UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS DE TELECOMUNICAÇÕES

EDUARDO TONDIN FERREIRA DIAS

MAPA DE SALIÊNCIAS: ESTUDO E IMPLEMENTAÇÃO NAS ARQUITETURAS CPU E GPU

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2011

EDUARDO TONDIN FERREIRA DIAS

MAPA DE SALIÊNCIAS: ESTUDO E IMPLEMENTAÇÃO NAS ARQUITETURAS CPU E GPU

Trabalho de Conclusão do Curso Superior de Tecnologia em Sistemas de Telecomunicações do Departamento Acadêmico de Eletrônica da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Hugo Vieira Neto

CURITIBA
2011

EDUARDO TONDIN FERREIRA DIAS

MAPA DE SALIÊNCIAS: ESTUDO E IMPLEMENTAÇÃO NAS ARQUITETURAS CPU E GPU

Este trabalho de conclusão de curso foi apresentado no dia 15 de dezembro de 2011, como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Telecomunicações, outorgado pela Universidade Tecnológica Federal do Paraná. O aluno foi arguído pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Ms. César Janeczko
Coordenador de Curso
Departamento Acadêmico de Eletrônica

Prof. Dr. Décio Estevão do Nascimento
Responsável pela Atividade de Trabalho de Conclusão de Curso
Departamento Acadêmico de Eletrônica

BANCA EXAMINADORA

Prof. Dr. Marcelo Victor Wüst Zibetti

Prof. Dr. Hugo Vieira Neto
Orientador

Prof. Ms. Juliano Mourão Vieira

Prof. Dr. Gustavo Benvenutti Borba

"A Folha de Aprovação assinada encontra-se na Coordenação do Curso"

RESUMO

DIAS, Eduardo Tondin Ferreira. **Mapa de Saliências: Estudo e Implementação nas Arquiteturas CPU e GPU**. 2011. 46 f. Trabalho de Conclusão de Curso. Curso Superior de Tecnologia em Sistemas de Telecomunicações. Universidade Tecnológica Federal do Paraná, 2011.

Este trabalho descreve o estudo e implementação de um método de atenção visual utilizando-se do modelo do mapa de saliências nas arquiteturas CPU e GPU. O algoritmo do mapa de saliências é primeiramente implementado no software de cálculos numéricos Matlab, com a finalidade de entendimento do modelo pesquisado. O modelo do mapa de saliências é então implementado na linguagem de programação C++. Analisam-se as funções desenvolvidas na programação C++, visando encontrar as funções que possuem características de paralelização. Com as funções definidas, o modelo é implementado na arquitetura GPU, utilizando-se da API de desenvolvimento CUDA. São efetuadas comparações entre as arquiteturas e descritos trabalhos futuros.

Palavras-chave: Atenção Visual, Mapa de Saliências, GPU, *CUDA*.

ABSTRACT

DIAS, Eduardo Tondin Ferreira. **Saliency Map: Study and Implementation on CPU and GPU Architectures**. 2011. 46 f. Trabalho de Conclusão de Curso. Curso Superior de Tecnologia em Sistemas de Telecomunicações. Universidade Tecnológica Federal do Paraná, 2011.

This work describes the study and implementation of a visual attention method using the saliency map model on CPU and GPU architectures. First, the saliency maps algorithm is implemented in the numerical calculations software Matlab with the purpose of understanding the model researched. After that, the model is implemented in the C++ programming language. The functions implemented in C++ are analyzed, in order to find functions that have characteristics of parallelization. With these functions located, the model is implemented in the GPU architecture, using the CUDA development API. Comparisons are made between the architectures and future research is described.

Palavras-chave: Visual Attention, Saliency map, GPU, *CUDA*.

LISTA DE FIGURAS

LISTA DE SIGLAS

API	do inglês *Application Programming Interface*, Interface de Programação de Aplicativos
CUDA	do inglês *Compute Unified Device Architecture*
CPU	do inglês *Central Processing Unit*, Unidade de Processamento Central
GPU	do inglês *Graphics Processing Unit*, Unidade de Processamento Gráfico
GPGPU	do inglês *General Processing on GPU*
SIFT	do inglês *Scale Invariant Feature Transform*
SURF	do inglês *Speeded Up Robust Features*
OpenCV	do inglês *Open Source Computer Vision*
FFT	do inglês *Fast Fourier Transform*, Transformada Rápida de Fourier
IFFT	do inglês *Inverse Fast Fourier Transform*, Transformada Rápida de Fourier Inversa

SUMÁRIO

1 INTRODUÇÃO

A visão computacional é uma área da computação que se utiliza de ferramentas e métodos para interpretar imagens, visando a implementação de sistemas visuais artificiais. Um de seus objetivos é o estudo e modelagem do sistema visual humano e a implementação artificial das suas características.

Uma das mais importantes características do sistema visual humano é a rapidez em direcionar o olhar para objetos de interesse em nosso ambiente visual (ITTI, 2000). Na visão computacional, o mecanismo que simula artificialmente essa função é denominado modelo de atenção visual. Podemos definir atenção visual como o processo de focalizar o objeto de interesse na imagem, colocando os demais objetos em segundo plano.

Para o presente projeto, propõe-se a implementação do mecanismo de atenção visual utilizando-se do modelo de mapa de saliências (ITTI, 2000). Existem outros modelos, como o SIFT (*Scale Invariant Feature Transform*) (LOWE, 1999) e o SURF (*Speeded Up Robust Features*) (BAY et al., 2008), porém, estes não serão estudados nesse momento.

O mecanismo de atenção visual descrito é baseado no comportamento e arquitetura do sistema neural dos primatas, analisando as imagens recebidas sob aspectos diferentes como cor, intensidade e orientação e combinando seus resultados com a finalidade de descobrir o seu ponto de atenção (ITTI et al., 1998).

Técnicas de processamento de imagens serão utilizadas na implementação do mecanismo de atenção visual, usando métodos e algoritmos que permitem a manipulação de imagens digitais, extraindo características ou efetuando transformações nas imagens visando sua utilização em um processamento futuro. Podemos citar o reconhecimento de padrões (TRUCCO; VERRI, 1998) como uma aplicação que pode utilizar-se de mecanismos de atenção.

Porém, a utilização de processamento de imagens em tempo real requer um alto poder de processamento das informações, causando, em alguns sistemas de visão computacional, lentidão ou perda de informações. Para a verificação da velocidade do processamento do mecanismo de atenção visual, o mesmo será desenvolvido em dois conceitos diferentes, CPU e GPU (unidades gráficas de processamento). O conceito CPU baseia-se na utilização de somente um processador para efetuar

o processamento serial das informações, enquanto no conceito de GPU são utilizados vários processadores para processamento paralelo.

No conceito CPU, o projeto foi desenvolvido em linguagem de programação C++, utilizando-se de uma biblioteca de processamento de imagens desenvolvida pelo professor Hugo Vieira Neto e também em Matlab, software para simulação e resolução de cálculos numéricos. Para o conceito GPU, foi utilizada a arquitetura CUDA (NVIDIA, 2010), utilizando-se da linguagem de programação C++ para seu desenvolvimento. O projeto não foi desenvolvido em Matlab no conceito GPU.

Com a utilização da tecnologia CUDA, plataforma desenvolvida pela empresa Nvidia, que se utiliza das unidades gráficas de processamento (GPU) para o processamento paralelo de informações, espera-se um ganho no processamento do algoritmo, permitindo a execução do mecanismo de atenção visual em tempo real. A utilização de multiprocessamento permite que as informações processadas sejam disponibilizadas mais rapidamente para o sistema. Tradicionalmente, a maioria dos aplicativos são escritos em programas sequenciais, utilizando-se somente de um processador (CPU) para efetuar todo o processamento (KIRK; HWU, 2010).

As GPU são multi-processadores dedicados presentes nas placas aceleradoras de vídeo para computadores, desenvolvidas inicialmente para processamento gráfico. Em virtude do crescimento da utilização do conceito CPU/GPU em diversas aplicações, sendo esse conceito uma tecnologia em evolução, optou-se por pesquisar o sistema para sua implementação no projeto.

Esse projeto também visa o aprimoramento dos métodos e expressões estudadas na disciplina de Processamento Digital de Imagens e Processamento Digital de Sinais, conceitos de processos estudados na disciplina de Sistemas Operacionais e programação orientada a objetos, ministrada na disciplina de Programação Avançada, do curso de Tecnologia em Sistemas de Telecomunicações.

1.1 PROBLEMA

Na área de visão computacional, um dos problemas de processamento de imagens é o reconhecimento de padrões. Existem diversas técnicas utilizadas para o reconhecimento, determinadas para cada aplicação, como por exemplo Transformada de Hough (BALLARD, 1981) e Shape from Focus (NAYAR, 1992) para detecção de

objetos, Fluxo Óptico (NEWCOMBE, 2004) para determinar o movimento de objetos e o Mapa de Saliências (ITTI, 2000) para definir o ponto de atenção.

As técnicas de processamento de imagens para o reconhecimento de informações podem necessitar de elevado processamento computacional, dificultando sua implementação em tempo real. O processamento de imagens nessas situações não deve sofrer atrasos, pois isto pode comprometer o desempenho de todo o sistema, gerando erros e não atingindo os objetivos propostos, como por exemplo alterar a rota em um sistema de navegação robótica.

Buscando a resolução do problema de reconhecimento de informações e a disponibilidade em tempo real, o presente projeto visa a implementação do mecanismo de atenção visual do mapa de saliências através da implementação de funções de processamento de imagens utilizando programação paralela.

1.2 JUSTIFICATIVA

A utilização da visão computacional permite o desenvolvimento de diversos sistemas de visão artificial. Alguns algoritmos de processamento de imagens efetuam a extração das características relevantes das imagens, permitindo o desenvolvimento de sistemas para diversas áreas da ciência, como por exemplo engenharia biomédica, navegação autônoma e reconhecimento de padrões (TRUCCO; VERRI, 1998).

O algoritmo de atenção visual implementado neste projeto necessita de elevado processamento computacional, dificultando sua implementação em tempo real, quando utiliza-se de processadores convencionais. A utilização do conceito GPGPU, *General Processing on GPU*, visa acelerar o processamento, permitindo a utilização do algoritmo em tempo real, como por exemplo no *tracking*, sistema de visão que acompanha o movimento do objeto de interesse, no qual o objeto não deve sair do campo de visão da câmera.

O desenvolvimento do projeto também visa o estudo das técnicas de visão computacional e sua implementação no conceito GPGPU, além de permitir a utilização futura como ferramenta para novos pesquisadores para a implementação das técnicas de programação paralela na área de processamento de imagens.

A utilização de multiprocessadores para o processamento de imagens digitais é uma tecnologia recente, que surgiu com o avanço nas pesquisas nos últimos

anos. Através desta pesquisa, pretende-se permitir a criação de sistemas com processamento de imagens com maior velocidade do que os sistemas atuais, utilizando-se dos resultados do estudo efetuado.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O presente projeto tem como objetivo implementar o mecanismo de atenção visual do mapa de saliências nas arquiteturas CPU e GPU.

1.3.2 Objetivos Específicos

1. Pesquisar as técnicas de processamento de imagens, atenção visual, programação paralela utilizando-se multiprocessadores gráficos.
2. Estudar o algoritmo do mapa de saliências.
3. Estudar a biblioteca de processamento de imagens desenvolvida em linguagem C++ pelo professor Hugo Vieira Neto.
4. Definir as funções relevantes de processamento de imagens para o projeto.
5. Desenvolver as funções relevantes de processamento de imagens utilizando programação paralela.
6. Implementar o algoritmo de atenção visual em CPU e GPU.
7. Comparar o desempenho GPU x CPU (C++) x CPU (Matlab) para o algoritmo de atenção visual proposto.
8. Compreender e documentar o aprendizado da implementação de algoritmos na abordagem GPU.
9. Redigir a documentação do projeto.

1.4 METODOLOGIA

O projeto foi dividido em quatro etapas. A figura 1 ilustra essas etapas.

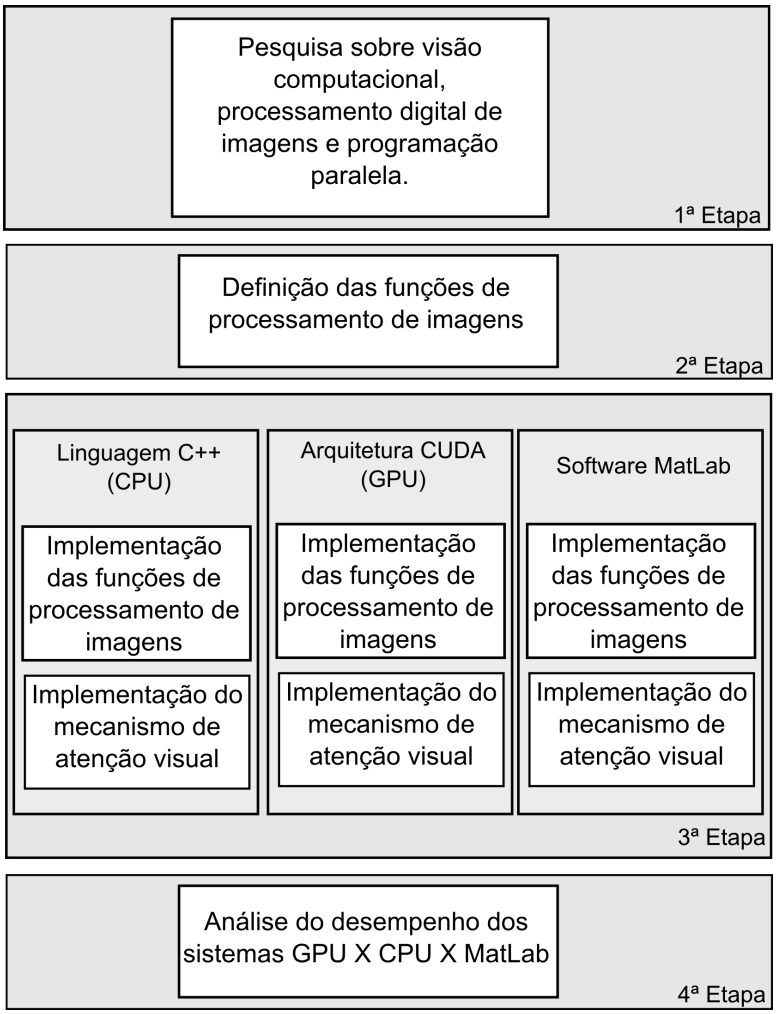

Figura 1: Diagrama em blocos das etapas do projeto
Fonte: Autoria própria.

A primeira etapa consistiu na pesquisa bibliográfica dos conceitos e métodos a serem utilizados no projeto. A pesquisa bibliográfica visa o estudo dos conceitos de atenção visual utilizando-se do mapa de saliências, programação paralela, bibliotecas gráficas e os métodos de processamento de imagens a serem implementados.

Para o desenvolvimento do projeto, foi necessário o entendimento de conceitos nas áreas de visão computacional, processamento de imagens, sistemas operacionais e programação paralela.

Após o estudo, as funções de processamento de imagens foram definidas com o propósito de atender o desenvolvimento do mecanismo de atenção visual proposto. Com essa definição, as funções foram implementadas na linguagem de programação

C++, na arquitetura CUDA e no software Matlab, todos em plataforma PC.

Na etapa seguinte, com as funções de processamento de imagens implementadas, efetuou-se o desenvolvimento do algoritmo de atenção visual do mapa de saliências em todos os sistemas propostos (plataforma CPU x GPU x Matlab). O mapa de saliências utilizado será baseado no descrito por Itti e colaboradores (ITTI et al., 1998). Nesse modelo, a imagem de entrada é decomposta em mapas de características de acordo com três componentes de atenção - intensidade, cor e orientação - combinando-se esses mapas para gerar o mapa de saliências.

A última etapa foi de testes e análise de desempenho de cada sistema proposto. Os testes dos sistemas foram efetuados com imagens estáticas, para a validação do mecanismo nas diversas plataformas propostas. Para a análise de desempenho, é importante salientar que a implementação será executada em imagens capturadas em tempo real, verificando a velocidade de seu processamento.

1.5 FUNDAMENTAÇÃO TEÓRICA

O sentido da visão pode ser considerado o meio mais eficiente de que o ser humano dispõe para captar as informações originadas no ambiente que o cerca (FACON, 2005). Uma de suas características principais é o mecanismo de atenção visual.

A atenção visual pode ser definida, de maneira simplista, como a focalização da região de interesse, descartando as outras em seu redor. Com essa característica, o sistema visual humano consegue efetuar o processamento das informações que recebemos de forma eficaz e eficiente, processando somente as regiões de maior interesse.

Sistemas visuais artificiais podem ser desenvolvidos baseando-se nesta característica. A visão computacional é a área da ciência que compreende o estudo do sistema visual humano para o desenvolvimento desses sistemas visuais artificiais. Esses sistemas podem ser aplicados em diversas áreas, como medicina, robótica e biologia.

Para a implementação do mecanismo de atenção visual em questão, foram utilizados os conceitos do modelo desenvolvido por Itti e colaboradores (ITTI et al., 1998). O modelo consiste na decomposição da imagem em três componentes de

atenção: intensidade, cor e orientação. Para cada componente, diferentes mapas de características são computados e para cada mapa, áreas de atenção são analisadas para a definição da mais significativa.

Os mapas de características são combinados para a criação do mapa de saliências principal, definindo a região de atenção. A figura 2 descreve o fluxo de processamento do método do mapa de saliências.

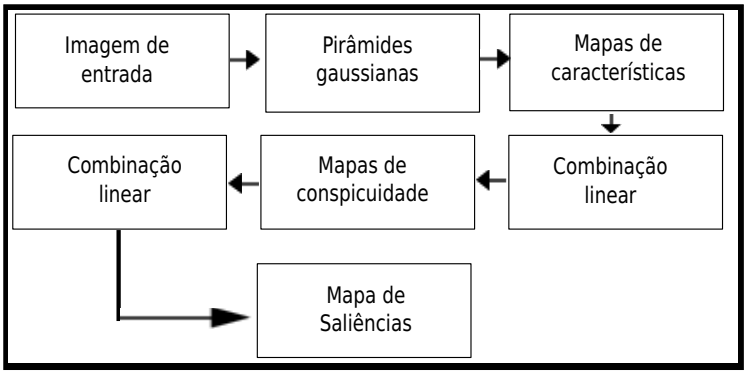

Figura 2: Mecanismo de atenção visual do mapa de saliências
Fonte: Autoria própria.

Para solucionar o problema de disponibilidade rápida das informações, o conceito de programação paralela utilizando multiprocessadores gráficos será empregado. O conceito de programação paralela consiste em dividir o processamento das informações para ser executado em cada núcleo da placa de vídeo desenvolvida para esta finalidade, segmentando o processamento em partes menores paralelizáveis.

As unidades gráficas de processamento (GPU) são responsáveis pelo processamento das informações, possuindo paralelismo de execução e obtendo assim um ganho significativo em seu processamento em relação às unidades de processamento central (CPU) usuais (CUDA, 2010).

A programação das GPU necessita de um paradigma próprio, sendo definida no escopo do projeto a utilização do modelo de programação paralela desenvolvido pela empresa Nvidia, denominado CUDA, que é integrada à linguagem de programação C/C++ para facilitar o desenvolvimento de sistemas.

Os sistemas baseados em linguagem C/C++ são programados sequencialmente, sendo suas informações processadas pela CPU desta forma. A utilização de outros processadores não necessariamente segmenta o processamento das informações, dividindo entre os processadores os processos em execução como um todo.

Na abordagem da programação paralela, o processamento das informações é segmentado entre os processadores. A exemplificação do processo é demonstrada na figura 3. Na programação serial, somente um processo é executado em um determinado período de tempo. Ao finalizar o processamento, um novo processo se inicia. Na programação paralela, os processos são divididos entre os processadores (no exemplo da figura 3, vinte processadores), sendo todas as tarefas executadas simultaneamente.

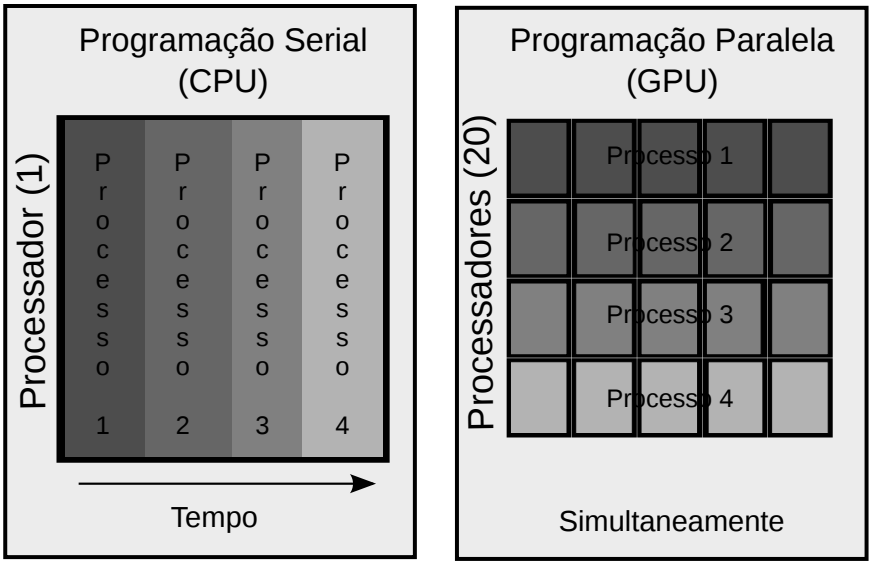

Figura 3: Processamento em programação serial x programação paralela
Fonte: Autoria própria.

O conceito de programação C/C++ foi utilizado no desenvolvimento do algoritmo de atenção visual na plataforma CUDA e na utilização da biblioteca gráfica desenvolvida na própria linguagem C/C++, no ambiente Linux. O algoritmo também foi desenvolvido na plataforma Matlab.

2 ATENÇÃO VISUAL

O mecanismo de atenção é uma parte do sistema visual, sendo que o olho humano pode se mover rapidamente para localizar os objetos de interesse em uma cena (XU et al., 2009).

A atenção visual pode ser definida como a habilidade de focalização de um ponto de interesse em uma região, descartando as outras regiões do campo visual. Essa habilidade permite que somente as informações de interesse sejam processadas de forma detalhada.

O sistema visual humano se utiliza de um controle de atenção para a escolha da região de interesse. O controle de atenção tem como parâmetros dois tipos de informações: o modelo *bottom-up*, em que elementos da própria imagem (*pixels*) são analisados ou o modelo *top-down*, onde elementos de ordem cognitiva mais alta são considerados para o controle (HEINEN, 2008).

Baseando-se no comportamento do sistema visual dos primatas, Itti e colaboradores (ITTI et al., 1998) propuseram um modelo de atenção *bottom-up*. Este modelo é formado pela decomposição de mapas de características, baseando em aspectos do sistema visual humano, para a formação de um mapa de saliências. O modelo em questão é descrito a seguir.

2.1 MAPA DE SALIÊNCIAS

Na figura 4, adaptada de Itti et al. (1998), é apresentado o modelo do mapa de saliências. O modelo pode ser descrito nas seguintes etapas: extração de características, filtragem linear, diferenças centro-periferia, normalização e soma dos mapas de características.

A imagem de entrada é decomposta em três mapas de características: intensidade, cor e orientação. Os mapas de características são criados através de pirâmides de Gauss e Gabor, através de sucessivas filtragens e sub-amostragens da imagem de entrada (ITTI et al., 1998).

Com uma imagem de resolução 640 x 480 *pixels*, as pirâmides são criadas com 9 níveis, sendo o nível zero a escala 1:1 e o nível 8 a escala 1:256. O número de níveis da pirâmide depende do tamanho da imagem de entrada, sendo limitados pela sua

sub-amostragem.

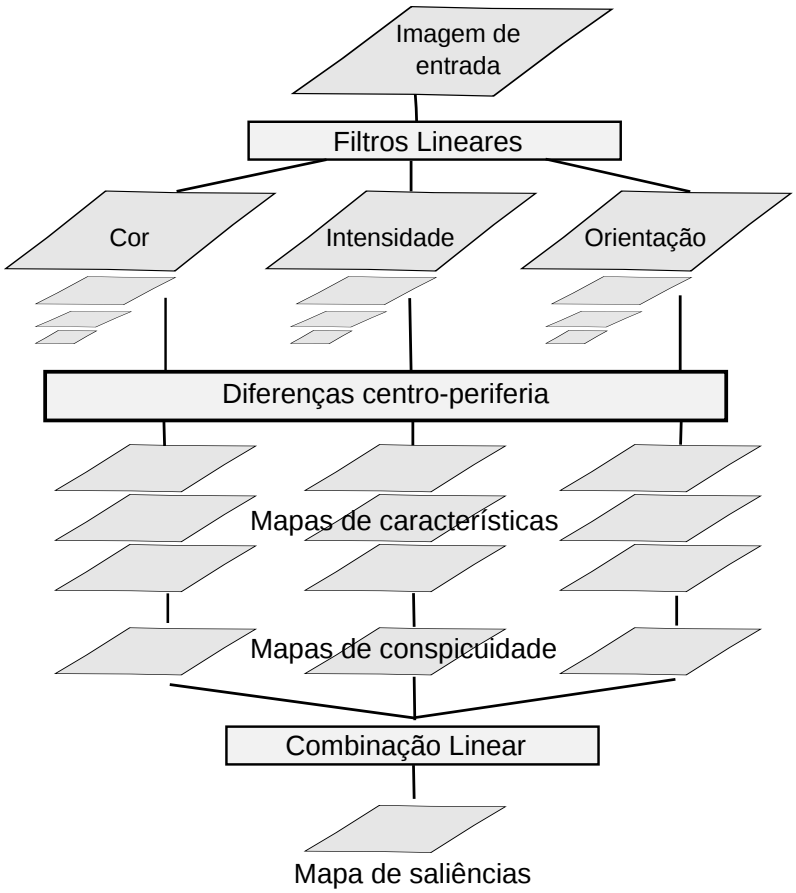

Figura 4: Arquitetura do modelo do mapa de saliências
Fonte: Adaptado de Itti et al. (1998).

A primeira etapa é a extração de características, na qual é criado o canal de intensidade (I) da imagem adquirida, a partir dos canais de cores vermelho (r), azul (b) e verde (g) da imagem de entrada:

$$I = \frac{(r + g + b)}{3}. \tag{1}$$

Após a criação do canal de intensidade, os canais de cores são normalizados por I, porém somente nas áreas onde I for maior que $1/10$ da maior intensidade (I_{max}):

$$\widehat{r} = \begin{cases} r/I & \text{se } I > I_{max}/10\ , \\ 0 & \text{caso contrário} \end{cases} \tag{2}$$

$$\widehat{g} = \begin{cases} g/I & \text{se } I > I_{max}/10\ , \\ 0 & \text{caso contrário} \end{cases} \tag{3}$$

$$\widehat{b} = \begin{cases} b/I & \text{se } I > I_{max}/10\ . \\ 0 & \text{caso contrário} \end{cases} \tag{4}$$

Em seguida, quatro novos canais de cores são criados, vermelho (R), azul (B), verde (G) e amarelo (Y):

$$R = max\{0, \widehat{r} - \frac{(\widehat{g} + \widehat{b})}{2}\}; \tag{5}$$

$$B = max\{0, \widehat{b} - \frac{(\widehat{r} + \widehat{g})}{2}\}; \tag{6}$$

$$G = max\{0, \widehat{g} - \frac{(\widehat{r} + \widehat{b})}{2}\}; \tag{7}$$

$$Y = max\{0, \frac{(\widehat{r} + \widehat{g})}{2} - \frac{|\widehat{r} - \widehat{g}|}{2} - b\}. \tag{8}$$

Depois, para cada canal de cor R, G, B e Y e para o canal de intensidade I são criadas pirâmides gaussinas $R(\sigma), G(\sigma), B(\sigma), Y(\sigma)$ e $I(\sigma)$, respectivamente, onde σ corresponde aos níveis das pirâmides. O canal de intensidade I também é utilizado para a criação da pirâmide de Gabor $O(\sigma, \theta)$, onde σ corresponde ao nível da pirâmide e θ às orientações do filtro de Gabor, com valores de $0°$, $45°$, $90°$ e $135°$.

O processo de criação dos níveis das pirâmides gaussianas pode ser descrito em 2 passos: filtragem linear passa-baixa e sub-amostragem por fator 2 da imagem de nível superior.

Para o processo de criação dos níveis das pirâmides de Gabor, é efetuada a convolução dos níveis da pirâmide de intensidade com os filtros de Gabor, descrito na equação 9 (WALTHER; KOCH, 2006), onde M_1 é o nível atual da pirâmide de intensidade e G o filtro de Gabor com fase $0°$ e $90°$. As respostas dos filtros de Gabor

se aproximam das respostas dos neurônios de orientação seletiva no cortéx primário visual humano (COSTA, 2011).

$$M_\theta = ||M_1(\sigma) * G_0(\theta)|| + ||M_1(\sigma) * G_{\pi/2}(\theta)||, \text{ com } \theta \in 0°, 45°, 90° \text{ e } 135°. \quad (9)$$

Os filtros de Gabor são definidos por (WALTHER; KOCH, 2006):

$$G_\Psi(x, y, \theta) = \exp\Big(-\frac{x'^2 + \gamma^2 y'^2}{2\delta^2}\Big)\cos\Big(2\pi\frac{x'}{\lambda} + \Psi\Big), \text{ com } \Psi = \{0, \tfrac{\pi}{2}\}. \quad (10)$$

onde γ define a proporção, δ o desvio padrão, λ o comprimento de onda e Ψ a fase da função de Gabor nas coordenadas x' e y', com as respectivas orientações θ:

$$x' = x\cos(\theta) + y\sin(\theta); \quad (11)$$

$$y' = -x\sin(\theta) + y\cos(\theta). \quad (12)$$

Com as pirâmides definidas, os mapas de características são obtidos efetuando-se a diferença entre os canais em diferentes escalas. Este processo é definido como diferença centro-periferia.

O processo de diferença centro-periferia é inspirado biologicamente, utilizando-se da estrutura de campos visuais receptivos. Segundo Itti (2000), os campos visuais dos neurônios respondem mais a uma pequena região do espaço visual central. Por sua vez, os estímulos apresentados nas regiões vizinhas inibem a resposta neural.

As operações centro-periferia são implementadas no modelo como diferenças entre as escalas finas, níveis de maior resolução da pirâmide, e as escalas grossas, níveis de menor resolução da pirâmide. Para efetuar a diferença entre escalas (denominada $\ominus$), é utilizada a técnica de processamento de imagens denomidada interpolação bilinear, que expande ou reduz a escala das imagens. O centro é um *pixel* na escala $c \in \{2, 3, 4\}$ e a periferia é o *pixel* correspondente na escala $s = c + \delta$ com $\delta \in \{3, 4\}$.

Os mapas de intensidade (I) estão relacionados com o contraste detectado pelos neurônios sensíveis ao brilho no centro e ao escuro na periferia e pelos neurônios

sensíveis ao escuro no centro e ao brilho na periferia. Os dois tipos sensíveis de contraste de intensidade são calculados simultaneamente da seguinte forma:

$$I(c,s) = |I(c) \ominus I(s)|. \tag{13}$$

Os mapas de cores são gerados com relação à rivalidade de cores, onde os neurônios no centro são excitados por uma cor e inibidos por outra, ocorrendo o inverso na periferia. Através dessa relação, são criados os mapas entre as rivalidades vermelho X verde e azul X amarelo, conforme segue:

$$RG(c,s) = |(R(c) - G(c)) \ominus (R(s) - G(s))|; \tag{14}$$

$$BY(c,s) = |(B(c) - Y(c)) \ominus (B(s) - Y(s))|. \tag{15}$$

Os mapas de orientação (O) são criados com relação ao contraste da orientação entre as escalas do centro e da periferia, sendo gerados separadamente para cada orientação:

$$O(c,s,\theta) = |O(c,\theta) \ominus O(s,\theta)|. \tag{16}$$

Com a geração dos mapas de características, obtêm-se seis mapas para intensidade, 12 mapas para cores rivais e 24 mapas para orientação, totalizando 42 mapas.

Para combinar os mapas de características em somente um mapa de saliências, é necessária a utilização de um operador de normalização $N(.)$. Esse operador simula a competição local entre os locais de vizinhança saliente, permitindo que os mapas que possuem um menor número de fortes picos de atividade se sobressaiam aos mapas com uma grande quantidade de picos de atividade comparável (COSTA, 2011).

O operador de normalização é calculado da seguinte forma:

1. Normalização dos mapas de características;
2. Para cada mapa normalizado, encontra-se o máximo global M e a média $\bar{m}$ de todos os outros máximos locais;
3. Multiplica-se globalmente o mapa por $(M - \bar{m})^2$.

Os mapas de características normalizados são combinados em três mapas de conspicuidade, sendo um para cada característica. Os mapas de conspicuidade são gerados através de uma adição entre escalas (definido como $\oplus$), que consiste em reduzir ou expandir cada mapa para a escala $\sigma = 4$ e efetuar uma soma *pixel* a *pixel*:

$$\bar{I} = \bigoplus_{c=2}^{4} \bigoplus_{s=c+3}^{c+4} N(I(c,s)); \tag{17}$$

$$\bar{C} = \bigoplus_{c=2}^{4} \bigoplus_{s=c+3}^{c+4} [N(RG(c,s)) + N(BY(c,s))]; \tag{18}$$

$$\bar{O} = \sum_{\theta \in \{0°,45°,90°,135°\}} N\left[\bigoplus_{c=2}^{4} \bigoplus_{s=c+3}^{c+4} N(O(c,s,\theta))\right], \tag{19}$$

Onde $\bar{I}$ é o mapa de conspicuidade para intensidade, $\bar{C}$ é o mapa de conspicuidade para cores e $\bar{O}$ é o mapa de conspicuidade para orientações.

O motivo da criação de três mapas de conspicuidade diferentes é a premissa de que características diferentes contribuem para a saliência independentemente, enquanto que características iguais competem entre si. Para a construção do mapa de saliências final (S), os três mapas de conspicuidade são normalizados, somados e efetuada a média:

$$S = \frac{1}{3}(N(\bar{I}) + N(\bar{C}) + N(\bar{O})). \tag{20}$$

3 PROCESSAMENTO PARALELO

3.1 GPU

A indústria de semicondutores, desde 2003, vem seguindo por duas trajetórias no desenvolvimento de processadores: os processadores com múltiplos núcleos, buscando aumentar a velocidade do processamento sequencial e a trajetória dos muitos núcleos, focando-se na execução de várias tarefas em paralelo.

Uma evolução dessas trajetórias pode ser analisada com os atuais processadores Intel Core i7, que possuem quatro núcleos de processadores independentes em comparação à GPU Nvidia GeForce GTX 280, que possui 240 núcleos compartilhados (*multithreaded*). A evolução do desempenho de CPUs e GPUs ao longo dos anos é ilustrado na figura 5 (CUDA, 2010).

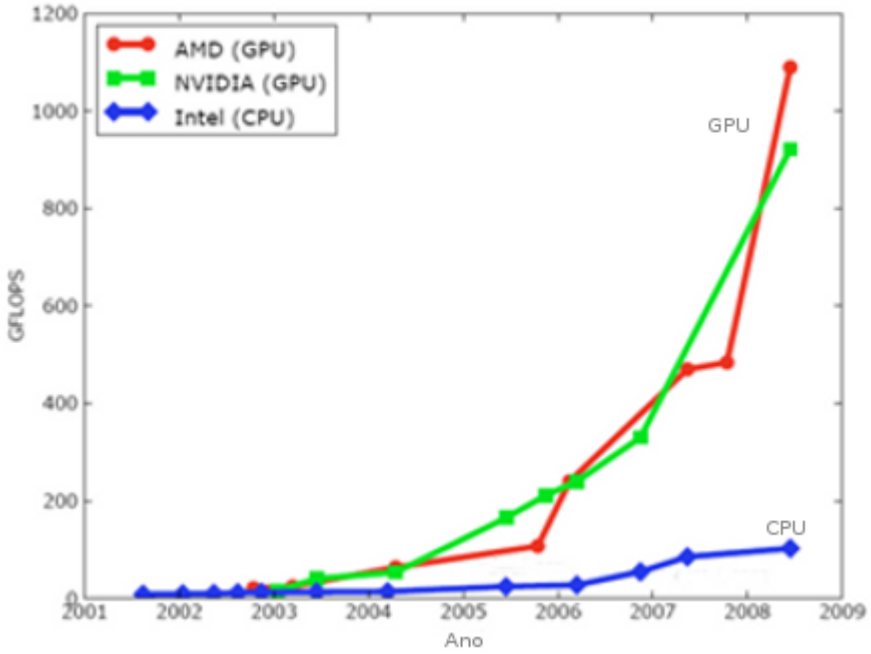

Figura 5: Evolução entre GPU x CPU
Fonte: Adaptado de Cuda (2010).

A grande diferença de desempenho na evolução das CPUs e GPUs se deve ao fato da dedicação exclusiva das GPUs ao processamento das informações, possuindo mais transistores para esse processamento, reduzindo o controle de memória cache e o fluxo de controle em relação às CPUs, e principalmente ao seu suporte ao processamento paralelo intenso, conforme ilustrado na figura 6 (CUDA, 2010).

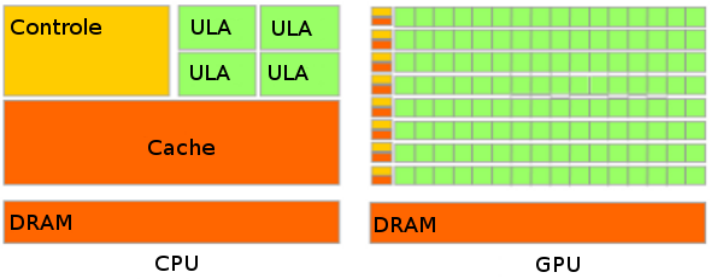

Figura 6: Comparação entre as arquiteturas CPU e GPU
Fonte: Adaptado de Cuda (2010).

As GPUs são projetadas para efetuar de forma otimizada e maximizada cálculos numéricos. Porém, deve se deixar claro que em algumas situações, o desempenho do modelo sequencial será superior. Para integrar esses dois modelos, a empresa Nvidia, desenvolveu em 2007 um modelo de programação denominado CUDA (*Compute Unified Device Architecture*), permitindo a execução conjunta de uma aplicação CPU/GPU.

3.2 CUDA

CUDA é uma arquitetura de programação paralela desenvolvida pela empresa Nvidia que permite o acesso de forma simplificada às unidades de processamento gráfico em algumas de suas placas de vídeo.

A arquitetura CUDA é composta por *multithreads*, isto é, a execução de várias tarefas de um processo em paralelo. A GPU controla desde a criação até a execução das *threads* automaticamente, permitindo uma interface transparente ao desenvolvimento em CUDA.

A GPU é considerada pela CPU como um processador auxiliar, com a capacidade de executar uma grande quantidade de *threads* em paralelo. Por esse motivo, na arquitetura CUDA, a GPU é denominada *device*, sendo a CPU denominada *host*.

Para um programa em CUDA, as tarefas que possuem pouco ou nenhum paralelismo são executados no *host* (CPU). Por outro lado, as tarefas que possuem uma grande quantidade de paralelismo são executadas no *device* (GPU).

3.2.1 API CUDA

O desenvolvimento de aplicações em CUDA é realizado com uma extensão da linguagem de programação C com comandos específicos para GPU. Também é disponibilizada a utilização de funções já otimizadas para GPU, como a biblioteca de FFT (Transformada Rápida de Fourier). O modelo da arquitetura CUDA é apresentado na figura 7.

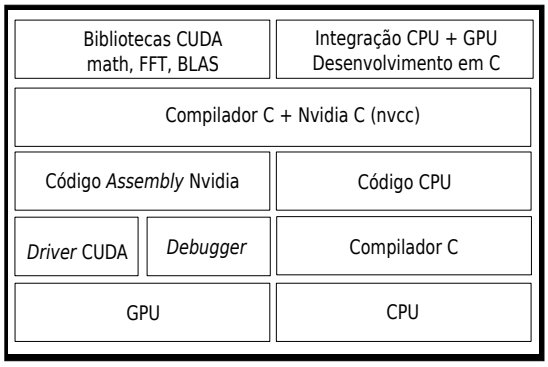

Figura 7: Arquitetura CUDA
Fonte: Adaptado de Halfhill (2008).

O programa escrito em CUDA deverá ser compilado utilizando-se o compilador específico da Nvidia (nvcc), gerando um código assembly para a GPU e um código em linguagem C para a CPU. O código assembly é inserido na GPU pelo *driver* CUDA, enquanto o código C é compilado utilizando-se o compilador específico para C (gcc) e disponibilizado na CPU.

Para o desenvolvimento em CUDA, é necessário que a placa gráfica seja compatível com essa arquitetura e a instalação de um *driver* específico e de um *toolkit* para a programação, que contém o compilador e as bibliotecas adicionais. Na figura 8 pode-se visualizar a sequência de processamento na arquitetura CPU/GPU. O código é executado sequencialmente na CPU até encontrar uma chamada de função CUDA, definida como *kernel*, que irá efetuar o processamento paralelo das informações da função na GPU. Após o processamento paralelo, o código é executado novamente sequencialmente na CPU.

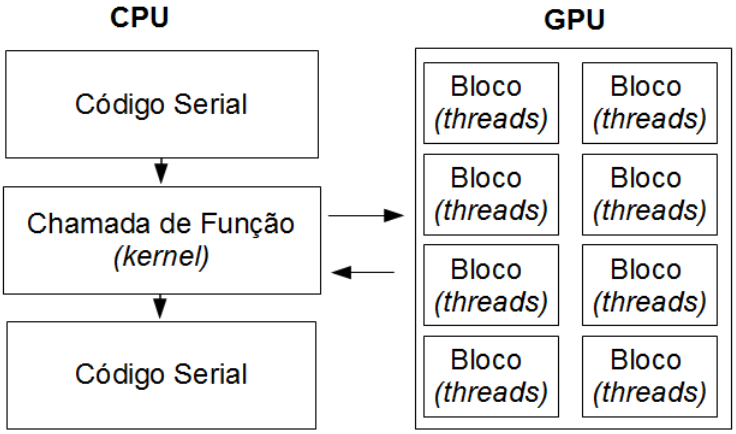

Figura 8: Fluxo de execução de um programa na arquitetura CUDA
Fonte: Autoria própria.

Com a instalação da API CUDA, algumas extensões da linguagem C são inseridas, tais como quantificadores do tipo de função e quantificadores de tipo de variável.

Os quantificadores do tipo de função definem em qual plataforma será executado o código, em CPU ou GPU, existindo três tipos:

- _device_ : define que a função será executada em GPU. Somente poderá ser chamada a partir da GPU.
- _global_: define que a função será executada em GPU, porém, será chamada a partir da CPU. Esse tipo de função é utilizado na arquitetura CUDA através da função *kernel*.
- _host_: define que a função será executada na CPU. Somente poderá ser chamada a partir da CPU.

Algumas restrições são impostas nas funções pelo tipo de quantificadores:

- As funções _device_ e _global_ não suportam recursividade.
- As funções _device_ e _global_ não permitem a declaração de variáveis estáticas.
- As funções _device_ e _global_ não permitem número variável de argumentos.
- As funções _global_ e _host_ não podem ser utilizadas simultaneamente.

As funções do tipo _global_ são chamadas a partir da CPU pela função *kernel*. Essa função é definida através de um escalonamento de *threads*, definidos como grade e bloco. O *kernel* efetua a divisão dos dados para o processamento paralelo, assim como a distribuição para as *threads*.

Uma grade é uma entidade onde estão distribuídos os blocos. Nas grades estão definidos os números de blocos de *threads* que serão criados pela GPU para uma determinada função. Uma grade pode ter uma ou duas dimensões (COSTA, 2011).

O bloco é uma unidade de organização das *threads*. Sua função é definir a quantidade de *threads* que serão utilizadas, assim como alocar recursos da GPU. Um bloco poderá ter até três dimensões de *threads*.

A figura 9 exemplifica a hierarquia de *threads* em CUDA. Para cada chamada de *kernel* do *device* (GPU) pelo *Host* (CPU), é criada uma grade de blocos de *threads*, para a execução das informações em paralelo. No exemplo, é criada uma grade de blocos com dimensões 3 x 2 e cada bloco de *threads* com dimensão 5 x 3.

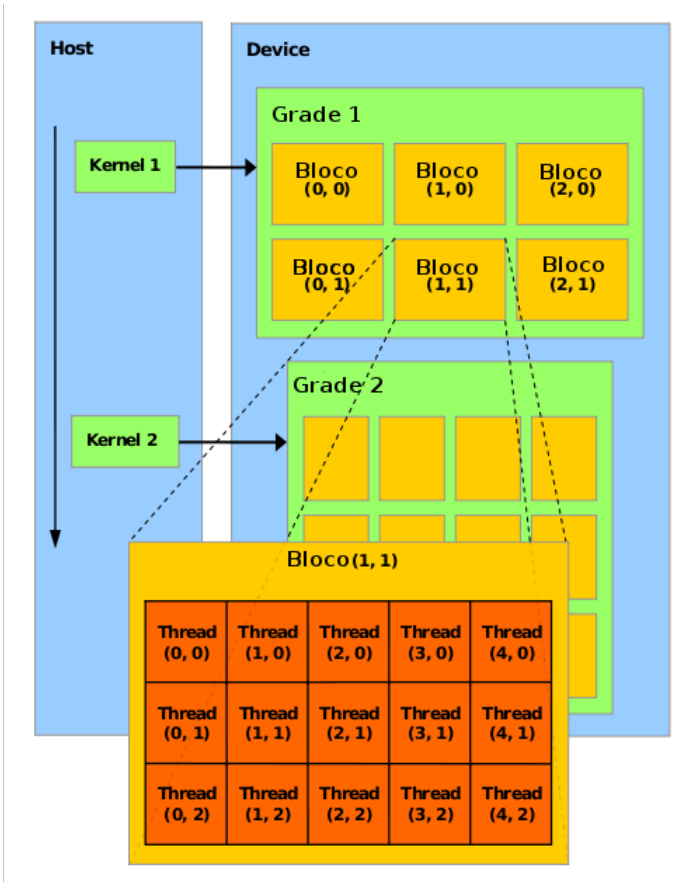

Figura 9: Exemplo da criação de uma hierarquia de *threads* em CUDA
Fonte: Adaptado de Kirk; Hwu (2010).

Os quantificadores do tipo de variável definem a localidade de memória que será utilizada. São definidos por:

- _device_ : define que a variável vai residir na memória global do *device* (GPU). Seu tempo de vida é da duração da aplicação e podem ser acessadas por todas as *threads* de uma grade. Também poderá ser acessada pela CPU através do uso da biblioteca de execução do CUDA.

- _constant_ : define que a variável vai residir na memória de constantes da GPU e seu tempo de vida é o da duração da aplicação.

- _shared_ : tipo de variável que é acessada somente pelas *threads* de um mesmo bloco. O seu tempo de vida é a duração de um bloco.

Na programação do código, para efetuar a chamada da função do *device* (GPU), é utilizado o *kernel*, que possui uma nova sintaxe de chamada de função: entre o nome da função e seus parâmetros é adicionado um *array* bidimensional, definindo o tamanho da grade e do bloco, entre os símbolos $<<<$ e $>>>$. As GPUs disponíveis atualmente permitem blocos com até 512 *threads*, sendo o tamanho da grade limitado pela memória disponibilizada pela GPU (COSTA, 2011).

O tamanho da grade e do bloco são definidos como vetores de três elementos, declarados na API CUDA com o tipo de variável *dim3*. No entanto, cada grade poderá tem no máximo duas dimensões de blocos de *threads*.

Para o acesso dentro da função do *device*, o índice das *threads* é acessado através das variáveis *threadIdx.x*, *threadIdx.y* e *threadIdx.z*, correspondentes à posição das *threads* dentro do bloco definido na chamada da função.

Para o acesso a cada índice de bloco, são usadas as variáveis *blockIdx.x* e *blockIdx.y*. Para o acesso aos valores das dimensões do bloco, utiliza-se as variáveis *blockDim.x*, *blockDim.y* e *blockDim.z* e para as dimensões da grade através das variáveis *gridDim.x* e *gridDim.y*.

Para demonstrar a programação em CUDA, um código simples é mostrado na figura 10. O código somente exemplifica uma chamada de função *kernel* e uma função no *device*, sem a implementação de alocação de memória e comandos da linguagem C.

Exemplo de uma função para somar 2 vetores de tamanho 256 na plataforma C / CUDA

```
__global__ void soma(float *vet_1, float *vet_2, float *resultado, int w)
{
int x = blockIdx.x*blockDim.x + threadIdx.x;
int y = blockIdx.y*blockDim.y + threadIdx.y;
resultado[x*w+y] = vet_1[x*w+y] + vet_2[x*w+y];
}
```

Chamada de kernel para a função soma, definindo o tamanho da grade e do bloco de *threads*

```
dim3 grade(1);
dim3 bloco(16,16);
soma <<< grade, bloco >>> (vet_1, vet_2, resultado, 256);
```

Figura 10: Exemplo do código na arquitetura CUDA
Fonte: Autoria própria.

As execuções das *threads* nas funções são assíncronas, sendo necessária a criação de uma barreira de execução através da função **"_syncthreads()"**. O mesmo conceito ocorre na chamada da função, onde a execução do código na CPU independe da execução do código na GPU. A função **"cudaThreadSynchronize()"** interrompe o processo na CPU até que toda a execução da aplicação seja completada na GPU.

4 IMPLEMENTAÇÃO

Com a finalidade de visualizar os resultados do estudo teórico do mecanismo de atenção visual e analisar seu desempenho, foram efetuadas implementações do algoritmo em três arquiteturas diferentes: Matlab, linguagem de programação C++ e API CUDA.

As implementações seguiram o mesmo modelo descrito anteriormente, porém, com algumas alterações na forma de processamento de acordo com a arquitetura escolhida.

Buscando facilitar o desenvolvimento e a análise dos resultados, o projeto foi subdividido em três partes: mapa de intensidade, mapa de cor e mapa de orientação.

Inicialmente foi implementado o modelo do mapa de saliência no software de cálculos numéricos Matlab. O software foi escolhido por ser uma ferramenta de fácil utilização.

Na segunda etapa, o algoritmo foi desenvolvido em linguagem C++, utilizando-se da biblioteca de processamento de imagens desenvolvida pelo professor Hugo Vieira Neto.

Com o algoritmo desenvolvido em C++, foi possível determinar as funções para efetuar o paralelismo, desenvolvendo o código na API CUDA.

4.1 MATLAB

A primeira etapa no desenvolvimento do código foi a captura da imagem, na resolução 640 x 480 *pixels*. Com a função própria do Matlab, denominada “imread”, a imagem é carregada em forma de uma matriz bidimensional.

Os valores dos pixels da imagem são normalizados para a faixa $[0, 1]$ e transformados para o formato *double*. A conversão da imagem para esse formato se deve ao fato do Matlab processar os dados de imagens como números inteiros, gerando erros no processamento das informações e posteriormente, erros no mapa de saliência.

Na próxima etapa, efetua-se a decomposição dos canais de cores da imagem de entrada e gerados os novos canais: R, G, B e Y. Foi implementada uma nova função, denominada *zero*, que efetua a saturação de valores negativos em zero.

Para a implementação das pirâmides gaussianas, optou-se por efetuar o desenvolvimento separado entre os mapas de características, criando três arquivos de funções: intensidade, cor e orientação.

A função intensidade recebe o canal de intensidade criado na etapa anterior para a criação da pirâmide. Uma função denominada *piramide_gauss* define o primeiro nível da pirâmide com o canal de intensidade recebido e cria os outros níveis.

Os outros níveis são gerados efetuando-se uma filtragem passa-baixa e uma sub-amostragem do nível anterior. O processo de filtragem é efetuado através da convolução do filtro gaussiano com a imagem. Para esse processo, foi utilizada a função “imfilter” do Matlab. Somente com a finalidade de aprendizagem, para efetuar a sub-amostragem, foi desenvolvida uma função que efetua a interpolação bilinear da imagem, pois o Matlab possui a função “imresize” para redimensionamento de imagens.

Com a pirâmide de intensidade gerada, conforme figura 11, foi efetuado o método de diferenças centro-periferia. As resoluções dos níveis da pirâmide foram redimensionadas utilizando a função “imresize”, redimensionando as imagens para a resolução necessária.

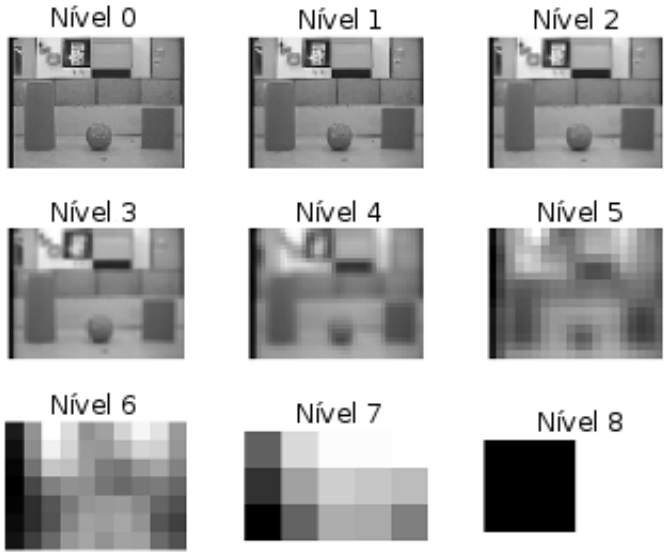

Figura 11: Pirâmide de intensidade (normalizada)
Fonte: Autoria própria.

O mapa de conspicuidade de intensidade foi gerado efetuando-se uma soma

normalizada do resultado do método de diferenças centro-periferia. Uma nova função foi criada para normalizar as imagens, baseando-se no método de normalização descrito anteriormente. No mapa da intensidade, os objetos com maiores intensidades são realçados.

A próxima etapa foi a criação do mapa de conspicuidade de cor. Para efetuar a implementação das pirâmides de cores, a mesma função utilizada na pirâmide de intensidade foi utilizada, porém executada quatro vezes, recebendo como parâmetro os quatro canais de cores: R, G, B e Y.

O método de diferença centro-periferia foi utilizado para a criação do mapa de conspicuidade de cor, sendo gerado da mesma forma descrita no mapa de intensidade, porém, sendo implementado para efetuar a rivalidade entre cores ($R - G$ e $B - Y$). Na geração do mapa de cor, os objetos que possuem cores mais intensas são realçados perante os objetos que possuem cores menos intensas.

Na etapa de criação do mapa de conspicuidade de orientação, foi utilizada a pirâmide de intensidade gerada anteriormente. Para cada nível da pirâmide foram gerados dois filtros de Gabor, um para a orientação 0° e outro para 90°.

Os filtros de Gabor foram gerados com os seguintes parâmetros: $\gamma = 1$, $\delta = 7/3$ *pixels*, $\lambda = 7$ *pixels*, $\Psi = \{0^\circ, 90^\circ\}$ e seu tamanho definido como dimensões 19 x 19, conforme descrito por (WALTHER; KOCH, 2006).

Com os filtros de Gabor gerados, foram efetuadas as convoluções entre os filtros e os níveis da pirâmide, utilizando-se novamente da função “imfilter” do Matlab.

A etapa seguinte foi a geração do mapa de conspicuidade de orientação, sendo efetuado o mesmo processo dos mapas anteriores (intensidade e cor), porém, para o mapa de conspicuidade de orientação, devido à convolução com o filtro de Gabor, foi necessário efetuar uma atenuação das bordas das imagens, após efetuar o método de diferença centro-periferia. Os objetos com orientações definidas nos filtros de Gabor são realçados no mapa de orientação.

Os mapas de conspicuidades são demonstrados na figura 12.

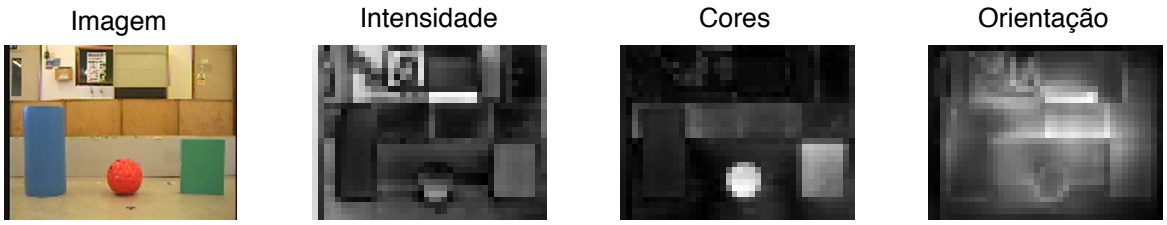

Figura 12: Mapas de conspicuidade
Fonte: Autoria própria.

A última etapa do código em Matlab, após a criação dos três mapas de conspicuidade, foi a geração do mapa de saliência, somando-se os três mapas normalizados na resolução 40 x 30 *pixels* (ITTI et al., 1998). O mapa de saliência final da imagem de teste é mostrado na figura 13, onde o objeto central, uma bola laranja, constitui a região mais saliente da imagem.

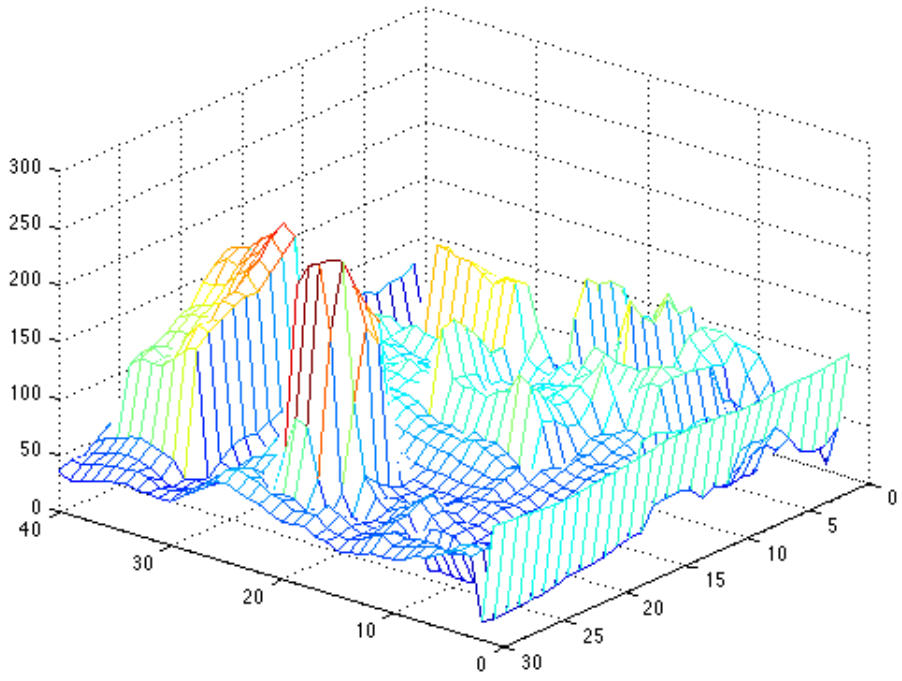

Figura 13: Exemplo de Mapa de Saliências
Fonte: Autoria própria.

4.2 C++

Na segunda etapa, o algoritmo do mapa de saliência foi implementado na linguagem C++, no sistema operacional Linux. Para sua implementação, a mesma

estrutura de funções desenvolvidas no Matlab foi utilizada.

A imagem com resolução 640 x 480 *pixels* foi capturada utilizando-se uma função da biblioteca de processamento de imagens desenvolvida pelo professor Hugo Vieira Neto. Essa biblioteca cria um objeto denominado *imagem*, possuindo alguns métodos e funções definidos. A função para a captura de imagens permite a utilização tanto de imagens estáticas quanto imagens capturadas em tempo real por uma câmera, sendo a função de captura de imagens da câmera implementada na biblioteca de processamento de imagens em projetos anteriores.

Assim como em Matlab, foram criadas funções para a decomposição dos canais de cores, para a criação das pirâmides de características e dos mapas de conspicuidade.

Porém, para a execução do algoritmo em C++, foram criadas funções específicas para efetuar cálculos de soma, subtração e multiplicação entre matrizes e entre matrizes e escalares. As funções internas do Matlab, como a verificação do máximo e mínimo também foram implementadas em C++.

Com a finalidade de estudo, os filtros foram convertidos para o domínio da frequência. Outro motivo para efetuar essa conversão para o domínio da frequência espacial é que, segundo descrito em (COSTA, 2011), a utilização da convolução com um filtro de dimensões 19 x 19 em C++ se torna inviável por consumir muitos recursos.

As imagens e os filtros de Gabor são convertidos para o domínio da frequência utilizando-se de uma função da biblioteca de imagens para FFT. Porém, para efetuar o cálculo da multiplicação entre os números complexos resultantes, foi criada uma função específica.

Após o término do cálculo, utiliza-se a função inversa da FFT, denominada IFFT para retornar ao domínio espacial.

Com a criação das pirâmides de características, o processo de criação dos mapas de conspicuidade foi efetuado redimensionando todas as imagens para a escala 4, com dimensões (40 x 30 *pixels*) e efetuando os mesmos métodos descritos em Matlab.

Os resultados obtidos na linguagem de programação C++ diferem um pouco dos resultados obtidos no *software* Matlab por diferenças de implementação. A diferença entre um mapa de saliências obtido no *software* Matlab e em linguagem C++ é mostrada na figura normalizada 14, onde a parte mais clara define a maior diferença e a parte escura a menor diferença entre os mapas.

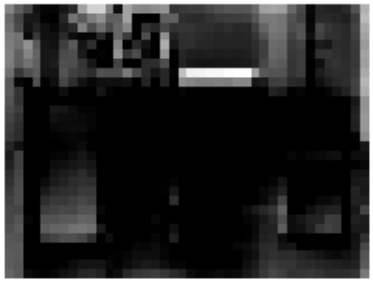

Figura 14: Comparação do Mapa de Saliências no *software* Matlab e na linguagem de programação C++ (normalizada)
Fonte: Autoria própria.

4.3 API CUDA

O desenvolvimento na API CUDA foi realizado utilizando-se do CUDA *Toolkit* 4.0, disponibilizado no *website* da empresa Nvidia.

Para se utilizar a API CUDA, primeiramente a placa gráfica deve possuir suporte ao CUDA. A placa gráfica utilizada no projeto foi a Nvidia Geforce 9400M, com 16 núcleos para processamento paralelo, sendo que *drivers* atualizados foram instalados e configurados, estes também disponibilizados no *website* da empresa Nvidia.

O código foi implementado utilizando-se da biblioteca de imagens com base em C++. Foi implementado um código para a captura de imagens em *OpenCV* (BRADSKI; KAEHLER, 2008), porém optou-se por manter a utilização da biblioteca já utilizada anteriormente. O fluxo de dados da implementação do mapa de saliência em CUDA é demonstrado na figura 15.

Após a imagem com resolução 640 x 480 *pixels* ser capturada, esta é convertida de uma matriz bidimensional para um vetor, pois a estrutura utilizada não permite a transferência de matrizes para a GPU. Nesse processo a imagem também já é normalizada para a faixa de valores [0,1].

Na próxima etapa, é efetuada a decomposição dos canais de cores da imagem de entrada e gerados novos canais, conforme o modelo matemático do mapa de saliências. Uma grade de dimensões (32,32) e um bloco de dimensões (20,15) são criados para efetuar a chamada de função na GPU. A primeira dimensão da grade e do bloco correspondem ao tamanho horizontal da imagem, totalizando 640 *pixels* (32 x 20 = 640), enquanto que a segunda dimensão define o tamanho vertical, totalizando 480 *pixels* (32 x 15 = 480).

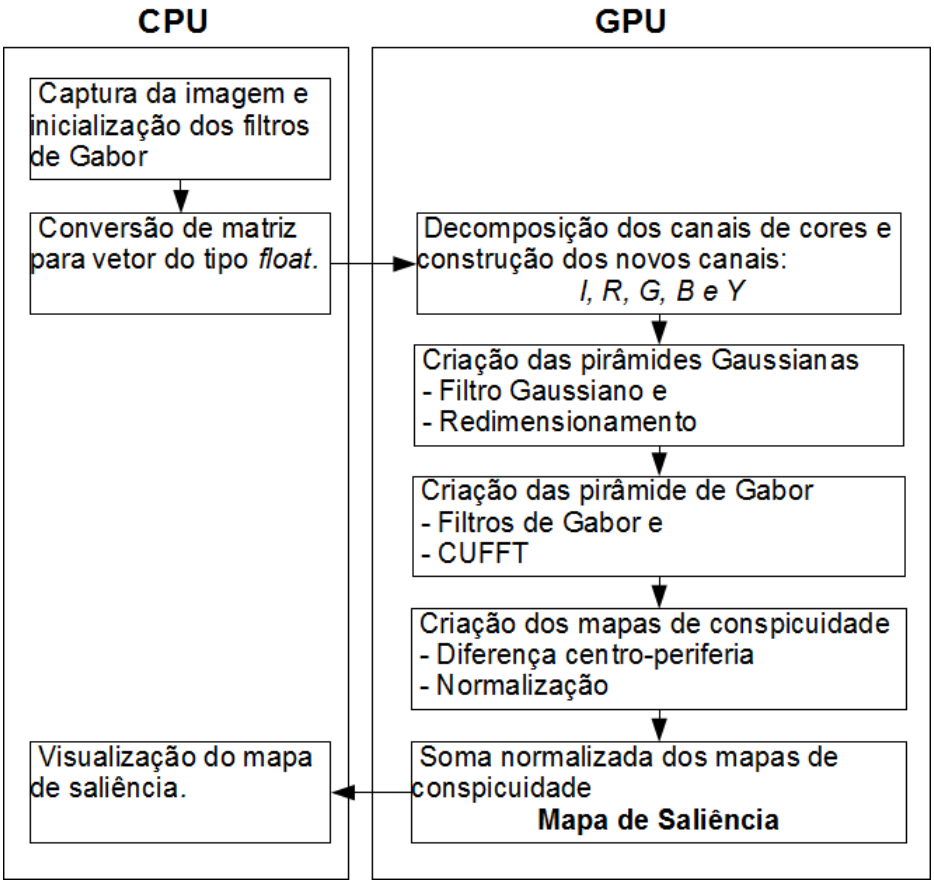

Figura 15: Fluxo de processamento das funções implementadas em CUDA
Fonte: Autoria própria.

Mantendo as mesmas dimensões de bloco e grade, as pirâmides de características são criadas executando os *kernels* para a filtragem gaussiana na direção horizontal, na direção vertical, e após a filtragem, efetuando a sub-amostragem através da função de interpolação bilinear implementada em CUDA.

Com a geração das pirâmides de características, todos os níveis foram redimensionados para a escala 4, com dimensões (40 x 30 *pixels*), executando novamente a função de interpolação bilinear.

Na próxima etapa foi efetuado o método de diferença centro-periferia, sendo criado um novo bloco de dimensões (20,15) e com uma nova grade, com dimensões (2,2), totalizando as dimensões da imagem (40 x 30 *pixels*).

Um vetor com o resultado do método centro-periferia foi criado para cada mapa de característica, sendo esses vetores normalizados através do operador *N*. Para o cálculo dos mapas de conspicuidade, uma função que efetua a soma entre os elementos do vetor foi criado.

4.4 COMPARAÇÃO ENTRE AS PLATAFORMAS

Os códigos de programação foram implementados nas plataformas propostas visando o estudo do mapa de saliências e a arquitetura de processamento paralelo.

Para efetuar a filtragem gaussiana no *software* Matlab, foi utilizada a função nativa “imfilter”, que efetua a convolução entre uma imagem e um filtro. O filtro utilizado foi um filtro gaussiano de dimensões 5 x 5, gerado utilizando a função “fspecial”. A implementação é descrita na figura 16.

```
function [gl] = filtro(img)
 flt = fspecial('gaussian', [5 5], 2);
 img2 = padarray(img,[2 2],'symmetric', 'both');
 gi = imfilter(img2,flt);
 [l c] = size(gi);
 gl = gi(3:l-2,3:c-2);
end
```

Figura 16: Filtragem gaussiana em Matlab
Fonte: Autoria própria.

Na linguagem de programação C++, foi utilizada a função implementada na biblioteca de processamento de imagens, que efetua a geração e convolução do filtro internamente na classe imagem. Em CUDA, a filtragem gaussiana é efetuada em duas etapas, sendo efetuada a convolução primeiramente horizontalmente e depois verticalmente. A necessidade de executar a filtragem em duas etapas se deve ao fato da imagem estar descrita em um vetor, facilitando a implementação. O filtro horizontal é demonstrado na figura 17, sendo o mesmo filtro executado verticalmente posteriormente.

```
__global__ void filtroX(float *in, int w, int h)
{
 int x = blockIdx.x*blockDim.x + threadIdx.x;
 int y = blockIdx.y*blockDim.y + threadIdx.y;

 if(x > w - 2 || y >  h - 2 || x < 2 || y < 2)
 return;

 int idx = y*w;
 int a = x-2;
 int b = x-1;
 int c = x;
 int d = x+1;
 int e = x+2;

 if(a < 0) a = 0;
 if(b < 0) b = 0;
 if(c >= w) c = w;
 if(d >= w) d = w;
 if(e >= w) e = w;
 __syncthreads();

in[y*w+x] = 0.0625f*in[idx+a] + 0.25f*in[idx+b] + 0.375f*in[idx+c] + 0.25f*in[idx+d] + 0.0625f*in[idx+e];
}
```

Figura 17: Filtragem gaussiana em CUDA
Fonte: Autoria própria.

A implementação da interpolação bilinear para a criação das pirâmides gaussianas foi efetuada utilizando-se de uma nova função, implementada utilizando dos conhecimentos adquiridos nas aulas de processamento de imagens, conforme descreve a figura 18. Para o redimensionamento das imagens, a função interna do Matlab “imresize” foi utilizada.

```
function m_fin = interpola(m_ent, interx,intery)
 tam = size(m_ent);
 linha = tam(1);
 coluna = tam(2);
 tam(1) = floor(tam(1)*interx);
 tam(2) = floor(tam(2)*intery);
 x = 0;
 y = 0;
 for i=1:tam(1)
  for j=1:tam(2)
   x_ent = (x/interx);
   y_ent = (y/intery);
   x_int = floor(x_ent)+1;
   x_fract = x_ent-floor(x_ent);
   y_int = floor(y_ent)+1;
   y_fract = y_ent-floor(y_ent);

   a = ((m_ent(x_int, y_int)*(1-x_fract)*(1-y_fract)));
   b = ((m_ent(x_int+1, y_int)*(x_fract)*(1-y_fract)));
   c = ((m_ent(x_int, y_int+1)*(1-x_fract)*y_fract));
   d = ((m_ent(x_int+1, y_int+1)*x_fract*y_fract));

   m_fin(i,j) = (a + b + c + d);

   y = y + 1;
  end
  x = x + 1;
  y = 0;
 end
end
```

Figura 18: Interpolação bilinear em Matlab
Fonte: Autoria própria.

A função de interpolação bilinear na linguagem de programação C++ também foi implementada utilizando a biblioteca de processamento de imagens. Na arquitetura CUDA foi utilizado o mesmo algoritmo implementado em Matlab, adequando o código para a API CUDA. O código na arquitetura CUDA é mostrado na figura 19.

```
__global__ void interpola(float *in, int w1, int h1, float *out, int w2, int h2)
{
 int x = blockIdx.x*blockDim.x + threadIdx.x;
 int y = blockIdx.y*blockDim.y + threadIdx.y;
 float a, b, c, d, menos_x, menos_y;
 float fraction_x, fraction_y;
 int ceil_x, ceil_y, floor_x, floor_y;
 float x_frac = (float)w1/(float)w2;
 float y_frac = (float)h1/(float)h2;

 if( (x <= w2) && (y <= h2) ) {
  floor_x = (int)floor(x * x_frac);
  floor_y = (int)floor(y * y_frac);
  if (floor_x <0) floor_x = 0;
  if (floor_y <0) floor_y = 0;
  if(floor_x >= w1) floor_x = w1;
  if(floor_y >= h1) floor_y = h1;

  ceil_x = floor_x + 1;
  ceil_y = floor_y + 1;

  if (ceil_x >= w1) ceil_x = floor_x;
  if (ceil_y >= h1) ceil_y = floor_y;

  fraction_x = x * x_frac - floor_x;
  fraction_y = y * y_frac - floor_y;
  menos_x = 1 - fraction_x;
  menos_y = 1 - fraction_y;

  a = in[floor_y*w1 + floor_x];
  b = in[floor_y*w1 + ceil_x];
  c = in[ceil_y *w1 + floor_x];
  d = in[ceil_y *w1 + ceil_x];

  out[y*w2+x] = menos_y*(menos_x *a +fraction_x*b) + fraction_y*(menos_x*c+fraction_x*d);
 }
}
```

Figura 19: Interpolação bilinear em CUDA
Fonte: Autoria própria.

A construção das pirâmides gaussianas foi efetuada utilizando-se de um vetor de imagens. Na primeira etapa é efetuada a filtragem linear e após, o redimensionamento das imagens através da interpolação bilinear. Em Matlab, foi gerada uma função para gerar as pirâmides, possuindo como entrada o primeiro nível da pirâmide e a quantidade de níveis desejados. A função é definida na figura 20.

```
function [piramide_] = piramide_gaussiana(feature_, nivel_)
 piramide_ = {};
 piramide_{1} = feature_;
 for i = 2:nivel_
  filtro_ = filtro(piramide_{i-1});
  piramide_{i} = reduz(filtro_,1);
 end
end
```

Figura 20: Construção das pirâmides gaussianas em Matlab
Fonte: Autoria própria.

Na linguagem de programação C++, a construção das pirâmides foi efetuada já definindo as características a serem utilizadas, intensidade e canais de cores. A função implementada é descrita na figura 21, efetuando o mesmo processo descrito na implementação em Matlab.

```
void piramide_gaussiana(int nivel)
{
 p_I[0] = I;
 p_R[0] = R;
 p_G[0] = G;
 p_B[0] = B;
 p_Y[0] = Y;

 for (int i = 0;i < nivel-1;i++)
 {
  p_I[i].LowPass5(p_I[i]);
  p_R[i].LowPass5(p_R[i]);
  p_G[i].LowPass5(p_G[i]);
  p_B[i].LowPass5(p_B[i]);
  p_Y[i].LowPass5(p_Y[i]);

  p_I[i+1].Scale(p_I[i],p_I[i].GetWidth()/2,p_I[i].GetHeight()/2,BILINEAR);
  p_R[i+1].Scale(p_R[i],p_R[i].GetWidth()/2,p_R[i].GetHeight()/2,BILINEAR);
  p_G[i+1].Scale(p_G[i],p_G[i].GetWidth()/2,p_G[i].GetHeight()/2,BILINEAR);
  p_B[i+1].Scale(p_B[i],p_B[i].GetWidth()/2,p_B[i].GetHeight()/2,BILINEAR);
  p_Y[i+1].Scale(p_Y[i],p_Y[i].GetWidth()/2,p_Y[i].GetHeight()/2,BILINEAR);
 }
}
```

Figura 21: Construção das pirâmides gaussianas em C++
Fonte: Autoria própria.

Na arquitetura CUDA, o código de programação para a construção das pirâmides gaussianas foi implementado diretamente na estrutura do programa, sem a utilização de funções. Foi definida uma grade de blocos com dimensões 32 x 32 e com blocos de *threads* com dimensões 20 x 15, totalizando o resolução definida da imagem de entrada, 640 x 480 *pixels*. Porém, o código não foi otimizado para a alocação das *threads*, acionando a totalidade das *threads* para todas as dimensões da pirâmide. O código na arquitetura CUDA é mostrado na figura 22.

```
cudaMemcpy(gpu_I_pyramid[0],I, (640*480) *sizeof(float), cudaMemcpyDeviceToDevice);

dim3 block1(20,15);
dim3 grid1(32,32);

w = 640;
h = 480;
int w1 = 0;
int h1 = 0;
for (int i=1;i<9;i++)
{
 w1 = (int)ceil(w/2);
 h1 = (int)ceil(h/2);

 filtroX <<< grid1, block1 >>> (gpu_I_pyramid[i-1],w,h);
 filtroY <<< grid1, block1 >>> (gpu_I_pyramid[i-1],w,h);
 interpola <<< grid1, block1 >>> (gpu_I_pyramid[i-1],w,h,gpu_I_pyramid[i],w1,h1);
 w = w1;
 h = h1;
 cudaThreadSynchronize();
}
```

Figura 22: Construção das pirâmides gaussianas em CUDA
Fonte: Autoria própria.

5 RESULTADOS E CONCLUSÃO

5.1 ASPECTOS TÉCNICOS

A proposta do projeto como um estudo de uma nova tecnologia, a arquitetura CUDA, e sua integração a um método de visão computacional, o mapa de saliências, foi atingida parcialmente, não tendo sido possível implementar o mapa de conspicuidade de orientação na arquitetura CUDA, por limitações de tempo.

A pesquisa do modelo de mapa de saliências, assim como sua implementação, permitiu a visualização da amplitude da área de visão computacional, contribuindo para o surgimento de ideias para trabalhos futuros. Em contraponto, foi possível perceber a dificuldade de implementação dos modelos e técnicas existentes, como a filtragem linear e o próprio mapa de saliências nas arquiteturas propostas.

O algoritmo do mapa de saliências foi primeiramente implementado no *software* Matlab, para estudo, análise e consistência do método. Com esse desenvolvimento, percebeu-se que a descrição do método efetuado por Itti em seu trabalho contém um erro em uma fórmula (ITTI, 2000). Esse erro ocorre quando se calcula o mapa de rivalidade entre cores, invertendo os canais de cores vermelho e azul no processo de diferença entre escalas.

Na implementação do algoritmo em C++, novos conceitos foram estudados, como a implementação de matrizes de imagens utilizadas na implementação do código para a geração dos mapas de saliências.

Comparando os resultados obtidos, utilizando-se de uma placa gráfica Nvidia 9400M e sistema operacional Linux, com imagens estáticas, a implementação em Matlab e em linguagem C++ tiveram o mesmo tempo de execução, 2,4 segundos, enquanto a implementação em CUDA teve tempo de execução em 71,16 milisegundos. Considerando que o mapa de orientação corresponde a aproximadamente 50% do tempo de processamento total (COSTA, 2011), temos um tempo de processamento aproximado de 142,32 milisegundos. É importante salientar que o algoritmo não foi otimizado em nenhum momento e por esse motivo, os resultados obtidos podem ser melhorados.

5.2 APRENDIZADO

Com o desenvolvimento do projeto, novos conhecimentos foram adquiridos e os conceitos aprendidos durante o curso foram ampliados.

O desenvolvimento do projeto também possibilitou verificar a grande importância da matéria de Processamento de Imagens, sendo seus conteúdos aplicados na criação das funções auxiliares para a correta implementação do código e para facilitar o entendimento do método estudado.

A arquitetura CUDA proporcionou uma quebra de paradigma na área de programação. A utilização de programação paralela para aumentar a velocidade de processamento demonstrou-se muito eficiente, além de ser uma tecnologia promissora. No desenvolvimento do código, porém, muitas dificuldades foram encontradas. A arquitetura CUDA foi desenvolvida sendo uma extensão da linguagem de programação C, não possuindo todas as funcionalidades de uma linguagem de programação orientada a objetos, como por exemplo a criação de métodos e classes. A dificuldade de entendimento das bibliotecas disponíveis, como a FFT, também ocasionou problemas no desenvolvimento. Porém, a maior dificuldade na implementação foi o entendimento da estrutura do CUDA, por não ter sido estudado formalmente durante o curso. Por essas dificuldades, diferentemente do proposto no projeto, o mapa de saliências não foi implementado por completo na arquitetura CUDA, faltando implementar o mapa de conspicuidade de orientação.

O desenvolvimento do projeto também proporcionou a utilização de conceitos adquiridos durante o curso, estágio e iniciação científica, na área de programação.

5.3 TRABALHOS FUTUROS

Como proposta para trabalhos futuros propõe-se a criação de uma biblioteca de processamento de imagens utilizando-se da arquitetura CUDA e a correção das implementações em Matlab, C++ e CUDA, com a finalidade de se obter exatamente os mesmos resultados do mapa de saliências em todas as plataformas.

Também é proposto, para o desenvolvimento e integração de modelos de atenção visual, um novo modelo integrando o método do mapa de saliências com o modelo log-polar, buscando a simulação do comportamento do sistema visual humano. O

mapeamento log-polar simula a conversão das imagens capturadas pela retina para o córtex dos seres humanos, antes de serem analisadas pelo cérebro (HEINEN, 2008).

REFERÊNCIAS

BALLARD, Dana H. Generalizing the Hough transform to detect arbitrary shapes. **Pattern Recognition**, v. 13, p. 111–122, 1981. Disponível em: <http://www.cs.utexas.edu/~dana/HoughT.pdf>. Acesso em: 06 set. 2010.

BAY, Herbert et al. SURF: Speeded Up Robust Features. **Computer Vision and Image Understanding**, v. 110, p. 346–359, 2008. Disponível em: <http://www.vision.ee.ethz.ch/~surf/papers.html>. Acesso em: 09 set. 2010.

BRADSKI, Gary R.; KAEHLER, Adrian. **Learning OpenCV** - computer vision with the OpenCV library. California, US: O' Reilly, 2008.

COSTA, Pedro Miguel Caetano França. **Implementação de Algoritmos de Saliência em Tempo-Real numa GPU.** 2011. 75 f. Dissertação (Mestrado em Engenharia Eletrotécnica e Computação) — Universidade de Coimbra, Coimbra, Portugal, 2011. Disponível em: <http://paloma.isr.uc.pt/mrl/people/peopleinformation.php?ID=113>. Acesso em: 15 ago. 2011.

CUDA. **NVIDIA CUDA C Programming Guide Version 3.2**. NVIDIA, 2010. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf>. Acesso em: 25 jan. 2011.

FACON, Jacques. **Apostila de Processamento e Análise de Imagens**. Curitiba: Pontifícia Universidade Católica do Paraná. Curso de Mestrado em Informática Aplicada, 2005.

HALFHILL, Tom R. Parallel Processing With CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading. **Microprocessor Report**, v. 2, p. 1–8, 2008. Disponível em: <http://www.nvidia.com/docs/IO/47906/220401_Reprint.pdf>. Acesso em: 16 ago. 2011.

HEINEN, Milton Roberto; ENGEL, Paulo Martins. Modelo de atenção visual para robôs inteligentes. **Revista do CCEI,** Bagé, v. 12, p. 116–127, 2008. Disponível em: <http://www.urcamp.tche.br/ccei/portal/images/Revista_CCEI/numero22_p116_p183.pdf>. Acesso em: 31 ago. 2011.

ITTI, Laurent. **Models of Bottom-Up and Top-Down Visual Attention.** 2000. 216 f. Tese (Doutorado) — California Institute of Technology, Pasadena, California, 2000. Disponível em: <http://iLab.usc.edu/publications/doc/Itti00phd.pdf>. Acesso em: 08 set. 2010.

ITTI, Laurent; KOCH, Christof; NIEBUR, Ernst. A Model of Saliency-Based Visual Attention for Rapid Scene Analysis. **Pattern Analysis and Machine Intelligence**, v. 20, p. 1254–1259, 1998. Disponível em: <http://iLab.usc.edu/publications/doc/Itti_etal98pami.pdf>. Acesso em: 07 set. 2010.

KIRK, David B.; HWU, Wen-mei W. **Programming Massively Parallel Processors:** A Hands-on Approach. California, US: Morgan Kaufmann, 2010.

LOWE, David G. Object Recognition from Local Scale-Invariant Features. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER VISION, Corfu, Greece, 1999. **Proceedings of the Seventh IEEE International Conference**, p. 1150–1157, 1999. Disponível em: <http://www.cs.ubc.ca/~lowe/papers/iccv99.pdf>. Acesso em: 09 set. 2010.

NAYAR, Shree K. Shape from focus system. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 1992. **Proceedings of the CVPR 1992**, p. 302–308, 1992. Disponível em: <http://www1.cs.columbia.edu/CAVE/publications/pdfs/Nayar_CVPR92.pdf>. Acesso em: 09 set. 2010.

NEWCOMBE, Richard A. **A Simple Quantitative Comparison of a Few Techniques Used in the Computation of Gradient Based Optical Flow**. CC466 – Visual Guidance for Robotics, Colchester, UK: Universidade de Essex, 2004.

NVIDIA. Disponível em: <http://www.nvidia.com>. Acesso em: 06 set. 2010.

TRUCCO, Emanuele; VERRI, Alessandro. **Introductory Techniques for 3-D Computer Vision**. New Jersey, US: Prentice Hall PTR Upper Saddle River, 1998.

WALTHER, Dirk; KOCH, Christof. Modeling attention to salient proto-objects. **Neural Networks**, v. 19, p. 1395–1407, 2006. Disponível em: <http://papers.klab.caltech.edu/220/1/sdarticle.pdf>. Acesso em: 16 ago. 2011.

XU, Tingting et al. A High-Speed Multi-GPU Implementation of Bottom-up Attention Using CUDA. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2009, Kobe, Japão. **In: Proceedings of ICRA 2009**, 2009. Disponível em: <http://www.lsr.ei.tum.de/fileadmin/publications/Xu_2009_ICRA_GPU.pdf>. Acesso em: 22 ago. 2011.