uma função que expressa o nível de disparidade entre o resultado produzido pela segmentação e um conjunto de segmentos definidos como referência.

3.3.2. Função de disparidade

A fim de se comparar o resultado obtido por determinada segmentação com uma segmentação tida como referência, utiliza-se, como mencionado, uma função de disparidade. Quando a função de disparidade tem seu valor correspondente à zero significa que a segmentação resultante é idêntica à referência.

Neste trabalho, a função de disparidade utilizada foi a RBSB (*Reference Bounded Segments Booster*) (Feitosa et al., 2006; Costa et al., 2008), representada pela Equação 3.2 a partir das seguintes definições:

- S_i é o conjunto formado pelos *pixels* do i-ésimo segmento pertencente a um dado conjunto de referências S;
- $O(P)_i$ é o conjunto de *pixels* constituído pelo segmento com maior interseção com S_i e resultante da segmentação utilizando um conjunto de parâmetros P;
- a quantidade de falsos positivos fp_i é o número de *pixels* em $O(P)_i$ que não pertencem a S_i;
- o número de falsos negativos fn_i é formado pelos *pixels* em S_i que não pertencem a $O(P)_i$;
- a quantidade de verdadeiros positivos vp_i é constituída pelos *pixels* em $O(P)_i$ que pertencem a S_i; e
- o número de segmentos existentes no conjunto S é dado por n.

$$F(S,P) = \frac{1}{n}\sum_{i=1}^{n}\frac{fp_i + fn_i}{vp_i + fn_i} \tag{3.2}$$

3.4. Computação paralela

A necessidade de lidar com problemas cada vez mais complexos tem gerado uma crescente demanda por desempenho na comunidade técnico-científica. São necessários computadores cada vez mais potentes de forma a viabilizar a execução de certas aplicações computacionalmente custosas.

A tecnologia dos processadores evoluiu bastante nas últimas décadas de forma a alcançar um aumento considerável de desempenho. Contudo, restrições físicas como a dissipação de energia fizeram com que estes avanços atingissem um limite (Asanovic et al., 2009). Para contornar esta barreira houve uma mudança no paradigma de construção de arquiteturas de computadores e recentemente na construção de processadores: optou-se por investir em uma abordagem baseada na exploração do processamento paralelo.

O processamento paralelo consiste na utilização simultânea de diferentes recursos computacionais, como processadores, para a execução de trechos independentes de uma determinada tarefa. Atualmente, o grande progresso tecnológico neste setor resultou em um barateamento de arquiteturas específicas que comportam o paralelismo.

A exploração do paralelismo pode ser realizada em níveis diferentes. Tanto em máquinas paralelas como dentro do próprio processador.

3.4.1. Máquinas paralelas

As arquiteturas de computadores podem ser genericamente divididas em quatro categorias de acordo com a classificação apresentada por Flynn (Flynn, 1966):

- SISD (**S**ingle **I**nstruction Stream, **S**ingle **D**ata Stream)
- SIMD (**S**ingle **I**nstruction Stream, **M**ultiple **D**ata Streams)
- MISD (**M**ultiple **I**nstruction Streams, **S**ingle **D**ata Stream)
- MIMD (**M**ultiple **I**nstruction Streams, **M**ultiple **D**ata Streams)

As arquiteturas SISD possuem um fluxo único de instruções operando sobre um único conjunto de dados. Uma única unidade de controle busca uma única instrução da memória e transmite sinais de controle para um único elemento

processador que irá operar sobre um único fluxo de dados. Esta categoria representa os modelos convencionais de computadores seqüenciais (arquitetura de Von Neumann) e é ilustrada pela Figura 3.7.

Figura 3.7. Esquema da arquitetura SISD

Em arquiteturas SIMD um fluxo único de instruções opera sobre múltiplos conjuntos de dados. Existe uma única unidade de controle e múltiplos elementos processadores. A unidade de controle busca uma instrução da memória e envia sinais de controle para todos os elementos de processamento. Os elementos processadores executam de forma síncrona a mesma instrução, porém com diferentes conjuntos de dados. A figura 3.8 demonstra esta arquitetura.

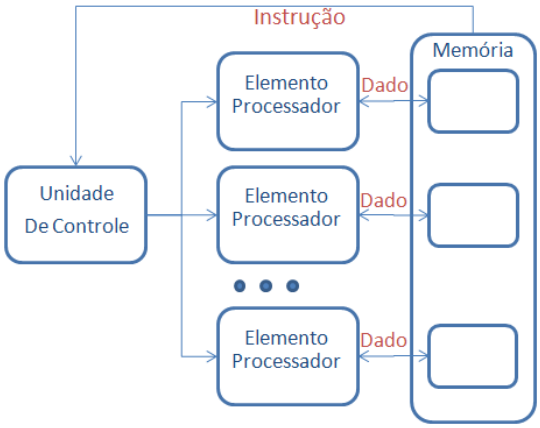

Figura 3.8. Esquema da arquitetura SIMD

As arquiteturas MISD são responsáveis por executar múltiplas instruções sobre um único fluxo de dados. Desta forma, múltiplos elementos de processamento executam diferentes instruções sobre um único conjunto de dados. Esta classe é considerada sem representantes por muitos cientistas da computação (Weper et al., 1999). Um esquema deste modelo pode ser visto na Figura 3.7.

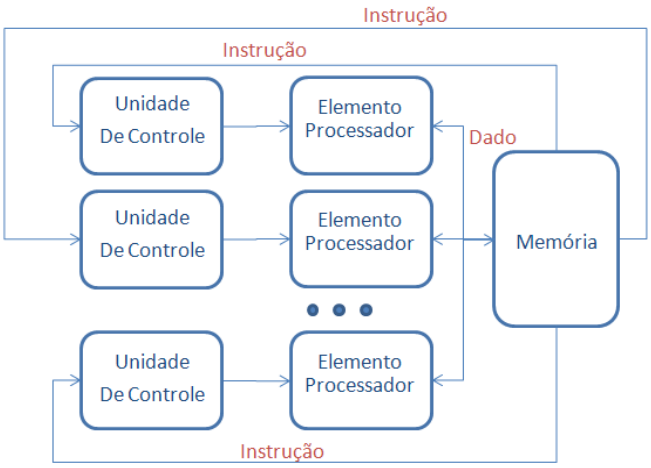

Figura 3.9. Esquema da arquitetura MISD

A arquitetura MIMD é caracterizada pela presença de múltiplas instruções operando sobre diferentes conjuntos de dados. Desta maneira, múltiplos processadores atuam de forma independente executando suas respectivas instruções para processar um conjunto de dados a eles alocado. Nesta categoria, existe uma unidade de controle independente para cada elemento processador. A Figura 3.10 representa este modelo.

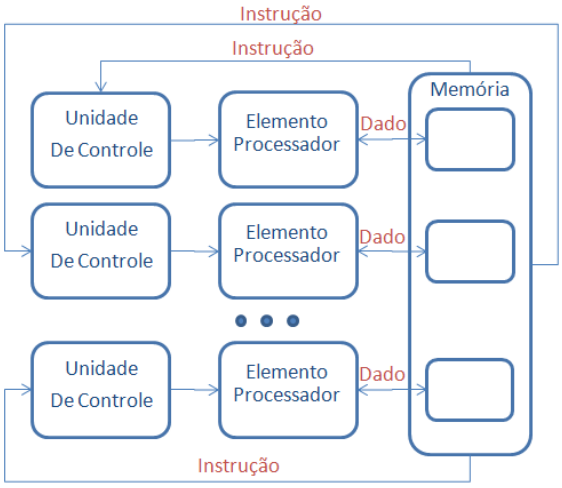

Figura 3.10. Esquema da arquitetura MIMD

As arquiteturas paralelas mais comuns são baseadas no modelo MIMD. Estas oferecem maior flexibilidade para a execução de programas paralelos devido aos elementos de processamento operarem de modo assíncrono (Almasi & Gottlieb, 1994). Podemos citar principalmente a existência dos *clusters* e *grids*.

Clusters são compostos da agregação de uma série de computadores homogêneos e geograficamente centralizados, denominados nós e conectados a uma rede local de alta velocidade de forma a distribuir o processamento entre estes.

Grids são conjuntos de computadores distribuídos geograficamente com a finalidade de compartilhar recursos através de uma conexão de rede externa como, por exemplo, a internet.

3.4.2. Processadores paralelos

Nesta categoria destacam-se os processadores de múltiplos núcleos (*multicore*). A forma de trabalho desses processadores é realizada através da utilização de linhas de execução ou *threads*, onde cada uma destas é processada simultaneamente pelos diferentes núcleos. A execução dos núcleos é independente e utiliza memória compartilhada, o que facilita o controle de execução do processamento paralelo.

Recentemente, as unidades de processamento gráfico (GPUs) se disseminaram amplamente e se tornaram uma opção com alto grau de paralelismo. Estas também trabalham com divisão por *threads* em diferentes núcleos localizados em processadores com uma arquitetura diferenciada. Neste caso, existe uma dependência na execução destes núcleos.

Nesta dissertação o interesse recai sobre a utilização de GPUs. A seguir serão descritos mais detalhes deste tipo de *hardware*.

3.5. GPUs

Uma Unidade de Processamento Gráfico (GPU – *Graphic Processing Unit*) é um processador produzido inicialmente para realizar cálculos e tarefas relacionadas ao processamento de gráficos em tempo real.

Motivado pelo amplo mercado de jogos, as placas gráficas foram ficando cada vez mais velozes. Tornaram-se então um componente capaz de oferecer um alto grau de paralelismo, através de múltiplas threads e múltiplos processadores, atingindo assim, um enorme poder computacional.

Se o enfoque a princípio era a execução de jogos, logo se notou o potencial das GPUs para outros tipos de aplicação que demandam grande capacidade de

processamento. A GPU passou a ser programável através dos chamados *shaders*. Estes programas permitem a codificação de algumas etapas específicas existentes no *pipeline* gráfico da GPU de forma a substituir o código padrão existente. Cada etapa programável, no entanto, é independente e requer um *shader* diferente.

Este tipo de programação era ainda bastante limitado e requeria o conhecimento do processamento gráfico por parte do programador. Além disto, os elementos de processamento existentes na GPU eram exclusivos para o processamento de cada etapa do *pipeline*.

Surgem então adaptações na arquitetura de forma a unificar os elementos processadores e permitir que estes atuem sobre qualquer tipo de *shader*. Desta forma, recursos que antes ficavam ociosos devido à utilização de um *shader* específico, se tornavam disponíveis de forma dinâmica e transparente ao programador.

A Computação de Propósito Geral em GPUs (GPGPU) se refere ao aproveitamento da capacidade de processamento das unidades de processamento gráfico para aplicações de propósito geral, ou seja, na utilização de GPUs para fins não-gráficos. Este tipo de computação passou a ser tão utilizado que os fabricantes de GPUs desenvolveram novas linguagens especificamente para este fim. Surge o CTM ("Close to Metal") para placas da AMD e o CUDA ("Compute Unified Device Architecture") da Nvidia.

A seguir, é descrita de forma objetiva a arquitetura básica de GPUs da Nvidia, bem como o modelo de programação CUDA. Maiores informações podem ser obtidas em (Nvidia, 2010).

3.5.1. Arquitetura de GPU

Uma Unidade de Processamento Gráfico ou GPU pode ser vista como um processador de diversos núcleos cuja capacidade computacional provém de uma arquitetura altamente paralela.

A organização de uma GPU é composta de um conjunto de multiprocessadores (*streaming multiprocessors*). Cada um destes consiste basicamente de um conjunto de núcleos (*cores*), chamados de *stream processors*, de unidades para cálculos de funções especiais (*special function units*) e de uma memória compartilhada. O número de multiprocessadores e de núcleos em cada

multiprocessador de uma GPU depende da arquitetura e do modelo da GPU. GPUs modernas podem possuir no até 512 *núcleos* no total. A Figura 3.11 ilustra um esquema básico deste multiprocessador.

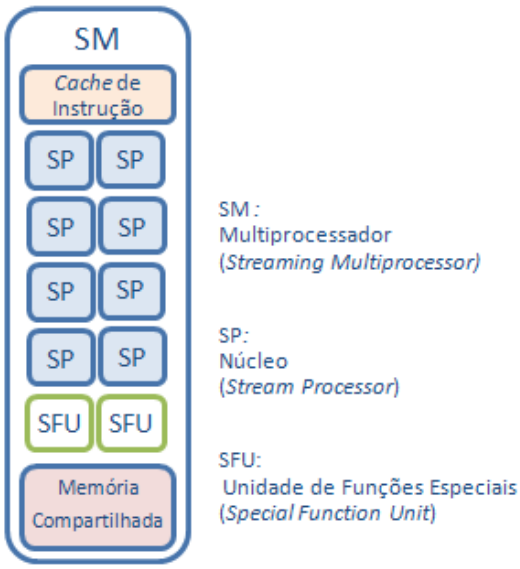

Figura 3.11. Esquema genérico de um multiprocessador da GPU (SM)

A arquitetura das GPUs da Nvidia é denominada SIMT (*Single Instruction Multiple Thread*). Esta arquitetura é semelhante à SIMD, pois uma única instrução controla múltiplos elementos de processamento. A diferença fundamental é que as instruções SIMT especificam a execução e comportamento de ramificação de uma única *thread* (Nvidia, 2010).

Cada multiprocessador SIMT é responsável por criar, gerenciar e executar *threads* em grupos de 32 *threads* paralelas. Este grupo de *threads* é chamado de *warp*. Dentro de um *warp* todas as *threads* executam necessariamente a mesma instrução. Quando uma das *threads* diverge das demais no processamento, devido a, por exemplo, um desvio no código, é gerada uma ramificação. Um *warp* executa cada ramificação em seqüência. As *threads* que não pertencem àquela ramificação são desabilitadas. Quando todas as ramificações de um *warp* são processadas, suas *threads* convergem novamente para a mesma instrução.

Para um bom aproveitamento da capacidade de processamento paralelo das GPUs é necessário que haja uma distribuição de carga de trabalho pelos conjuntos de *threads*. Além disso, como o acesso à memória global possui uma elevada latência, é importante que a execução das *threads* possua uma baixa

granularidade. Desta forma, os algoritmos para GPUs tendem a possuir um grande número de *threads* que executam tarefas menores a fim de otimizar o desempenho.

3.5.2. Programação em GPUs

O modelo de programação da Nvidia, CUDA, foi criado para o desenvolvimento de aplicações para esta plataforma. CUDA permite que o programador crie funções especiais em C que são executadas em paralelo em *threads* na GPU.

A execução na GPU é baseada numa organização hierárquica de grade de blocos de *threads*, conforme ilustrado na Figura 3.12. A grade (*grid*) é a estrutura máxima que pode ser subdivida em blocos (*blocks*), que, por sua vez, contêm uma quantidade de *threads*. Tanto as *threads* quanto os blocos podem possuir até três dimensões, com a finalidade de melhor representarem um vetor, uma matriz ou uma estrutura volumétrica. Para cada *thread* de um bloco é definido um índice (*thread ID*) para maior facilidade de acesso e mapeamento. O mesmo ocorre para cada bloco existente na grade (*block ID*).

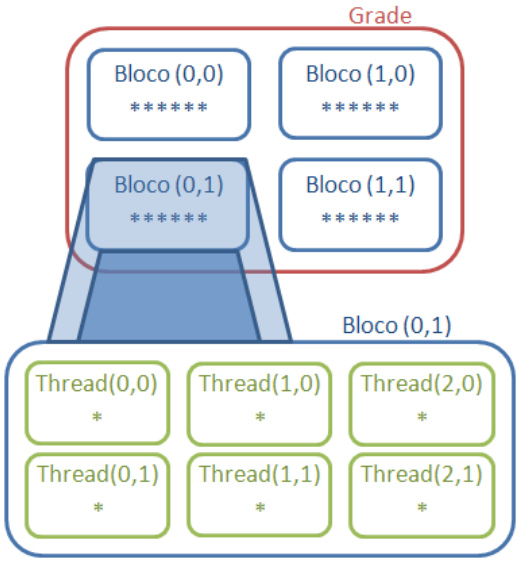

Figura 3.12. Hierarquia de grade de blocos de *threads* presente na GPU

Quando um programa é enviado para ser executado na GPU, os blocos de uma grade são enumerados e distribuídos para multiprocessadores que possuam

capacidade de execução disponível. *Threads* de um mesmo bloco são processadas em paralelo em um mesmo multiprocessador. Dependendo da capacidade, um multiprocessador pode processar mais de um bloco.

Durante a execução, as *threads* têm acesso a dados em diferentes níveis da hierarquia de memória: registradores, memória local, memória compartilhada e memória global.

A memória local é de acesso exclusivo para cada *thread. Threads* que pertencem a um mesmo bloco podem partilhar da memória compartilhada. A memória global é acessível a todas as *threads* existentes, mas seu tempo de acesso é em média 500 vezes mais longo do que o tempo de acesso à memória compartilhada e à memória local.

Além disso, há dois outros tipos de memória disponíveis para todas as *threads*: a memória constante e a memória de textura. Estas são memórias utilizadas apenas para leitura de dados dentro da GPU. A figura 3.13 ilustra a hierarquia de memória na GPU.

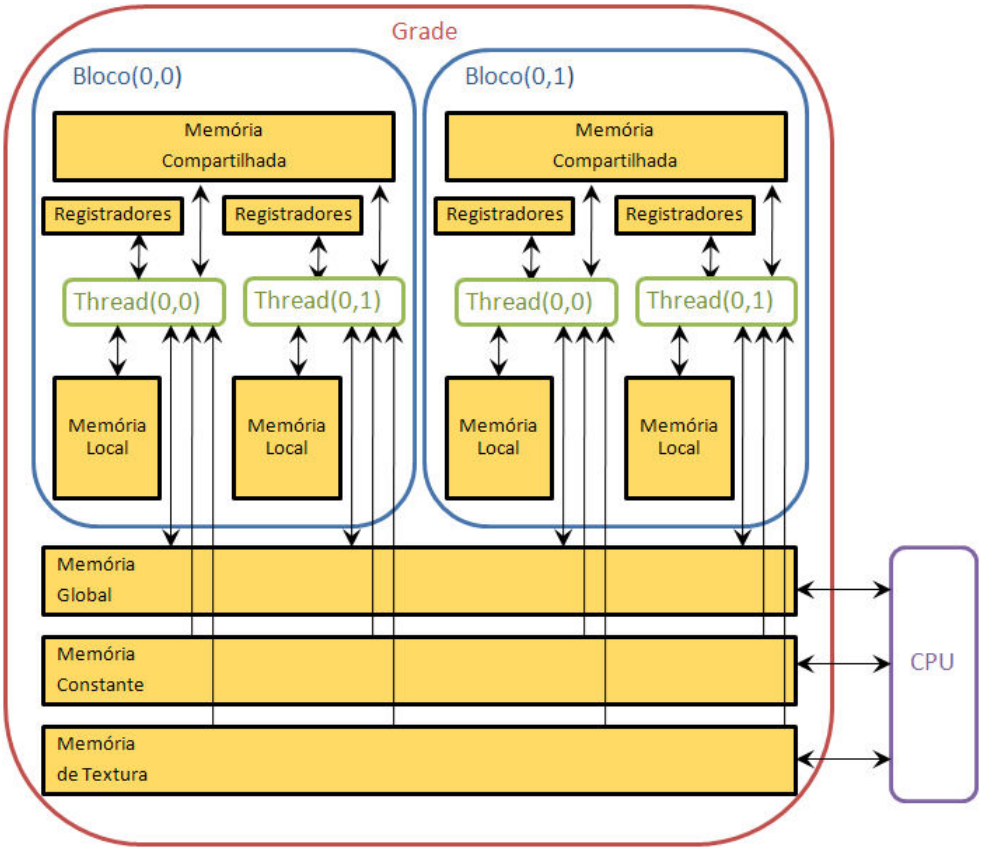

Figura 3.13. Hierarquia de memória na GPU

O processamento de programas desenvolvidos em CUDA alterna entre execuções na CPU e na GPU. O trecho de código ou função que é invocado pela CPU para ser executado na GPU é chamado de *kernel*.

Geralmente, antes de se processar na GPU, ocorre uma cópia dos dados da memória principal do computador para a memória global da GPU. Um *kernel* é então invocado com as especificações das dimensões da grade e do bloco, ou seja, a quantidade de blocos e de *threads* por bloco. Ao final, os dados são copiados de volta para a memória principal. A Figura 3.14 ilustra essa troca de dispositivos e o fluxo de execução.

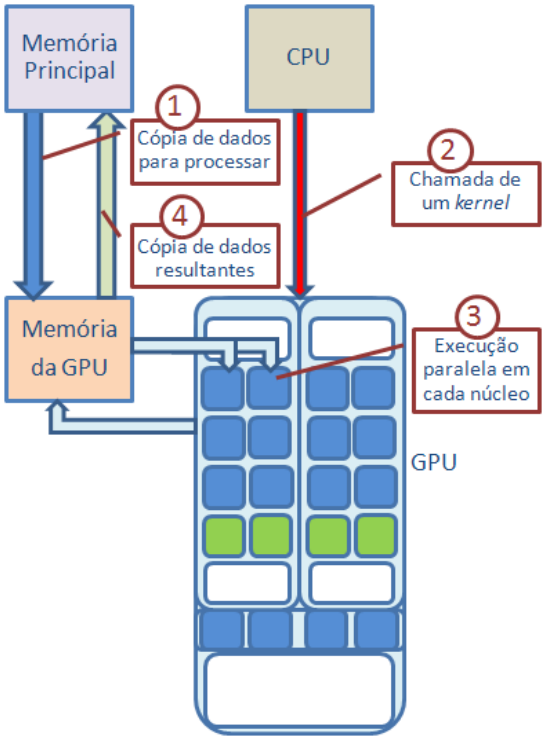

Figura 3.14. Esquema de processamento alternado entre CPU (seqüencial) e GPU (paralelo)

É permitida a utilização de funções para sincronização de *threads* pertencentes a um mesmo bloco através de mecanismos de barreira. *Threads* de diferentes blocos só podem ser sincronizadas ao final da execução de um determinado *kernel*.

CUDA também oferece funções atômicas com a finalidade de realizar a escrita nas memórias compartilhada e global de forma segura. Este tipo de função também é utilizado para criação de seções críticas. As funções atômicas estão disponíveis em GPUs com capacidade de computação igual ou superior a 1.1.