4
Metodologia

Neste capítulo, é apresentado o algoritmos de segmentação utilizado no desenvolvimento das versões paralelas do segmentador. Uma descrição mais detalhada das funções principais e de modificações introduzidas também é exposta.

As duas versões paralelas do algoritmo de segmentação foram desenvolvidas em C e CUDA da NVidia. Os segmentadores gerados são múltiplataforma e podem ser executados em GPUs com arquiteturas iguais ou superiores à 1.1.

4.1. Alteração na estrutura de dados

Antes de se construir o código paralelo, foi necessário introduzir algumas alterações no programa seqüencial de modo a operar sobre estruturas de dados mais adequadas à programação em GPU.

A versão original utiliza alocação dinâmica de memória, geralmente através de listas encadeadas contendo segmentos e seus atributos, além de outras variáveis importantes para o bom funcionamento do algoritmo. Nesta etapa do estudo idealizaram-se estruturas diferentes baseadas em vetor.

Essa solução foi escolhida, pois a GPU possui uma memória privada distinta da memória principal da CPU. Desta maneira, deve-se realizar uma cópia entre as variáveis existentes na memória RAM para a memória da GPU e vice-versa, sempre que for necessário alternar a unidade que controlará a execução. Manter uma lista encadeada requereria que cada lista fosse inteiramente percorrida sempre que o chaveamento entre CPU e GPU fosse necessário, o que oneraria o processamento.

A nova estrutura de dados armazena as características de *pixels* e de segmentos, além do mapeamento entre eles na memória global da GPU. Esta estrutura consiste de um vetor cujos índices representam os identificadores (Id) dos *pixels*. São armazenadas informações sobre o *pixel,* como:

- o identificador do segmento a que pertence,
- se o *pixel* faz parte da borda do segmento e
- o identificador do *pixel* anterior e do próximo *pixel* para o mapeamento de *pixels* do segmento a que pertence.

São armazenadas também as informações sobre o segmento como:

- a área total,
- as características espectrais,
- as características espaciais,
- a identificação do melhor vizinho encontrado para fusão e
- o custo de fusão calculado.

Cada elemento deste vetor possui, portanto, dados referentes tanto ao *pixel* quanto ao segmento a que pertence, conforme ilustrado na Figura 4.1. Cabe informar que os dados referentes aos segmentos só são válidos caso o segmento ainda não tenha sido unido por outro no processo de fusão de segmentos. Um segmento é considerado válido caso o Id do *pixel* seja igual ao Id do segmento, sendo este *pixel* considerado como representante do segmento.

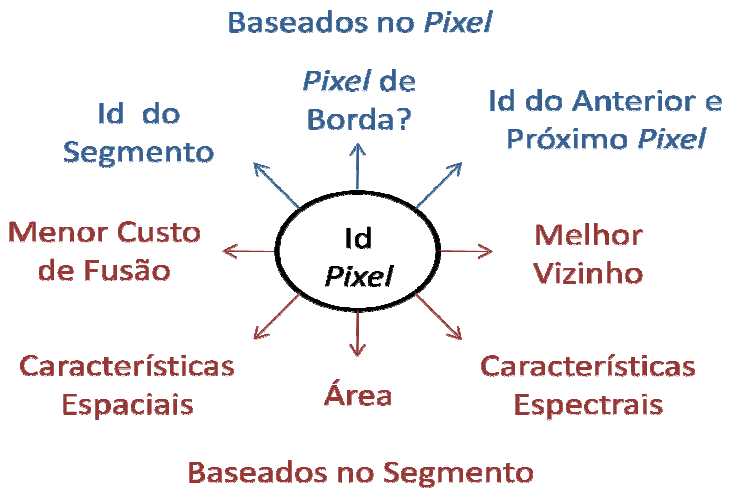

Figura 4.1. Estrutura de dados armazenada na memória global da GPU contendo características do *pixel* e do segmento

4.2. Algoritmo paralelo

A idéia central do algoritmo proposto consiste em distribuir o processamento relativo a cada *pixel* da imagem em uma diferente *thread*, de forma a tirar proveito da alta capacidade de processamento paralelo da GPU que permite a execução de um número bastante elevado de *threads*. Este tipo de abordagem favorece também o balanceamento de carga, pois cada *thread* sempre se refere à execução de um único *pixel*, resultando sempre em um processamento de granularidade mais fina.

Outra vantagem é a possibilidade de se processar todos os segmentos existentes de forma direta, sem a necessidade de dividir a imagem e ter que enfrentar problemas de execução em suas fronteiras.

Foram desenvolvidas duas versões paralelas do algoritmo de segmentação baseadas em duas das heurísticas de decisão para fusão de segmento. A primeira, denominada de "PBF" (*parallel best fitting*), é baseada na heurística de melhor vizinho e a segunda chamado "PLMBF" (*parallel local mutual best fitting*) considera a heurística de melhor vizinho mútuo.

Ambas as versões consistem em seis principais funções (*kernels*) a serem executadas na GPU: *inicializar sementes*, *calcular vizinhos*, *realizar fusões*, *redefinir segmentos*, *recalcular bordas* e *escrever resultado*. A Figura 4.2 ilustra a interação entre estas funções no algoritmo.

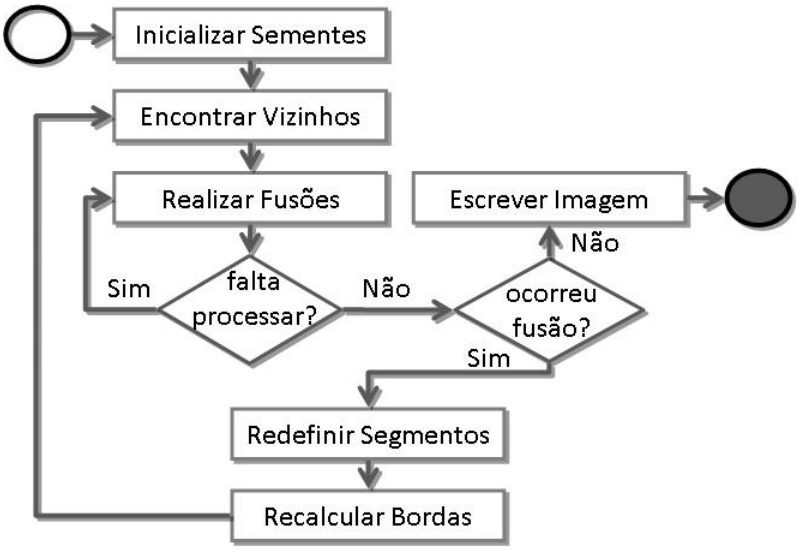

Figura 4.2. Esquema do algoritmo paralelo de segmentação

As duas versões presentes nesta dissertação possuem basicamente o mesmo algoritmo, com exceção de algumas diferenças principalmente na função *Realizar Fusões*. A seguir é descrita a versão PBF e em seguida são apresentadas as modificações realizadas na versão PLMBF.

4.3. Versão PBF

A versão PBF é baseada na heurística de melhor vizinho e possui de uma maneira geral seis funções principais. A seguir são descritas sucintamente cada uma destas seis funções em relação a esta versão.

4.3.1.Função *Inicializar Sementes*

No início do processo de segmentação, cada *pixel* da imagem é considerado como semente e, por conseqüência, passa a representar um segmento. De forma paralela, cada um desses *pixels* é processado pela função *Inicializar Sementes* e tem seus respectivos valores de atributos armazenados. O identificador inicial de cada segmento equivale ao identificador do *pixel* que o representa e suas características espectrais e espaciais também são baseadas nos valores referentes ao *pixel* em questão.

Ao término desta etapa, a estrutura de dados estará devidamente preenchida com todas as informações iniciais sobre *pixels* e segmentos necessárias para os passos seguintes que constituem a segmentação propriamente dita.

4.3.2. Função *Encontrar Vizinhos*

A função *Encontrar Vizinhos* inicia com um teste que visa restringir o processamento somente aos *pixels* da borda de segmentos, evitando, assim um esforço computacional desnecessário, já que *pixels* internos do segmento não possuirão *pixels* adjacentes em algum segmento vizinho.

São calculados, para os *pixels* de borda, os custos de fusão entre o seu segmento e os segmentos adjacentes. O segmento adjacente que implicar no menor custo é considerado como melhor vizinho do segmento.

Como o processamento é realizado de forma paralela para todos os *pixels* da borda de um segmento, faz-se necessário garantir que seja atribuído ao segmento o melhor entre todos os vizinhos encontrados.

Para fins de otimização, esta identificação do melhor entre todos vizinhos de um segmento é realizada em duas etapas: uma no interior de cada bloco e uma externa. A primeira garante que para cada segmento somente um melhor vizinho será definido dentro de cada bloco. Desta forma, é necessário utilizar a memória compartilhada e um mecanismo de sincronização por barreira.

A segunda etapa é responsável por comparar o valor obtido para cada segmento existente no bloco com o valor armazenado na estrutura do segmento. Caso seja necessário atualizar a estrutura do segmento, esta é feita por intermédio de uma seção crítica de forma a garantir a escrita na memória global.

4.3.3. Função *Realizar Fusões*

O primeiro passo desta função consiste na verificação dos *pixels* que representam segmentos válidos, conforme definido na seção 4.1. Somente as *threads* referentes a estes *pixels* devem ser processadas, já os demais *pixels* representam segmentos que já foram anexados.

Após a verificação, caso um segmento possua um melhor vizinho que tenha um custo de fusão menor do que o custo de fusão máximo (definido pelo parâmetro de escala) este é selecionado para a fusão.

Uma seção crítica garante que um segmento não seja processado simultaneamente por mais de uma *thread*. Desta maneira, os segmentos envolvidos em determinada fusão são marcados, fazendo com que qualquer outra *thread* que tente acessá-los fique bloqueada. A fim de não gerar um bloqueio definitivo desses processos ou *deadlock*, é necessário que a marcação dos dois segmentos envolvidos em uma fusão obedeça a determinado critério, no caso a ordem de seus Ids.

O segmento escolhido para fusão anexa seu melhor vizinho e os atributos do segmento resultante são calculados. Por conseqüência, o segmento que foi anexado deixa de ser considerado como segmento (passa a ser inválido) e o *pixel* que o representava passa a fazer parte do segmento que o anexou.

Cabe ressaltar que, a fim de evitar perda de desempenho, criou-se um mecanismo para realizar um controle sobre as *threads* que estão em espera. Assim, quando uma *thread* está para ser bloqueada, um indicador global é ativado designando que ainda há *pixels* não processados e em seguida a *thread* é forçada a abortar. Desta forma, não é necessário manter elementos de processamento ociosos enquanto aguardam um segmento ser liberado.

Ao término desta etapa, caso o indicador global tenha sido ativado, este é então desativado e a função *Realizar Fusões* é novamente invocada. Este procedimento se repete até que o indicador não mais seja ativado.

4.3.4. Função *Redefinir Segmentos*

Após a fusão entre dois segmentos, os *pixels* pertencentes ao segmento que foi anexado devem fazer parte do segmento que o anexou. Para obter um maior desempenho computacional, este procedimento pode ser realizado de forma paralela. A função *Redefinir Segmentos* é responsável por percorrer paralelamente estes *pixels* e atribuir-lhes o novo identificador de segmento.

Esta função deve ser executada somente se alguma fusão houver ocorrido na etapa *Realizar Fusões* processando-se apenas as *threads* dos *pixels* pertencentes a segmentos envolvidos nestas fusões.

4.3.5. Função *Recalcular Bordas*

Após a fusão entre dois segmentos, *pixels* que antes faziam parte da borda de um segmento podem não pertencer à borda do segmento resultante da fusão. A função *Recalcular Bordas* tem o papel de identificar e assinalar estes *pixels*.

De forma paralela, para cada *pixel* de borda, é verificado se este se encontra na borda da imagem ou se pelo menos um de seus *pixels* adjacentes pertence a um segmento diferente do seu próprio. Não sendo este o caso, o *pixel* deixa de ser considerado como *pixel* borda.

4.3.6. Função *Escrever Imagem*

Quando mais nenhuma fusão é possível, antes do algoritmo ser finalizado, a função *Escrever Resultado* é invocada.

Nesta etapa, as *threads* de cada *pixel* são processadas paralelamente de modo a escrever em uma imagem de saída, que contém o resultado final da segmentação, a média dos valores espectrais do segmento correspondente.

Os *pixels* que fazem parte da borda dos segmentos podem ser representados por um valor pré-determinado em suas bandas espectrais.

4.4. Versão PLMBF

A versão PLMBF utiliza as mesmas funções básicas descritas na versão PBF na seção 4.3. A diferença fundamental está na função *Realizar Fusões*: enquanto para a PBF basta haver um segmento definido como melhor vizinho que atenda ao critério de homogeneidade, para que ocorra uma fusão na PLMBF é preciso também que o melhor vizinho encontrado para um dado segmento tenha como seu melhor vizinho o próprio segmento em questão.

Desta forma, após selecionar um segmento para fusão conforme descrito na função *Realizar Fusões* (4.3.3) é necessário incluir um novo teste para verificar se a condição de mutualidade entre o segmento e seu vizinho é atendida. Em caso negativo, não será realizado o processo de fusão do segmento em questão.

Entretanto, ao utilizar a heurística de decisão de melhor vizinho mútuo presente nesta versão, podem ocorrer problemas no crescimento de regiões. Este problema fica evidente quando mais de um par de candidatos à fusão têm o mesmo valor de custo de fusão.

Trata-se de uma situação relativamente freqüente em regiões homogêneas da imagem. Para agilizar o processamento e evitar falhas na segmentação, propõe-se um procedimento na versão PLMBF que envolve o armazenamento de até quatro melhores vizinhos para cada segmento, desde que estes possuam o mesmo valor de custo de fusão calculado.

Para tanto, é necessária, além da modificação na estrutura de dados, alterações nas funções *Encontrar Vizinhos* e *Realizar Fusões*. Esta mudança resultou em um ganho de desempenho principalmente na versão seqüencial do algoritmo.

4.5. Cálculo dos atributos espaciais

O cálculo dos dois atributos de forma descritos anteriormente (Equações 3.6 e 3.7) envolve o cálculo do comprimento da borda b. Este cálculo não pode ser realizado de maneira direta, pois *pixels* que fazem parte da borda de um segmento podem não pertencer à borda do segmento resultante da fusão.

A maneira mais eficiente de calcular tal medida é pela soma do comprimento de borda dos dois segmentos envolvidos e pela subtração deste total pela quantidade de *pixels* de borda que fazem o contato entre estes segmentos, ou seja, o número de *pixels* que não mais fará parte da borda.

Desta forma, é necessário que, no mínimo, sejam percorridos todos os *pixels* de borda do segmento com menor comprimento de borda. Assim, é possível verificar o número de *pixels* de borda que tem como *pixel* adjacente um elemento do outro segmento envolvido e multiplicar este total por 2, para totalizar a quantidade de *pixels* de ambos os segmentos.

À medida que os segmentos crescem, este cálculo passa a ser bastante caro computacionalmente, pois envolve um caminhamento sobre as bordas de um segmento. Este caminhamento deve ser realizado para cada *pixel* de borda e requer um grande número de acessos à estrutura dos segmentos presente na memória global da GPU, cuja latência é alta. Além disso, este procedimento provoca o desbalanceamento de carga entre as *threads*, pois o processamento é proporcional ao tamanho da borda de cada segmento.

Com o intuito de contornar este problema, é proposta neste trabalho a substituição destes atributos espaciais por outros que não envolvam o cálculo do comprimento da borda dos segmentos e que sejam considerados similares, isto é, que encontrem seu mínimo para formas semelhantes.

Cabe ressaltar que atributos que envolvam o perímetro também não são interessantes, tendo em vista que para tanto também se faz necessário o caminhamento dos *pixels* de borda.

Russ (1998) apresenta uma tabela que resume alguns dos parâmetros mais utilizados para cálculo de forma de objetos. Nesta tabela, é descrito um atributo de compacidade ($Comp_{Russ}$) tal qual apresentado na Equação 4.1. Este atributo envolve o cálculo da área n e do diâmetro $dmax$.

$$Comp_{Russ} = \frac{\sqrt{\frac{4}{\pi} n}}{dmax} \tag{4.1}$$

Este atributo tem por objetivo identificar objetos que tenham forma compacta, assim como o atributo $Comp$ proposto por (Baatz & Schäpe, 2000) e definido na Equação 3.6. Por sua similaridade e por não envolver o cálculo do comprimento de borda, o atributo $Comp_{Russ}$ é passível de ser utilizado no algoritmo de segmentação proposto nesta dissertação.

Observando mais uma vez os atributos de forma expostos em (Russ, 1998), encontra-se um atributo que, assim como o atributo Svd definido por (Baatz & Schäpe, 2000) e apresentado na Equação 3.7, varia de acordo com a convexidade dos objetos. Este atributo, denominado solidez Sol_{Russ}, é definido na Equação 4.2 e necessita apenas da área n do objeto e da área do retângulo mínimo envolvente n_{box}. O atributo Sol_{Russ} também foi escolhido para ser utilizado no algoritmo de segmentação sugerido neste trabalho.

$$Sol_{Russ} = \frac{n}{n_{box}} \tag{4.2}$$

4.6. Supersegmentação no PLMBF

Ao serem executados testes preliminares com os segmentadores elaborados foi verificado um problema de supersegmentação em algumas regiões das imagens resultantes na versão PLMBF. Tendo em vista que este problema tem causa inesperada aos neófitos no uso de GPUs considerou-se importante dedicar uma seção para descrevê-lo e à correspondente solução concebida neste estudo.

Verificou-se que o problema da supersegmentação decorre de erros de arredondamento gerados na GPU durante o cálculo dos atributos espectrais e espaciais e conseqüentemente na geração do valor de custo de fusão. O fato é que a aritmética da GPU é diferente da CPU e possui uma precisão menor para problemas de ponto flutuante. Novas arquiteturas de GPU como a Fermi da Nvidia já possuem uma maior precisão.

O erro pode ser explicado da seguinte forma: uma expressão aritmética em diferentes ordens equivalentes algebricamente não necessariamente produz o

mesmo resultado, quando este resultado é em módulo muito menor do que os termos envolvidos na expressão.

Assim, em alguns casos, ao se calcular o custo de fusão do segmento A em relação ao segmento B o valor obtido era diferente do valor encontrado ao se calcular o custo do segmento B em relação ao segmento A. Neste cenário, é possível ocorrer em um erro na determinação do melhor vizinho mútuo dos segmentos.

A figura 4.3 ilustra um exemplo onde a determinação do melhor vizinho mútuo fica prejudicada. No exemplo, o segmento A tem seus valores de custo de fusão calculados corretamente para os segmentos B e C, definindo B como seu melhor vizinho. O segmento C também obtém seus custos de fusão calculados corretamente, tendo seu melhor vizinho determinado como sendo o segmento A. Porém, um erro de arredondamento ocorre no cálculo do custo de fusão para o segmento B em relação ao segmento A, fazendo com que o valor resultante do custo entre B e C seja menor do que entre B e A e, por conseqüência, que C seja definido como melhor vizinho de B.

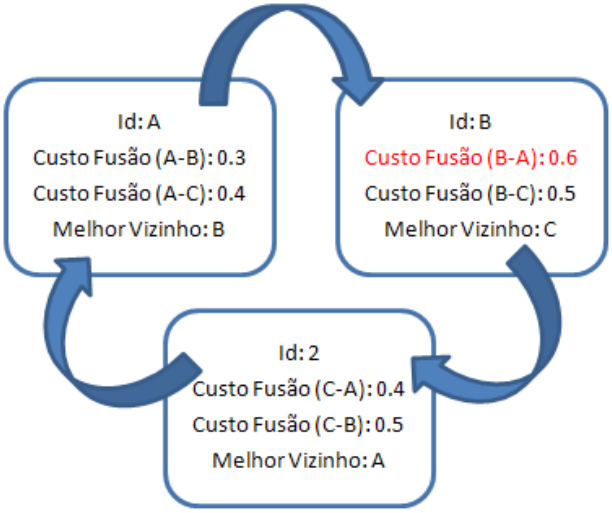

Figura 4.3. Exemplo de caso onde há problemas de crescimento devido a erro de arredondamento

A fim de contornar este tipo de problema, foi definida uma heurística para o cálculo do custo de fusão entre dois segmentos: o cálculo passa a ser sempre executado em uma ordem fixa, independente de quais são os segmentos envolvidos.

Desta maneira, antes de realizar o cálculo, verifica-se qual dos segmentos possui o menor Id. Este então é considerado como segmento principal e o cálculo dos atributos é feito sempre deste em relação ao seu vizinho.