3
Processamento em Baixo Nível Utilizando Padrões Paralelos

Neste capítulo são apresentadas abordagens paralelas em formulações que se beneficiam da eficiência de processamento da GPU para temas selecionados da camada de processamento do conteúdo visual denominada *Processamento de Baixo Nível*. São abordados temas de processamento de imagens, nos quais o conteúdo visual manipulado corresponde ao esboço primitivo e temas de localização no espaço da imagem de conjuntos de objetos simples.

As manipulações do esboço primitivo abrangem operações sobre uma imagem adquirida para gerar outra imagem. Grande parte de suas formulações são descritas com operações similares e independentes entre si sobre os pixels da imagem de entrada para gerar os pixels da imagem de saída. Portanto, atendem às características ideais de processamento em GPU apresentadas no Capítulo 2, por serem compostas de computações replicadas sobre um grande volume de dados, definidos pela resolução da imagem. Em suporte à formulações em GPU para tais operações, este tema de manipulação de imagens é apresentado na Seção 3.2.

Outro tema de visão computacional abordado neste capítulo apresenta mecanismos paralelos de análise de imagens para identificação de um conjunto de objetos, no qual cada objeto possui uma característica cromática única que permita distingui-lo do restante da cena retratada pela imagem (Seção 3.3). A partir dessa identificação, os mecanismos a serem apresentados podem ser usados para extrair informações de localização dos objetos no espaço da imagem, tais como a massa (número de pixels), posição central e caixa envolvente ("bounding box"), além de possibilitarem a construção de estimativas sobre as profundidades relativas a que se encontram os objetos e sobre a distância dos objetos à câmera, quando conhecidas, a priori, suas dimensões reais ou informações sobre a câmera.

De forma complementar aos temas anteriores, abordamos na Seção 3.4 o conceito de regiões de interesse, o qual é bastante utilizado na área de processamento de imagens. A definição de regiões de interesse permite limitar a área a ser processada por um operador. Para permitir a distribuição de operações realizadas no interior de regiões de interesse em subdivisões

independentes, introduzimos uma formulação paralela.

As abordagens em GPU propostas neste capítulo fazem adaptações de padrões de programação paralela. Os padrões de programação paralela podem ser vistos como blocos construtores a serem utilizados na formulação de algoritmos. Por esse motivo, sua compreensão se posiciona no segundo nível de abstração de sistemas de informação propostos por Marr (Mar80), correspondendo ao nível de mecanismos para operação de um algoritmo.

Os padrões de programação selecionados para dar suporte ao tratamento dos temas escolhidos são apresentados na Seção 3.1.

3.1 Padrões de Programação Paralela

O estudo de padrões de programação paralela antecede o desenvolvimento das GPUs programáveis (Hil86, Szy95, Par99) e é explorado em diferentes arquiteturas. A comunidade de GPGPU incentiva o uso dessas construções como blocos construtores de algoritmos mais complexos em GPU, por apresentarem soluções propícias ao processamento de fluxos em operações independentes entre seus elementos.

Os padrões mais frequentemente destacados pela comunidade de GPGPU são: mapeamento ("map"); coleta ("gather") e dispersão ("scatter"); redução ("reduction"); filtragem; escaneamento ou prefixo paralelo ("scan"/"parallel prefix"); ordenação e busca. Dois desses padrões são de especial interesse aos temas abordados por esta tese, o operador de mapeamento e o de redução, os quais são apresentados a seguir. Análises recentes de frameworks de processamento utilizando esses dois padrões implementados com a linguagem CUDA podem ser encontradas em (Cat08, He08). Informações sobre os padrões utilizados em GPGPU podem ser localizadas em ampla literatura específica (Kru03, Hor05, Rog07, Owe07, Owe07b).

3.1.1 Mapeamento

O operador de mapeamento representa a construção típica na qual se deseja percorrer todos os elementos de um conjunto, executando uma determinada operação, na qual um resultado é produzido para cada elemento. Mais formalmente, dado um fluxo de elementos A e uma função $f(x)$, definida sobre os elementos de A, o operador de mapeamento $map(A, f)$ aplica a função f a cada elemento $a_i \in A$ produzindo $f(a_i)$.

Em abordagens sequenciais, o operador de mapeamento é construído com um laço de repetição que percorre o conjunto de elementos e calcula um valor

de saída a cada passada do laço.

Esse é um operador extremamente simples de ser implementado em GPU. A implementação desse operador em formulações de processamento genérico via pipeline gráfico segue explicitamente o modelo de aplicação apresentado na Seção 2.2. Uma textura pode ser usada para representar o fluxo de entrada A, na qual cada texel representa um de seus elementos a_i. Seguindo o modelo, basta desenhar um quadrilátero cobrindo tantos pixels quantos forem os elementos de A e escrever um "fragment program"implementando a função $f(x)$. O "fragment program"é responsável por fazer uma consulta ao elemento a_i na coordenada de textura a ele correspondente. O valor de $f(a_i)$ é armazenado como valor do fragmento produzido, o qual pode vir a ser armazenado em uma textura (renderização para textura) ou projetado no "frame-buffer".

Em uma versão para linguagem de programação genérica em GPU, mais especificamente, CUDA, uma grade de processamento é definida sobre as dimensões do domínio de A e um "kernel"é codificado implementando $f(x)$. Os elementos de A podem ser previamente armazenados, tanto em espaço de memória de textura, se beneficiando de mecanismos de cache da linguagem, quanto em memória global. Na segunda alternativa, para diminuir a latência associada a esse tipo de memória, o "kernel"pode ser usado para codificar um pré-carregamento de uma cópia local das regiões de memória global acessadas por seu bloco para a memória compartilhada de rápido acesso. A consistência de tal pré-carregamento pode ser assegurada pelo uso dos mecanismos de sincronização disponíveis para as "threads"de mesmo bloco.

3.1.2 Redução

Existem diversas situações nas quais se faz necessário processar um fluxo para produzir um outro de tamanho menor, cujos elementos indicam métricas de características globais do fluxo original. Em hardware sequencial, tais computações são tradicionalmente realizadas por operações de escrita indireta ("scatter") em espaços de memória pré-definidos. No hardware gráfico, possíveis soluções para processar o fluxo inicial, gerando os valores do fluxo final via operações de escrita indireta ("scatter") incluem a utilização de um "vertex shader"juntamente com o hardware dedicado a operações de "blending"e a utilização de operações atômicas, quando disponíveis, em espaços de memória global (CUDA08). Na primeira solução o "vertex shader"fornece o deslocamento da posição de escrita, enquanto o hardware dedicado a operações de "blending"(Sch07), especializado em combinações ponderadas de valores de

fragmentos sobre o mesmo pixel, permite combinar os valores armazenados na mesma posição. Entretanto, tanto as operações de "blending"quanto as atômicas são, em última instância, operações realizadas sequencialmente por endereço de escrita, não sendo escaláveis com o aumento do número de processadores.

O operador de redução oferece um modelo para executar em paralelo, de maneira distribuída, tarefas de conversão de um fluxo inicial em um fluxo de saída, quando é conhecida a operação a ser utilizada para a aglomeração de elementos do fluxo original nos elementos do fluxo produzido. Mais formalmente: dado um operador binário associativo $\otimes$, uma redução é um operador $(S, \otimes)$, onde S representa o fluxo de elementos de entrada, no qual $\otimes$ está definida. Dado o fluxo de n elementos x_0, x_1, ..., x_{n-1}, a operação de redução é definida como o cálculo de $x_0 \otimes x_1 \otimes x_2 \ldots \otimes x_{n-1}$. Exemplos comuns para o operador $\otimes$ são $+$, $\times$, $\vee$, $\wedge$, $\otimes$, $\cup$, $\cap$, max e min.

Na literatura de processamento paralelo, Parhami (Par99) mostra que a arquitetura de processadores distribuídos em árvore binária é ideal para a construção desse operador. Cada nó interno da árvore recebe duas entradas pela leitura de seus filhos. Aplica o operador sobre essas entradas e passa o resultado para o nó na camada mais alta. Depois de $O(\lg n)$ passos com um total da ordem de $O(n)$ operações, o processador na raiz da árvore recebe o valor do resultado.

Para o cálculo do operador de redução em GPU, a árvore binária de processadores não é fisicamente construída, mas simulada. Os níveis da árvore são simulados com múltiplas passadas de processamento, sendo cada passada responsável por aglomerar o fluxo gerado pela passada anterior. Os nós na mesma altura da árvore são computados em paralelo durante uma passada de processamento e seus resultados armazenados em uma textura (ou memória global, se disponível) para serem consultados na próxima passada. No modelo original, a cada passo o tamanho do fluxo produzido é reduzido pela metade. Além da binária, outras árvores n-árias podem ser usadas para estruturar a aglomeração, aumentando a taxa de redução, de acordo com o número de elementos aglomerados por nó.

A vantagem de simular a arquitetura de árvore binária (ou n-ária) de processadores em formulações de GPGPU é esta construção fornecer uma distribuição equilibrada das tarefas entre os processadores da GPU e propiciar a escrita de processamentos independentes. Cada "shader"ou "kernel"executa operações de coleta ("gather") das informações correspondentes aos filhos do nó corrente pela leitura da memória de textura (ou global, se disponível) utilizada para o armazenamento dos resultados da passada anterior.

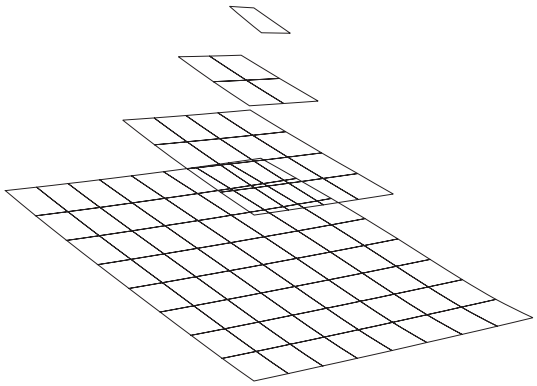


Figura 3.1: Operador de redução: pirâmide de texturas

Analisando a ordem de grandeza do processamento em GPU de um operador de redução, segundo o modelo apresentado, supondo uma arquitetura com p processadores, p elementos são processados em paralelo. Logo, o tempo para a redução de um conjunto de n elementos é da ordem de $\mathrm{O}(\frac{n}{p}\log n)$ passos de tempo. Em contraposição, observa-se que em uma abordagem sequencial $O(n)$ passos de tempo seriam necessários para computar a redução (Owe07b).

Em aplicações de GPGPU é comum definir um operador de redução sobre uma textura 2D. Ao invés de uma árvore binária, a estrutura simulada assemelha-se a uma pirâmide (Figura 3.1), onde cada nó interno possui quatro filhos e a raiz da pirâmide corresponde ao valor final desejado. Neste caso, a aglomeração agrupa quatro elementos por operação, dois a dois, em cada dimensão da textura. Estendendo o mesmo conceito, a redução de dados volumétricos pode ser feita agrupando oito elementos por operação. Uma arquitetura paralela baseada na construção de pirâmide foi proposta em (Tan83).

Além de variações relacionadas ao número de dimensões do fluxo de entrada (1D, 2D ou 3D), existem ainda variações na formulação do operador de redução relativas ao número de elementos desejados no fluxo de saída. Em tais variações, o fluxo de entrada é examinado em paralelo por múltiplos operadores de redução, tantos quantos forem os elementos desejados no fluxo de saída. A variação denominada Redução Múltipla 1D (Owe07) utiliza uma textura 2D para representar o fluxo de entrada, o qual passa a ser reduzido sucessivamente em apenas uma das dimensões por esse operador. Em outra variação, o operador de redução múltipla agrupa informações sobre um domínio 2D, executando múltiplas reduções em paralelo em ambas as dimensões (Flu06, Vas08b, Vas08c).

Redução Múltipla

O operador de redução múltipla oferece um padrão para processar, de maneira distribuída e paralela, múltiplos elementos de um fluxo, por múltiplos operadores, mas, apesar de sua importância, não é amplamente divulgado. A motivação para seu uso nesta tese, em oposição ao processamento de um mesmo fluxo por múltiplos operadores, porém um de cada vez, é justificada pela busca de ganho em tempo de processamento pela consideração de questões relacionadas à latência e banda de memória.

Conforme apresentado no Capítulo 2, as modernas arquiteturas de GPU favorecem aplicações de alta intensidade aritmética. A formulação do operador de redução múltipla promove o reaproveitamento de dados pelas múltiplas operações e diminuição da latência observada, nos casos em que o operador é implementado considerando-se localidade de acesso e mecanismos transparentes de cache, oferecidos pela linguagem ao tipo de memória que contém os dados do fluxo sendo processado, ou quando implementado via programação de pré-carregamentos para a memória compartilhada de rápido acesso, se disponível.

Tais recursos permitem que a adoção deste padrão acarrete melhoria do tempo de processamento pela diminuição da sobrecarga total em tempo de latência e transferência de dados, quando comparado com a aplicação de um mesmo conjunto de operadores sobre um fluxo, processados um de cada vez, fazendo com que os dados precisem ser retransferidos vez a vez. Este argumento pode ser comprovado nos resultados obtidos em sua utilização, em comparação a processamentos de operadores distintos em diferentes passadas expostos em (Vas08b) (Anexo A).

Não é do conhecimento desta autora uma definição formal anterior do operador de redução múltipla, seja na literatura de processamento paralelo, seja na de GPGPU. Sua formalização se faz necessária neste contexto para embasar sua adoção nas soluções apresentadas nesta tese.

Dado um fluxo S de elementos e um conjunto ordenado Ω composto por M operações binárias associativas independentes Ω_i definidas sobre S, a operação de redução múltipla pode ser representada como um operador $(S,\ \Omega = \{\Omega_0, \Omega_1, ..., \Omega_{M-2}, \Omega_{M-1}\})$ e que produz o fluxo ordenado contendo $\{\Omega_0(S), \Omega_1(S), ..., \Omega_{M-2}(S), \Omega_{M-1}(S)\}$.

A computação em paralelo de uma redução múltipla é processada em duas etapas distintas: a primeira é composta de múltiplas avaliações locais, no interior de subfluxos de tamanho M do fluxo S, para construir uma base de resultados intermediários; a segunda é responsável pela aglomeração dos resultados locais em M resultados globais. Mais formalmente,

a base é construída subdividindo o fluxo S em subfluxos de tamanho M, $s_0, s_1, ..., s_{p-2}, s_{p-1}$. Durante a construção da base são processadas em paralelo avaliações de S por Ω, cada uma restrita a um subfluxo. Em detalhes, cada subfluxo s_i define o conjunto de elementos a serem operados durante a construção da base por uma avaliação local de Ω, que resulta no novo subfluxo ordenado contendo $\{\Omega_0(s_i), \Omega_1(s_i), ..., \Omega_{M-2}(s_i), \Omega_{M-1}(s_i)\}$. Ao término da construção da base, ela é composta de um novo fluxo ordenado, contendo $\{\Omega\{s_0\}, \Omega\{s_1\}, ..., \Omega\{s_{n-2}\}, \Omega\{s_{n-1}\}\}$. O paralelismo e independência de procedimentos acontece tanto entre o processamento de subfluxos s_i e s_j distintos, quanto em operações distintas sobre um mesmo subfluxo ($\Omega_i(s_i)$ e $\Omega_j(s_i)$) [1].

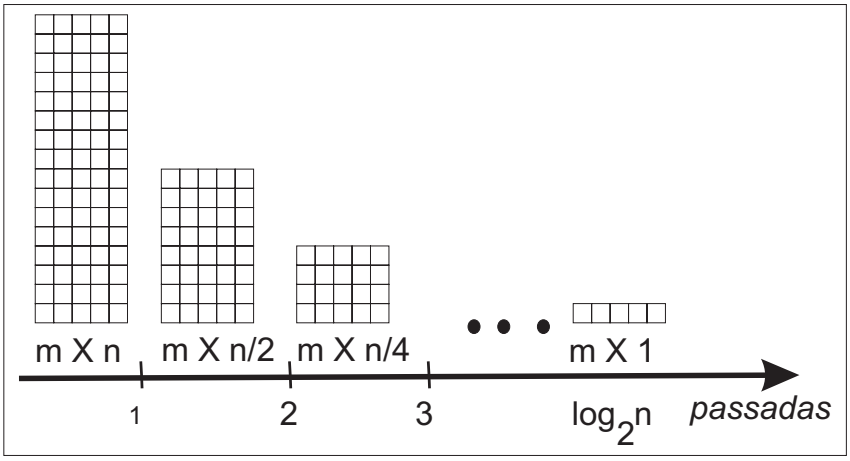
Para qualquer que seja o número de elementos no fluxo inicial S, a base gerada representa um fluxo de tamanho múltiplo de M, uma vez que a construção da base produz as avaliações locais de todos os M operadores em Ω para cada subfluxo em um fluxo ordenado de tamanho M. A regra também se aplica ao último subfluxo s_{M-1}, mesmo que este não contenha M elementos.

Formada a base, o redutor múltiplo agrupa em paralelo os elementos representativos de uma mesma operação Ω_i computados sobre diferentes subfluxos. A quantidade de resultados agrupados por interação de um mesmo operador Ω_i sobre subfluxos distintos define o fator de redução do fluxo por interação. Múltiplas interações são aplicadas para reduzir o fluxo da base, até sobrarem apenas M elementos no fluxo.

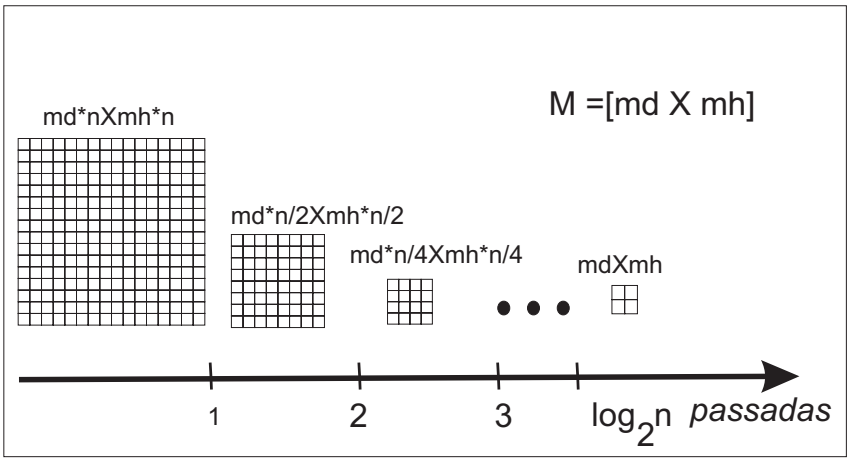
Ressaltamos a distinção da estrutura formada pelo operador de redução múltipla 1D ou 2D em comparação ao operador de redução simples, no qual as operações de redução em 1D ou 2D são representadas, respectivamente, com uma estrutura de árvore ou de pirâmide, cuja base representa o fluxo inicial e a raíz contém o resultado da redução. No caso de redução múltipla, além de ser acrescentada uma nova camada entre o fluxo inicial e as estruturas, composta pela base criada, sobre ela são levantadas múltiplas árvores ou pirâmides. Esse conjunto sobre a mesma base forma múltiplas estruturas independentes, cada uma representando uma operação distinta sobre o fluxo $\Omega_i(S)$, cujo resultado é representado em sua raiz (ver Figura 3.2). As computações de um mesmo nível, em qualquer das diversas estruturas, são processadas em paralelo e as de níveis distintos, em múltiplas passadas do redutor.

Sendo N o tamanho do fluxo inicial, M o tamanho do conjunto Ω, e r o fator de redução por interação, o operador de redução múltipla computa aglomerações em $(0(log_r N)$ passos, com um total de $0(MN)$ operações.

[1] Por restrições do hardware existente (SPMD), o conjunto de operações é na prática formado por um conjunto de parâmetros distintos para a mesma operação



3.2(a): unidimensional



3.2(b): bidimensional

Figura 3.2: Operador de redução múltipla

3.2 Manipulações em Pré-processamento

A partir desta seção são apresentadas formulações propícias à computação em GPU por adaptação dos padrões de programação mapeamento e redutor, para os temas selecionados.

Entre as funções tipicamente encontradas em um sistema de visão computacional, logo após uma fase inicial de aquisição de imagens, encontra-se uma fase de tratamento das imagens para ressaltar determinadas características, de acordo com o contexto da aplicação em questão (Gon01, For02). Exemplos de manipulações dos valores dos pixels da imagem de entrada para gerar uma imagem de saída incluem: operações de quantização e de escala do sinal da imagem, transformações do espaço de cor, manipulação de contraste, computação do gradiente da imagem, redução de ruído, filtros (seja no domínio espacial, seja no de frequências) e operadores morfológicos.

Mais formalmente, do ponto de vista do processamento de imagens

(Gon01), a imagem de entrada é vista como um sinal bidimensional $f(x, y)$ a ser processado para produzir um sinal $g(x, y)$, utilizando um operador T definido sobre a vizinhança de tamanho N de cada amostra (x, y).

Esta formulação pode ser traduzida diretamente em abstrações de processamento paralelo a dados. O sinal de entrada $f(x, y)$ é visto como um fluxo de pixels, e T é codificado em um "kernel"operando independentemente na produção de cada $g(x, y)$ pela coleta ("gather") das amostras da vizinhança N de $f(x, y)$. Esta descrição é equivalente à do padrão de programação de mapeamento. Por sua simplicidade, as operações de processamento de imagens foram as primeiras operações a serem propostas com a introdução da programação da GPU.

3.3 Localização de objetos

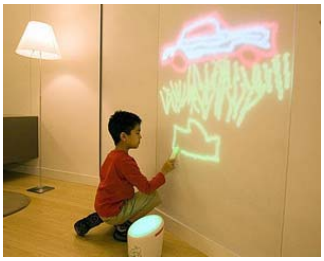
Nesta seção apresentamos o operador MOCT ("Multi-Object Chromatic Tracking") de processamento baixo nível para localização em tempo real de conjuntos de objetos identificáveis pela sua característica cromática. Apesar da simplicidade dessa formulação, a necessidade de localização de objetos no espaço da tela é inerente a diversas aplicações. Tal operador pode ser aplicado tanto em processamento de imagens naturais, capturadas com uma câmera, onde objetos reais possam ser identificados por sua característica cromática de forma única na cena, quanto em imagens virtuais resultantes de um processo de renderização (ou computação) no qual determinados pixels sejam intensionalmente coloridos de forma a identificar as áreas que representam.

Dois cenários principais de aplicação se destacam onde objetos reais de característica cromática única na cena são utilizados em imagens naturais (Figura 3.3): a composição de vídeo e o processamento de novas formas de interação. No primeiro cenário de aplicação, os objetos funcionam como demarcadores de regiões locais de "chroma-key", ou seja, delimitam regiões para serem substituídas em composições de vídeo por outras texturas ou objetos renderizados. No segundo cenário, os objetos de identificação única podem fornecer novas formas de interação, seja como marcadores da cena ou como apontadores dinâmicos de interação. Em um exemplo desse tipo de aplicação, objetos esféricos podem ser usados como marcadores da cena, tendo suas dimensões reais previamente calibradas para serem usadas em estimativas de profundidade que permitem transformar as coordenadas do espaço da tela para coordenadas $2D\frac{1}{2}$.

A geração de imagens virtuais contendo regiões identificadoras é comum aos chamados mapas de imagens, onde cada área de interesse é representada

3.3(a): composição de vídeo

3.3(b): marcadores de cena (composição em realidade aumentada)

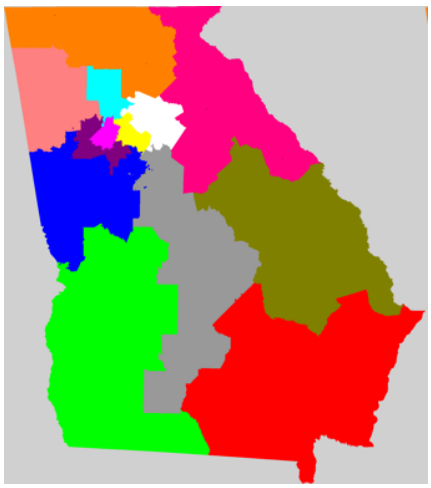


3.3(c): apontadores (interação)

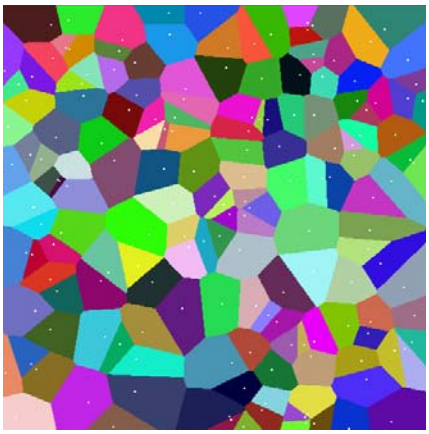
Figura 3.3: Localização de Objetos em Imagens Naturais

com uma característica cromática única, equivalente a um rótulo específico (Figura 3.4). Os mapas de imagens definem regiões de acordo com o interesse da aplicação e podem ser produzidos usando os canais RGB ou utilizando apenas um canal (tons de cinza). Desta forma, a origem dessas regiões pode estar relacionada a processos algorítmicos com estruturas de segmentação espacial (Figura 3.4 b).

O operador MOCT proposto viabiliza o uso de informações de localização de objetos coloridos em inúmeras aplicações de realidade aumentada, uma vez que possibilita a extração quadro-a-quadro de tais informações de uma forma extremamente eficiente.



3.4(a): Mapas de imagens



3.4(b): Voronoi

Figura 3.4: Localização de Objetos em Imagens Renderizadas

A diferença existente entre o tratamento do operador proposto em imagens naturais e imagens renderizadas é que a cor de identificação dos objetos em mapas de cores pode ser tratada por um teste de igualdade. Entretanto, nas naturais, a cor que identifica objetos sofre variações, causadas

pela interação da luz com o material que cobre a superfície do objeto e, por esse motivo, é avaliada dentro de intervalos de aceitação definidos por limiares.

O operador MOCT e os resultados demonstrados nesta seção foram publicados no artigo (Vas08b) (Anexo A) em uma abordagem propícia a implementação via pipeline gráfico. Sua formulação usando os conceitos de processamento paralelo de dados é apresentada a seguir.

3.3.1 MOCT: rastreamento de conjunto de objetos cromáticos em GPU

O núcleo do algoritmo de rastreamento proposto é baseado em uma adaptação do padrão de programação paralela de redução múltipla bidimensional. Utilizando-se da definição formal desse operador apresentada na Seção 3.1.2, apresentamos a seguir a descrição das adaptações das etapas do operador de redução múltipla, além de pseudo-códigos para os "kernels"de processamento paralelo que efetuam a tarefa de localização de objetos na imagem.

Inicialmente, o conjunto dos M objetos que se deseja rastrear é associado a uma disposição espacial que estabelece uma ordenação para as operações $\Omega = \{\Omega_0, ..., \Omega_{M-1}\}$, que representam testes de características cromáticas, cada um comparando com as características do objeto associado a sua posição na ordenação. Três tipos de disposição desse conjunto foram testadas, seguindo arranjos horizontal ou vertical de tamanho M ou quadrangular, de dimensões $\left\lceil \sqrt{M} \right\rceil$ (Figura 3.5). Observamos que o arranjo quadrangular pode causar desperdício de espaço na base gerada.

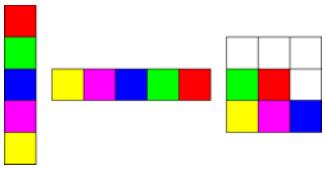


Figura 3.5: Arranjos do conjunto de objetos: horizontal, vertical e quadrangular

Após estabelecido o arranjo utilizado, o algoritmo MOCT constrói um fluxo bidimensional, a ser preenchido pelas análises locais da imagem de entrada, de acordo com o arranjo utilizado. É esse fluxo de elementos criado que compõe a chamada base do operador de redução múltipla. Deve, portanto, cobrir a imagem de entrada por múltiplas disposições dos arranjos selecionados, tendo sua dimensão espacial como múltiplo inteiro das dimensões do arranjo utilizado. Cada elemento do fluxo criado é então processado de forma independente pelo "kernel"de processamento, descrito a seguir, que preenche o fluxo com os valores que compõem a base de avaliações locais de forma intercalada na ordenação estabelecida pelo arranjo escolhido.

Pseudo-código das Avaliações Locais

```
constantes : dimensoesArranjo;

%posicao define qual elemento da base está sendo processado
elementoNaBase AvaliacaoLocal(posicao){

    %subfluxo corrente:
    s_i = posiçãoGlobalParaSubfluxo(posicao, dimensoesArranjo);
    %objeto corrente:
    i = posiçãoGlobalParaObjeto(posicao, dimensoesArranjo, s_i);

    avaliacaolocal = 0;
    %varre pixeis vizinhos cobertos pelo subfluxo
    para ( 0 < j < dimensoesArranjo ){
     posicao2D = posicaoInicial2D(s_i)
                  + posicao2DnoInteriordoArranjo(j);
      se(posição2D no interior da imagem){
          % lê imagem na posição corrente:
          cor = imagem(posição2D);
          se(Omega_i(cor)){
              atualiza(avaliacaolocal);
          }
      }
    }
    escreve no fluxo de saída(avaliacaolocal);

}
```

O "kernel"de avaliação local varre amostras da imagem de entrada que estejam posicionadas no espaço da imagem no intervalo correspondente ao coberto por um subfluxo no interior da base. Os contadores ilustrados na Figura 3.6 indicam essa avaliação no interior da área coberta por subfluxos gerados em arranjos verticais e não apenas sobre o pixel posicionado na mesma posição do elemento criado na base.

Na segunda fase do operador, o fluxo de dados da base contendo análises sobre regiões distintas da imagem para cada um dos objetos rastreados é reduzido para agrupar as avaliações locais em avaliações globais, uma para cada objeto rastreado.

Para iniciar a segunda fase, o algoritmo cria um novo fluxo, cujas dimensões são proporcionais à redução da base pelo fator de aglomeração,

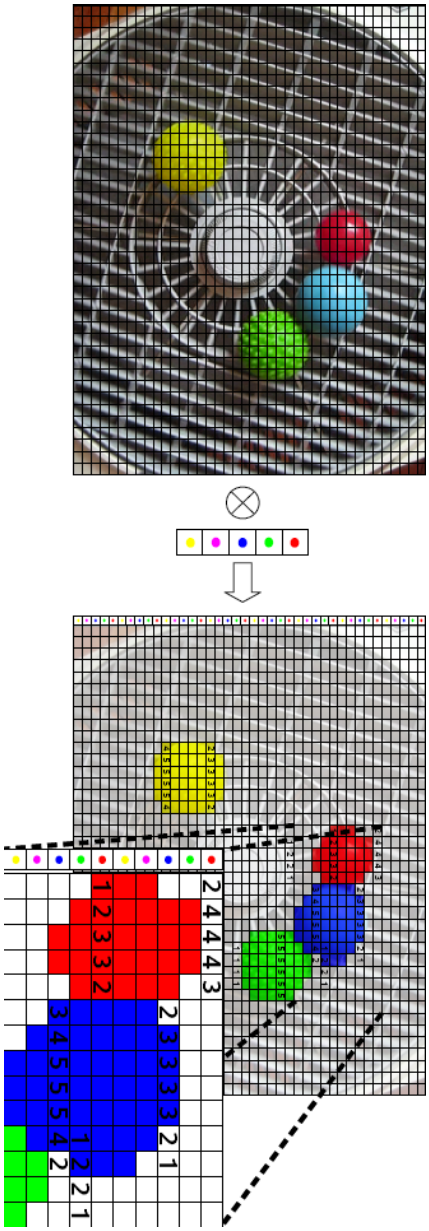


Figura 3.6: Criação da Base: subfluxos em arranjos verticais

mantendo a disposição do arranjo especificado. Esta etapa é repetida diversas vezes, criando novos fluxos reduzidos e chamando o "kernel"descrito abaixo para computar os valores de seus elementos, até que seja obtido um fluxo contendo apenas um único arranjo.

Pseudo-código da Avaliação Global em Coleta Paralela

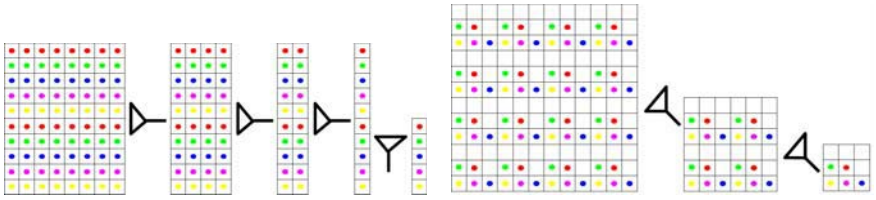
```
constantes : fatorRedução;

%posicaoCorrente define qual elemento
% do fluxo aglomerado está sendo processado
elemento Aglomeração(posicaoCorrente){
```

```
    aglomeracaoParcial = 0;
    para (0 < n < fatorRedução){
       pai = leElementoPai(posicaoCorrente, n);
       aglomera(aglomeracaoParcial,pai);
    }
    escreve no fluxo de saída(aglomeracaoParcial);

}
```

Este "kernel"codifica um nó interno à estrutura de redução, responsável por coletar informações anteriormente computadas com operações de leitura indireta de memória ("gather") representadas genericamente no pseudo-código com a chamada à função "leElementoPai"(Figura 3.7).



3.7(a): subfluxos em arranjos verticais

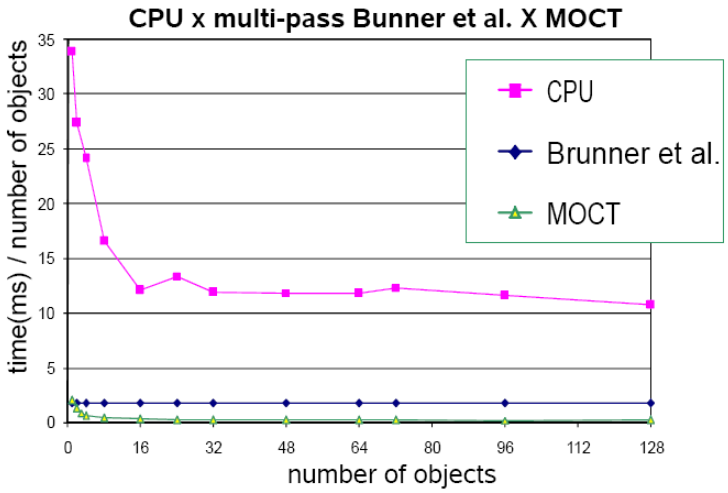
3.7(b): subfluxos em arranjos quadrangular

Figura 3.7: Obtenção dos resultados globais

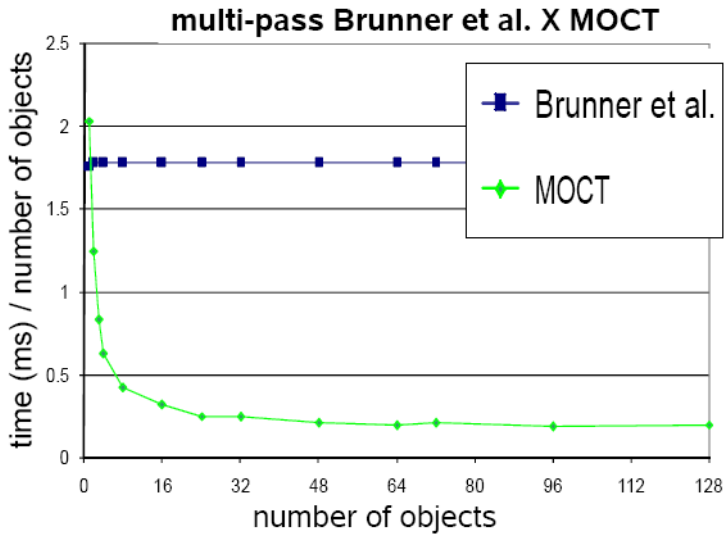
3.3.2 Resultados do operador MOCT

Os resultados obtidos para o operador MOCT foram comparados com uma implementação sequencial. A implementação em CPU foi feita utilizando uma única "thread"com laços de repetição aninhados examinando todos os pixels para cada um dos objetos. Também foi testada uma versão baseada em um operador de redução simples para detectar cada objeto separadamente, a qual é descrita em (Bru07). Os testes foram computados usando um processador Intel Core 2 Duo E6550 2.33Ghz com 2GB de RAM e uma placa gráfica nVidia GeForce 8800 GTX (768MB).

O gráfico na Figura 3.8 demonstra que a implementação sequencial em CPU obteve os piores tempos. Quando observada a razão entre o tempo gasto pela computação e o número de objetos rastreados, a aplicação de múltiplas passadas da formulação proposta por Bruner et al. (Bru07) obtém uma taxa constante. Já para o MOCT, a taxa medida por essa razão declina com o aumento do número de objetos. O único cenário no qual a versão da proposta por Brunner et al. (Bru07) apresentou uma melhor performance foi para o



3.8(a): comparação geral



3.8(b): comparação entre versões em GPU

Figura 3.8: Comparação entre CPU, aplicação de (Bru07) repetidas vezes e o MOCT

rastreamento de um único objeto. Neste caso, a pequena diferença observada pode ser associada ao custo das computações de posicionamento em relação ao arranjo. Para todos os outros casos, o operador MOCT atingiu melhor desempenho. Os resultados comprovam a viabilidade de adotar o operador MOCT em aplicações de rastreamento em tempo real, uma vez que atinge de 500 fps a 40 fps nos testes apresentados, utilizando de 1 a 128 objetos.

Em um cenário mais geral, os resultados demonstram também que o operador de redução múltipla desenvolvido fornece um acesso otimizado à textura (ou memória global), confirmando ser mais eficiente que a aplicação de múltiplos operadores de redução simples, um para cada função a ser aplicada

ao fluxo original.

3.3.3 Avaliação do acesso à textura pelo operador MOCT

A análise apresentada nesta seção é inspirada por duas observações principais: a de que o número total de acessos à textura diminui com o aumento da taxa de redução por interação, e que os acessos à textura na placa gráfica são mais eficientes quando as amostras solicitadas foram previamente carregadas no cache do multiprocessador correspondente ao seu uso.

Analisando a quantidade de acessos à textura utilizados pelo operador de redução múltipla em sua segunda fase, podemos observar que na primeira passada de redução o número de acessos de textura é igual ao de entradas na base. O tamanho do fluxo gerado é reduzido e uma nova interação acontece, na qual novamente, todos os elementos gerados na etapa anterior precisam ser lidos. Tal ciclo se repete, até que, na última interação, são feitos tantos acessos de textura quanto for o produto do número de elementos no fluxo da análise global pelo fator de redução. Portanto, o total de acessos de textura da segunda etapa do operador de redução múltipla é calculado pela soma a seguir, na qual p é o tamanho do fluxo da base, r o fator de redução por interação, M o número de objetos rastreados, e as parcelas descrevem, respectivamente, da iteração inicial de redução da base, até a última iteração:

$$p + \lceil p/r \rceil + \lceil p/r^2 \rceil + ... + M * r^3 + M * r^2 + M * r \tag{3-1}$$

Na prática, nem sempre o valor de p é tal que a sequência acima possa ser estendida perfeitamente utilizando r fixo. Entretanto, podemos encontrar um limite superior para o número total de acessos à textura. Partindo da parcela da última interação, o somatório é desdobrado nas novas parcelas, tantas quanto for o mínimo de interações com r fixo para cobrir um fluxo de tamanho igual ou imediatamente maior que p. Portanto, um limite superior para o número de acessos de textura pelo operador de redução múltipla é obtido por:

$$M * r + M * r^2 + M * r^3 + ... + M * r^{i-2} + M * r^{i-1} + M * r^i \tag{3-2}$$

onde

$$i = min\left\{k \mid p \leq M * r^k\right\} \tag{3-3}$$

A soma descrita pela equação 3-2 é uma progressão geométrica de i termos, sendo o primeiro termo $a_1 = M * r$, e a progressão $a_n = r * a_n$. Portanto, o número de acessos à textura pelo operador de redução múltipla é limitado por $\frac{M*r(r^i-1)}{r-1}$ onde i é definido pela equação 3-3. Este limite indica

que o aumento do fator de redução acarreta um menor número de passos de processamento e diminuição do total de operações de leitura. No caso limite, uma única interação de redução é usada ($i = 1$), com um total de $M * r$ acessos, ou seja, cada amostra é lida apenas uma vez. Apesar de utilizar o mínimo de acessos à textura, esta formulação não representa necessariamente a melhor performance se vier a provocar a ociosidade de processadores, pela diminuição extrema do número de tarefas para M. Portanto, a análise proposta ressalta o equilíbrio necessário entre a distribuição de tarefas para computação paralela e um número total de acessos à textura, para promover uma utilização eficiente dos recursos da GPU pelo operador de redução múltipla.

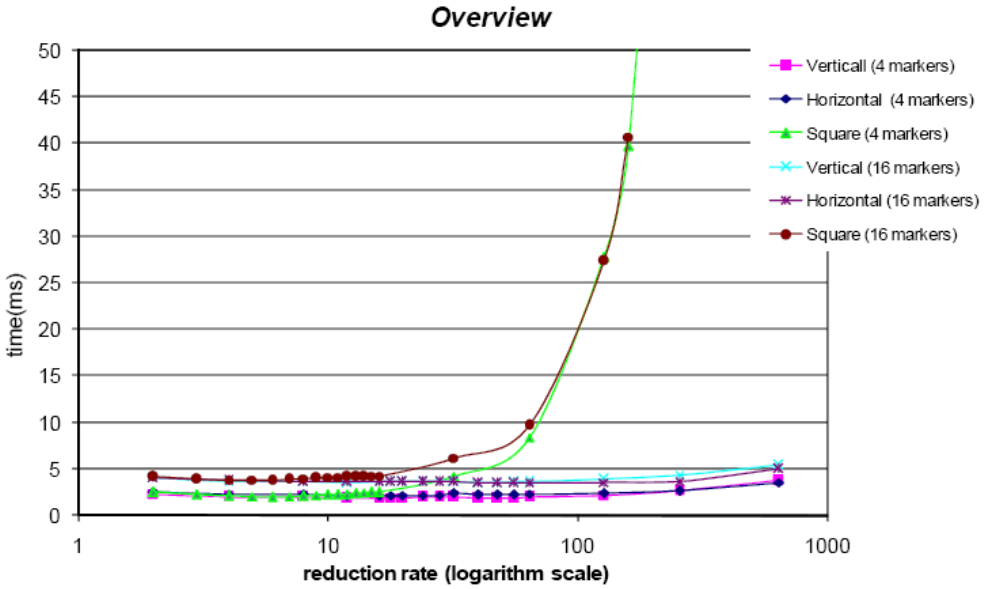
Outro fator testado com influência na performance do MOCT é a resposta aos três arranjos espaciais propostos. Apesar de não serem divulgadas as políticas adotadas pelos mecanismos de cache das placas gráficas, é comumente reconhecido que padrões de acesso às texturas interferem severamente na eficiência dos algoritmos.

Os gráficos apresentados na Figura 3.9 ilustram como o tempo de processamento é alterado em função do fator de redução para cada tipo de arranjo espacial. A melhor performance obtida pelos arranjos vertical e horizontal em relação ao quadrangular é explicada pela observação dos padrões de acesso à textura que acarretam. Durante a etapa de redução, as amostras de texturas lidas nos "kernels"codificando os arranjos verticais e horizontais são vizinhas, com alta chance de representar um "cache hit"em políticas de carregamento de regiões de textura para o cache. Por outro lado, nos "kernels"codificando arranjos quadrangulares, as amostras de textura correspondentes aos elementos a serem aglomerados são espaçadas pela distância igual ao lado do quadrado, aumentando a possibilidade de ocorrer um "cache miss".

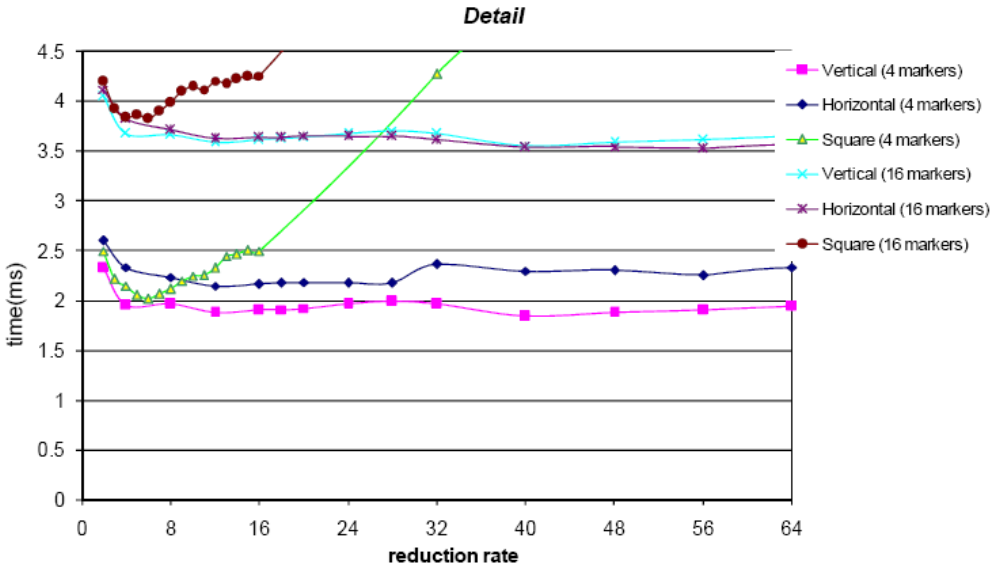
Os resultados apresentados podem ser estendidos a outros operadores de redução múltipla. Apesar de os tempos mostrados refletirem a placa gráfica específica utilizada, incentivam o uso de fatores de redução diferentes do tradicional 2×2 e confirmam a necessidade de compreensão dos padrões de acesso à textura no desenvolvimento de operadores de redução múltipla.

Implementação CUDA

Em CUDA, além da formulação para o operador de redução múltipla armazenando o fluxo original em textura, é possível armazená-lo em espaço de memória global. Permite estender as observações sobre padrões de acesso à textura e pré-carregamento apresentadas de forma controlada via programação. As disposições espaciais dos arranjos podem ser traduzidas em blocos de "threads"que utilizam o suporte oferecido ao pré-carregamento em memória



3.9(a): Visão Geral



3.9(b): Detalhes

Figura 3.9: Análise do fator de redução e arranjos

compartilhada de rápido acesso via programação dos dados contidos em memória global. Seja na primeira ou na segunda etapa do operador de redução múltipla, esse recurso permite controlar, via programação, o pré-carregamento de dados a serem acessados pelo respectivo arranjo.

3.4

Localização de objetos com Região de Interesse

O operador de redução múltipla apresentado na seção anterior processa cada um dos elementos do fluxo de entrada com cada operador a ser computado sobre esse fluxo. No MOCT, por exemplo, para a construção da base de avaliações locais, cada pixel da imagem de entrada é comparado com cada objeto rastreado. Entretanto, nem sempre as operações definidas no conjunto Ω procuram extrair características de todo o fluxo, mas apenas de alguns de seus trechos. Por este motivo, nestes casos a formulação do redutor múltiplo provoca desperdícios, tanto em recursos de memória quanto de operações realizadas. Tal desperdício pode tornar sua adoção inviável pelo aumento do número de elementos no fluxo de entrada ou do conjunto Ω.

Ainda seguindo o exemplo da aplicação de localização de objetos, é intuitivo perceber que quando aplicado a uma sequência de vídeo, supondo o movimento contínuo dos objetos na cena, o quadro corrente em processamento pode ser avaliado apenas pela busca dos objetos em regiões da imagem definidas ao redor das posições dos objetos no quadro anterior. Esta observação é válida para diversos algoritmos de rastreamento e correspondência em que a definição de regiões de interesse permite limitar a área a ser processada por um operador.

Na pesquisa de processamento de imagens, uma região de interesse (ROI – do inglês "Region of Interest") representa uma seleção de um subconjunto de amostras a partir de um conjunto de dados. Este conceito é aplicado a diferentes contextos e tipos de dados como por exemplo: pela escolha de um intervalo de tempo ou frequências, em um sinal 1D; pela escolha de uma área definida pela borda de um objeto no espaço 2D; de um volume definido pelo contorno de uma superfície em bases de dados tridimensionais; ou, ainda, por pares de contorno e intervalo de tempo em bases de dados tempo-volumétricas (4D). Portanto, a importância do conceito de ROI cresce em aplicações nas quais processar todo o conjunto de dados não é viável ou causa desperdício computacional.

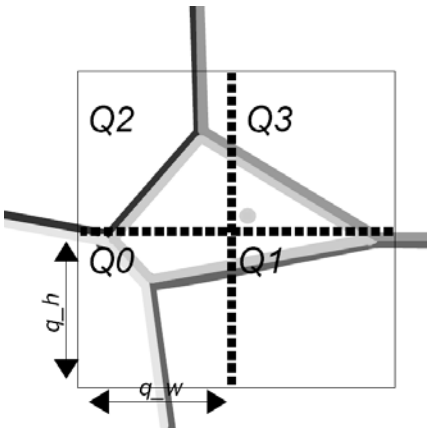
A seguir, formulamos um operador de processamento paralelo denominado operador de redução regional múltipla que incorpora o conceito de regiões de interesse ao padrão de programação paralela de redução. Sua proposição limita a aplicação dos operadores às suas respectivas regiões de interesse e permite distribuir o processamento realizado no interior de uma região em subdivisões independentes.

O operador paralelo e os resultados demonstrados nesta seção foram publicados no artigo (Vas08c) (Anexo B) em uma aplicação para computação em GPU de localização de centros de regiões definidas por um Diagrama Centroidal de Voronoi.

3.4.1
Operador de Redução Regional Múltipla

Propomos uma definição para o Operador de Redução Regional Múltipla (RRM) que altera a redução múltipla pela inclusão de um conjunto ordenado descrevendo M regiões de interesse $R = \{r_0, r_1, ..., rM-2, rM-1\}$, definidas sobre S, cada uma associada a um operador. O fluxo produzido pelo RRM é definido como a sequência de resultados da aplicação de cada operador em Ω sobre os elementos de sua região de interesse: $\{\Omega_0(r_0(S)), \Omega_1(r_1(S)), ..., \Omega_{M-2}(rM-2(S)), \Omega_{M-1}(rM-1(S))\}$.

Apresentamos, a seguir, a computação paralela do RRM sobre fluxos bidimensionais representando uma imagem e regiões de interesse definidas por quadrados idênticos ao redor de M pontos de interesse $p = p_0, p_1, .., p_{M-2}, p_{M-1}$. Para avaliar em paralelo os subfluxos gerados por essa combinação, construímos uma hierarquia de subdivisão espacial tomada em relação a pontos centrais dos quadrados (Figura 3.10).



3.10(a): Quadrantes

r24	r25	r26	r33	r34	r35
r21	r22	r23	r30	r31	r32
r18	r19	r20	r27	r28	r29
r6	r7	r8	r15	r16	r17
r3	r4	r5	r12	r13	r14
r0	r1	r2	r9	r10	r11

3.10(b): Retalhos

Figura 3.10: Hierarquia de fluxos de processamento das regiões de interesse

O nível mais alto da hierarquia é definido como o nível dos quadrantes. Ele é composto de um conjunto de quatro quadrantes tomados ao redor de cada ponto de interesse e ordenados da esquerda para a direita, de baixo para cima, $Q(p_i) = \{Q_0(p_i), Q_1(p_i), Q_2(p_i), Q_3(p_i)\}$. A área coberta pelos quatro quadrantes de um ponto de interesse define a região de interesse associada a este ponto.

O próximo nível da hierarquia subdivide cada quadrante em unidades regulares, aqui denominadas retalhos. O conjunto de retalhos no interior de um quadrante é ordenado da esquerda para direita, de baixo para cima. Cada retalho define uma sub-área no interior de um quadrante (subsequentemente, no interior da imagem), para ser considerada como um elemento independente,

sobre o qual o "kernel"traduzindo a operação a ser realizada é aplicado, para produzir um único elemento de saída. Este nível oferece um mecanismo de subdivisão do processamento das regiões de interesse em tantas unidades de processamento independentes quanto desejadas.

Cada retalho recebe um número identificador que representa sua posição no conjunto ordenado de retalhos associados ao mesmo ponto de interesse. A numeração é iniciada no retalho inferior esquerdo do quadrante Q_0 e incrementada sequencialmente, um a um, seguindo a ordenação imposta aos retalhos no interior de um quadrante. Após numerados todos os retalhos de um quadrante, a numeração continua obedecendo à mesma ordem para varrer os retalhos do quadrado seguinte. Portanto, o número de identificação de um quadrante, $Id_{retalho}$, é determinado usando:

$$Id_{retalho} = Q * \alpha + y' * \beta + x' \tag{3-4}$$

na qual Q representa o número do quadrante onde o retalho é situado (entre 0 e 3); x' e y' são as coordenadas horizontal e vertical do retalho dentro de seu quadrante, tomadas em unidades de retalho; e α e β são constantes representando, respectivamente, o número de retalhos por quadrante e por linha do quadrante. Na ilustração 3.10 são usados $\alpha = 9$ e $\beta = 3$.

A escolha dos valores de α e β pode ser feita dinamicamente pela análise do número de tarefas geradas e de multiprocessadores disponíveis. Idealmente, cada processador deve receber um conjunto de tarefas para alternar entre elas em tratamentos de ociosidade por "time sharing".

Assim como no operador de redução múltipla, o operador de redução regional múltipla também opera em duas etapas. A primeira é responsável por computações em subdivisões locais e a segunda coleta tais resultados parciais para produzir os resultados finais. A hierarquia apresentada define os subfluxos de aplicação de cada operação pelo quarteto de quadrantes ao redor do ponto de interesse e a maneira como os domínios locais são extraídos para o operador regional, no qual os retalhos fazem a delimitação antes proporcionada pelos arranjos.

Na primeira etapa do operador regional um fluxo bidimensional é criado, sendo a sua largura suficiente para cobrir tantos quantos forem os operadores em Ω e sua altura, para cobrir tantos quantos forem os retalhos criados por região de interesse. O fluxo armazenando as avaliações em subdivisões locais é ilustrado na Figura 3.11.

O "kernel"de processamento dessa etapa é então responsável por processar um retalho e produzir sua avaliação, preenchendo a entrada respectiva no fluxo 2D criado. A área coberta por cada processamento é determinada

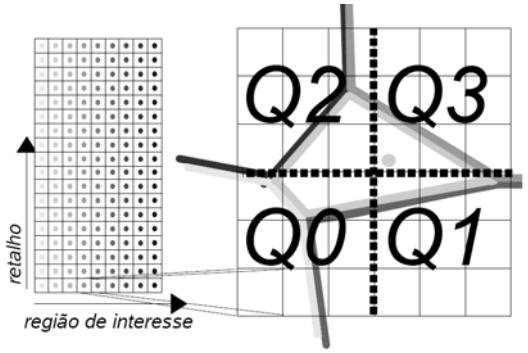


Figura 3.11: Base de avaliações Locais

revertendo o procedimento de numeração dos retalhos. Com esta finalidade, utilizam-se as coordenadas horizontal e vertical do elemento de saída a ser gerado, os valores constantes selecionados para α e β e o número de pixels por região de interesse.

Formada a base de avaliações, em uma segunda etapa do operador, um novo "kernel"agrupa os resultados, reduzindo o fluxo na dimensão vertical.

3.4.2 Aplicação Ilustrativa: Cálculo do Diagrama Centroidal de Voronoi Discreto

A seguir apresentamos um exemplo de aplicação desenvolvida com o operador de Redução Regional Múltipla, a qual possui o objetivo de computar o Diagrama Centroidal de Voronoi (CVD) discreto em GPU.

O CVD é uma variação do Diagrama de Voronoi na qual os geradores do diagrama (assumidos aqui como pontos) possuem a propriedade extra de estarem localizados nos centros de massa das regiões de Voronoi. O algoritmo clássico para sua computação sequencial é conhecido como algoritmo de Lloyd (ou "k-means", quando peso é associado a amostra). O algoritmo é composto por um ciclo de iterações que se encerra com a convergência dos pontos gerados em posições correspondentes aos centros de massa. O núcleo desse algoritmo basicamente alterna entre uma etapa que constrói o Diagrama de Voronoi utilizando as posições dos geradores e a função de densidade de ocupação do espaço e outra etapa que, a partir do Diagrama de Voronoi produzido, move os geradores para os centros de massa correspondentes às suas regiões(Figura 3.12).

Sobre a computação em paralelo do algoritmo de Lloyd em GPU, ainda que a etapa de computação do Diagrama de Voronoi possa ser computada utilizando diversas abordagens existentes na literatura para resolver este problema no domínio discreto, o mesmo não é válido para a etapa de cálculo de centróides. A desvantagem de uma solução parcialmente processada em GPU,

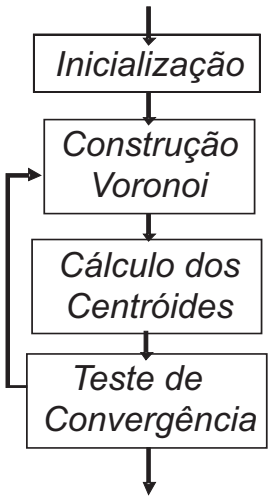


Figura 3.12: Algoritmo de Lloyd

na qual apenas o cálculo do Diagrama de Voronoi é processado na placa gráfica, é a necessidade de fazer transferências custosas de dados entre a GPU e CPU. Portanto, uma das motivações para processar as etapas do algoritmo de Lloyd em GPU é eliminar as repetidas transferências da representação do diagrama de Voronoi (normalmente armazenado em uma textura) entre a GPU e a CPU ao longo do ciclo do algoritmo.

Na aplicação apresentada em (Vas08b), apresentamos o cálculo dos centros de massa das regiões de Voronoi como um problema composto de múltiplos processamentos independentes sobre regiões de interesse distintas, definidas de maneira a cobrir regiões de Voronoi. Para encontrar o centro de massa correspondente a cada ponto gerador, uma região de interesse quadrangular ao redor de tal ponto gerador é analisada. Assim, a computação do conjunto de centróides é calculada a partir de um operador de Redução Regional Múltipla.

A princípio, o cálculo de centróides também pode ser feito utilizando o operador de Redução Múltipla. Seguindo a descrição apresentada para esse operador, durante a construção da base, as amostras de Voronoi no interior de um arranjo são comparadas com os identificadores de cada de região. Já na primeira fase do operador, este utiliza $(nP \times nO \times nI)$ acessos a textura, no qual nP é o número de pixels representando o espaço coberto pelo Diagrama de Voronoi, nO é o número de geradores e nI representa o número de interações do algoritmo de Lloyd até a convergência. Uma consequência imediata da utilização do operador de Redução Múltipla é que, mantendo a resolução de representação do Diagrama de Voronoi, o total de acessos a textura aumenta a cada novo gerador. Isto ocorre devido ao fato de o algoritmo de Redução Múltipla desconsiderar a repartição do espaço coberto pelo Diagrama de

Voronoi entre suas regiões e comparar todas as amostras do Diagrama de Voronoi *versus* todos os geradores para extrair as frequências locais (primeira fase do operador).

Por outro lado, a computação dos centróides na versão com o Operador de Redução Regional Múltipla utiliza um total de $(nR \times nO \times nI)$ acessos a textura, onde nR é o número de pixels na região de interesse (ao redor de cada primitiva geradora). Logo, quanto menor for a região de interesse comparada ao tamanho da imagem, ou seja, nR em relação a nP, maior a diferença entre os dois operadores. Com isso, apenas um grupo de amostras do Voronoi é comparado com cada gerador. Se considerarmos que tal grupo de amostras, definido pela extensão da região de interesse, pode ser reduzido com o aumento do número de geradores, então a utilização do operador de redução regional múltipla consegue manter uma taxa de acessos à textura aproximadamente constante para um número crescente de geradores.

Maiores detalhes sobre a proposta para computação em GPU do algoritmo de Lloyd podem ser encontradas no anexo B.