UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ACELERANDO O *MEAN-SHIFT* COM OPENCL

TRABALHO DE GRADUAÇÃO

Guilherme Gonçalves Schardong

Santa Maria, RS, Brasil

2011

ACELERANDO O *MEAN-SHIFT* COM OPENCL

por

Guilherme Gonçalves Schardong

Trabalho de Graduação apresentado ao Curso de Ciência da Computação
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Prof. Dr. Marcos Cordeiro d'Ornellas

Trabalho de Graduação N. 330

Santa Maria, RS, Brasil

2011

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

ACELERANDO O *MEAN-SHIFT* COM OPENCL

elaborado por
Guilherme Gonçalves Schardong

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Marcos Cordeiro d'Ornellas
(Presidente/Orientador)

Prof. Dr. Cesar Tadeu Pozzer (UFSM)

Prof. Dr. José Antônio Trindade Borges da Costa (UFSM)

Santa Maria, 15 de Dezembro de 2011.

“Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction.”

ALBERT EINSTEIN

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

ACELERANDO O *MEAN-SHIFT* COM OPENCL
Autor: Guilherme Gonçalves Schardong
Orientador: Prof. Dr. Marcos Cordeiro d'Ornellas
Local e data da defesa: Santa Maria, 15 de Dezembro de 2011.

Na área de análise de imagens existem vários métodos de segmentação de imagens que compõem processos mais complexos em uma sequência de filtros e operadores aplicados a um determinado conjunto de imagens de entrada. O mean-shift se enquadra nesse contexto, porém, o seu desempenho deixa a desejar em algumas situações, como o aumento do conjunto de dados e do número de dimensões do mesmo. Para corrigir esse problema, é proposta uma solução que implemente uma parte do algoritmo em OpenCL, visando acelerar a obtenção dos resultados. Os resultados obtidos mostram que a melhoria de desempenho é significativa, o que viabiliza mais pesquisas usando a metodologia do trabalho.

Palavras-chave: Mean-shift; GPGPU; processamento de imagens; OpenCL.

ABSTRACT

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

ACCELARATING THE *MEAN-SHIFT* USING OPENCL

Author: Guilherme Gonçalves Schardong
Advisor: Prof. Dr. Marcos Cordeiro d'Ornellas

In the field of image analysis there are several methods of image segmentation that make up for a more complex sequence of filters and operators applied to a certain set of images. The mean-shift fits in this context, however, their performance falls short in some situations, such as increasing the dataset and its number of dimensions. To fix this problem, we propose a solution that implements a part of the algorithm in OpenCL, looking forward to speed up the process of obtaining the results. The results show a significant increase of the algorithm's performance, which enables further research using this metodology.

Keywords: mean-shift, GPGPU, image processing, OpenCL.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision
GPGPU	General Purpose computation on Graphics Processor Units
LaCA	Laboratório de Computação Aplicada
API	Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO

O *mean-shift* é um método não paramétrico e não supervisionado com o objetivo de localizar as modas em um conjunto de dados quaisquer (FUKUNAGA; HOSTETLER, 1975). O método foi adaptado por Cheng (1995) para a análise de imagens e foi extendido por Comaniciu & Meer (2002) para as áreas de segmentação de imagens, suavização e detecção de movimento.

Porém, de acordo com Comaniciu & Meer (2002) a implementação do *mean-shift* possui problemas de desempenho. O algoritmo comum possui complexidade de tempo da ordem de $O(Tn^2)$, onde T é o número de iterações e n é o número de pontos do conjunto de dados (THIRUMURUGANATHAN, 2010). Diversas mudanças foram feitas sobre o algoritmo clássico com o fim de fazê-lo segmentar o conjunto inicial de dados mais rapidamente, entre elas encontram-se: Melhores algoritmos de busca multi-dimensionais (COMANICIU; MEER, 2002), algoritmos que permitem a variação da largura de banda para cada ponto do conjunto e alterar os pontos iniciais à medida que o algoritmo roda (THIRUMURUGANATHAN, 2010).

A maior parte das implementações existentes do *mean-shift* são voltadas para a área de processamento de imagens, e trabalham apenas com três canais de cor, o que dificulta o seu uso para outros propósitos mais generalizados, como mineração de dados e até mesmo a segmentação de imagens onde são necessários parâmentros extras além da cor de cada pixel. Um exemplo de uma implementação desse tipo é a existente na API OpenCV. Porém, existe uma implementação Matlab, feita por Bart Finkston (2006) que recebe dados com N dimensões. Essa implementação foi escolhida como base para a construção do protótipo do presente trabalho por ser mais genérica e por poder ser aplicada para outros propósitos onde o conjunto de dados precisa receber outros parâmetros além da cor.

Este trabalho propõe uma implementação do *mean-shift* utilizando o poder de processamento da GPU para diminuir o tempo de obtenção dos resultados. Visa-se a implementação parcial do algoritmo, principalmente o cálculo da distância euclidiana, em GPU, já que, estima-se, que essa etapa é a que mais influencia no tempo de resposta do algoritmo, já que o cálculo da distância é realizado de todos os pontos do conjunto para todos os outros pontos.

1.1 Objetivos do Trabalho

Esse trabalho tem por objetivo principal a implementação do *mean-shift* em GPU, mais especificamente, utilizando a API OpenCL, visando uma melhor distribuição do uso de recursos computacionais e consequentemente, uma diminuição do tempo de execução necessário para segmentar diversos conjuntos de dados.

Para verificar e validar a implementação serão realizados testes com imagens de diferentes tamanhos, bem como serão utilizadas diversas larguras de banda como entrada para o algoritmo. Também será testada a influência do conteúdo da imagem no resultado, ou seja, será verificado se imagens mais ricas em detalhes e variações de cor influenciam no tempo de processamento.

Os objetivos específicos propostos pelo trabalho são:

- Estudo e implementação do *mean-shift* para determinar possíveis melhorias a serem aplicadas ao algoritmo clássico;
- Estudo da arquitetura OpenCL com o fim de descobrir suas capacidades, limitações e possíveis contribuições para o projeto;
- Estudo da *API* OpenCV para manipular apropriadamente as imagens de entrada e resultantes do processamento;
- Alteração do protótipo para que faça uso da arquitetura OpenCL para paralelizar o processamento.
- Avaliação dos resultados obtidos com o protótipo e sugestões de melhorias para a aplicação produzida.

1.2 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma. O capítulo 2 apresenta trabalhos relacionados ao *mean-shift*, seus conceitos e aplicações. O capítulo 3 apresenta a proposição do trabalho e a justificativa para a escolha dos algoritmos utilizados no trabalho. A lógica do *mean-shift*, bem como o fluxo de execução da aplicação e sua arquitetura são explicadas no capítulo 4. O capítulo 5 apresenta e discute os resultados obtidos até então com a implementação construída. Por fim, o capítulo 6 apresenta as conclusões que podem ser inferidas do trabalho, bem como os possíveis trabalhos futuros e sugestões de melhoria a serem implementadas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem por objetivo a apresentação dos conceitos teóricos utilizados no projeto, principalmente aqueles relacionados ao *mean-shift* e as APIs envolvidas no desenvolvimento do projeto.

2.1 Conceitos Fundamentais

As subseções seguintes tem por objetivo a apresentação de alguns conceitos básicos fundamentais para o entendimento do presente trabalho.

2.1.1 Imagens, Pontos e Dimensões

Uma imagem é a representação da forma de um objeto, que é obtida pelo mapeamento de uma de suas propriedades físicas. As imagens digitais são grades de pontos, conhecidos como *pixels*, aos quais são associados valores discretos dos sinais mapeados (ROSA, 2008)

No caso de uma imagem em níveis de cinza, o sinal mapeado pode ser representando apenas por uma variável discreta associada a cada *pixel* (ou ponto). Nesse caso, a imagem é chamada de monocromática. Já em uma imagem colorida, é necessário que haja mais de uma váriavel discreta associada a cada *pixel*. Nesse caso as váriaveis são representadas na forma de um vetor. As imagens coloridas fazem parte dessa última classificação, pois normalmente são utilizadas três váriaveis, ou componentes espectrais, para representar as cores presentes na imagem.

No contexto do presente trabalho, as componentes espectrais de uma imagem são chamadas de **dimensões**. Ou seja, uma imagem monocromática possui apenas uma componente espectral, ou dimensão e uma imagem colorida em RGB possui três componentes espectrais ou dimensões.

Figura 2.1: Exemplo de segmentação de imagens.

2.1.2 Segmentação de Imagens

A segmentação de imagens trata da divisão de uma imagem em seus objetos constituintes. Do ponto de vista do processamento de imagens digitais, a segmentação pode ser vista como a classificação dos *pixels* como pertencentes a diferentes objetos presentes na imagem. Assim, os objetos passam a ser considerados como classes de *pixels* e ao final do processo, todos os *pixels* são rotulados como pertencentes a uma dessas classes (ROSA, 2008).

A imagem 2.1 é um exemplo de uma imagem de entrada e da imagem resultante da segmentação da mesma.

2.2 *Mean-shift*

O *mean-shift* é um método não paramétrico, iterativo, com a finalidade de estimar o gradiente de uma função densidade de probabilidade, dado um conjunto de dados discretos amostrados dessa função.

Este método é normalmente utilizado para encontrar modas de funções densidade, clusterização e segmentação de sinais (THIRUMURUGANATHAN, 2010). A sua concepção se deu em 1975 por Fukunaga & Hostetler (FUKUNAGA; HOSTETLER, 1975) como um algoritmo que estima gradiente de uma função densidade com aplicações em reconhecimento de padrões. Desde então foi expandido para outras áreas, como processamento e análise de imagens, visão computacional e mineração de dados.

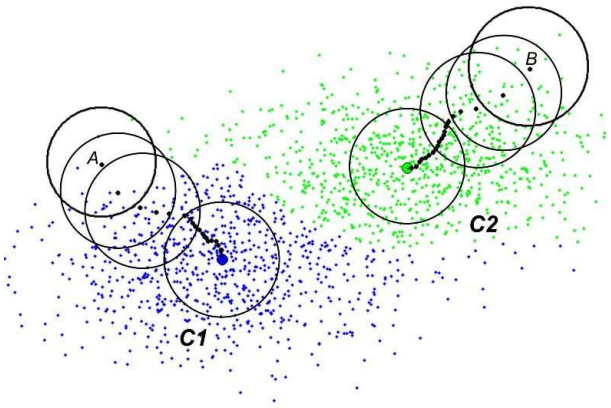


Figura 2.2: Ilustração do funcionamento do *mean-shift*. Os clusters encontrados foram representados em azul (C1) e verde (C2).

Na figura 2.2 pode ser observado o comportamento do *mean-shift*, o algoritmo escolhe um ponto arbitrário do conjunto de dados e determina uma janela ao redor desse ponto. Após essa etapa, é calculada a média dos pontos que estão dentro dessa janela, assim, o novo centro de massa é determinado e a janela é deslocada de forma que fique centrada no ponto calculado. Esse procedimento é repetido até que a média não se desloque mais, ou que atinja um determinado critério de parada definido. Esse ponto final é o centro de massa do *cluster*, também conhecido como centróide.

Os *clusters* são regiões do espaço onde a densidade de pontos é mais alta. Se a função densidade $f(x)$ fosse conhecida, os *clusters* poderiam ser identificados como regiões ao redor das modas da função. Já que esta função não é conhecida, pode-se estimá-la a partir de amostras. Os máximos da função densidade estimada $\hat{f}(x)$ podem ser encontrados usando o gradiente, ou seja, a busca dos *clusters* pode ser feita pelo cálculo do gradiente da densidade estimada $\nabla \hat{f}(x)$ (ROSA, 2008).

Para realizar a segmentação via *mean-shift* basta que sejam adicionados alguns passos após a execução do algoritmo em si. Esses passos consistem em associar cada ponto do conjunto de dados, ao seu centro de massa, assim, todos os pontos serão associados a um centróide e cada centróide definirá um *cluster* de dados.

Podem ocorrer casos em que um ou mais pontos não sejam associados a nenhum centróide, pois estão fora da janela do mesmo. Esses pontos podem ser adicionados ao *cluster* mais próximo, ou podem ser descartados se o seu número for pequeno o suficiente.

Dado $\{x_i\}_{i=1...n}$ um conjunto arbitrátio de n pontos em um espaço euclidiano d-dimensional $\Re^d$, a estimativa da densidade multivariada do kernel obtida com o *kernel* $K(x)$ e o raio h computada no ponto x é definido por:

$$\hat{f}(\vec{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{\vec{x} - \vec{x_i}}{h}\right). \tag{2.1}$$

O uso de um *kernel* diferenciável permite que a estimativa do gradiente da densidade seja definida como o gradiente do *kernel* estimador de densidade definido em 2.1 (COMANICIU; MEER, 1999);

$$\nabla \hat{f}(\vec{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} \nabla K\left(\frac{\vec{x} - \vec{x_i}}{h}\right) \tag{2.2}$$

O vetor de *mean-shift* pode ser calculado usando a equação 2.3 definida inicialmente por Fukunaga & Hostetler (1975):

$$M_h(\vec{x}) = \frac{1}{n_{\vec{x}}} \sum_{\vec{x_i} \in S_h(\vec{x})} \vec{x_i} - \vec{x} \tag{2.3}$$

O termo $\frac{1}{n_{\vec{x}}} \sum_{\vec{x_i} \in S_h(\vec{x})} \vec{x_i}$ em 2.3 é a média dos pontos amostrados dentro de uma hiperesfera $S_h(\vec{x})$ com raio h e centrada em $\vec{x}$. Esta média corresponde ao centro de massa da hiperesfera. O *mean-shift* é a diferenca entre essa média da amostra e o ponto $\vec{x}$ (ROSA, 2008).

Assim, o algoritmo de localização de modas baseado no *mean-shift* pode ser definido como:

1. Compute o vetor *mean-shift* $m(\vec{x})$ para um ponto de entrada;
2. Translade a janela (*kernel*) de forma que fique centrada em $m(\vec{x})$;
3. Repita os passos acima até que a janela não se mova, ou que algum critério de parada seja atingido.

O *mean-shift* trata os pontos do conjunto de dados como estimativas de uma função de densidade de probabilidade (THIRUMURUGANATHAN, 2010). Isso siginifica que regiões com uma grande concentração de pontos, ou densas, correspondem a máximos locais (também chamados de modas) da função. Para cada ponto, é feita uma operação de ascenção em gradiente na densidade local até que ocorra a convergência, em outras palavras, são descobertos os máximos locais mais próximos ao ponto. Todos os pontos que forem associados a mesma moda, pertencem ao mesmo *cluster*.

Com essas informações em mente, o algoritmo de segmentação por *mean-shift* definido por Comaniciu & Meer (COMANICIU; MEER, 1999) é uma extensão do algoritmo de localização de modas citado acima, e é definido a seguir:

Dado um conjunto de dados $\{x_j\}_{j=1...n}$, um conjunto $\{z_j\}_{j=1...n}$ de pontos de convergência e um conjunto de *labels* $\{L_j\}_{j=1...n}$, o algoritmos de segmentação via *mean-shift* é definido como:

1. Para cada $j = 1...n$ rode o algoritmo *mean-shift* para x_j e armazene o ponto de convergência em z_j;
2. Identifique os *clusters* $\{C_p\}_{p=1...n}$ de pontos de convergência relacionando todos os z_j que estiverem a uma distância menor que o raio da janela uns dos outros;

3. Para cada $j = 1...n$ faça $L_j = \{p|z_j \in C_p\}$ de forma que cada ponto j seja rotulado com pertencente ao *cluster* p;

4. *(Opcional)*: Elimine as regiões espaciais menores que M pontos.

Nas seções seguintes serão definidos alguns conceitos importantes para o entendimento do funcionamento do *mean-shift*.

2.3 Distribuição de Probabilidade

Na área da teoria de probabilidade, uma distribuição de probabilidade é uma função que descreve as probabilidades de uma variável aleatória assumir determinados valores (StarTrek.com, 2011).

Para uma descrição mais detalhada, deve haver a distinção entre variáveis contínuas e variáveis discretas. No caso de variáveis discretas, pode ser feita uma associação entre um valor da variável e a probabilidade de sua ocorrência. Por exemplo, quando é jogada uma moeda, pode-se atribuir uma probabilidade para cada face, no caso, 1/2.

Em contrapartida quando uma varíavel é contínua, as probabilidades possuem valores diferentes de zero apenas se elas se referem a intervalos finitos. Por exemplo, na área de controle de qualidade, pode ser exigido que a probabilidade de uma pacote conter entre 500 e 510 gramas de conteúdo, não seja menor que 96%.

Nas subseções a seguir, serão explicadas as diferenças entre funções densidade e massa de probabilidade.

2.3.1 Função de Massa de Probabilidade

No campo da estatística, uma função massa de probabilidade é uma função que resulta na probabilidade de uma variável aleatória discreta assumir um determinado valor exato (JOHNSON; KEMP; KOTZ, 2005).

A diferença entre uma função massa de probabilidade e uma função densidade de probabilidade, é que a última trata de variáveis contínuas ao invés de discretas, e não resulta na probabilidade da ocorrência de um valor exato, mas sim do valor estar em um intervalo infinitesimal no domínio.

Dada X, uma váriavel definida em $A \rightarrow \Re$, A um espaço de amostras. Então, a função massa de probabilidade $f(x) : \Re \rightarrow [0, 1]$ é definida como:

$$f_X(x) = Pr(X = x) = Pr(a \in A : X(a) = x) \tag{2.4}$$

2.3.2 Função de Densidade de Probabilidade

A função densidade de probabilidade de uma variável aleatória contínua descreve a probabilidade relativa da ocorrência de um determinado valor em um certo intervalo do domínio dessa variável.

A probabilidade de uma variável pertencer a um determinado intervalo é dada pela integral da função densidade da variável nessa região.

Dada X, uma variável qualquer e $f(x)$ a sua função de densidade, a probabilidade de X ocorrer no intervalo $[a, b]$ é dada por (WEISSTEIN, 2011):

$$P[a \leq X \leq b] = \int_a^b f(x)dx \tag{2.5}$$

Apesar disso, a probabilidade de X possuir um valor definido a é zero, pois a integral de uma função onde os limites superior e inferior coincidem, é zero.

Formalmente, cada valor possui uma probabilidade infinitesimalmente pequena de ocorrer, essa probabilidade é estatisticamente igual a zero.

2.4 Ascenção em Gradiente

O algortitmo de ascenção em gradiente visa encontrar um máximo local de uma função com base em um ponto pertencente ao domínio da mesma.

Para encontrar um máximo local, o algoritmo recebe um ponto como entrada e calcula o novo ponto baseado em um valor proporcional ao gradiente da função no ponto escolhido. Após esse passo ele recalcula o gradiente para o novo ponto. Esse processo é repetido enquanto o gradiente for não-nulo, ou seja, enquanto o máximo local não for atingido.

Dada $F(x)$ uma função definida e diferenciável na vizinhança do ponto x_n, a função cresce mais rapidamente quando, a partir de x_n, ela se move na direção do gradiente de $F(x)$. A forma geral da ascenção em gradiente é dada a seguir (VENKATARAMA, 1999):

$$x_{n+1} = x_n + \eta\nabla(F) \tag{2.6}$$

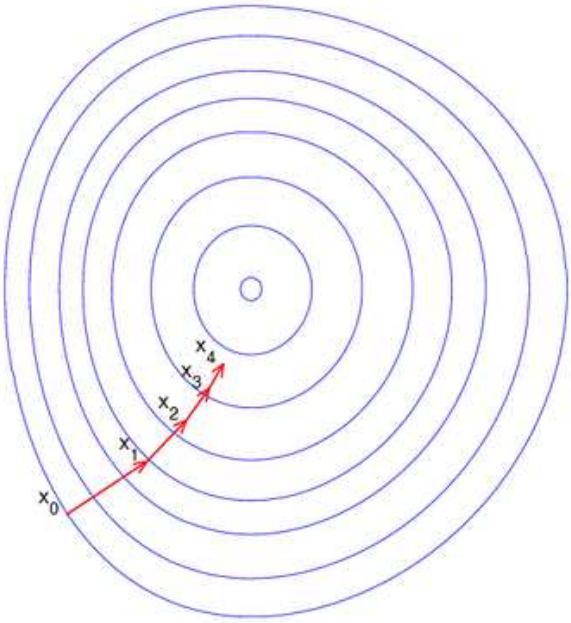


Figura 2.3: Ilustração da ascenção em gradiente

onde $\eta > 0$ é um número real suficientemente pequeno e ∇ é o operador de gradiente. O método continua até que atinja um máximo local da função $F(x)$.

Na figura 2.3 é possível observar o comportamento do algoritmo de ascenção em gradiente.

2.5 Clusterização via *K-Means*

Na área da estatítica, o *k-means* é um algoritmo com o objetivo de particionar um conjunto de dados em clusters distintos, associando cada observação pertence ao cluster mais próximo.

O procedimento recebe como entrada um número definido k de clusters existentes e um conjunto de dados com n elementos. Com esses dados, o algoritmo define k centróides, um para cada cluster, os posiciona no conjunto de entrada. Esse posicionamento é extremamente importante, pois ele definirá o resultado obtido, por isso, normalmente os centróides são postos o mais longe possível uns dos outros (MATTEUCCI, 2011). O próximo passo envolve a associação de cada ponto do conjunto de dados ao centróide mais próximo. Após esse procedimento, o primeiro agrupamento está completo, assim, os centróides devem ser recalculados como sendo os baricentros dos clusters resultantes, gerando um novo conjunto de centróides. Novamente, os pontos do conjunto devem ser

associados aos novos centróides. Esse processo é repetido até que os centróides não se desloquem. O resultado desse processo é um conjunto de dados segmententado em k clusters distintos (MACQUEEN, 1967).

O algoritmo em si é composto dos seguintes passos:

1. Posicione os k centróides no conjunto de dados;
2. Associe cada ponto do conjunto de dados ao centróide mais próximo;
3. Assim que todos os pontos forem associados, recalcule as posições dos centróides;
4. Repita os passos 2 e 3 até que os centróides parem de se mover.

2.6 OpenCL

OpenCL é um padrão de desenvolvimento aberto, multi-plataforma, que visa possibilitar a programação de plataformas heterogêneas (Khronos Group, 2008). Ou seja, é possível desenvolver aplicações que utilizem o poder de paralelização de dispositivos com características diferentes, como, CPUs, GPUs, e outras arquiteturas. Dessa forma, é possível manter a portabilidade e obter um bom ganho de desempenho proporcinado pelo poder de paralelização dos dispositivos utilizados.

O padrão inclui uma linguagem baseada em C99 para o desenvolvimento dos trechos de códigos a serem paralelizados (*kernels*), bem como uma API usada para configurar e controlar as plataformas de execução (Khronos Group, 2010).

O OpenCL é análogo aos padrões OpenGL e OpenAL, para gráficos e som, respectivamente, tanto que existe a possibilidade de interoperabilidade entre OpenCL e OpenGL. Até agora, o padrão foi adotado pela AMD, ARM, Intel e Nvidia, e está sendo desenvolvido pelo grupo Khronos (2008). Sua proposta inicial foi lançada pela Apple em 2006.

O código 2.6.1 é um exemplo de um *kernel* escrito em OpenCL. Esse trecho de código recebe um vetor de números inteiros sem sinal e atribui para cada posição o seu índice no vetor. O código presente no exemplo será rodado de forma a aproveitar o paralelismo da plataforma utilizada, sendo GPU, ou CPU, ou outros tipos de dispositivos.

Para possibilitar essa variedade de plataformas, a preparação da plataforma antes da execução é mais extensa que em outros padrões. O código do exemplo 2.6.2 ilustra essa preparação. O código do *kernel* é codificado na forma de uma cadeia de caracteres, e então é passado para o contexto apropriado, para que o código executável seja corretamente

compilado. Isso permite que o OpenCL possua suporte para uma grande variedade de plataformas distintas.

```
__kernel void memset(__global uint* a)
{
  int i = get_global_id(0);
  a[i] = index;
}
```

Código 2.6.1: Exemplo de um *kernel* em OpenCL.

```
int main(int argc, char** argv) {
  char* source = "__kernel void memset(__global float* a)"
                 "{"
                 "  int index = get_global_id(0);"
                 "  a[index] = index;"
                 "}";
  cl_platform_id platform;
  cl_device_id device;
  cl_context context;
  cl_command_queue queue;
  cl_program program;
  cl_kernel kernel;
  cl_mem buffer;
  clGetPlatformIDs(1, &platform, NULL);
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
  context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
  queue = clCreateCommandQueue(context, device, 0, NULL);
  program = clCreateProgramWithSource(context, 1, &source, NULL,
                 NULL);
  clBuildProgram(program, 1, &device, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "memset", NULL);
  buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                 ITEMS * sizeof(cl_uint), NULL, NULL);
  size_t global_work_size = 512;
  clSetKernelArg(kernel, 0, sizeof(buffer), (void*) &buffer);
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
                 &global_work_size, NULL, 0, NULL, NULL);
  clFinish(queue);
  cl_uint* ptr;
  ptr = (cl_uint*) clEnqueueMapBuffer(queue, buffer, CL_TRUE,
                 CL_MAP_READ, 0, ITEMS * sizeof(cl_uint), 0,
                 NULL, NULL, NULL);
  for(int i = 0; i < 512; i++)
    printf("%d\n", ptr[i]);
  return 0;
}
```

Código 2.6.2: Preparação do contexto e execução do *kernel* em OpenCL.

2.6.1 Modelo de Execução

Como o OpenCL foi projetado para uma grande variedade de dispositivos, a flexibilidade é dada pelo tipo de *kernel* especificado. Os *kernels* podem possuir paralelismo de dados, que se adequa muito bem a arquitetura das GPUs, ou paralelismo de tarefas, que se adequa a arquitetura das CPUs.

Um *kernel* pode ser considerado uma unidade básica de código executável, similar a uma função na linguagem C. Diversos eventos são disponibilizados para que o desenvolvedor possa verificar o estado da execução do *kernel*.

Em termos de organização, o modelo de execução do *kernel* pode ser definido como um domínio de computação N-dimensional de itens de trabalho, por exemplo, um domínio com uma dimensão seria um vetor, duas dimensões, uma matriz de itens de trabalho. Cada elemento do domínio de execução é um item de trabalho, ou seja, uma *thread*, e o OpenCL agrupa esses itens de trabalho em grupos de trabalho para facilitar a sincronização e a comunicação entre os mesmos.

O conceito de programa usado em OpenCL é definido como uma coleção de *kernels* e outras funções relacionadas, ou seja, pode ser considerado uma biblioteca dinâmica (Khronos Group, 2010).

Os passos gerais para a preparação do contexto para a execução de um *kernel* seguem abaixo, todos eles podem ser vistos no exemplo 2.6.2.

1. Identificação da plataforma disponível;
2. Identificação dos dispositivos presentes na plataforma;
3. Criação de um contexto de execução com um ou mais dispositivos encontrados na etapa anterior;
4. Definir uma fila de execução de comandos para cada dispositivo;
5. Carregar o código dos *kernels* e compilá-los;
6. Alocar as regiões de memória no dispositivo e transferir os dados da memória principal para a memória do dispositivo;
7. Executar o *kernel* e aguardar o seu término;
8. Copiar os resultados de volta para a memória principal.

Todas as etapas anteriores envolvem verificação de erros. Estas verificações não foram exibidas no exemplo para facilitar o entendimento.

2.6.2 Modelo de Memória

O modelo de memória do OpenCL é constituído de diversas camadas distintas, com a memória privada sendo visível apenas para um item de trabalho, até a memória global, visível por todos os itens de trabalho (AMD, 2008).

O OpenCL 1.0 define quatro níveis de memória: privada, local, constante e global. A figura 2.4 ilustra a hierarquia de memória utilizada pelo OpenCL e o código 2.6.3 ilustra a passagem de parâmetros em diferentes espaços de memória.

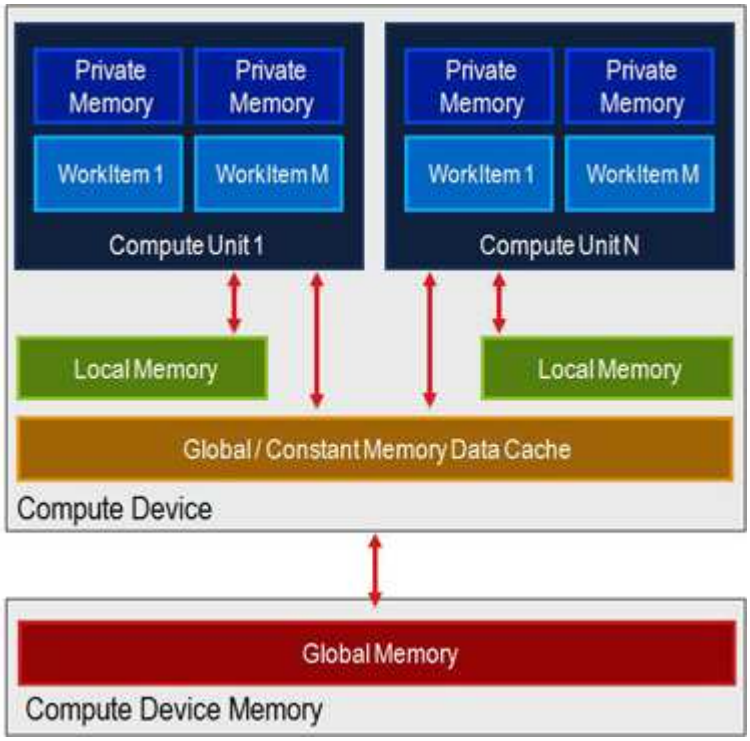


Figura 2.4: Hierarquia de memória do OpenCL.

Para definir a qual espaço de memória uma variável pertence, basta que sejam adicionadas algumas palavras chave antes do tipo da variável. A palavra **__private** indica que a variável pertence a memória privada de um item de trabalho. Já a palavra chave **__local** indica que a variável pertence a memória local de um grupo de trabalho. A palavra **__constant** faz a variável pertencer a memória constante. Por fim, a palavra chave **__global** indica que uma variável pertence a memória global do dispositivo.

A memória privada é visível apenas para um item de trabalho, funcionando como um conjunto de registradores para o mesmo. A memória local é vísivel para todos os itens de trabalho em um grupo, permitindo a comunicação e compartilhamento de dados entre os mesmos. A memória constante é utilizada para armazenar dados que não podem ser

alterados durante a execução do *kernel*, ou seja, esses dados podem ser lidos, mas não alterados, e são visíveis para todos os itens de trabalho de um dispositivo. E a memória global é visível por todos os itens de trabalho de um dispositivo e pode ser lida e alterada por eles.

É importante salientar que as memórias privadas, locais e constantes possuem um tempo de acesso menor que a memória global, porém são menores e impõem restrições a visibilidade e acesso aos dados. Com isso em mente é possível projetar *kernels* que utilizem melhor essas memórias, aumentando o desempenho da aplicação.

```
__kernel void mykernel(__constant float* input,
                       __private int num_items)
{
  int index = get_global_id(0);
  input[index] = index \% 255;
}
```

Código 2.6.3: Exemplo de declaração de variáveis em diferentes espaços de memória em OpenCL.

3 PROPOSIÇÃO DO TRABALHO

Conforme as justificativas apresentadas no capítulo 1, e as explicações fornecidas no capítulo 2, este trabalho propõe uma abordagem para melhorar o desempenho do *mean-shift*. A abordagem proposta envolve o uso do poder de paralelização das GPUs modernas para melhorar o tempo de execução do algoritmo e o uso mais eficiente dos recursos computacionais disponíveis por parte do *mean-shift*. Outras abordagens foram utilizadas em trabalhos anteriores. Elas envolvem a implementação de algoritmos para pré processar os dados a serem enviados para o *mean-shift* (SAHBA; VENETSANOPOULOS, 2010) e algoritmos de busca em espaços multidimensionais para localizar os pontos da vizinhança de um *kernel* (COMANICIU; MEER, 2002).

O objetivo principal do trabalho é melhorar o desempenho do *mean-shift*. Para isso será utilizada a arquitetura OpenCL para paralelizar o trecho de código responsável pelo cálculo da distância euclidiana entre o centro de massa atual e todos os pontos do conjunto de dados. O uso da plataforma OpenCL se justifica pelo fato de que ela proporciona flexibilidade quanto a plataforma de execução, ou seja, ela permite que o algoritmo possa ser executado tanto em GPU quanto em CPU, resultando um aumento de desempenho mesmo em caso de ausência de uma GPU adequada, e possibilitando que o código seja portável, conforme explicado na seção 2.6 do presente trabalho.

A partir das imagens de entrada é possível montar um espaço de características N-dimensional, onde cada elemento é representado por um vetor neste espaço. O *mean-shift* consiste de localizar o centro de massa mais próximo a cada elemento do espaço de características. Para isso, a partir de cada ponto, são feitas diversas iterações, em que é calculada a distância euclidiana do ponto sendo processado a todos os pontos do conjunto de dados. Então é calculada a média de todos aqueles pontos que estiverem dentro da hiperesfera centrada no ponto atual e de raio igual a largura de banda. Esse

processo é repetido para todos os pontos do espaço de características. O resultado desse processamento é o conjunto inicial segmentado em diversos *clusters* distintos.

A proposta do trabalho é paralelizar o cálculo da distância euclidiana utilizada para determinar os pontos dentro das hiperesferas determinadas por cada um dos centros de massa. Esse trecho particular foi escolhido pois ele é repetido para todos os elementos do espaço de características em relação a todos os outros elementos do mesmo espaço, tornando-o particularmente lento quando o número de elementos é muito elevado. Assim é esperado que haja uma melhora substancial no desempenho geral do *mean-shift* e para determinar se há uma melhoria e avaliar se ela é significativa, será construído um protótipo que implementa o *mean-shift* com o cálculo da distância euclidiana implementado tanto em OpenCL quanto em C++.

As próximas seções justificam a escolha do algoritmo *mean-shift* ao invés do *k-means*, bem como o uso da plataforma OpenCL em detrimento da arquitetura CUDA.

3.1 *Mean-shift* Vs *K-Means*

O *k-means*, como explicado no capítulo 2 recebe como parâmetro um número de clusters a serem encontrados, posiciona-os de acordo com um critério escolhido pelo desenvolvedor no conjunto de dados, associa os pontos do conjunto a cada centróide mais próximo, recalcula os centróides e repete o processo até que os centróides não se movam mais. Esse processo é todo baseado no fato do usuário informar a quantidade de *clusters* a serem encontrados pelo algoritmo, o que exige que o usuário estime o número de regiões presentes, ou que seja construída uma heurística para tentar determinar esse valor.

Diferentemente do *k-means* o *mean-shift* precisa apenas ser informado da largura de banda, também chamado de raio, para poder operar, dado que ele escolhe um ponto central do conjunto de dados e calcula os novos centros de massa com base nos pontos que se encontram a um determinado raio do ponto central, após esse cálculo, ele centra a janela no novo ponto e recalcula a média, até que a janela não se mova mais, assim, ele associa os pontos que estão dentro da janela ao centróide correspondente.

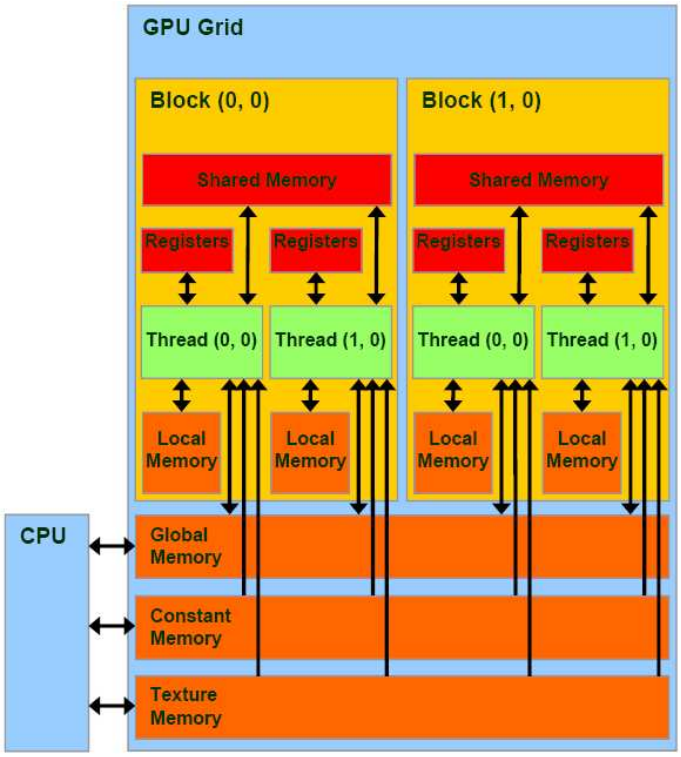
Dado que o *mean-shift* não exije que o usuário estime o número de *clusters* presentes no espaço de características, ele foi escolhido para ser paralelizado no presente trabalho. O único problema que persistirá é a escolha de um valor de largura de banda apropriado, esse problema não será tratado no contexto do presente trabalho.

3.2 OpenCL e CUDA

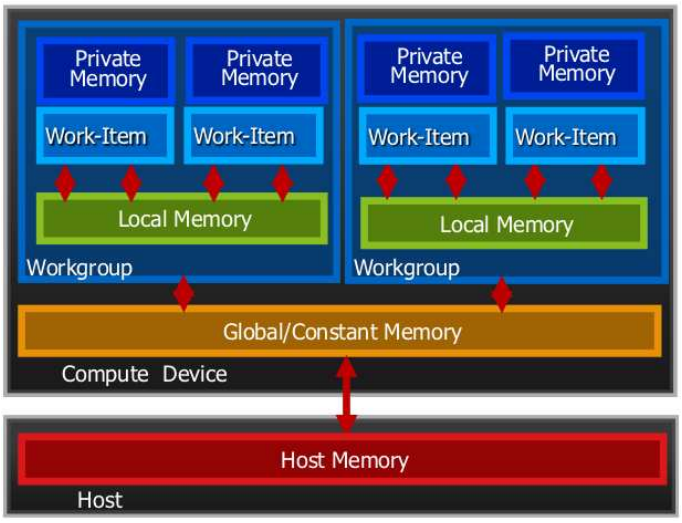
De acordo com o grupo Khronos (Khronos Group, 2008), o OpenCL é um padrão de desenvolvimento aberto e voltado para a programação de plataformas heterogêneas. Ou seja, o usuário de uma aplicação que utiliza OpenCL não precisa de uma GPU para obter os benefícios fornecidos pela paralelização da aplicação. Há também o aumento da portabilidade de código escrito em OpenCL, visto que o mesmo código pode ser executado em diversas plataformas, basta que o usuário escolha a plataforma que o código será compilado de acordo.

Já a arquitetura CUDA desenvolvida pela NVIDIA (NVIDIA, 2011) é definida como uma arquitetura de computação paralela de propósito geral que usa o hardware das GPUs da NVIDIA para resolver diversos problemas computacionais. Embora as arquiteturas OpenCL e CUDA sejam similares em muitos pontos, o fato de CUDA precisar de uma GPU da NVIDIA diminui significativamente a portabilidade da aplicação. Esse foi o principal motivo da escolha do OpenCL para a implementação do protótipo do *mean-shift*.

Na figura 3.1 temos uma comparação das arquiteturas CUDA e OpenCL respectivamente. É possível observar uma grande semelhança entre elas. Os modelos de memória são praticamente os mesmos, bem como a organização dos elementos de execução em grupos de *threads*. A diferença entre CUDA e OpenCL é que a última tem suporte à execução em outras arquiteturas diferentes de uma GPU, mas utiliza o mesmo modelo de execução e memória, aumentando drasticamente a portabilidade do código produzido.



(a) Arquitetura CUDA



(b) Arquitetura OpenCL

Figura 3.1: Comparação entre as arquiteturas CUDA e OpenCL respectivamente.

4 IMPLEMENTAÇÃO

Para a implementação do *mean-shift* foi utilizada a versão 1.1 do OpenCL, bem como as versões 2.1 e 2.3 do OpenCV. Como a aplicação foi desenvolvida tanto sobre Linux quanto Windows nenhuma biblioteca específica desses sistemas foi utilizada na implementação. Para o desenvolvimento no Windows foi utilizado o ambiente de desenvolvimento integrado Visual Studio 2010, já para o Linux, o desenvolvimento foi feito predominantemente sobre um editor de texto emacs, bem como o aplicação *make* para automatizar a compilação e a ligação da aplicação.

Ultimamente o OpenCV tem sido atualizado com maior frequência o que gerou uma necessidade de adaptação da aplicação para que não utilizasse uma versão muito antiga desta API. Porém não existe um instalador para Windows para a última versão da API (2.3), mas após um estudo das funções utilizadas, foi determinado que não havia a necessidade de uma grande refatoração do código, houve apenas a necessidade de adaptar as bibliotecas ligadas ao executável e os arquivos de cabeçalho utilizados. Para o Linux não houveram problemas com a versão do OpenCV, já que os repositórios da distribuição utilizada possuem a última versão disponível do OpenCV.

Durante o processo de desenvolvimento houve uma atualização do OpenCL para a versão 1.2, adicionando novas funcionalidades à API e potencialmente modificando algumas funcionalidades existentes. Portanto, foi necessário realizar um estudo básico sobre os impactos dessa atualização no trabalho desenvolvido e foi determinado que a versão 1.2 é retrocompatível com as versões 1.0 e 1.1. Essa retrocompatibilidade se mostrou vantajosa, já que não houve a necessidade de uma reestruturação do trabalho feito até então. Mas é interessante que haja um estudo futuro mais aprofundado para determinar se as funcionalidades novas implementadas nessa atualização podem auxiliar a melhoria de desempenho do *mean-shift*.

Neste capítulo são discutidos alguns dos aspectos mais relevantes da implementação do *mean-shift*, bem como a arquitetura da aplicação confeccionada, o fluxo de execução da aplicação, bem como uma descrição da implementação do método principal do *mean-shift*.

4.1 Arquitetura da Aplicação

A aplicação implementada possui uma estrutura relativamente simples, já que o objetivo principal do trabalho é a melhoria de desempenho do *mean-shift*. Existem apenas quatro classes implementadas, uma que implementa o *mean-shift*, outra classe de abstração, que, entre outras funções, realiza o pré-processamento das imagens a serem passadas para o *mean-shift* e fornece o acesso ao *mean-shift* em si. Há também uma classe que implementa um *timer* de alta resolução para a maior precisão na medição do tempo de execução do algoritmo. A última classe implementa o padrão de programação *Singleton*, que tem por função garantir que exista apenas uma instância da classe de abstração, evitando assim potenciais problemas de concorrência pelos recursos da máquina.

A classe MeanInterface foi feita para abstrair o acesso ao *mean-shift*, facilitando a construção de uma interface gráfica ou por linha de comando para utilizar o algoritmo. Além das funções de acesso, a classe MeanInterface é responsável pelo pré-processamento das imagens de entrada para a forma de uma matriz com D linhas e N colunas, onde D é o número de imagens e N é o número de pixels de cada imagem. Para que o algoritmo funcione apropriadamente, todas as imagens devem possuir o mesmo número de pixels.

Há a opção de cada imagem constituir apenas uma dimensão, ou seja, uma linha da matriz de características, para isso, a imagem é carregada em níveis de cinza. Caso a imagem for colorida, há a opção de carregar todos os seus canais, dessa forma, cada canal constitui uma dimensão na matriz de características. Por enquanto a aplicação trabalha apenas com as imagens codificados em formato RGB, porém ela pode ser alterada para aceitar outros formatos de representação de cor, basta que eles sejam escalados apropriadamente para serem processados pelo *mean-shift*.

Nos trechos de código 4.1.1 e 4.1.2 temos as declarações das classes MeanInterface e meanshift respectivamente.

```
typedef enum {
  cl_gpu,
  cl_cpu,
  no_cl
} openclModeEnum;
class MeanInterface : public Core::Singleton<MeanInterface> {
private:
  friend class Core::Singleton<MeanInterface>;
  openclModeEnum m_openclMode;
  int m_imgWidth;
  int m_imgHeight;
  float m_bandwidth;
  double m_breakCriteria;
  std::vector<IplImage*> m_imgVector;
  MeanInterface() {
    m_openclMode = cl_gpu;
    m_imgWidth = 0;
    m_imgHeight = 0;
    m_bandwidth = STD_BANDWIDTH;
    m_breakCriteria = STD_BREAK_CRITERIA;
    m_imgVector.clear();
  }
  ~MeanInterface() {
    m_openclMode = cl_gpu;
    m_imgWidth = 0;
    m_imgHeight = 0;
    m_bandwidth = 0.f;
    m_breakCriteria = 0.f;
    for(unsigned int i = 0; i < m_imgVector.size(); i++)
      delete m_imgVector[i];
    m_imgVector.clear();
  }
  std::vector<IplImage*> splitChannels(const IplImage* img);
  IplImage* grayscaleToPseudocolor(const IplImage* img);
public:
  //Getters and setters.
  inline void setBandwidth(float bandwidth) {
    m_bandwidth = bandwidth;
  }
  inline float getBandwidth() {
    return m_bandwidth;
  }
  inline void setBreakCriteria(double breakCriteria) {
    m_breakCriteria = breakCriteria;
  }
  inline double getBreakCriteria() {
    return m_breakCriteria;
  }
  inline void setOpenCLMode(openclModeEnum openclMode) {
    m_openclMode = openclMode;
  }
  inline openclModeEnum getOpenCLMode() {
    return m_openclMode;
  }
  void addImageVec(std::vector<IplImage*> imgVector);
  void loadImages(std::vector<std::string> imgs, int n_channels);
  void addImage(IplImage* img);
  void loadImage(std::string img, int n_channels);
  IplImage *doMeanShift();
};
```

Código 4.1.1: Declaração da classe MeanInterface.

```
typedef struct cluster_vote_element_t{
  int cluster;
  int value;
} ClusterVotesElement;
class MeanShift {
private:
  /**
   *Variaveis MeanShift
   */
  CvMat* dataset;
  int numOfPoints;
  int numOfDimensions;
  CvMat* indexedClusters;
  float bandwidth;
  float breakCriteria;
  bool randomMode;
  int pointsToVotes;
  int* thisClusterVotes;
  int thisClusterVotesSize;
  std::vector< std::list<ClusterVotesElement*> *> clusterVotes;
  int *indexToVisit;
  int indexToVisitSize;
  float *currentKMean;
  float *lastKMean;
  std::list<int> *indexOnSquaredDistToAll;
  float *squaredDistToAll;
  int squaredDistToAllSize;
  int kClusters;
  int *visitedPoints;
  bool mergeClusters;
  std::vector<float*> clusterCenter;
  bool usingOpenCL;
  /**
   * Variaveis de configuracao OpenCL
   */
  cl_platform_id platform_id;
  cl_device_id device_id;
  cl_context context;
  cl_kernel kernel;
  cl_command_queue queue;
  cl_program program;
  cl_mem input;   // device memory used for the input array
  cl_mem output;  // device memory used for the output array
  cl_int err;
  /**
   * Metodos
   */
  void init();
  char* load_program_source(const char *filename);
  void initOpenCL(int deviceType);
  void destroyOpenCL();
  void destroy();
  float norm(float* a, float* b, int dimension);
  float* biased(float* a, float* b, int dimension);
  int getClusterVotesValue(int index, int cluster);
  void setClusterVotesValue(int index, int cluster, int value);
  int indexMax(int* a, int size);
  void eclidianOpenCL(float *result,float *element);
  void clErrorCheck(cl_int error_code);
public:
  MeanShift();
  ~MeanShift();
  void doMeanShift(CvMat* indexedClusters,
                   CvMat* dataset,
                   float bandwidth,
                   float breakCriteria,
                   bool randomMode,
                   int openClArg);
};
```

Código 4.1.2: Declaração da classe meanshift.

4.2 Fluxo de Execução

Para invocar a aplicação, o usuário deve passar uma série de parâmetros para que seja iniciado o processamento. Esses parâmentros constituem no modo que o OpenCL deve ser utilizado (OpenCL em GPU, ou OpenCL em CPU ou sem OpenCL), a largura de banda, o número de canais das imagens a serem carregadas, o caminho para o arquivo resultante do processamento, e os arquivos de entrada para o algoritmo. É importante ressaltar que os arquivos de entrada e o arquivo de saída devem ser imagens, ainda não há suporte a outros tipos de arquivos. Também é importante citar que as imagens de entrada devem possuir o mesmo número de pixels para que a codificação dos dados seja feita corretamente.

Após a invocação do programa com os parâmetros acima, a largura de banda escolhida, bem como o modo de uso do OpenCL são passados diretamente para a instância de MeanInterface, enquanto as imagens de entrada são carregadas utilizando a API OpenCV e então passadas para a interface. Após o processamento dos parâmetros, as imagens de entrada são codificadas na forma de uma matriz de D linhas por N colunas, onde D é o número de imagens de entrada passadas e N é o número de pixels de cada imagem. Se todos os parâmetros forem carregados corretamente, a aplicação cria uma instância da classe meanshift e inicia o processamento das imagens, bem como a medição do tempo para averiguar o desempenho e, se foi definido que deve ser utilizado OpenCL, a API é então iniciada apropriadamente para o dispositivo escolhido (GPU ou CPU), senão, é utilizada uma versão serial do cálculo da distância euclidiana.

Ao terminar o processamento das imagens, o meanshift retorna uma matriz de características para a interface. Esta converte a matriz para a forma de uma imagem em escala de cinza. Como foi determinado durante o desenvolvimento, o *mean-shift* pode encontrar um elevado número de *clusters*. Se esse número for maior que 255, não há como representá-los apropriadamente utilizando níveis de cinza. Para resolver esse problema, foi adicionado um passo ao final da codificação do resultado, que consiste em realizar a conversão da imagem de escala de cinza para uma imagem de três canais, e atribuir uma cor (pseudo-cor) para cada valor de cinza encontrado. Com essa extensão, um número muito maior de *clusters* distintos pode ser representado na imagem resultante. A imagem resultante do processamento é então retornada para a função principal, onde ela é exibida para que o resultado seja conferido, bem como é salva em disco utilizando uma função

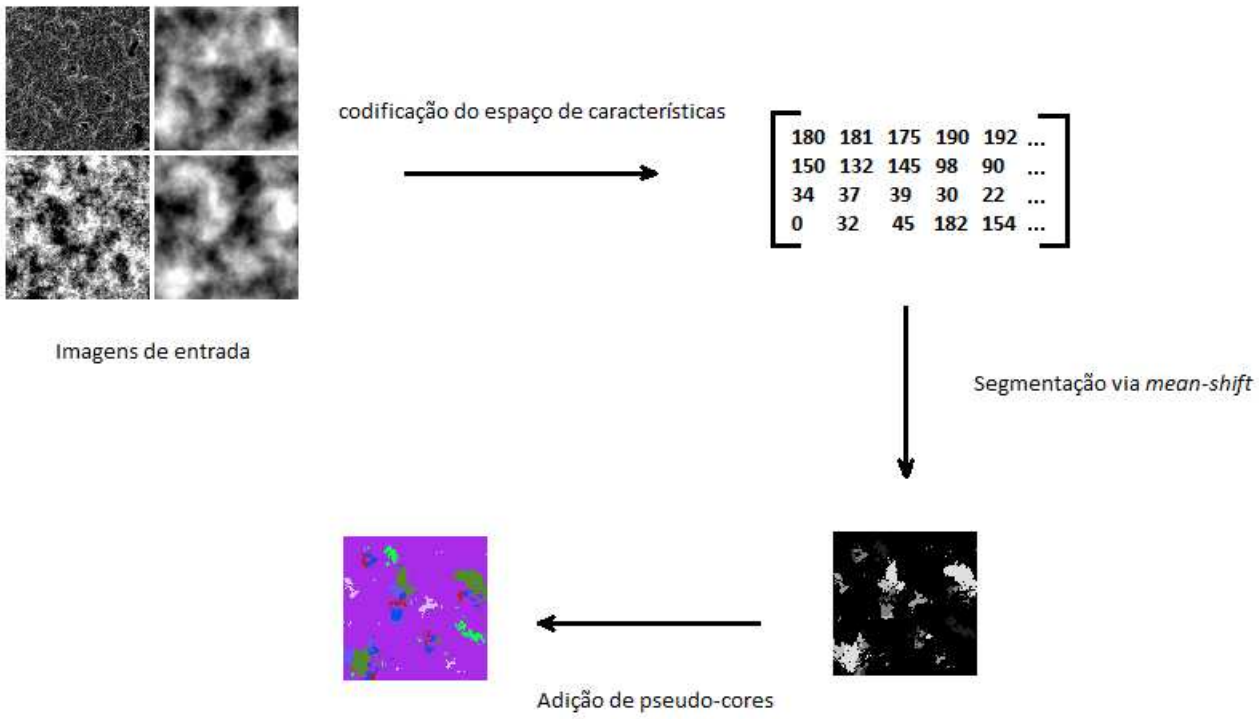


Figura 4.1: Fluxo de execução da aplicação.

do OpenCV que recebe o caminho para o arquivo de saída como parâmetro. Um fluxo de execução mais geral pode ser visto na figura 4.1.

4.3 Lógica por trás do *Mean-shift*

Após o a passagem e processamento dos parâmetros, é dado início ao *mean-shift*. O algoritmo roda para cada um dos pontos do conjunto de dados e calcula a média desses pontos, que consiste em um vetor, chamado de vetor de *mean*, que resulta na direção de maior crescimento da densidade, para maiores informações, verifique a seção 2.4.

Para calcular o vetor de *mean*, que aponta para o novo centro de massa, o algoritmo entra em um laço. Ele então calcula a distância euclidiana entre o ponto atual e todos

os pontos do conjunto de dados e se essa distância for menor que a largura de banda, o ponto é adicionado a um conjunto de dados, para que o vetor possa ser calculado. O laço termina apenas quando a distância entre o centro de massa calculado na iteração atual e o centro calculado na iteração anterior for menor que um critério de parada definido. Para os testes, o critério foi estabelecido em 10^{-6} unidades.

A paralelização do código em OpenCL entra na etapa do cálculo da distância euclidiana, que é executada a cada iteração do laço. Se o parâmetro de utilização do OpenCL foi passado ao iniciar a aplicação, então toda a parte de criação de um contexto de execução explicada em 2.6 é feita logo no início do algoritmo. Assim, a cada iteração, é chamada a função em OpenCL, responsável pelo cálculo da distância euclidiana, que recebe como parâmetros: o conjunto de pontos, o centro de massa atual, um vetor para armazenar o resultado, o número de dimensões do conjunto de dados de entrada e o número de elementos do mesmo. Essa função, ou *kernel* pode ser vista em 4.3.1.

```
__kernel void euclidean_distance(
  __global float *output,
  __global const float *dataset,
  int nPoints,
  int nDimensions,
  __global const float *currentKMean
)
{
  int index = get_global_id(0);
  float sum = 0.f;
  for(int d = 0; d < nDimensions; d++)
  {
    int acc = currentKMean[d] - dataset[d * nPoints + index];
    sum += pow((float) acc, 2);
  }
  output[index] = (float) sqrt((float) sum);
}
```

Código 4.3.1: Cálculo da distância euclidiana paralelizado em OpenCL.

Após o cálculo da distância entre o centro de massa atual e os pontos do conjunto de entrada, todos os pontos que estiverem a uma distância menor que a largura de banda em relação a centro de massa, são incluídos em uma estrutura de dados para que o novo centro seja calculado. Quando o novo centro de massa for calculado, a distância entre ele e o centro antigo é determinada, e se for menor que o critério de parada definido no início do algoritmo, então uma série de procedimentos a serem explicados abaixo tomam lugar e o laço é interrompido. Então o processo recomeça para o próximo ponto no conjunto de entrada, até que todos os pontos tenham sido processados.

Os procedimentos finais anteriores à interrupção do laço consistem em verificar se

há um outro centro de massa, ou *cluster* próximo ao centro de massa calculado, se esse resultado for positivo, os *cluster* representados pelos centros de massa são unidos e passam a formar apenas um *cluster* maior. O critério para determinar a união dos *clusters* é se os dois centros de massa estão a uma distância menor que a metade da largura de banda. Nesse caso, deve ser determinado um novo centro de massa para esse *cluster*. O novo centro de massa é o ponto médio entre os dois centros de massa dos *clusters* antigos. Caso os centros de massa não estiverem tão próximos, os *clusters* não serão unidos. Uma variável que indica o número de *clusters* encontrados é incrementada e o centro de massa calculado é adicionado a uma estrutura que armazena todos os centros de massa determinados até então.

Quando todos os pontos forem processados e todos os centros de massa presentes no conjunto de entrada forem encontrados, então os pontos do conjunto de entrada serão varridos e associados ao centro de massa mais próximo. Assim, o centro de massa juntamente com os pontos associados a ele formam um *cluster*. A associação entre o centro de massa e os pontos é feita de forma que os pontos assumam o mesmo valor que o centro de massa. Os valores desses pontos são codificados em uma matriz que após um processamento, formará a imagem segmentada resultante. Ao final do algoritmo, os cálculos de desempenho são realizados e salvos em um arquivo para futuras referências.

5 RESULTADOS

Nesse capítulo serão expostos os resultados obtidos com o protótipo implementado. Foram usados quatro conjuntos de imagens para determinar a melhoria do desempenho. Três dos conjuntos utilizados possuem imagens que variam de tamanho, com o objetivo de determinar se o crescimento linear do tamanho do conjunto de dados implica em um crescimento linear do tempo de execução do algoritmo. Também foram testadas as larguras de banda variando entre 30 e 59 unidades, com a finalidade de determinar a sua influência no resultado final e no tempo de execução.

Os conjuntos de imagens utilizados contêm imagens triviais, ou seja, totalmente brancas e totalmente pretas, bem como imagens de casos reais, e imagens com o ruído *salt and pepper*, com o fim de determinar se a riqueza de conteúdo das imagens interfere no tempo de execução do algoritmo. Algumas das imagens utilizadas podem ser vistas em 5.1 e 5.2, que representam os conjuntos 1 e 4 respectivamente. As imagens pretas constituem o conjunto de imagens 2, e as imagens brancas formam o conjunto 3.

O *hardware* utilizado para os testes é de uma geração anterior à atual, sendo sua maioria lançado no ano de 2010 no Brasil quando era considerado estado da arte. Os principais itens de *hardware* presentes na máquina utilizada para os testes estão descritos abaixo. Vale ressaltar que o *hardware* é de uso pessoal, não pertencendo a um grupo de pesquisa.

- Processador: Core i7 950 3.07 GHz
- 6 Gb de memória RAM DDR 3
- ATI Radeon HD 5870 1 Gb de RAM GDDR 5

Esse capítulo será dividido em quatro seções, sendo que as seções 5.1 5.2 e 5.3 apresentam e comentam os resultados obtidos com os testes realizados com base em cada um

Figura 5.1: Conjunto de imagens reais, denomidado conjunto 1.

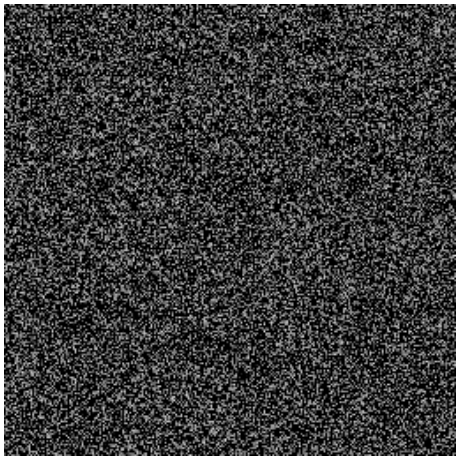


Figura 5.2: Conjunto de imagens com ruído *salt and pepper*, denomidado conjunto 4.

dos critérios estabelecidos no início do presente capítulo. A seção 5.4 visa apresentar os resultados de um teste de caso real do algoritmo para verificar a sua viabilidade. Assim, pode ser determinada a influência de alguns dos fatores que afetam o desempenho e, em alguns casos, os resultados obtidos com a segmentação via *mean-shift*.

5.1 Influência do Tamanho da Imagem

Para esse quesito foram usadas as imagens dos conjuntos 2, 3 e 4. As imagens testadas possuem tamanhos de 256x256, 512x512, 768x768 e 1024x1024 pixels.

Observe que no gráfico 5.3 o aumento das dimensões da imagem implica em um aumento do tempo de processamento necessário para segmentar a imagem. São necessários mais testes para determinar se esse comportamento se mantém. No gráfico 5.4 é possível notar que os tempos de processamento são maiores que os tempos do conjunto 2, embora as imagens possuam apenas uma cor em ambos os conjuntos. Já no gráfico 5.5 ocorre um comportamento anômalo, já que para a imagem de 256x256 pixels, o tempo de processamento em GPU é o maior, porém, conforme a dimensão da imagem aumenta, esse tempo tende a crescer menos em relação aos tempos em CPU com e sem OpenCL.

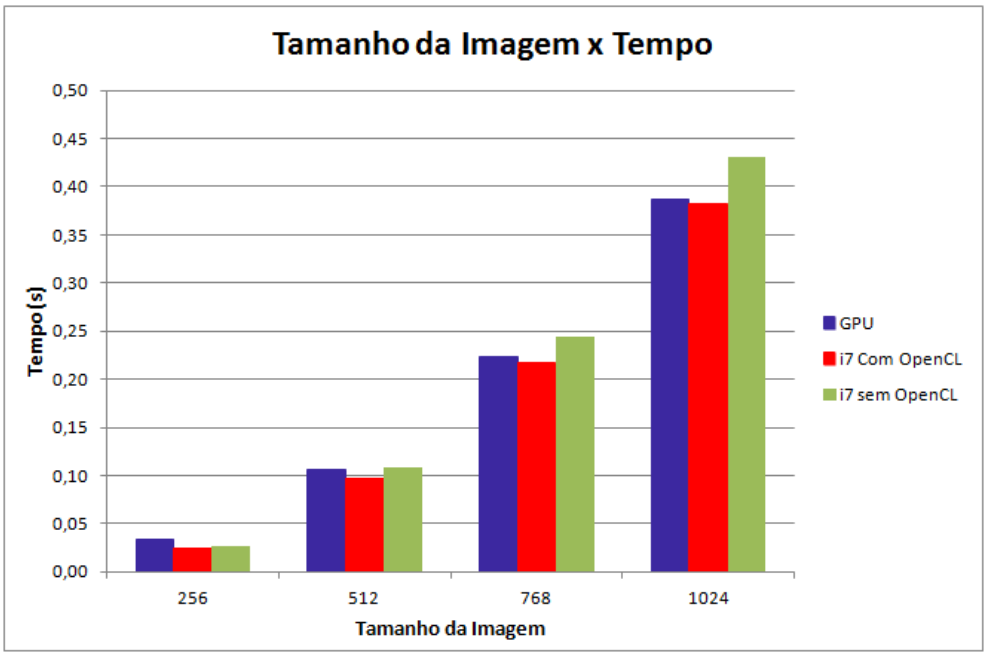


Figura 5.3: Gráfico referente ao processamento do conjunto de imagens 2 (brancas).

Conforme os gráficos 5.3, 5.4 e 5.5, referentes aos conjuntos 2, 3 e 4, de imagens pretas, brancas e com ruído *salt and pepper*, respectivamente, é possível observar que ocorre um crescimento dos tempos de processamento conforme as dimensões das imagens

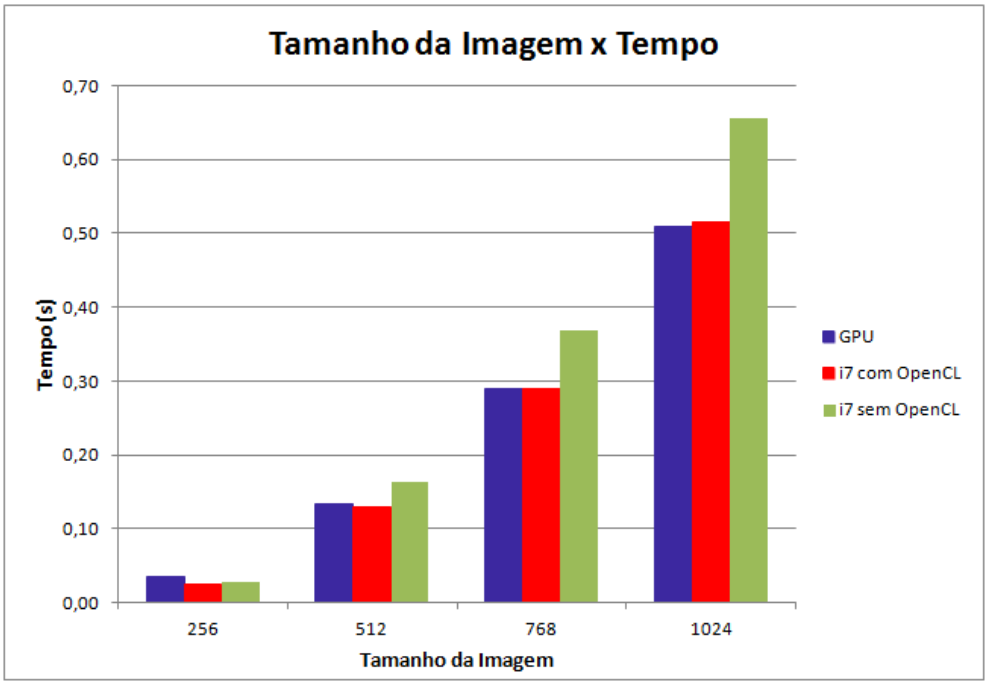


Figura 5.4: Gráfico referente ao processamento do conjunto de imagens 3 (pretas).

de entrada aumentam. Esse crescimento ocorre tanto no algoritmo rodando em GPU quanto em CPU com e sem OpenCL.

Também pode ser observado nos gráficos 5.3 e 5.4 que os tempos de processamento do conjunto 3 são maiores que os tempos do conjunto 2, embora a única diferença entre as imagens desses conjuntos é que as do conjunto 2 são totalmente pretas, e as do conjunto 3 são totalmente brancas. Essa discrepância entre os tempos aumenta conforme as dimensões das imagens aumentam. Uma possível explicação para essa diferença de tempo pode ser vista na seção 5.3.

5.2 Influência da Largura de Banda

Para a obtenção dos dados a seguir, foram utilizados todos os conjuntos de imagens citados no início do capítulo. Para o conjunto 1, representado pelo gráfico 5.6 os testes foram realizados em GPU, para os outros conjuntos, os testes foram realizados em CPU sem OpenCL. Essa metodologia de teste foi escolhida pois, os tempos médios de execução do algoritmo em CPU sem OpenCL para o conjunto 1 são de aproximadamente 578 segundos (9 minutos e 36 segundos). Isso para a largura de banda com valor de 50 unidades e usando apenas quatro canais de cor, dos doze disponíveis. Estima-se que para a largura de banda de 30 unidades esse tempo seria muito maior.

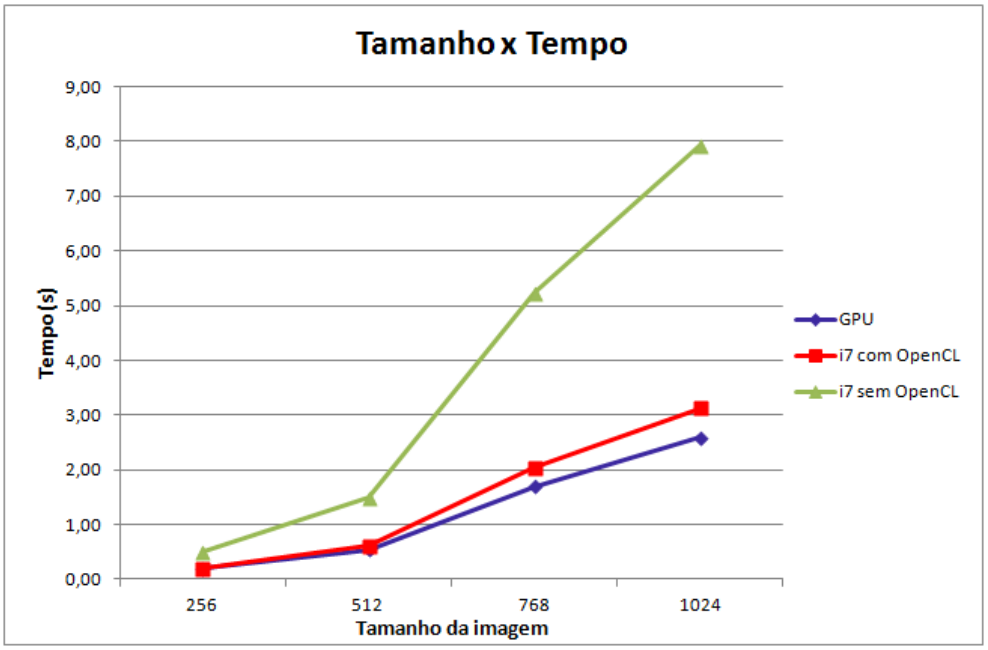


Figura 5.5: Gráfico referente ao conjunto de imagens 4 (*salt and pepper*).

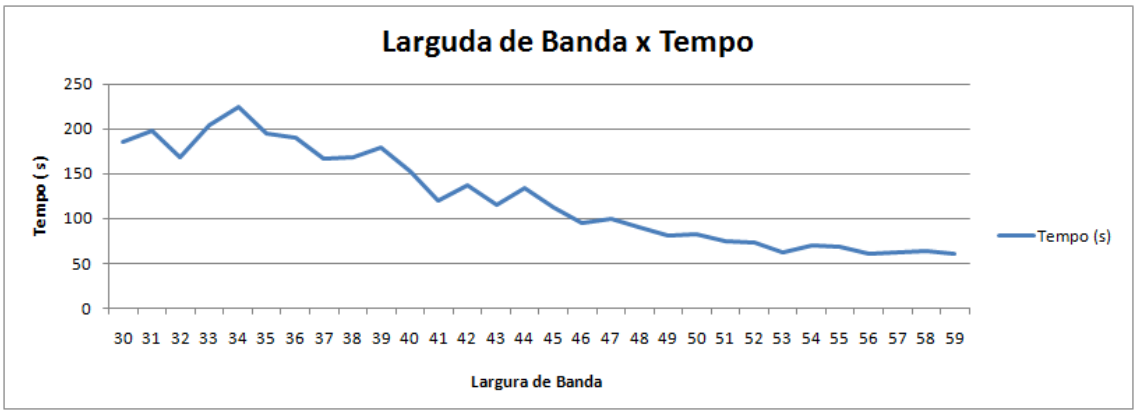


Figura 5.6: Gráfico referente aos tempos de processamento para diferentes larguras de banda usando o conjunto 1.

É possível observar no gráfico 5.7 que para os conjuntos 1 e 4, de imagens com ruído *salt and pepper*, o decrescimento da largura de banda implica, em geral, no decrescimento do tempo de processamento necessário. Porém, esse tempo voltou a crescer para alguns valores. Os testes com esses valores foram repetidos e foram obtidos os mesmos resultados.

Para os conjuntos de imagens 2 e 3, o decrescimento da largura de banda não implica no decrescimento do tempo de processamento. Os tempos de processamento das imagens são constantes para qualquer largura de banda, pois as imagens possuem apenas uma cor, ou seja, todos os pontos são coincidentes, e a segmentação desses pontos resulta em um

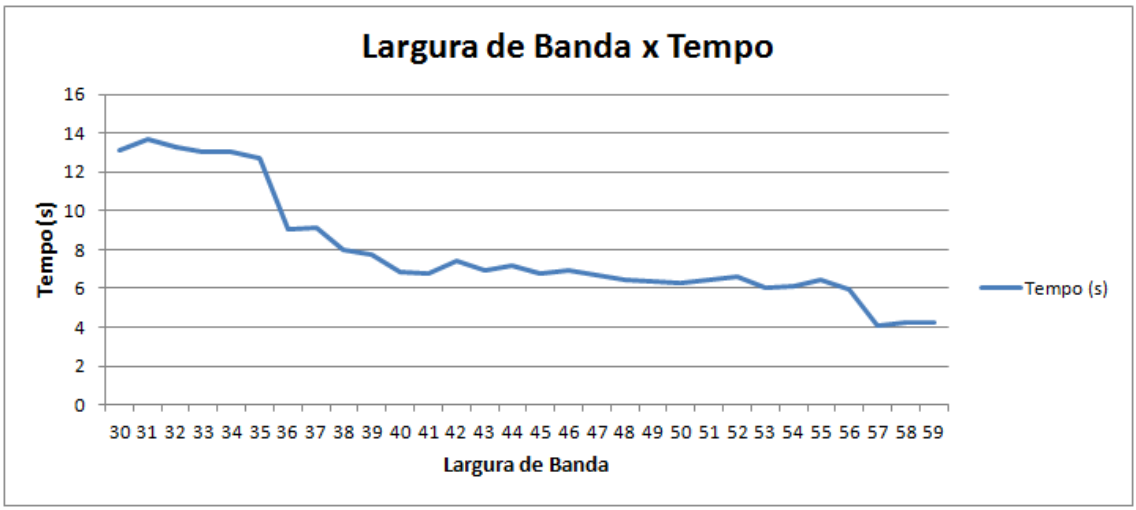


Figura 5.7: Gráfico referente aos tempos de processamento para diferentes larguras de banda usando o conjunto 4.

cluster apenas. O fator da simplicidade da imagem é verificado na seção 5.3.

5.3 Influência do Conteúdo da Imagem

Essa seção visa apresentar os resultados obtidos do protótipo usando como entrada os conjuntos de imagens 2, 3 e 4.

Foi observado ao final dos testes que a riqueza de conteúdo das imagens de entrada influencia significativamente no tempo de processamento do algoritmo, podendo, no caso de imagens com dimensões de 1024x1024 pixels processadas em CPU sem OpenCL, gerar um aumento de 12 vezes no tempo final de execução do algoritmo.

Houve também uma pequena diferença entre os tempos de processamento entre as imagens dos conjuntos 2 e 3, imagens totalmente pretas e totalmente brancas, respectivamente. Os tempos de execução do método nas imagens do conjunto 3 foram maiores em praticamente todos os casos de teste, com exceção apenas dos testes em imagens de 256x256 pixels, onde as médias permaneceram praticamente as mesmas. Uma possibilidade para essa diferença entre os tempos de processamento envolve o fato de ocorrerem otimizações durante a compilação do protótipo, pois muito embora as imagens dos conjuntos 2 e 3 possuam o mesmo conteúdo (apenas uma cor, nesse caso), ainda assim os *pixels* das imagens do conjunto 2 possuem valor 0 enquanto os das imagens do conjunto 3 possuem valor 255.

As tabelas 5.1, 5.2 e 5.3 exibem as médias dos tempos de processamento em segundos, obtidos com os testes dos conjuntos de imagens 2, 3 e 4, com o algoritmo rodando tanto

X	256x256	512x512	768x768	1024x1024
Conjunto 2	0,03	0,11	0,22	0,39
Conjunto 3	0,04	0,13	0,29	0,51
Conjunto 4	0,19	0,54	1,70	2,59

Tabela 5.1: Médias dos tempos dos conjuntos 2, 3 e 4, rodando em GPU.

X	256x256	512x512	768x768	1024x1024
Conjunto 2	0,02	0,10	0,22	0,38
Conjunto 3	0,02	0,13	0,29	0,51
Conjunto 4	0,20	0,61	2,04	3,13

Tabela 5.2: Médias dos tempos dos conjuntos 2, 3 e 4, rodando em CPU usando OpenCL.

em GPU quanto em CPU com e sem OpenCL.

É possível observar nas tabelas 5.1, 5.2 e 5.3 que há uma grande diferença entre os tempos de processamento dos conjuntos 2 e 3 em relação ao conjunto 4, esse fato se deve à presença de maior variação entre os pontos das imagens deste conjunto, portanto, é possível concluir que o algoritmo levará mais tempo para processar imagens mais complexas em relação a imagens mais simples.

Também é evidente que, para as imagens dos conjuntos 2 e 3, os tempos de processamento em GPU e em CPU com OpenCL são os mesmos, quando os tempos em CPU não são menores. Isso pode ser atribuído a simplicidade das imagens destes conjuntos, a própria passagem dos dados para a memória da GPU, bem como a invocação da função de processamento tomam uma parte do tempo de processamento, que nesse caso, acabou superando o tempo em CPU.

5.4 Exemplo da Segmentação de Imagens Reais

Para demonstrar o funcionamento do algoritmo, foi obtido um conjunto de mapas de altura para os testes, essas imagens compõem o conjunto de imagens 1 e foram a base dos testes de desempenho realizados. As images podem ser vistas na figura 5.1.

Para demonstrar a validade do método, os testes realizados foram divididos em duas

X	256x256	512x512	768x768	1024x1024
Conjunto 2	0,03	0,10	0,24	0,43
Conjunto 3	0,03	0,16	0,37	0,66
Conjunto 4	0,50	1,49	5,23	7,92

Tabela 5.3: Médias dos tempos dos conjuntos 2, 3 e 4, rodando em CPU sem OpenCL.

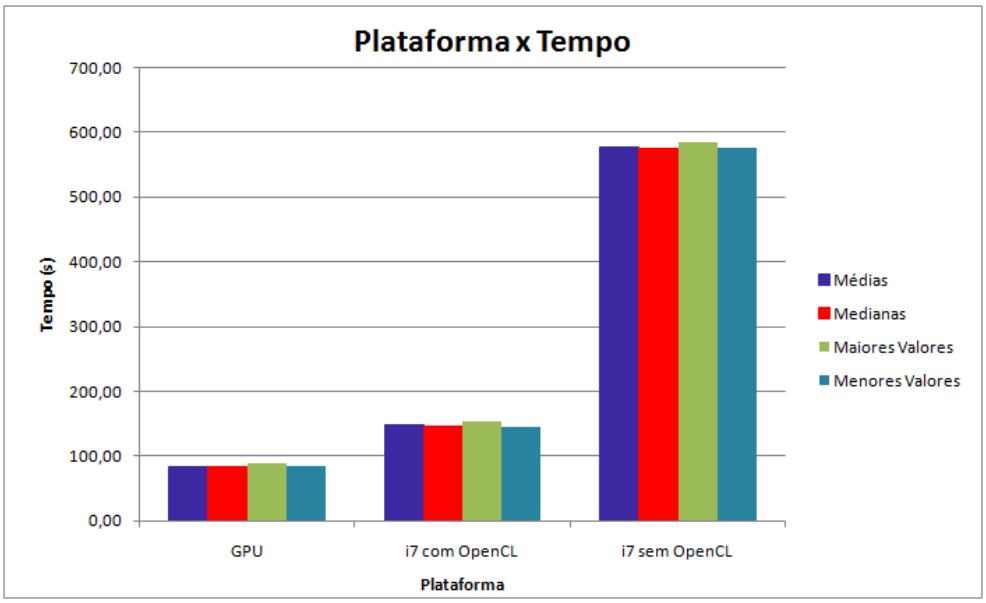


Figura 5.8: Comparação dos tempos de execução em GPU e CPU.

categorias:

1. Foi realizada uma série de testes nos quais a largura de banda foi fixada em 50 unidades e o algoritmo foi rodado com as nove imagens em GPU e CPU.

2. Foram realizados os mesmos testes anteriores, mas a largura de banda variou entre 30 e 59 unidades.

O gráfico 5.8 ilustra o melhor desempenho da implementação em GPU do algoritmo em relação às implementações em CPU.

Muito embora a implementação em CPU que faz uso de OpenCL já tenha provocado uma melhora significativa em relação a implementação comum, o fato do algoritmo ter sido adaptado para usar a capacidade de paralelização da GPU, faz com que haja uma diminuição de, aproximadamente, 16 vezes no tempo necessário para segmentar as imagens do conjunto 1, contra 5 vezes de melhoria proporcionada pela implementação em CPU.

Os resultados da segunda categoria de testes foram exibidos e comentados na seção 5.2, porém, de acordo com Comaniciu & Meer (2002), a escolha da largura de banda deve ser feita com cautela, pois, se for muito baixa, afeta drasticamente desempenho e o resultado da segmentação, podendo separar uma região em duas ou mais regiões distintas, e caso a largura de banda for muito grande, pode juntar duas ou mais regiões em uma só, gerando erros no resultado, porém consumindo muito menos tempo de processamento.

6 CONCLUSÃO

O objetivo deste trabalho foi melhorar o desempenho de uma implementação genérica do *mean-shift* utilizando OpenCL, assim possibilitando o acesso ao poder de processamento do *hardware* moderno. Para isso, foi realizado um estudo do *mean-shift* com o objetivo de determinar quais as etapas do algoritmo que mais influenciam no desempenho do mesmo.

Após a determinação da distância euclidiana como a etapa que mais influencia no desempenho, ela foi implementada em OpenCL e então uma série de testes com o objetivo de determinar se houve diferença no tempo de execução foi realizada. A próxima etapa envolveu a criação de uma classe de acesso ao *mean-shift*, que realizasse o pré-processamento das imagens de entrada e a codificação da matriz resultante para a imagem segmentada. Essa etapa se fez necessária para que o algoritmo pudesse ser apropriadamente utilizado em outros trabalhos desenvolvidos no LaCA (HENZ, 2011).

Durante o curso do trabalho, foi observado que a paralelização do algoritmo principal pode ser estendida. Como o algoritmo calcula a moda com base em cada ponto do espaço de características, esse cálculo pode ser paralelizado, criando um fluxo de execução distinto para cada ponto. Como este trabalho é um passo inicial na direção dessa nova abordagem de paralelização, apenas a distância euclidiada foi paralelizada afim de determinar se a melhoria de desempenho era significativa.

A escolha do OpenCL como API para paralelizar a distância euclidiana se mostrou vantajosa, pois ela fornece uma boa flexibilidade quanto a plataforma de execução, além de proporcionar portabilidade ao código produzido.

Os resultados obtidos se mostraram excelentes, possibilitando uma grande diminuição do tempo de execução do algoritmo, principalmente quando o número de dimensões considerado é grande.

6.1 Trabalhos futuros

Com os resultados obtidos pela execução da aplicação desenvolvida, é possível determinar que o ganho de desempenho obtido é expressivo, justificando a continuidade dos estudos afim de determinar outra possíveis melhorias que podem ser feitas utilizando OpenCL.

Uma das otimizações que podem ser implementadas é a paralelização do cálculo das modas para cada ponto, ou seja, como o algoritmo determina os máximos locais a partir de cada ponto sequencialmente. Esse processo pode ser paralelizado, melhorando ainda mais o desempenho do algoritmo.

O algoritmo também pode ser alterado para aceitar outras entradas que não sejam imagens, evitando assim que quaisquer dados a serem processados precisem ser codificados em uma imagem antes de serem processados.

REFERÊNCIAS

AMD. **An Introduction to OpenCL**. Available at: http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx.

COMANICIU, D.; MEER, P. Mean Shift Analysis and Applications. **The Proceedings of the Seventh IEEE International Conference**, [S.l.], v.2, p.1197–1203, 1999.

COMANICIU, D.; MEER, P. Mean Shift: a robust aproach toward feature space analysis. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.24, n.5, p.603–619, 2002.

FUKUNAGA, K.; HOSTETLER, L. D. The Estimation of The Gradient of a Density Funcion, with Applications in Pattern Recognition. **IEEE Transactions on Information Theory**, [S.l.], v.21, p.32–40, 1975.

HENZ, B. **Geração de Caminhos em Cenários 3D a Partir de Mapas de Custo Utilizando Meanshift**. Trabalho de Graduação, Ciência da Computação, UFSM.

JOHNSON, N. L.; KEMP, A. W.; KOTZ, S. **Univariate Discrete Distributions**. 3.ed. [S.l.]: Wiley, 2005.

Khronos Group. **OpenCL - The open standard for parallel programming of heterogeneous systems**. Available at: http://www.khronos.org/opencl/.

Khronos Group. **OpenCL Introduction and Overview**. Available at: http://www.khronos.org/assets/uploads/developers/library/overview/OpenCL-Overview-Jun10.pdf.

MACQUEEN, J. B. Some Methods for Classification and Analysis of Multivariate Observations. **Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability**, [S.l.], v.1, p.281–297, 1967.

MATTEUCCI, M. **A Tutorial on Clustering Algorithms**. Available at: http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html.

NVIDIA. **CUDA**. Available at: http://www.nvidia.com/object/cuda_home_new.html.

ROSA, M. **Segmentação de Grãos de Hematita em Amostras de Minério de Ferro por Análise de Imagens de Luz Polarizada**. 2008. Dissertação (Mestrado) — Universidade Federal de Santa Maria (UFSM).

SAHBA, F.; VENETSANOPOULOS, A. Mean Shift Based Algorithm for Mammographic Breast Mass Detection. **Proceedings of the 2010 IEEE 17th International Conference on Image Processing**, [S.l.], 2010.

StarTrek.com. **Statistics Tutorial**: probability distributions. Available at: http://stattrek.com/lesson2/probabilitydistribution.aspx.

THIRUMURUGANATHAN, S. **Introduction to Mean Shift Algorithm**. Available at: http://saravananthirumuruganathan.wordpress.com/2010/04/01/introduction-to-mean-shift-algorithm/.

VENKATARAMA, A. **Gradient Descent/Ascent**. Available at: http://www-speech.sri.com/people/anand/771/html/node33.html.

WEISSTEIN, E. W. **Probability Function**. Available at: http://mathworld.wolfram.com/ProbabilityFunction.html.