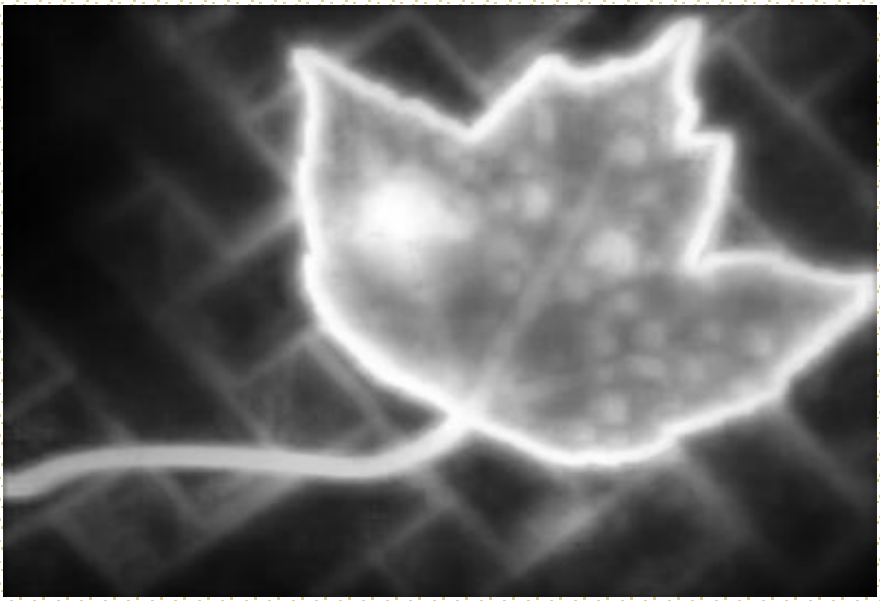

Pedro Miguel Caetano França Costa

IMPLEMENTAÇÃO DE ALGORITMOS DE SALIÊNCIA EM TEMPO-REAL NUMA GPU

Dissertação de Mestrado

11 de Julho de 2011

· U C ·

UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
Departamento de Engenharia Electrotécnica e de Computadores

Mestrado Integrado em Engenharia Electrotécnica e de Computadores
2010-2011

DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO DE ALGORITMOS DE SALIÊNCIA EM TEMPO-REAL NUMA GPU

Pedro Miguel Caetano França Costa

Júri:
Presidente: Prof. Doutor Rui Paulo Pinto da Rocha
Orientador: Prof. Doutor Jorge Nuno de Almeida e Sousa Almada Lobo
Co-Orientador: Prof. Doutor João Filipe de Castro Cardoso Ferreira
Vogal: Prof. Doutor Gabriel Falcão Paiva Fernandes

Coimbra, 11 de Julho de 2011

UNIVERSIDADE DE COIMBRA

Agradecimentos

Em primeiro lugar, quero agradecer aos meus orientadores, o Prof. Doutor Jorge Nuno de Almeida e Sousa Almada Lobo e o Prof. Doutor João Filipe de Castro Cardoso Ferreira, por me terem oferecido a oportunidade de realizar uma tese numa àrea do meu interesse e por me terem acompanhado ao longo do período em que estive a realizá-la.
Gostaria, por outro lado, de agradecer a todos os meus familiares que me apoiaram durante o período em que realizei a dissertação, em particular, aos meus pais, ao meu irmão e à minha namorada por me terem apoiado e ajudado em muitas ocasiões ao longo da mesma.
Gostaria, também, de agradecer a todos os meus amigos, os quais estiveram igualmente sempre presentes ao longo da realização da dissertação.
Por último, gostaria de agradecer a todas as pessoas que fazem parte do Instituto de Sistemas e Robótica, no qual eu realizei a tese, pelo apoio que me deram sempre que era necessário.

Resumo

Neste trabalho apresenta-se uma implementação em GPU (Graphics Processing Unit) do modelo de saliência baseado na atenção visual. O campo de visão por computador tornou-se numa parte importante da intervenção da robótica na sociedade de hoje em dia, tendo aplicações cruciais, nomeadamente na medicina, na vigilância e segurança militar e no domínio da vigilância. A atenção visual é um mecanismo perceptual que, descrito de forma simples, aplica os recursos de processamento disponíveis, limitados por natureza, na análise selectiva das partes mais relevantes do fluxo de dados sensoriais de entrada, de forma a usá-los mais eficiente e criteriosamente, e tem sido alvo de extensa investigação na área de visão por computador nos últimos anos. Neste contexto, a saliência é a qualidade subjectiva e distintiva de percepção visual que faz com que certos aspectos do mundo se destaquem dos demais e capturem a atenção do ser humano. O processo de saliência presta-se a SIMD (Single Instruction Multiple Data), onde a mesma instrução é aplicada a múltiplos dados. SIMD implica processamento em paralelo, para o qual as GPUs são adequadas porque são extremamente poderosas, devido à sua arquitectura paralela e à sua eficiência, permitindo a manipulação de grandes blocos de dados. Na última década as GPUs têm sido desenvolvidas de forma a permitir utilizações mais genéricas, onde o hardware gráfico é usado para cálculos além daqueles de natureza estritamente gráfica e em que uma das ferramentas para GPGPU (General-Purpose Computing on Graphics Processing Units) é o CUDA (Compute Unified Device Architecture). A solução implementada segue o algoritmo de saliência de Itti et al. e no decurso deste trabalho usa-se uma NVIDIA 9800GTX+ com 128 núcleos CUDA de forma a obter um desempenho de 25 imagens a cores processadas por segundo, com uma resolução de 640×480 píxeis. Os resultados deste trabalho demonstram que esta implementação é 27,5x mais rápida do que a sua homóloga em CPU, usando as funcionalidades da biblioteca OpenCV (Open Source Computer Vision) que já recorre a várias optimizações. Face a outras soluções publicadas, neste trabalho conseguiu-se um desempenho superior, ao contrário de outros está disponível em *open source* e disponibilizou-se um guia de utilizador com a finalidade de auxiliar a comunidade científica no uso deste sistema. Os resultados obtidos e a facilidade

de integração permitem, como proposto para trabalho futuro, a integração no IMPEP (Integrated Multimodal Perception Experimental Platform) e em num Ar.Drone da Parrot.

- Palavras-chave: GPU; paralelização; saliência; tempo-real

Abstract

In this work an implementationin a GPU (Graphics Processing Unit) of the salience model based on the visual attention is presented. Computer vision has become an important part in robotic intervention of nowadays society, with key usage on medicine, on military security and surveillance and on surveillance domain. Briefly, the visual attention is a perceptual mechanism that applies the available processing resources, innerantte limited by, on the selective analysis of the most relevant parts of the inbound sensory data flow, in order to use them more efficiently and wisely. It has been thoroughly studied on the computer vision field during the last few years. In this context the salience is the subjective and distinctive quality of visual perception that makes certain aspects of the world to standout from the others and capture the human being attention. The salience process may be executed following the SIMD method where the same instruction is applied to multiple data. SIMD (Single Instruction Multiple Data) implies parallel processing for which the GPUs are suitable since they are extremely powerful due to their parallel architecture and to their efficiency, allowing the manipulation of large data blocks. Over the last decade GPUs have been developed to allow more generic applications, where graphical hardware is used to computing operations beyond the ones of graphical nature, and CUDA is one of GPGPU (General-Purpose Computing on Graphics Processing Units) tools. The implemented solution follows the Itti et al. model and during this work uses a NVIDIA 9800GTX+ with 128 CUDA (Compute Unified Device Architecture) cores so that it achieves a performance of 25 processed colour images per second, with a 640×480 pixels resolution. the results of this work show that this implementation is 27.5 times faster than its homologous CPU, using the OpenCV (Open Source Computer Vision) library features, which has optimizations. Compared with other published solutions, this work is available in *open source* and offer a user guide to assist the scientific community in using this system. The results and the ease of integration allow, as proposed for future work, integration in the IMPEP (Integrated Multimodal Perception Experimental Platform) and in a Parrot AR.Drone.

- Keywords: GPU; parallelization; saliency; real-time

Conteúdo

Lista de Tabelas

Lista de Figuras

Abreviaturas

- API — Application Programming Interface
- B — Blue
- BY — Blue/Yellow — Rivalidade entre a cor azul e a cor amarela
- CPU — Central Processing Unit
- CUBLAS — Compute Unified Basic Linear Algebra Subprograms
- CUDA — Compute Unified Device Architecture
- CUFFT — Compute Unified Fast Fourier Transform
- FFT — Fast Fourier Transform
- FPGA — Field-Programmable Gate Array
- G — Green
- GPGPU — General-Purpose Computing on Graphics Processing Units
- GPU — Graphics Processing Unit
- IFFT — Inverse Fast Fourier Transform
- IMPEP — Integrated Multimodal Perception Experimental Platform
- OpenCL — Open Computing Language
- OpenCV — Open Source Computer Vision
- POP — Perception on Purpose

- RAM — Random-Access Memory
- R — Red
- RG — Red/Green — Rivalidade entre a cor vermelha e a cor verde
- RGB — Red, Green, Blue
- SIMD — Single Instruction Multiple Data
- WiFi — Wireless Fidelity

Capítulo 1

Introdução

1.1 Motivação

Nos últimos anos, o desenvolvimento de robôs que interagem com seres humanos tem ganho uma prevalência importante em Robótica. Há uma procura cada vez maior de soluções robóticas para problemas como as relações públicas primárias na recepção de visitantes em museus ou outras repartições públicas, ou para o desenvolvimento de companheiros artificiais ("robôs de estimação"). Devido às necessidade resultantes da construção de "robôs sociais" (e sociáveis) nessas aplicações — por outras palavras, de agentes robóticos que interajam convicentemente com seres humanos, tem-se assistido a um crescente interesse no desenvolvimento de sistemas cognitivos artificiais que permitam a percepção do ambiente circudante, de uma maneira consistente com a forma como os humanos percepcionam o mundo que os rodeia.

Uma componente chave de muitos destes sistemas perceptuais artificiais reside nos modelos computacionais de atenção visual subjacentes. Apesar destes sistemas serem aproximações rudimentares das funcionalidades fornecidas pelo sistema visual humano, ainda assim ofereceram um enorme avanço em termos da capacidade com que dotam sistemas perceptuais artificiais para lidar eficientemente, e de forma automática, com a quantidade enorme de informação sensorial que lhes é apresentada, sem necessidade de recorrer a processos cognitivos mais complexos.

Em 1998, L.Itti, C.Koch e E.Niebur [18] propuseram uma solução baseada no sistema visual dos primatas, modelando processos que evoluem desde a retina até às células do córtex visual, assente no conceito de **saliência**. De forma simples, este conceito corresponde à característica de uma entidade sensorial, como por exemplo, a sensibilização de um fotorreceptor, traduzido na intensidade luminosa de um píxel, se destacar

face aos seus vizinhos.

O modelo acima descrito, centrado neste conceito, permite a análise rápida de cenas perceptuais e possui potencial para que seja aplicado o princípio de "instrução única para dados múltiplos" (SIMD - *Single Instruction Multiple Data*) de forma a melhorar exponencialmente o seu desempenho através de **programação paralela**, utilizando, para tal, os novos meios que surgiram nos últimos anos, nomeadamente as FPGA (*Field-programmable Gate Arrays* — lógica programável) e GPUs (*Graphics Processing Units* — placas gráficas).

As GPUs são unidades de processamento, originalmente concebidas para manipulação gráfica mas correntemente acessíveis para qualquer tipo aplicação (GPGPU — *General Purpose GPU programming*), capazes de manipular dados e fazer cálculos paralelamente com uma velocidade superior às CPUs, particularmente ao nível do cálculo matricial e vectorial. Os algoritmos de processamento de imagem em tempo-real requerem que esse processamento seja muito rápido, tendo sido esse facto determinante na escolha pelas GPUs para implementar o modelo de saliência.

Nestes dispositivos é possível obter um grau de paralelização maior que em CPU, pois consegue-se obter uma maior paralelização dos estágios computacionais. Vários esforços para se adaptarem algoritmos de visão por computador às unidades de GPU mostram que é possível, por essa via, obter uma melhoria significativa ao nível de desempenho, como demonstrado por [12].

A fim de permitir aos investigadores e aos projectistas a utilização plena das capacidades das GPUs, várias ferramentas estão disponíveis, designadamente OpenCL [8] e CUDA (*NVIDIA Compute Unified Device Architecture*) [3], este último disponível desde a série G8X de placas gráficas da NVIDIA. Para este trabalho foi escolhido o CUDA como ferramenta para implementar o algoritmo, pois há mais exemplos disponíveis, é mais optimizado para o hardware NVIDIA e melhor documentado.

1.2 Objectivos

O principal objectivo da presente dissertação é a implementação do modelo de saliência baseado na atenção visual proposto por Itti et al. [18], tendo em conta as características da imagem, da intensidade, da rivalidade entre cores e da orientação de forma a que se obtenha um resultado em tempo-real, ou seja, de pelo menos 24 imagens por segundo.

A implementação servirá para disponibilizar como um pacote de *software* de forma gratuita para toda a comunidade científica (*open source*), devidamente documentado através de um guia de utilizador (ver

apêndice A). Esta implementação deve incluir um demonstrador "stand-alone" concebido de forma a ser facilmente integrado em software de terceiros, como por exemplo, através de uma simples ligação de ficheiros objecto, ou a inclusão de ficheiros objecto e inclusão de ficheiros cabeçalho com os protótipos das funções utilizáveis. Deve o pacote de *software* ser desenvolvido de maneira a serem facilmente adicionadas extensões por terceiros.

1.3 Estado da arte

Em 1985 C.Koch e S.Ullman [19] propuseram um modelo baseado na atenção visual onde diferentes características da imagem, como as cores, a orientação, a direcção do movimento e a disparidade contribuem para determinar o mapa de saliência final.

O modelo mais conhecido para modelização da percepção visual humana é o de [18]. Variando conforme a situação de aplicação, várias melhorias e optimizações foram propostas. No entanto, todas elas são baseadas no modelo proposto por [18].

A implementação que mais se pode comparar à que foi realizada é a proposta em [14], onde se usam os parâmetros propostos em [18]. Para uma imagem de entrada cuja resolução é de 640×480 pixéis, utilizando um processador Intel Xeon, o tempo de cálculo do mapa de saliência final é de 51,34 milissegundos (ms), considerando as características da intensidade, das cores e da orientação.

Numa implementação proposta por [15] obtém-se um resultado em tempo-real, tendo em conta as características de intensidade, as dos cantos e as das cores. Para se obter o resultado em tempo-real é implementada uma pirâmide Gaussiana de apenas 5 camadas e com uma imagem de entrada cuja resolução é de 160×120 píxeis.

Uma outra implementação distribuída foi proposta por [16], possuindo este sistema um *cluster* de 8 computadores, 5 deles equipados com 2 processadores Intel Xeon a 2,2GHz, 2 equipados com 2 processadores a 2,8GHz Intel Xeon e o último equipado com 2 processadores Opteron a 2.4GHz. Neste sistema todos os computadores estão ligados em rede a um *switch* com uma capacidade de transferência de 1 Gigabit, sendo obtidas 30 imagens por segundo para uma imagem de entrada cuja resolução é de 320×240 píxeis, tendo em conta as características da intensidade, das cores, da orientação, do movimento e da disparidade.

Uma implementação feita em GPU, mas que não usa CUDA, foi a proposta por [17], sendo a implementação feita numa NVIDIA 6600GT e obtendo-se um tempo de computação de 34ms para uma imagem de entrada cuja resolução é de 512×512 píxeis. Para se obterem os mapas de orientação são usados filtros de

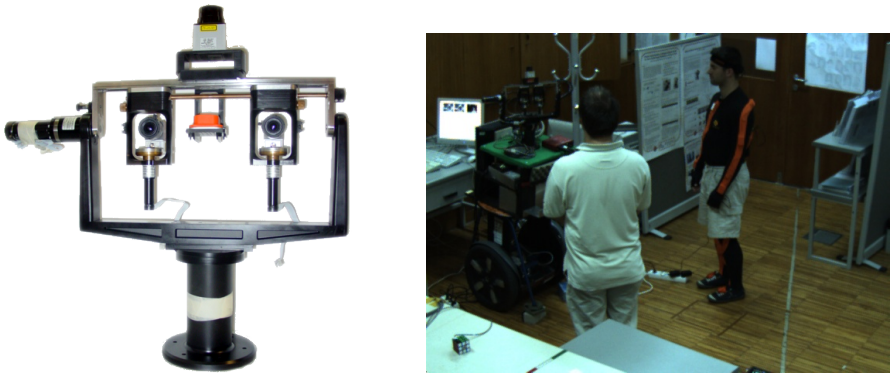

Figura 1.1: Exemplo de aplicação em robôs sociais. O sistema robótico IMPEP (*Integrated Multimodal Perception Experimental Platform*, à esquerda) desenvolvido no Instituto de Sistemas e Robótica no âmbito do projecto europeu POP (*Perception on Purpose*; projecto FP6-IST-2004-027268), observa dois interlocutores num contexto de interacção homem-robô (à direita) [1].

Sobel em vez de filtros de Gabor.

Finalmente uma implementação feita em GPU, usando CUDA, foi proposta por [13]. Baseia-se no modelo proposto por [18] e dali obtém-se um resultado final de 42,03ms para uma imagem de entrada cuja resolução é de 512×512 píxeis, usando uma NVIDIA GTX480.

A implementação proposta neste trabalho tem como principal característica, em relação às anteriormente apresentadas, a disponibilização de uma versão *open source* para que a comunidade científica tenha a liberdade de a utilizar e adaptá-la às suas necessidades e apesar de ter sido testada numa GPU inferior (NVIDIA 9800GTX+) em relação à GPU da solução proposta por [13], obtêm-se melhores resultados.

1.4 Contribuição

O presente sistema, a executar em tempo-real, que tem como principal unidade de processamento uma GPU, a qual, por via dos desenvolvimentos mais recentes, consome cada vez menos energia e, paralelamente, tem vindo a aumentar a capacidade de manipulação de dados. Estas evoluções, aliadas a um custo cada vez mais reduzido, permitem que a implementação tenha aplicações práticas mais generalizadas.

O sistema implementado possibilita, através da atribuição de pesos diferentes a cada característica visual básica (contraste, cor, intensidade, etc.) utilizada como componente na computação global da saliência, a adaptação do modelo a objectivos específicos em cada situação prática.

Esta implementação pode ser integrada num sistema de maior dimensão, como o sistema robótico IMPEP desenvolvido no Instituto de Sistemas e Robótica (figura 1.1), de forma a facilitar o controlo do foco da atenção do robô. O sistema pode ter várias aplicações sociais, tendo as mais importantes sido já detalhadas

e descritas na secção 1.1.

Alternativamente, usando uma unidade com capacidade de voo devidamente dotada de uma câmara, destacam-se as suas aplicações no apoio a sistemas mais complexos: detecção de zonas de incêndio (figura 1.2), controlo/vigilância de multidões (figura 1.3) e casos de catástrofes naturais — como a procura de corpos.

1.5 Estrutura da dissertação

No capítulo 2 é apresentado o modelo de saliência proposto por Itti et al. [18], na secção 2.1, enquanto a linguagem de programação de unidades de processamento gráfico da NVIDIA CUDA é apresentada na secção 2.2. No capítulo 3 explica-se como foi efectuada a implementação do modelo de saliência proposto por [18] recorrendo à linguagem CUDA, sendo mostrado o fluxo de dados e como foi efectuada a paralelização dos cálculos na unidade de processamento gráfico (recorrendo à linguagem de programação CUDA). No capítulo 4 demonstram-se os resultados temporais e visuais da implementação do modelo de saliência em CUDA e, por fim, no capítulo 5 apresentam-se as conclusões e propõe-se trabalho futuro.

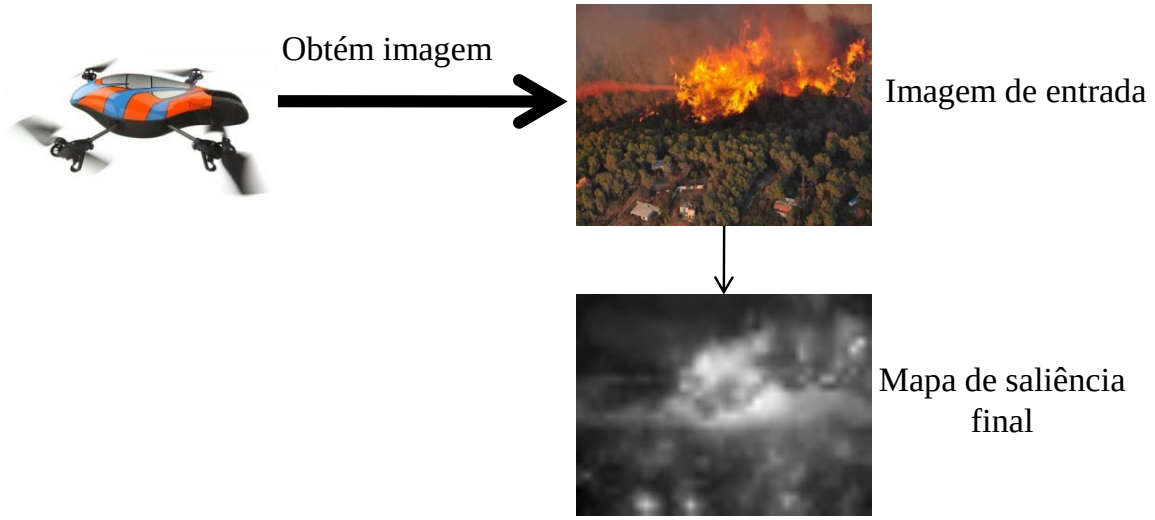

Figura 1.2: Detecção de zonas de incêndio através da patrulha de uma unidade com capacidade de voo devidamente dotada de uma câmara de vídeo (exemplo: AR.Drone da Parrot [23]).

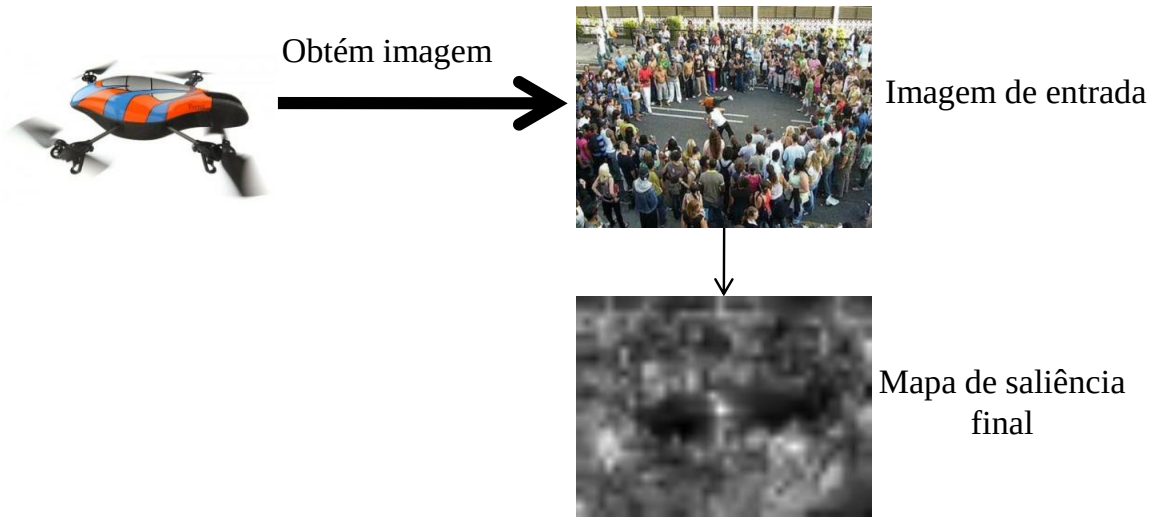

Figura 1.3: Vigilância de multidões. Unidade aérea dotada de uma câmara que patrulha uma zona com uma multidão e dirige-se para o foco da atenção autonomamente.

Capítulo 2

Teoria de suporte

2.1 Modelo de Saliência

O ser humano fixa a atenção em determinadas regiões da imagem devido a estímulos causados pelos contrastes entre as características da imagem, sendo responsáveis por guiar o mecanismo biológico de atenção visual. Nesta secção explica-se o modelo proposto por [18] em que a atenção visual é guiada por características da imagem, como a intensidade, a cor e a orientação. São computados mapas onde se representam estas características e existem mecanismos de inibição lateral em cada mapa, em que os locais que diferem significativamente da sua vizinhança são extraídos. O mapa de saliência final é obtido através da informação de conspicuidade ou interesse dos locais da imagem. A conspicuidade representa a saliência para cada tipo de característica.

No início apresenta-se uma imagem a cores, cuja resolução é de 640×480 píxeis, sendo esta imagem decomposta em vários mapas de características baseados em diferentes aspectos da visão humana.

A imagem de entrada é sub-amostrada. São criadas 9 escalas formando uma pirâmide Gaussiana onde se convolve linearmente usando um filtro de Gauss e se faz uma decimação por um factor de 2. A convolução é feita na direcção de x seguida de uma decimação na direcção de x, repetindo-se este processo para a direcção de y.

Obtêm-se os níveis de escala de 0 a 8 ($\sigma = [0, \cdots, 8]$) da pirâmide Gaussiana por repetição do processo de sub-amostragem. Cada nível da pirâmide tem uma resolução em termos de colunas e linhas reduzido por um factor de 2 em relação ao nível de escala anterior, sendo que o oitavo nível da pirâmide tem uma resolução 1/256 vezes menor que a imagem de entrada.

Sendo r, g e b os valores da imagem a cores, o mapa de intensidade é calculado através da seguinte equação:

$$M_I = \frac{r+g+b}{3}; \tag{2.1}$$

Obtêm-se todos os níveis da pirâmide Gaussiana, com $M_I = \sigma[0,\cdots,8]$.

Para se obterem os mapas da rivalidade entre as cores utilizam-se dois mapas, sendo que esta rivalidade existe para os seguintes pares de cores: vermelho/verde, verde/vermelho, azul/amarelo e amarelo/azul, que são representados no córtex primário dos humanos.

Através dos valores de vermelho, azul e verde obtidos da imagem de entrada, vão ser criados dois mapas:

$$M_{RG} = r - \frac{(g+b)}{2} - (g - \frac{r+b}{2}); \tag{2.2}$$

$$M_{BY} = b - \frac{(r+g)}{2} - (\frac{r+b}{2} - \frac{|r-g|}{2}); \tag{2.3}$$

Os mapas de orientação são obtidos através da convolução dos níveis da pirâmide de intensidade com filtros de Gabor, que aproximam os neurónios de orientação selectiva no córtex primário visual.

$$M_\theta = \|M_I(\sigma) * G_0(\theta)\| + \|M_I(\sigma) * G_{\pi/2}(\theta)\|, com \;\; \theta \in \{0^o, 45^o, 90^o, 135^o\}; \tag{2.4}$$

Os filtros de Gabor são obtidos através da seguinte equação:

$$G_\psi(x,y,\theta) = e^{-\frac{x'^2+\gamma^2 y'^2}{2\delta^2}} \cos(2\pi\frac{x'}{\lambda} + \psi), com \;\; \psi = \{0, \frac{\pi}{2}\}; \tag{2.5}$$

Com uma proporção γ, desvio padrão δ, comprimento de onda λ, fase ψ e coordenadas (x', y') transformadas com a respectiva orientação:

$$x' = x\cos(\theta) + y\sin(\theta); \tag{2.6}$$

$$y' = -x\sin(\theta) + y\cos(\theta); \tag{2.7}$$

Para se obterem os mapas de características são feitas diferenças de centro-vizinhança.

Tipicamente os neurónios visuais são mais sensíveis a uma pequena região do espaço visual (centro), enquanto os estímulos apresentados numa região concêntrica com o centro (vizinhança) inibem a resposta neuronal. Desta forma, as descontinuidades espaciais sensíveis ao local permitem que se possam obter lo-

calizações que sobressaem em relação à vizinhança, sendo um princípio computacional da retina, do núcleo geniculado lateral e do córtex primário da visão. As diferenças entre o centro e a vizinhança são implementadas como uma diferença entre as escalas finas e as escalas grossas. O centro é o píxel a uma escala $c \in \{2,3,4\}$ e a vizinhança é o píxel correspondente a uma escala $s = c + \delta$, com $\delta = \{3,4\}$. Os mapas de intensidade estão ligados ao contraste de intensidade, que nos mamíferos é detectado pelos neurónios sensíveis quer em centros escuros com uma vizinhança brilhante, quer em centros brilhantes com uma vizinhança escura. Estes dois tipos de sensibilidades são considerados para se obterem os seis mapas de características da intensidade:

$$I(c,s) = \mathcal{N}(|I(c) \ominus I(s)|); \tag{2.8}$$

Os mapas de cores são construídos tendo em conta o sistema de rivalidade entre cores. No centro os neurónios são excitados por uma cor (exemplo: vermelho) e inibidos por outra cor (exemplo: verde), sendo o inverso verdadeiro para a vizinhança. Através desta propriedade são construídos os mapas que se seguem, onde RG(c,s) representa a rivalidade entre vermelho/verde e verde/vermelho, e BY(c,s) representa a rivalidade entre azul/amarelo e amarelo/azul:

$$RG(c,s) = \mathcal{N}(|RG(c) \ominus RG(s)|); \tag{2.9}$$

$$BY(c,s) = \mathcal{N}(|BY(c) \ominus BY(s)|); \tag{2.10}$$

Os mapas de orientação representam o contraste da orientação local entre o centro e as escalas da vizinhança:

$$O(c,s,\theta) = \mathcal{N}(|O(c) \ominus O(s)|); \tag{2.11}$$

No total obtêm-se 42 mapas de características, sendo 6 para a intensidade, 12 para as cores e 24 para a orientação.

O operador $\mathcal{N}(\cdot)$, apresentado nas equações anteriores, é um operador iterativo não linear que simula a competição local entre os locais da vizinhança salientes. Os mapas onde há um menor número de fortes picos de actividade são promovidos em relação aos mapas onde existem numerosos picos de actividade comparáveis.

$\mathcal{N}(\cdot)$ consiste numa convolução entre os mapas com um filtro de diferenças de Gauss (equação 2.12), sendo este operador aplicado ao mapa resultante das diferenças centro-vizinhança. As áreas homogéneas do mapa são ignoradas e os locais onde há mais actividade são evidenciados. Este operador vai replicar os

mecanismos de inibição cortical lateral, onde as características na vizinhança similares se inibem umas às outras por via de conexões definidas anatomicamente.

O filtro de diferenças de Gauss é criado de acordo com a seguinte equação:

$$DoG(x,y) = \frac{c_{ex}^2}{2\pi\sigma_{ex}^2} e^{-\frac{x^2+y^2}{2\sigma_{ex}^2}} - \frac{c_{ini}^2}{2\pi\sigma_{ini}^2} e^{-\frac{x^2+y^2}{2\sigma_{ini}^2}}; \tag{2.12}$$

A cada iteração do processo de normalização, um dado mapa de características M é sujeito à transformação apresentada na equação 2.13.

$$M \leftarrow |M + M * DoG - C_{ini}|_{\geqslant 0}; \tag{2.13}$$

Os mapas de características, resultantes da aplicação posterior do operador $\mathcal{N}(\cdot)$, são combinados em três mapas de conspicuidade através de adições ao longo das escalas, consistindo numa redução de cada mapa à escala 4 e adição ponto-a-ponto.

Para se obter o mapa de conspicuidade para a intensidade:

$$I = \bigoplus_{c=2}^{4} \bigoplus_{s=c+3}^{c+4} (\mathcal{N}(I(c,s)); \tag{2.14}$$

Para o mapa de conspicuidade das cores:

$$C = \bigoplus_{c=2}^{4} \bigoplus_{s=c+3}^{c+4} (\mathcal{N}(RG(c,s)) + \mathcal{N}(BY(c,s))); \tag{2.15}$$

Para a orientação, os mapas intermédios, com ângulos 0º, 45º, 90º e 135º, são combinados num único mapa final:

$$O = \sum_{\theta \in \{0^o, 45^o, 90^o, 135^o\}} \bigoplus_{c=2}^{4} \bigoplus_{s=c+3}^{c+4} (\mathcal{N}(O(c,s))); \tag{2.16}$$

Com os três mapas de conspicuidade obtém-se o mapa de saliência final:

$$S = \frac{I + C + O}{3}; \tag{2.17}$$

Sendo a arquitectura geral do modelo apresentada na figura 2.1.

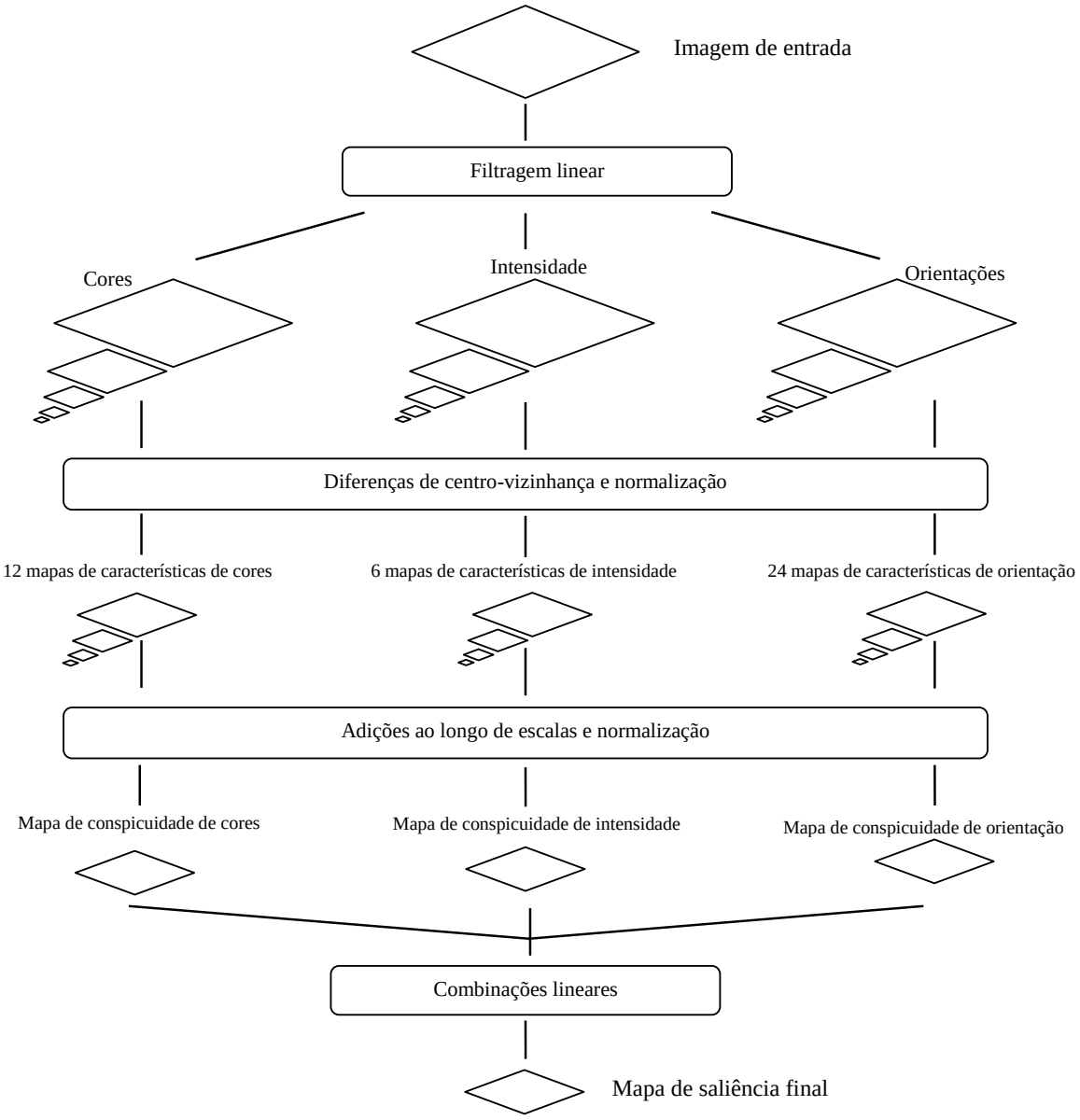

Figura 2.1: Arquitectura geral do modelo de Itti et al. [18]. À imagem de entrada é aplicado um filtro passo baixo e são criadas as pirâmides Gaussianas com escalas $\sigma = 0, \cdots, 8$. São efectuadas as diferenças de centro-vizinhança às imagens de escalas $\sigma = 2, \cdots, 8$, obtendo-se 12 mapas de características de cores, 6 mapas de características de intensidade e 24 mapas de características de orientações. Aos mapas de características fazem-se as adições ao longo das escalas, obtendo-se os 3 mapas de conspicuidade. Estes são combinados e obtém-se o mapa de saliência final.

2.2 CUDA

Como definido em [3], as unidades de processamento gráfico (GPUs) têm avançado rapidamente, passando de unidades de processamento de uma função específica para serem dispositivos altamente pro-

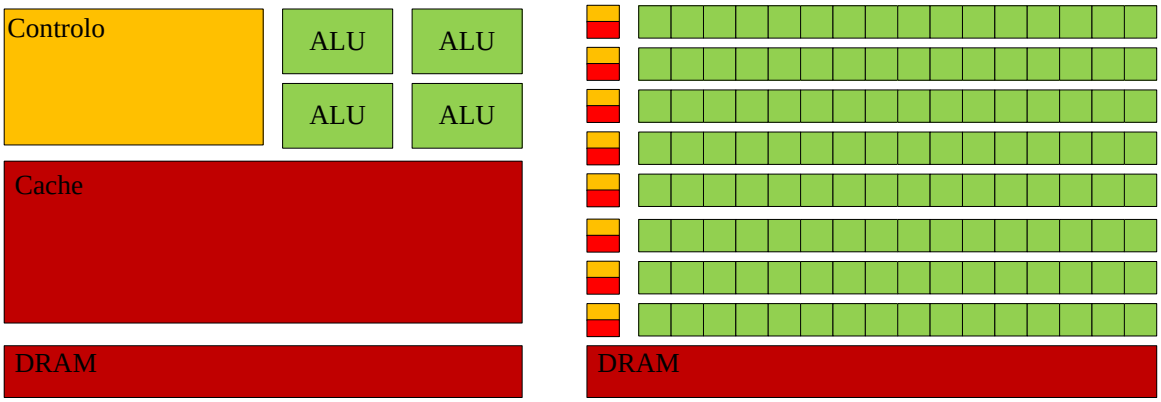

Figura 2.2: Comparação entre a aplicação dos transístores em GPUs e CPUs. As GPUs são especializadas para computação altamente paralela, pois mais transístores são dedicados ao processamento de dados, em vez de existirem transístores dedicados ao controlo de fluxo e à cache de dados como existem nas CPUs.

gramáveis e muito poderosos na computação paralela. Com a introdução por parte da NVIDIA do *Compute Unified Device Architecture* (CUDA) em 2007, que tira proveito do mecanismo de computação paralela destes dispositivos, é possível executar múltiplas operações de uma instrução a múltiplos dados (SIMD).

Nesta secção irá ser analisada esta tecnologia.

2.2.1 Arquitectura de GPUs da NVIDIA

O principal motivo do ganho de desempenho das GPUs em relação às CPUs é que aquelas dedicam-se exclusivamente ao processamento de dados, não tendo a preocupação de guardar informações em memórias cache, nem de tratar de um controlo de fluxo altamente complexo. Isto deve-se ao facto das aplicações gráficas serem paralelas, realizando a mesma operação num grande volume de dados. Desta forma, a área destinada à memória cache e ao controlo de fluxo nos processadores tradicionais é utilizada para processamento de dados nas GPUs. A figura 2.2 exibe uma comparação entre a aplicação dos transístores em GPUs e CPUs.

Esta nova arquitectura cria, gere, agenda e executa *threads* automaticamente em grupos de 32 *threads* paralelas. Quando cada multiprocessador recebe um bloco com mais de 32 *threads* para processar, quebra esse bloco em *warps*, agrupando as *threads* de acordo com o seu número de identificação.

Uma GPU NVIDIA consiste num *array* de multiprocessadores de *threads*, conforme observado na figura 2.3.

Cada multiprocessador consiste em 8 processadores escalares com 16384 registos de 32 bits de uso exclusivo (2048 registos por processador, que corresponde a 8Kb).

Cada multiprocessador possui uma memória partilhada (*shared memory*) com 16Kb de tamanho e

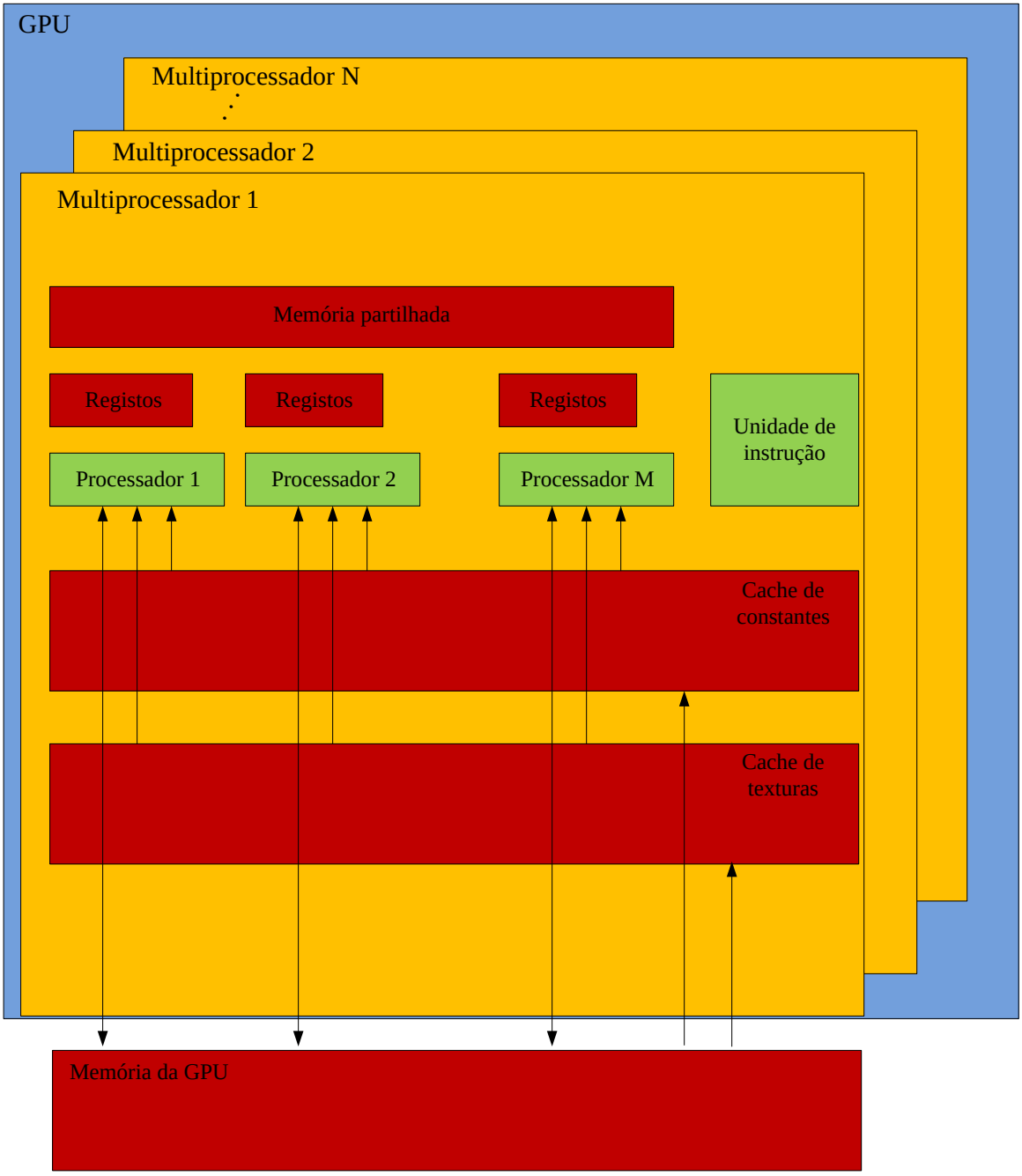

Figura 2.3: Disposição de multiprocessadores duma GPU NVIDIA. A característica de *hardware* chave é que cada multiprocessador tem 8 núcleos SIMD (única instrução, múltiplos dados). Todos os 8 núcleos executam a mesma instrução simultaneamente mas com dados diferentes.

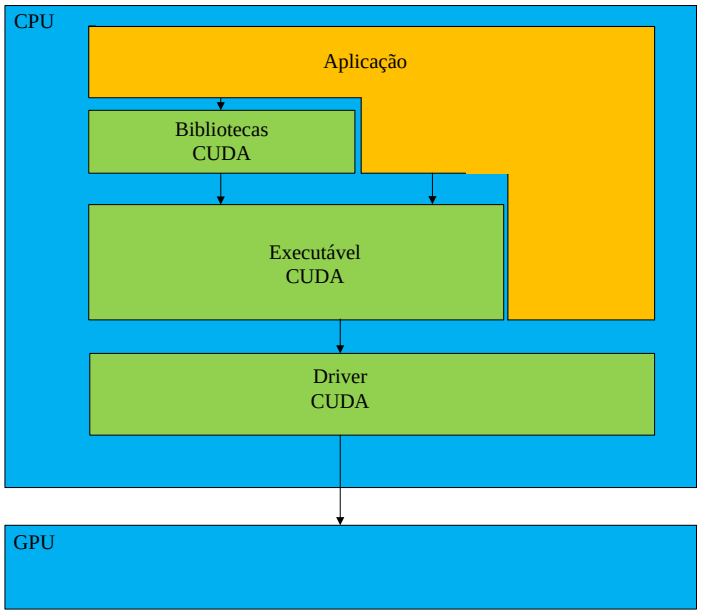

Figura 2.4: Pilha de *software* da plataforma CUDA. Esta pilha de *software* é composta pelo *driver* de acesso ao *hardware*, um componente de execução e duas bibliotecas matemáticas prontas para uso: CUBLAS (*Basic Linear Algebra Subroutines*) e CUFFT (*Fast Fourier Transform*). No topo da pilha encontra-se a API da plataforma CUDA.

acesso muito-rápido, apenas acessível pelas *threads* de um determinado bloco, além de duas memórias de leitura de acesso rápido: a memória de texturas (*texture memory*) e a memória constante (*constant memory*), com 64Kb. Na memória principal (*global memory*) da GPU residem as memórias locais de cada *thread* e a memória global da aplicação, sendo esta a memória com maior latência de acesso.

2.2.2 A API CUDA

Executar um programa escrito em CUDA requer alguns componentes de *software* e *hardware*. No que diz respeito ao *hardware*, uma extensa lista de GPUs capazes de processar CUDA pode ser encontrada em [5]. Uma vez satisfeita essa condição é necessário instalar algum software: um *driver* específico e um *toolkit* contendo um compilador e algumas ferramentas adicionais, sendo ambos fornecidos pela NVIDIA [6].

A figura 2.4 apresenta a pilha de *software* da plataforma CUDA, composta pelo *driver* de acesso ao *hardware*, um componente de execução e duas bibliotecas matemáticas prontas para uso: CUBLAS (*(Compute Unified Basic Linear Algebra Subprograms)*) e CUFFT (*Compute Unified Fast Fourier Transform*), estando no topo da pilha a API da plataforma CUDA.

A plataforma CUDA introduz dois novos conceitos para o escalonamento das *threads*: bloco e grelha. É com estes conceitos que se organiza a repartição dos dados entre as *threads*, bem como a sua organização

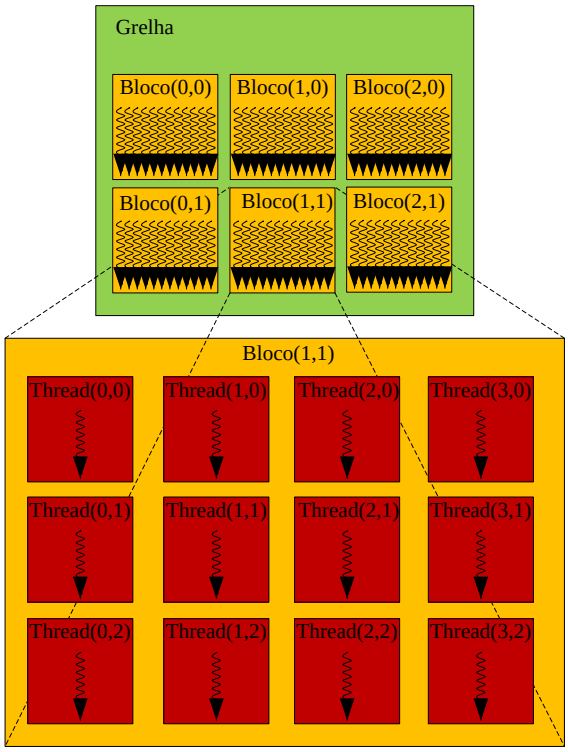

Figura 2.5: Exemplo de uma grelha com os seus blocos. A grelha é a unidade básica onde estão distribuídos os blocos. Nesta figura, ilustra-se o exemplo de uma grelha=(2,3) e cada bloco=(3,4).

e distribuição em *hardware*. Um bloco é a unidade básica de organização das *threads* e de mapeamento para o *hardware*, sendo alocado num multiprocessador da GPU. Desta forma, o tamanho mínimo recomendado de um bloco é de 8 *threads*, caso contrário haverá processadores ociosos. Os blocos podem ter uma, duas ou três dimensões.

Uma grelha é a unidade básica onde estão distribuídos os blocos, sendo responsável pela estrutura completa de distribuição das *threads* que se executam numa função. É nela que está definido o número de blocos e *threads* que serão criados e geridos pela GPU para uma dada função. Uma grelha pode ter uma ou duas dimensões.

Esta API fornecida pela NVIDIA introduz extensões à linguagem C, que são:

- Qualificadores do tipo de função, para definir a unidade lógica de execução do código CPU ou GPU, onde existem por sua vez três tipos de qualificadores:
  - `__device__`: define uma função que executa em GPU, onde só pode ser chamada a partir da GPU.

  - `__global__`: define o que a plataforma CUDA chama de *kernel*, ou seja, uma função que executa na GPU e é chamada a partir da CPU.
  - `__host__`: define uma função que será executada no CPU e que só pode ser chamada a partir da CPU.

- Qualificadores do tipo de variável:
  - `__device__`: define uma variável que reside na memória global da GPU — estas variáveis podem ser acedidas por todas as *threads* de uma grelha e também a partir da CPU através do uso da biblioteca de execução do CUDA, sendo o seu tempo de vida igual ao da aplicação.
  - `__constant__`: define apenas que a variável residirá no espaço de memória constante da GPU.
  - `__shared__`: residem na memória partilhada da GPU e só podem ser acedidas pelas *threads* de um mesmo bloco, sendo o seu tempo de vida igual ao tempo de vida do bloco.

Para se efectuar a chamada de uma função global é necessário informar no código fonte as dimensões da grelha e do bloco, sendo isto feito através de uma nova sintaxe na chamada da função. Utiliza-se, entre o nome da função e os argumentos passados a ela, um *array* bidimensional onde constam as dimensões da grelha e do bloco. Este *array* é delimitado pelos caracteres `<<<` e `>>>`. Dentro desta função é possível aceder aos valores dos índices das *threads* através da variável `threadIdx`, que é um vector de até três dimensões. O acesso a cada dimensão é feito através das componentes x, y e z, ou seja, `threadIdx.x`, `threadIdx.y` e `threadIdx.z`. Para se aceder a um determinado bloco são usadas as variáveis `blockIdx.x` e `blockIdx.y`. Os valores das dimensões dos blocos de *threads* podem ser acedidos através da variável `blockDim`, em `blockDim.x`, `blockDim.y` e `blockDim.z`. Por fim, é possível aceder aos valores das dimensões da grelha através da variável `gridDim`, em `gridDim.x` e `gridDim.y`.

Esta API apresenta algumas restrições: as funções `__device__` e `__global__` não suportam recursividade, não podem ser declaradas variáveis estáticas e não podem ter um número variável de argumentos. As funções `__global__` têm de retornar `void`.

As chamadas às funções `__global__` são assíncronas, ou seja, a execução é contínua na CPU mesmo que a respectiva função não tenha terminado na GPU. Os parâmetros de uma função `__global__` estão limitados a 256bytes.

A figura 2.6 ilustra o fluxo de processamento de um programa escrito em CUDA, que se inicia ao copiar os dados a serem processados na GPU da memória RAM para a memória principal da GPU. Após se

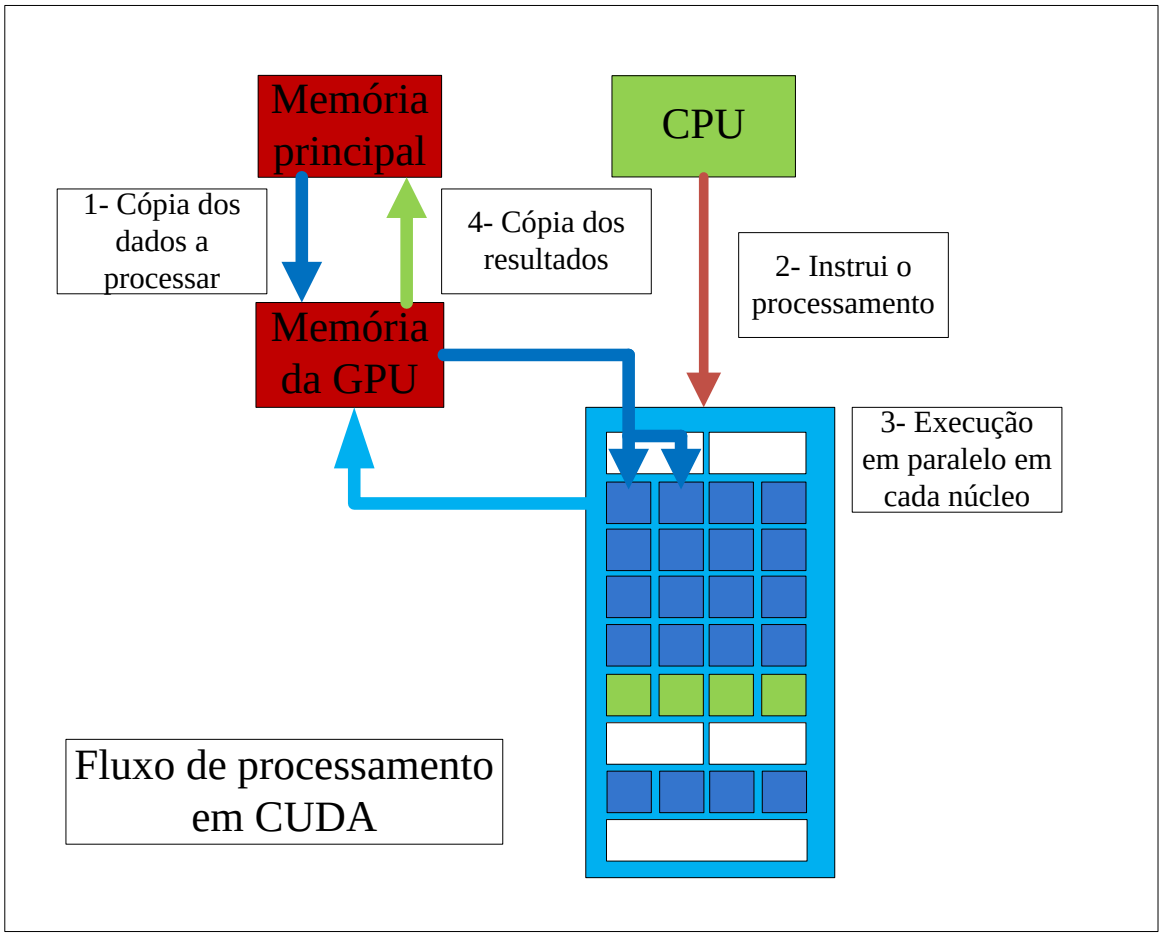

Figura 2.6: Fluxo de execução de um programa em CUDA. O programa inicia e são copiados os dados da memória RAM para a memória principal da GPU, depois configura-se uma grelha e é chamado um *kernel* que executa na GPU. No final, os dados processados na GPU são copiados para a memória RAM.

configurar uma grelha é realizada a chamada de um *kernel* para ser executado na GPU. Por fim, os dados resultantes do processamento em GPU são copiados para a memória RAM para posterior processamento.

Consequentemente, o ciclo de execução de uma aplicação utilizando a tecnologia CUDA alterna entre execuções na CPU e na GPU.

Capítulo 3

Implementação

Neste capítulo explica-se a implementação do modelo proposto por [18] e como foi efectuada a sua paralelização em CUDA.

Inicialmente o modelo foi implementado através de uma versão em Matlab [11], a qual se revelou extremamente lenta.

A segunda implementação foi efectuada recorrendo à linguagem de programação C/C++, com recurso à biblioteca OpenCV [9][10], onde se identificaram as partes do algoritmo que seriam susceptíveis de paralelização.

Finalmente procedeu-se à implementação do modelo em placa gráfica recorrendo à linguagem CUDA [3], sendo que nesta secção será descrito como foi efectuada esta implementação.

A figura 3.1 representa o mapa global do fluxo de dados do sistema. Nos parágrafos seguintes são apresentados e descritos mapas correspondentes às funções que integram o mapa global, conforme as figuras 3.2, 3.4, 3.6 e 3.7.

Na inicialização do programa são gerados os filtros de Gabor, com uma proporção $\gamma = 1$, desvio padrão $\delta = 7/3$ píxeis, comprimento de onda $\lambda = 7$ píxeis e fase $\psi \in \{0^o, 90^o\}$ [20]. São inicializados os filtros de diferenças de Gauss com os parâmetros $\sigma_{ex} = 2\%$ e $\sigma_{ini} = 25\%$ das colunas da imagem de entrada, $c_{ex} = 0,5$ e $c_{ini} = 1.5$ [21]. É também reservada toda a memória necessária para a execução do programa, tanto na CPU como na GPU, sendo os filtros inicializados anteriormente na CPU copiados para a memória global da GPU na função de inicialização. Ainda na inicialização, os filtros de diferenças de Gauss são copiados para a memória de constantes, pois o acesso a esta memória é mais rápido.

A imagem a cores é obtida da câmara de vídeo com recurso à biblioteca OpenCV. Na parte relativa ao

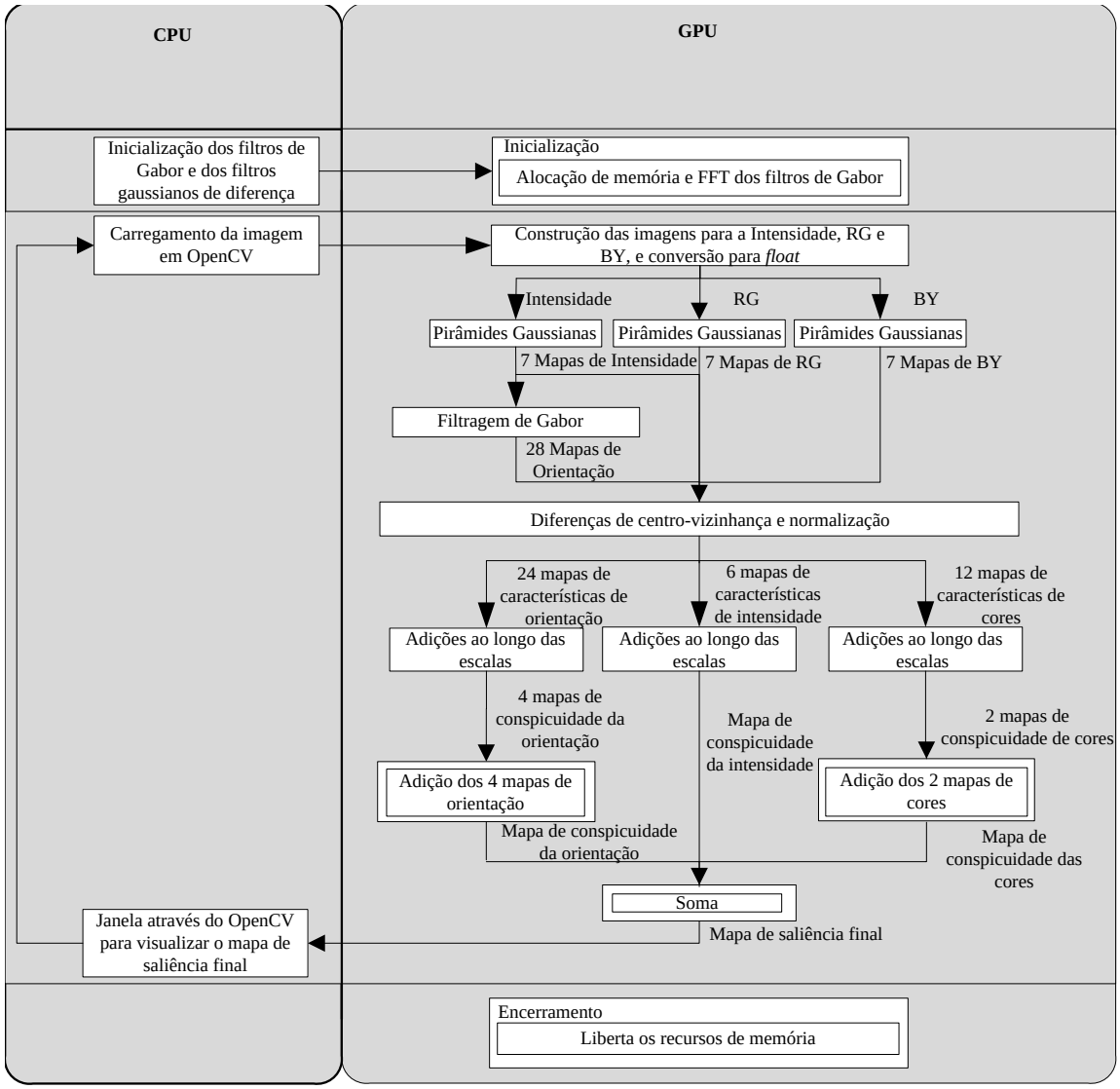

Figura 3.1: Fluxo de dados do sistema. Na função de inicialização é reservada toda a memória necessária e os filtros de Gabor são convertidos para o domínio da frequência. Da imagem de entrada são obtidos os mapas de intensidade e rivalidade entre cores e através destes mapas são criadas as pirâmides Gaussianas. Os 7 mapas de intensidade são convoluídos com os filtros de Gabor para se obterem os 28 mapas de orientação. São determinadas as diferenças centro-vizinhança. As adições ao longo das escalas são computadas através dos 6 mapas de características da intensidade, dos 12 mapas de características da rivalidade entre cores e dos 24 mapas de características da orientação. O mapa de saliência final é obtido através da combinação dos 3 mapas de conspicuidade.

processamento de CUDA, os canais da imagem a cores (r, g e b) são separados para que se formem os mapas de intensidade, os mapas RG e os mapas BY (equações 2.1, 2.2 e 2.3). Este processo é feito através de uma grelha, representada na equação 3.1, e cada bloco contém as *threads* representadas na equação 3.2.

$$grelha = (32, 32); \tag{3.1}$$

$$bloco = (20, 15); \tag{3.2}$$

Para se criarem as pirâmides Gaussianas é efectuada uma convolução entre um filtro de Gauss de 5 coeficientes e a imagem a ser reduzida, e são feitas sucessivas decimações por 2. A convolução é separada em duas direcções: uma na direcção horizontal e outra na direcção vertical, sendo que estas convoluções utilizam a memória partilhada da GPU para que o desempenho seja melhorado. A convolução na direcção horizontal tem uma grelha, representada na equação 3.3, e cada bloco tem o número de *threads* em x, representadas na equação 3.4. São usados 528 bytes de memória partilhada por cada bloco e são distribuídos 6 blocos por cada multiprocessador.

$$grelha = (divisao\_int\_acima(Colunas, 128), Linhas); \tag{3.3}$$

$$bloco = (128); \tag{3.4}$$

A função `divisao_int_acima` arredonda o resultado para o inteiro imediatamente acima. Quando a grelha na equação 3.3 tem um resultado de (1, Linhas) cada bloco tem o número de *threads* em x igual ao valor do número de colunas da imagem. A convolução na direcção vertical tem uma grelha representada na equação 3.5 e cada bloco tem o número de *threads* representado na equação 3.6. São usados 1120 bytes de memória partilhada por cada bloco e são distribuídos 4 blocos por cada multiprocessador.

$$grelha = divisao\_int\_acima(Colunas, 16), divisao\_int\_acima(Linhas, 8)); \tag{3.5}$$

$$bloco = (16, 8); \tag{3.6}$$

Na equação 3.5, quando a grelha em x tem um valor igual a 1, cada bloco tem um valor igual às linhas para as *threads* em x. Quando a grelha tem um valor igual a 1 em y, cada bloco tem um valor igual às linhas para as *threads* em y.

Para se obterem os mapas de orientação é feita uma convolução entre cada um dos 7 mapas de inten-

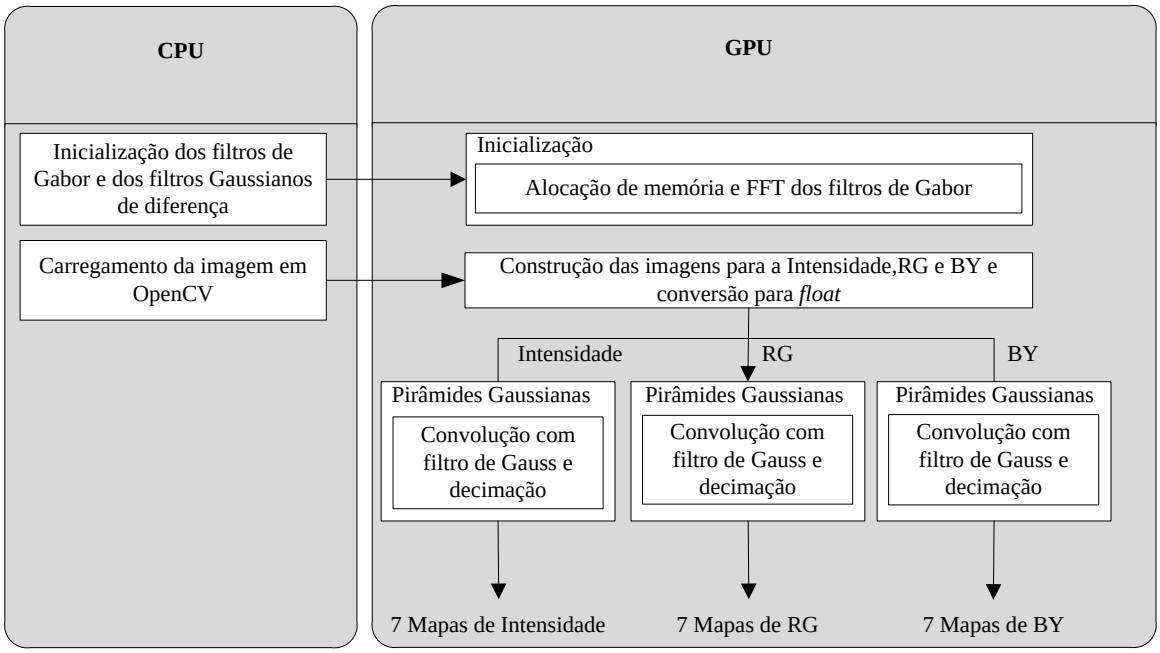

Figura 3.2: Fluxo de dados — Inicialização e criação das pirâmides Gaussianas. Os filtros de Gabor e de diferenças de Gauss são inicializados na CPU. A imagem é obtida da câmara de vídeo através da biblioteca OpenCV. Na inicialização é reservada toda a memória necessária para a execução do programa e os filtros de Gabor são convertidos do domínio do espaço para o domínio da frequência. Através da imagem de entrada a cores cuja resolução é de 640×480 píxeis são obtidos os mapas de intensidade (equação 2.1), RG (equação 2.2) e BY (equação 2.3), sendo através destes mapas que são criadas as pirâmides Gaussianas(escalas $\sigma = 2, \cdots, 8$) e obtêm-se os 7 mapas de intensidade, os 7 mapas de RG e os 7 mapas de BY necessários para a computação das diferenças centro-vizinhança.

sidade e os filtros de Gabor. Porém, como uma convolução com um filtro 19x19 consome muitos recursos, recorre-se ao domínio da frequência para que sejam efectuadas multiplicações, ou seja, recorre-se à FFT e IFFT para acelerar o processo de filtragem. Os filtros de Gabor e as imagens têm de ser convertidos para o domínio da frequência usando a FFT e depois multiplicados entre si, sendo o resultado convertido de volta ao domínio do espaço usando a IFFT. Usando a biblioteca CUFFT [2] calculam-se as 8 FFT dos filtros de Gabor com 4 orientações diferentes e duas fases diferentes, realizando-se este processo na inicialização. Como as imagens usadas são as de escalas $\sigma = (2, \cdots, 8)$, constrói-se uma imagem cuja resolução é de 256×256 píxeis (figura 3.3) onde cabem todas as imagens sendo feita uma FFT e 4 IFFTs em vez de 7 FFTs e 28 IFFTs. As IFFTs correspondem às imagens cuja resolução é de 256×256 píxeis convoluídos com os filtros de Gabor com $\theta = 0^o, 45^o, 90^o$ e 135^o.

Deverá ser dada particular atenção aos cantos das imagens quando estas são construídas, com recurso a um modo de textura chamado *clamp to border* [4] em que as imagens são redimensionadas e os píxeis fora dos cantos das imagens ficam com o valor do último píxel do canto da imagem — este método é aplicado às

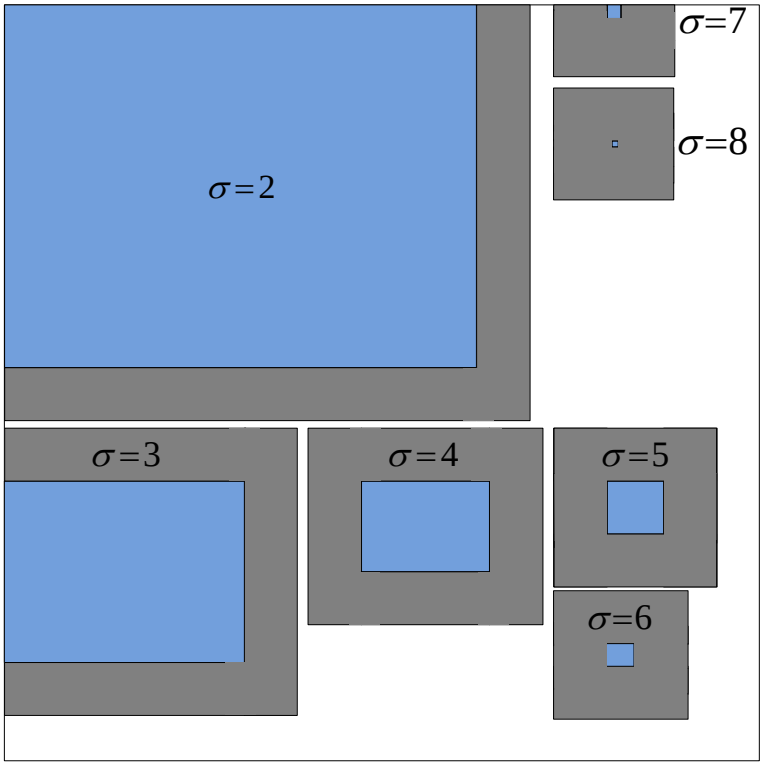

Figura 3.3: Imagem cuja resolução é de 256×256 píxeis. Contém os 7 mapas de intensidade (escalas $\sigma = 2, \cdots, 8$). A azul estão representados os 7 mapas de intensidade e é deixado espaço entre os mapas para que seja aplicado o modo de textura *clamp to border* [4] que está representado a cinzento.

7 sub-imagens de escalas ($\sigma = 2, \cdots, 8$). Os filtros de Gabor são redimensionados para o mesmo tamanho da imagem 256×256 píxeis, pois a convolução com FFT só se pode aplicar com entradas do mesmo tamanho. Na sequência retiram-se os 28 mapas de características da orientação das 4 imagens 256×256 píxeis resultantes do processo de filtragem .

Para se efectuarem as diferenças de centro-vizinhança, todos os 42 mapas de características têm de ser redimensionados para a escala 4 (resolução 40×30 píxeis). Aos mapas de escala 2 e 3 realiza-se uma decimação e aos mapas de escalas 5, 6, 7 e 8 é efectuada uma interpolação bilinear, executando-se primeiro a decimação na horizontal e só depois na vertical, com uma grelha, conforme representada na equação 3.7, e cada bloco contém o número de *threads* em x representado na equação 3.8.

$$grelha = (divisao\_int\_acima(Colunas, 128), Linhas); \tag{3.7}$$

$$bloco = (128); \tag{3.8}$$

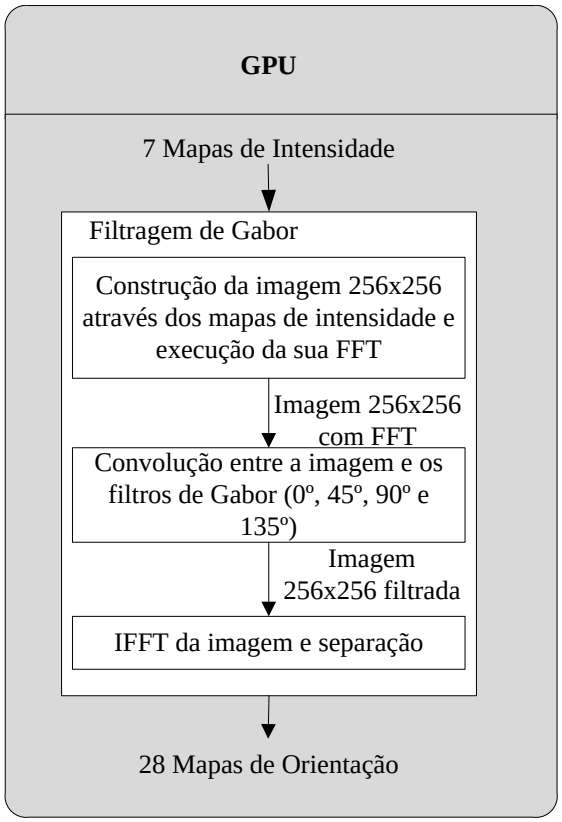

Figura 3.4: Fluxo de dados — Filtragem de Gabor. Cria-se a imagem, cuja resolução é de 256×256 píxeis, que contém os 7 mapas de intensidade ($\sigma = 2, \cdots, 8$). Esta imagem é multiplicada por cada filtro de Gabor ($\theta = 0^o, 45^o, 90^o, 135^o$). Depois da filtragem são realizadas 4 IFFTs e das 4 imagens resultantes obtêm-se os 28 mapas de orientação.

Quando a grelha tem um resultado de (1, Linhas) as *threads* em x têm o valor do número de colunas da imagem. Para a decimação na vertical há uma uma grelha representada na equação 3.9 e cada bloco contém o número de *threads* representado na equação 3.10.

$$grelha = (divisao\_int\_acima(Colunas, 16), divisao\_int\_acima(Linhas, 8); \tag{3.9}$$

$$bloco = (16, 8); \tag{3.10}$$

Quando a grelha em x tem um valor igual a 1, cada bloco tem um valor igual às linhas para as *threads* em x. Quando a grelha tem em y um valor igual a 1 cada bloco tem um valor igual às linhas para as *threads* em y. Para as interpolações bilineares é usada a grelha representada na equação 3.11 e cada bloco tem o número de *threads* representado na equação 3.12.

$$grelha = (divisao\_int\_acima(40, Colunas), divisao\_int\_acima(30, Linhas)); \tag{3.11}$$

$$bloco = (Colunas, Linhas); \tag{3.12}$$

Na sequência do redimensionamento de todos os mapas para a escala 4, os cálculos, para que se efectuem as diferenças de centro-vizinhança, são implementados com uma grelha representada na equação 3.13 e cada bloco tem o número de *threads* representado na equação 3.14.

$$grelha = (2, 2); \tag{3.13}$$

$$bloco = (20, 15); \tag{3.14}$$

O operador $\mathcal{N}(.)$, conforme representado nas equações 2.12, 2.13 e 2.14, é implementado recorrendo a filtros de diferenças de Gauss: um de excitação 5×5 píxeis e um de inibição 29×29 píxeis. Este operador $\mathcal{N}(.)$ é implementado através da convolução entre cada filtro e o mapa de características, onde os cálculos são efectuados na memória partilhada da GPU (ver figura 3.5) para que se obtenha um melhor desempenho. São usados 5232 bytes de memória partilhada por cada bloco e são distribuídos 2 blocos por cada multiprocessador. Esta convolução tem uma grelha representada na equação 3.15 e cada bloco tem o número de *threads* representado na equação 3.16, sendo que dentro da função global é assegurado que não se sai fora dos limites do mapa de características. São efectuadas 42 convoluções entre os filtros de diferenças de Gauss e cada mapa de características. Os máximos globais de cada mapa de características são calculados recorrendo-se à biblioteca *thrust* [7].

$$grelha = (5, 5); \tag{3.15}$$

$$bloco = (8, 6); \tag{3.16}$$

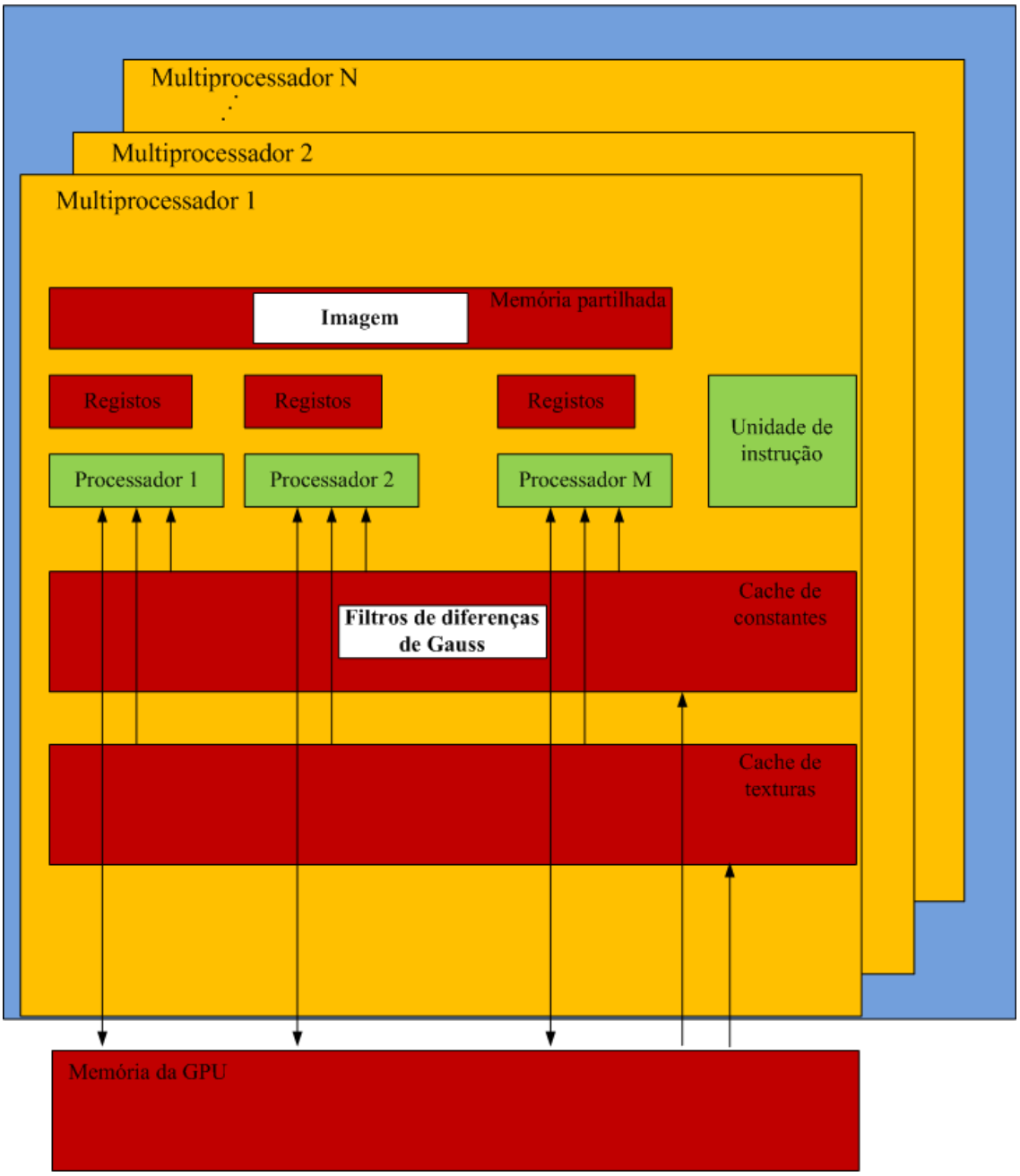

Figura 3.5: Convolução dos filtros de diferenças de Gauss com a imagem. Os filtros estão armazenados na memória de constantes e a imagem está armazenada na memória partilhada quando ocorre a convolução.

Para as adições ao longo das escala os cálculos são efectuados com recurso a uma grelha representada na equação 3.17 e tendo cada bloco um número de *threads* representado na equação 3.18.

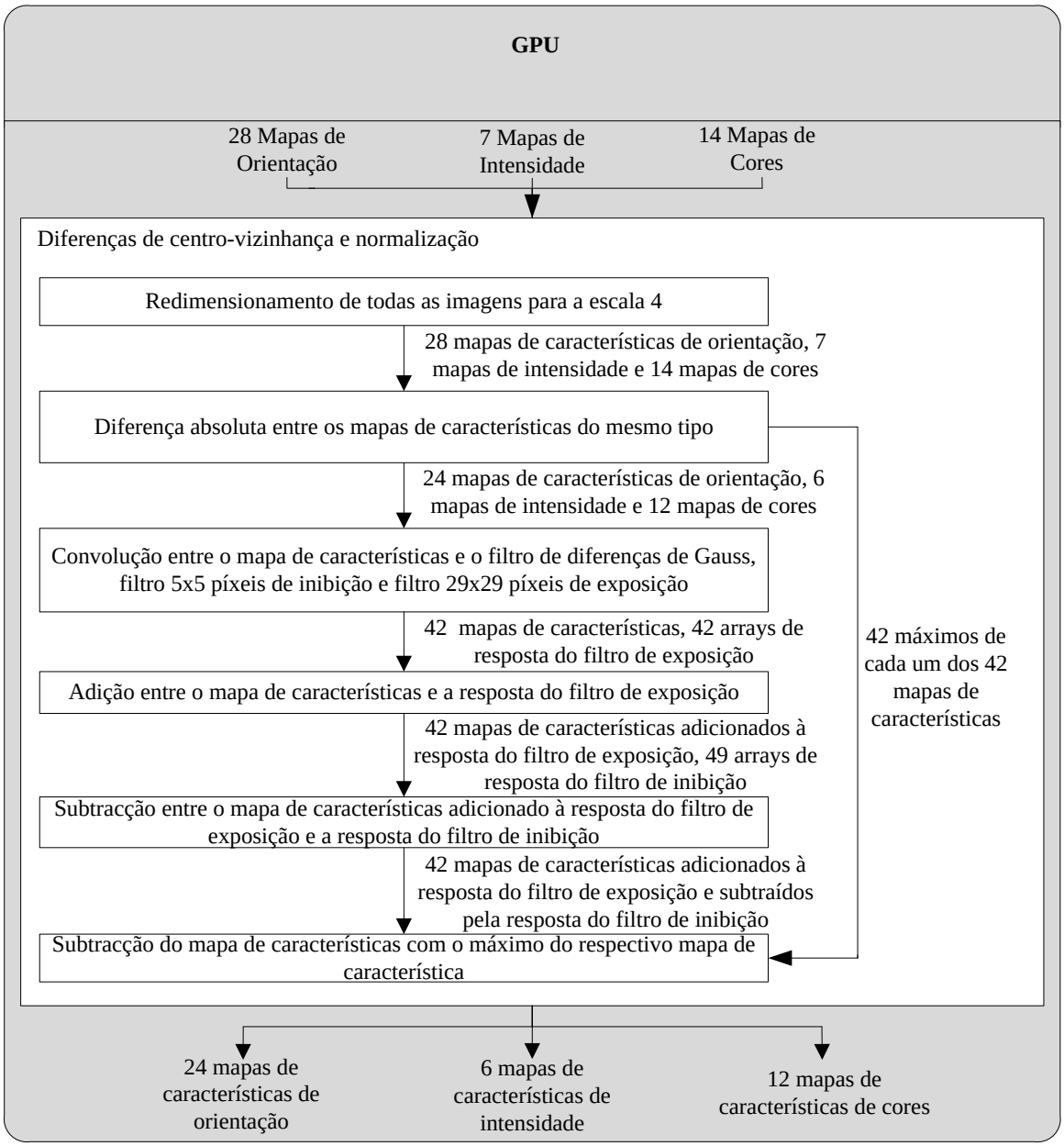

Figura 3.6: Fluxo de dados — Diferenças de centro-vizinhança. Os 7 mapas de intensidade, 14 mapas de rivalidade entre cores e 28 mapas de orientação são redimensionados para a escala $\sigma = 4$ (40×30 píxeis). As diferenças de centro-vizinhança são efectuadas aos mapas do mesmo tipo, ou seja, aos 7 mapas de intensidade é aplicada a equação 2.8, aos 7 mapas RG a equação 2.9, aos 7 mapas de BY a equação 2.10 e aos 28 mapas de orientação a equação 2.11, obtendo-se 6 mapas de características de intensidade, 12 mapas de características para a rivalidade entre cores e 24 mapas de características para a orientação.

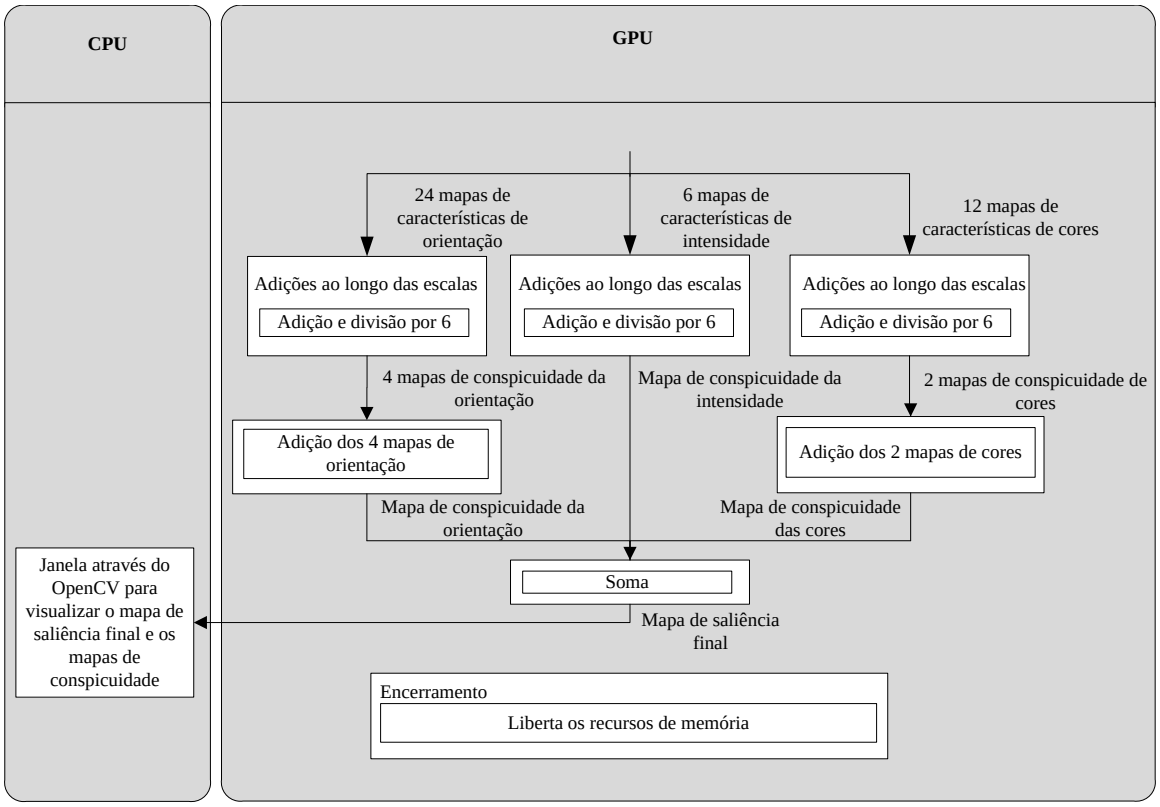

Figura 3.7: Fluxo de dados — Adições ao longo das escalas e cálculo do mapa de saliência final. As adições ao longo das escalas são aplicadas aos mapas de características da intensidade segundo a equação 2.14, aos mapas de características da rivalidade entre cores segundo a equação 2.15 e aos mapas de características da orientação segundo a equação 2.16. O mapa de saliência final é obtido através dos mapas de conspicuidade da intensidade, rivalidade entre cores e orientação (equação 2.17). O mapa de saliência final é copiado da memória da GPU para a memória RAM e é mostrado numa janela no ecrã. Quando o programa termina, na função de encerramento, é libertada toda a memória.

$$grelha = (2,2); \tag{3.17}$$

$$bloco = (20,15); \tag{3.18}$$

Por último, os mapas de conspicuidade são combinados formando o mapa de saliência final e são redimensionados de 40×30 píxeis para 640×480 píxeis, recorrendo a uma função da biblioteca OpenCV [9].

Capítulo 4

Resultados e discussão

Neste capítulo apresentam-se os resultados da implementação em GPU do modelo de saliência baseado na atenção visual, e faz-se a comparação com outras implementações realizadas em CPU. As implementações em CPU realizadas foram: uma em C/C++, recorrendo à biblioteca OpenCV, e uma versão em Matlab. De notar que se começou um implementação recorrendo-se à *AR.Drone open API platform* [24], para se receber imagens da câmara frontal do AR.Drone da Parrot e controlá-lo através de um *gamepad*. Realizou-se uma outra implementação à parte em C/C++, usando *sockets*, para receber imagens do quadricóptero.

Um mapa de saliência final é obtido no final da execução, onde se identificam as regiões salientes da imagem de entrada.

A obtenção do mapa de saliência final passa pelos seguintes passos:

- Obtenção da imagem da câmara de vídeo e cópia da imagem da memória da CPU para a memória global da GPU;
- Criação das pirâmides Gaussianas para a intensidade e rivalidade entre cores;
- Filtragem de Gabor;
- Redimensionamento de todos os 42 mapas para a escala 4;
- Diferenças de centro-vizinhança;
- Máximos globais dos 42 mapas;
- Convolução dos 42 mapas de características com os filtros de diferenças de Gauss;

- Adições ao longo das escalas;
- Normalização;
- Obtenção do mapa de saliência final através dos mapas de conspicuidade da intensidade, dos mapas da rivalidade entre cores e dos mapas da orientação, e a cópia do mapa de saliência final da memória global da GPU para a memória da CPU;
- Redimensionamento do mapa de saliência final na CPU.

De seguida demonstra-se os resultados obtidos para várias imagens de entrada de resolução 640×480 píxeis. A implementação foi testada usando um Pentium Dual Core 3.4Ghz com uma GPU NVIDIA 9800 GTX+ com 512Mb de RAM.

A alocação de memória ocorre apenas uma vez, tanto na CPU como na GPU. Obtendo a imagem de entrada duma câmara de vídeo conectada por *firewire*, a computação do mapa de saliência final necessita de 40ms, obtendo-se desta forma 25 quadros por segundo, em que o tempo de cálculo é a média de 10000 imagens de entrada estáticas.

Conforme pode ser observado na figura 4.1, o consumo de tempo de maior dimensão refere-se à aquisição de imagem da câmara de vídeo e respectivo processamento, com um tempo total de 14,74ms, representando 37% do tempo total da computação do mapa final de saliência.

A figura 4.2 demonstra que a computação do mapa de conspicuidade da orientação é a que tem mais peso no cálculo do mapa de saliência final, pois ocupa 54% desse tempo. As filtragens de Gabor aos mapas de intensidade ocupam um tempo de computação considerável e o número de mapas de orientação são sempre quatro vezes mais que os de intensidade, e duas vezes mais que os de rivalidade entre cores.

Para avaliar os ganhos de desempenho da computação do mapa de saliência final, compara-se o tempo da implementação em CUDA, numa NVIDIA 9800 GTX+, com uma implementação em C/C++ com recurso à biblioteca OpenCV [9][10] e Matlab [11], como demonstrado na tabela 4.1. O tempo de cálculo é a média de uma sequência de 500 imagens de entrada carregadas a partir do disco rígido do computador.

	Tempo
Matlab	3,52s
C/OpenCV	1,1s
CUDA	35,93ms

Tabela 4.1: Comparação entre as várias implementações

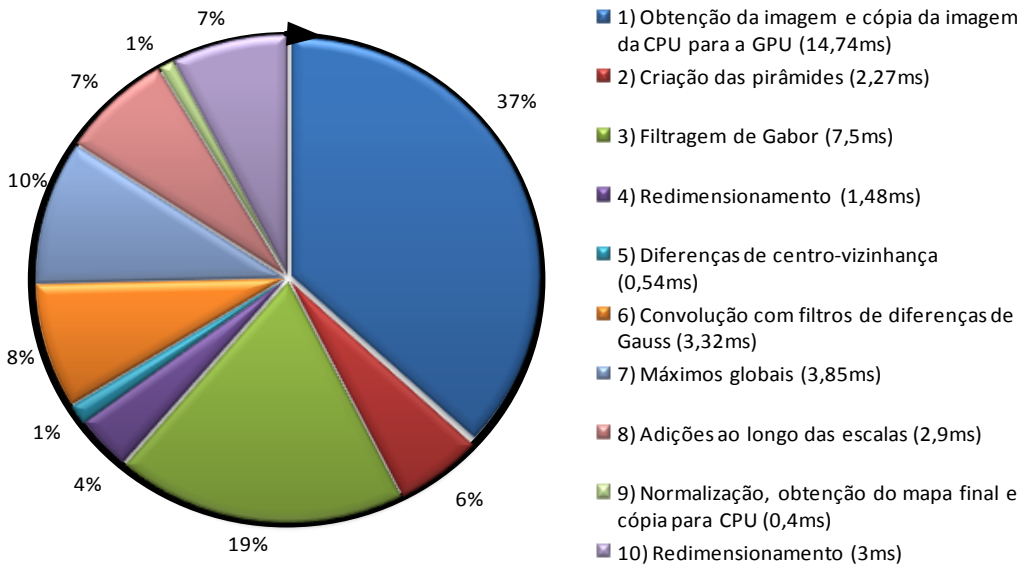

Figura 4.1: Proporção temporal entre os vários estágios do cálculo do mapa de saliência fina, sendo a imagem de entrada obtida de uma câmara de vídeo. Neste gráfico é demonstrado o peso de cada passo, em percentagem, no cálculo do mapa de saliência final. O primeiro passo é o que tem mais peso no tempo de computação do mapa de saliência final, representando 36,85% do tempo total, sendo incluído o tempo da obtenção da imagem a cores da câmara de vídeo.

	Nossa Implementação	Implementação proposta por [13]
Equipamento usado	NVIDIA 9800GTX+	NVIDIA GTX480
Número de núcleos CUDA	128	480
Tempo de cálculo do mapa de saliência final	35,93ms	42,03ms

Tabela 4.2: Comparação entre a nossa implementação e a proposta por [13]. O tempo de obtenção das imagens do disco rígido do computador é incluído para a nossa implementação.

Da tabela 4.1 observa-se que existe um *speedup* elevado da implementação em CUDA em relação à implementação sequencial em CPU. Esse *speedup* é demonstrado na figura 4.3.

Considera-se, em termos comparativos, o sistema utilizado de acordo com a referência [13], testando-se a implementação proposta nesta dissertação através de uma sequência de 500 imagens de entrada carregadas a partir do disco rígido do computador.

Através da tabela 4.2 verifica-se que o sistema implementado de acordo com a presente dissertação, apresenta características substancialmente mais favoráveis, designadamente ao nível dos tempos de cálculo do mapa de saliência final. Como tal, poderá ser demonstrado que a utilização de equipamentos de características equiparadas ao sistema de [13], permite apresentar um tempo de cálculo do mapa de saliência final

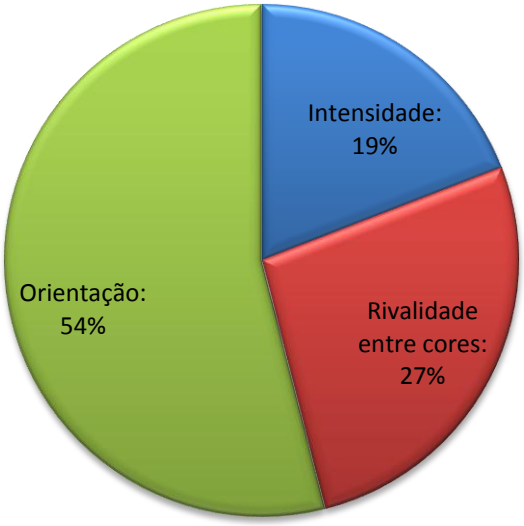

Figura 4.2: Proporção temporal do cálculo de cada mapa de conspicuidade.

substancialmente inferior.

De forma a avaliar de forma qualitativa o processo de cálculo da saliência proposto na nossa solução, testou-se a implementação com sete imagens de entrada diferentes. Da imagem de entrada são calculados os mapas de conspicuidade para a intensidade, os de rivalidade entre cores e os de orientação, sendo posteriormente através destes mapas calculado o mapa de saliência final. Deste último é seleccionado o máximo de saliência mais próximo do centro da imagem (caso haja mais do que um píxel com o mesmo valor máximo de saliência) e, através da imagem de entrada, indica-se a zona que representa o foco da atenção utilizando uma seta vermelha. Os resultados são apresentados nas figuras 4.4, 4.5 , 4.6 , 4.7 , 4.8 , 4.9 e 4.10.

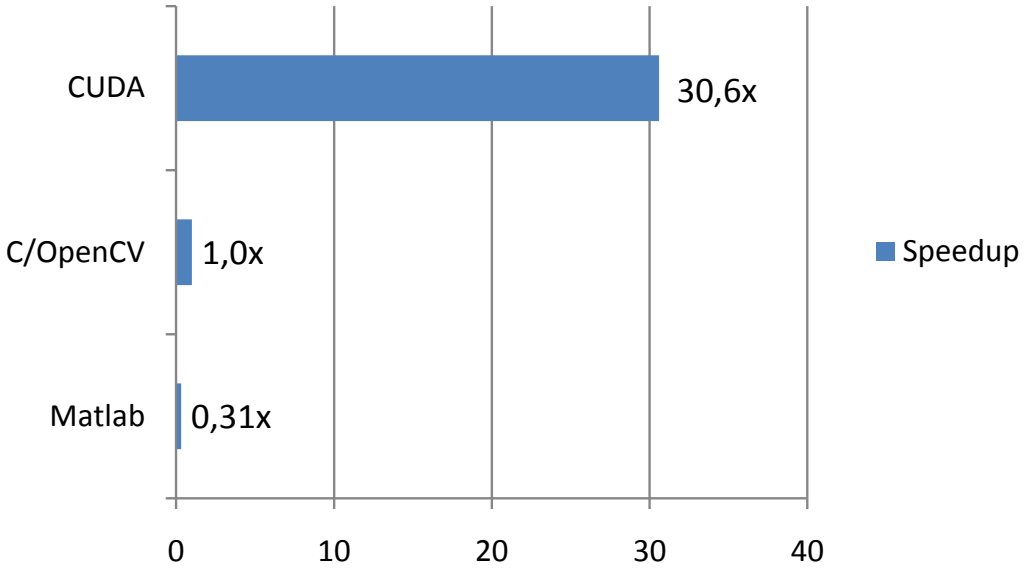

Figura 4.3: *Speedup* das implementações realizadas para o modelo de saliência. A implementação em CUDA tem um *speedup* de 30,6x em relação à implementação em C/C++ com recurso à biblioteca OpenCV.

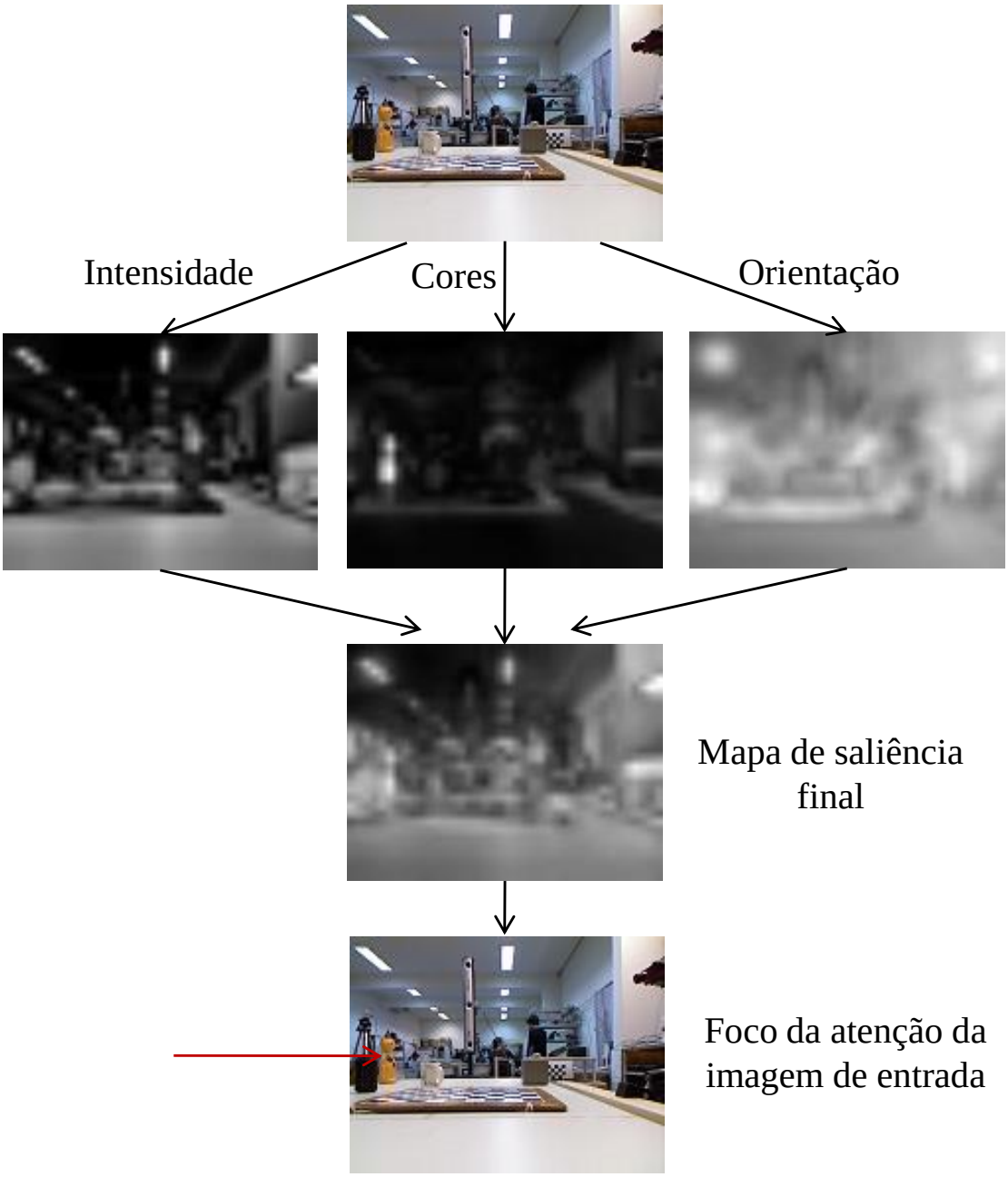

Figura 4.4: Resultados para a primeira imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

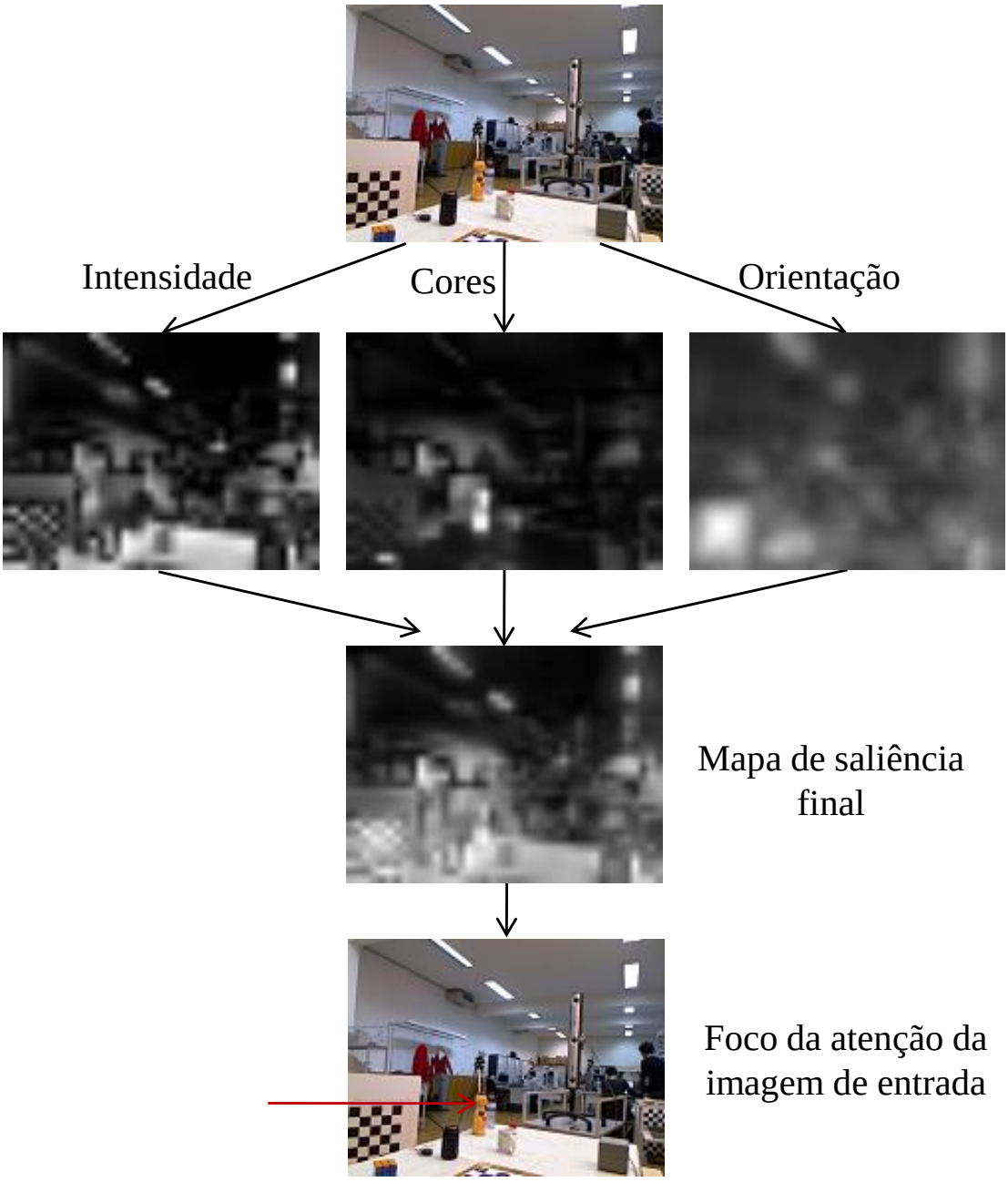

Figura 4.5: Resultados para a segunda imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

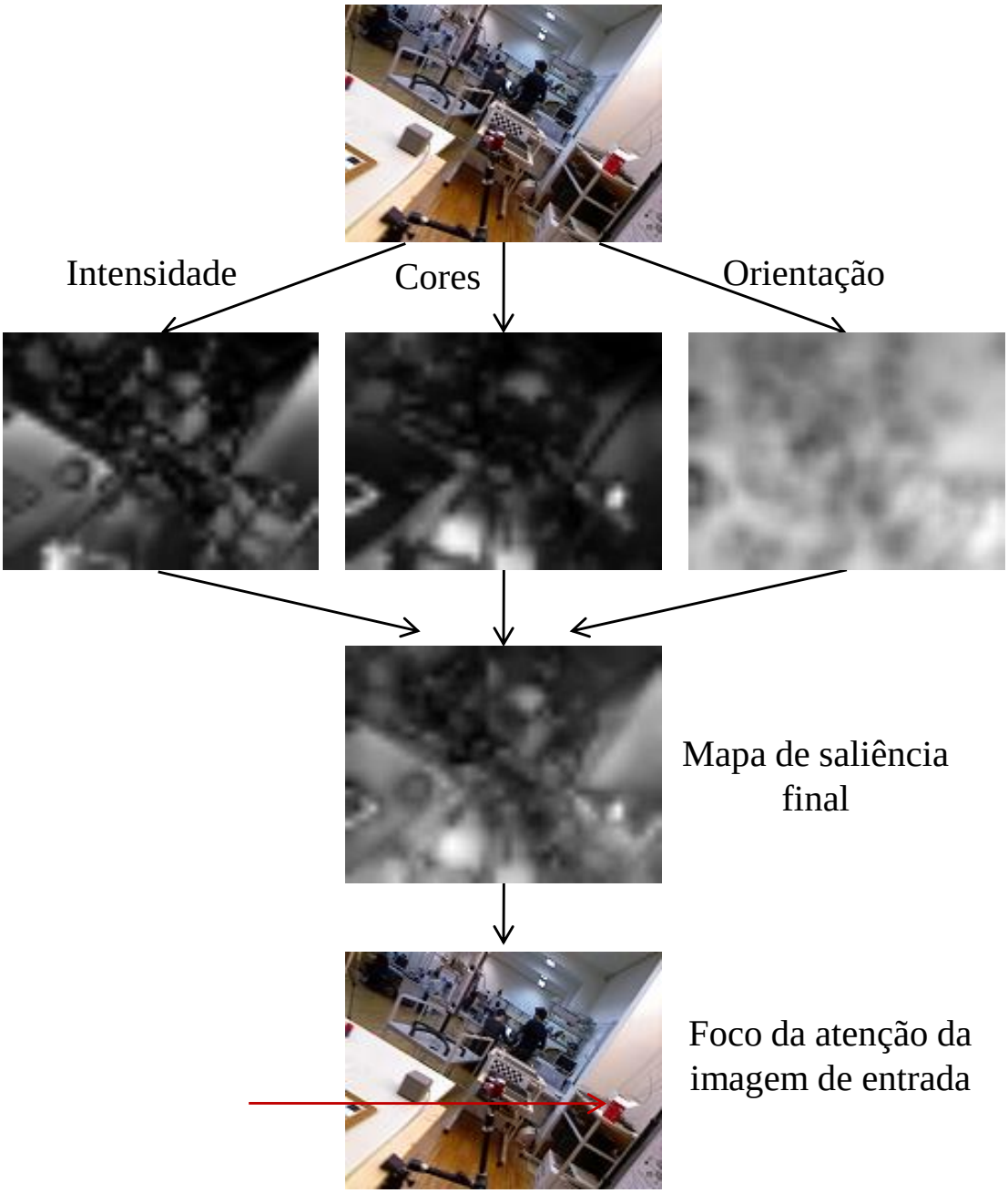

Figura 4.6: Resultados para a terceira imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

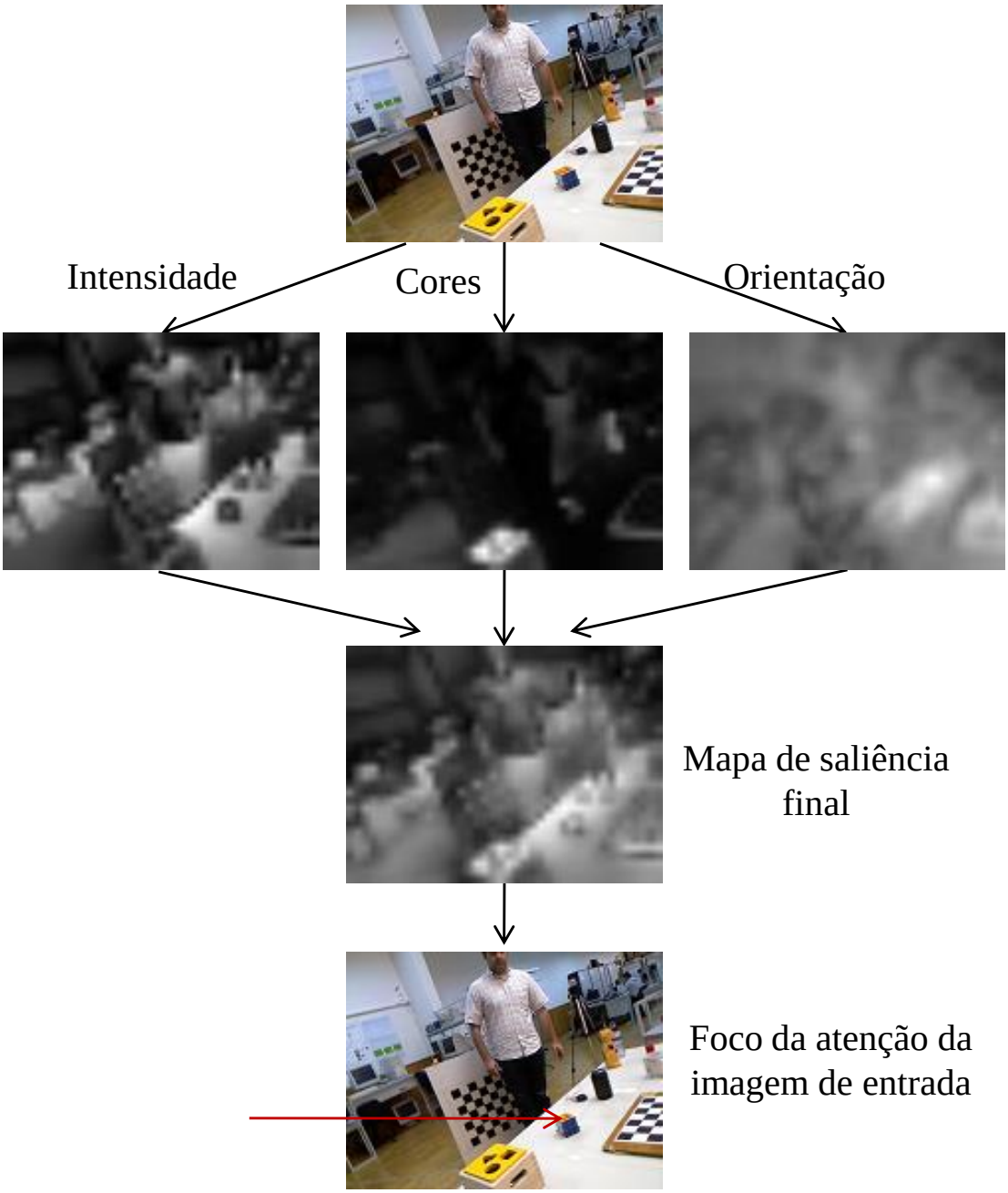

Figura 4.7: Resultados para a quarta imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

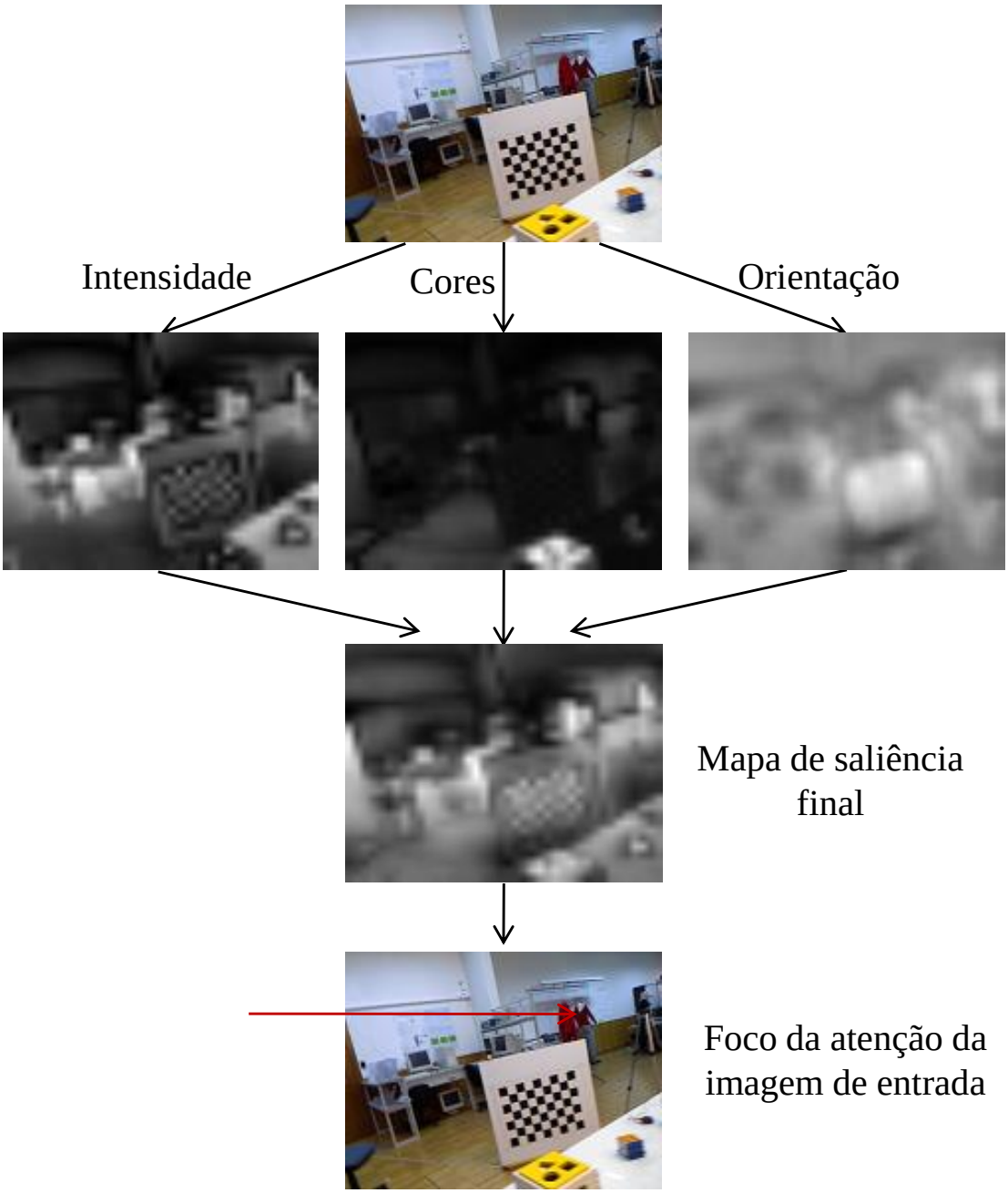

Figura 4.8: Resultados para a quinta imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

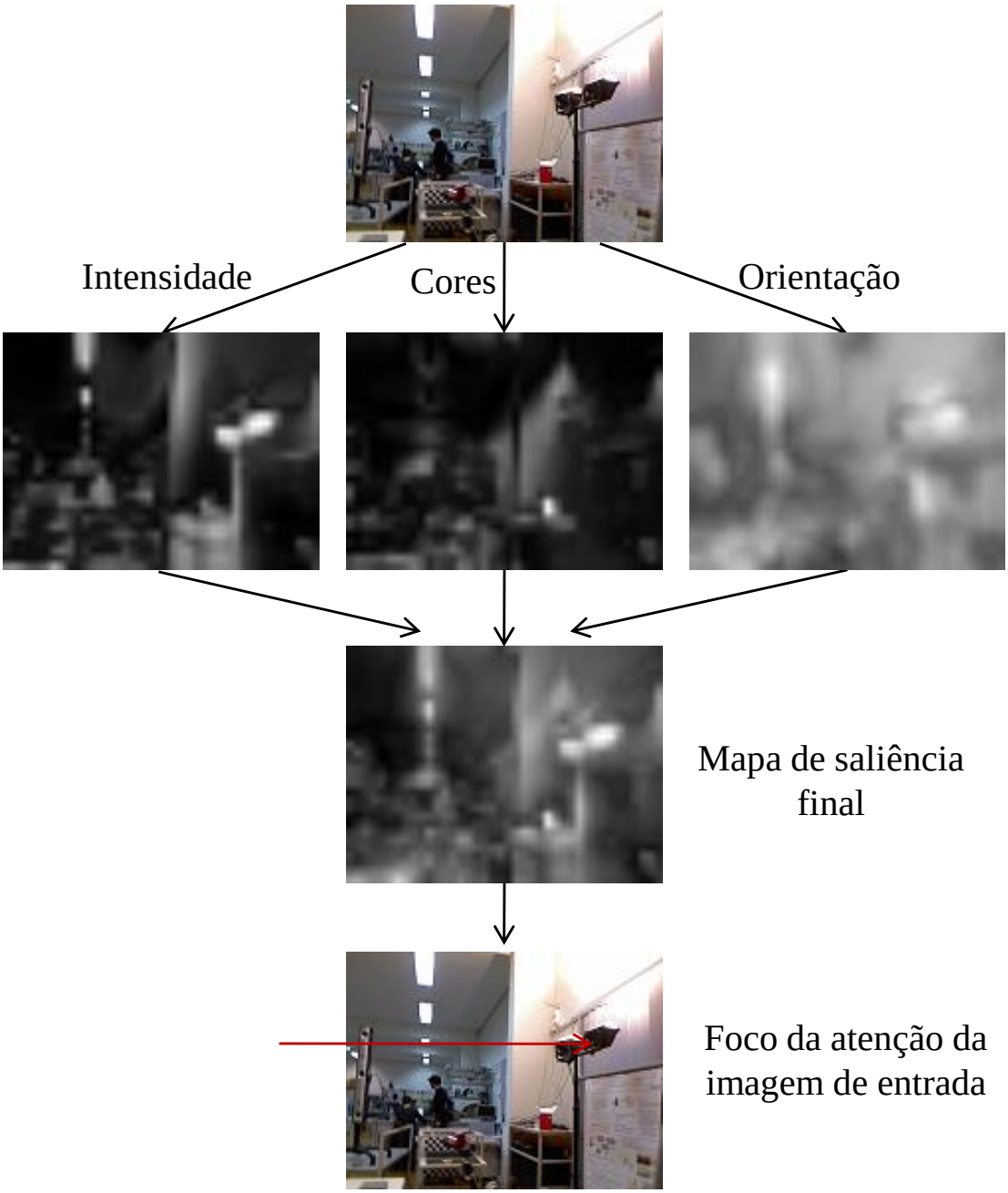

Figura 4.9: Resultados para a sexta imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

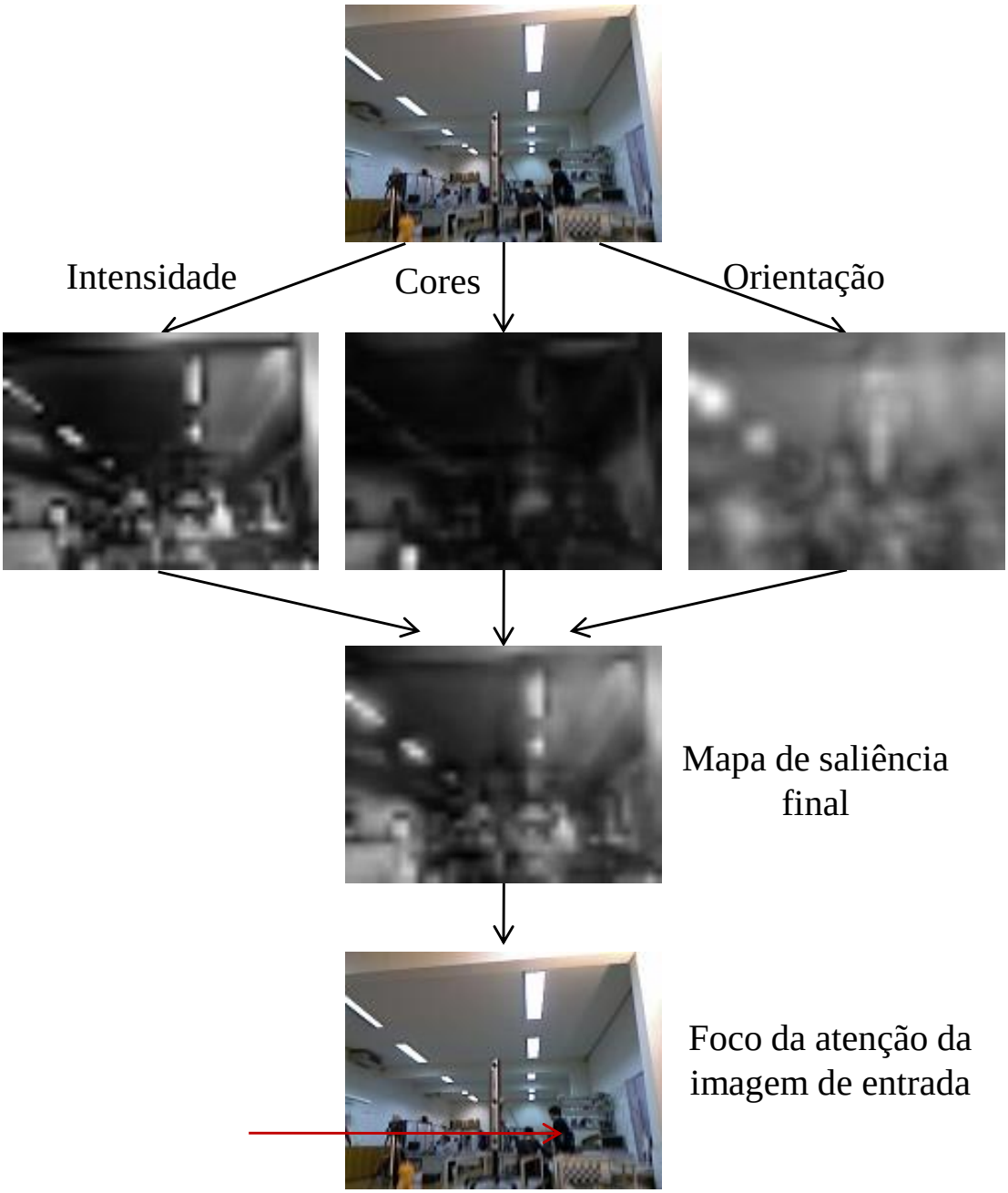

Figura 4.10: Resultados para a sétima imagem de entrada. À imagem de entrada é aplicada a implementação obtendo-se os mapas de conspicuidade da intensidade, os de rivalidade entre cores e os de orientação. Estes mapas de conspicuidade são combinados e obtém-se depois o mapa de saliência final. Na imagem de entrada é representado o foco da atenção com uma seta a vermelho.

Capítulo 5

Conclusão e trabalho futuro

Todos os objectivos propostos foram atingidos e formaram-se também ideias para trabalho futuro, concretamente na aplicação do módulo de saliência implementado em GPU proposto em situações práticas.

Ao longo desta dissertação foi apresentada uma solução prática para a paralelização do modelo de saliência numa unidade de processamento gráfico, obtendo-se um resultado em tempo-real de 25 imagens por segundo para uma imagem de entrada cuja resolução é de 640×480 píxeis, superando o objectivo inicialmente proposto de 24 imagens por segundo.

A maior dificuldade foi, sem dúvida, conseguir um sistema que possuísse um nível de precisão e de robustez satisfatório, e que atingisse as metas temporais delineadas. Porém, optimizando o código fonte da implementação este problema foi ultrapassado.

No apêndice A encontra-se um guia de utilizador para investigadores que desejem usufruir desta implementação e possam obter assistência no uso do sistema. Igualmente poderão efectuar optimizações ou acrescentar novos atributos, como por exemplo um novo mapa de conspicuidade para aplicar ou adaptar o sistema a uma determinada situação. Este guia facilita ainda a integração da implementação em sistemas de maior dimensão.

O guia de utilização consiste numa descrição do programa e da sua estrutura geral, com especial enfoque nas suas bibliotecas. Faz-se também referência ao *hardware* e *software* necessários, à forma de executar o demonstrador e à integração do *software* em terceiros. Por último descrevem-se as principais funções da implementação para que o sistema possa ser alterado e adequado às necessidades de futuros desenvolvimentos.

Existe uma versão pré-compilada do módulo de saliência, que recebe imagens da câmara de vídeo e

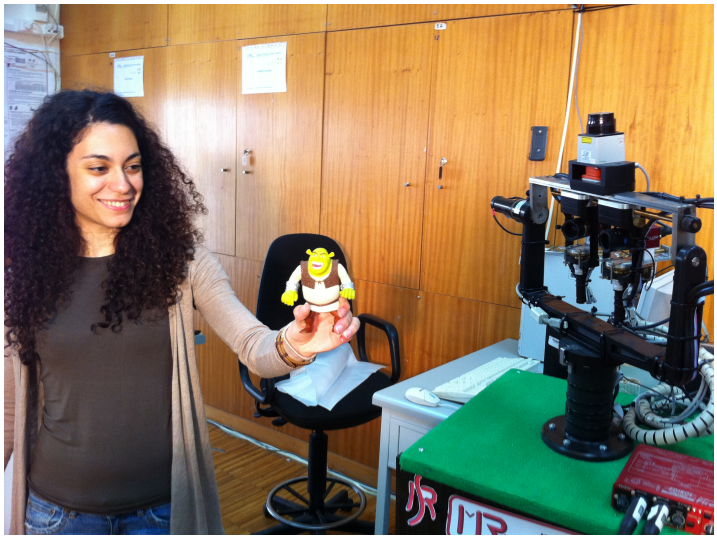

Figura 5.1: Atenção conjunta numa aplicação de interacção homem-robô em contexto social, usando a plataforma IMPEP.

através desta calcula os mapas de conspicuidade e o mapa de saliência final. Estes 4 mapas são enviados para o ecrã do computador para poderem ser visualizados pelo utilizador.

Como trabalho futuro propõe-se a adição mais mapas de conspicuidade, como um para detectar rostos humanos. Este mapa permite que seja dado mais peso no mapa de saliência final quando há a presença de uma pessoa na imagem.

Propõe-se que as grelhas utilizadas e os blocos utilizados ao longo da implementação sejam alterados de forma a obter a ocupação máxima de cada *kernel* que executa na GPU, pois assim é possível obter o máximo desempenho através da ocupação de todos os recursos existentes na GPU.

Ao longo do desenvolvimento do sistema não houve o cuidado com os acessos à memória global, onde pode haver optimizações através do acesso coalescido aos dados. A largura de banda da memória global é usada mais eficientemente quando acessos simultâneos à memória por threads de um meio-*warp* (durante a execução de uma instrução de leitura ou escrita) podem ser coalescidos em uma única transacção de 32, 64 ou 128 bytes de memória.

Num futuro próximo, este *software* será usado de forma integrada num sistema de percepção multisensorial activa para robôs sociais, recorrendo à plataforma IMPEP, apresentada na figura 5.1. O sistema terá a capacidade, especificamente, de emular o processo a que se dá o nome de "atenção conjunta" (*joint attention*), um tipo de imitação que o ser humano desenvolve durante o primeiro ano de vida aquando da interacção com os seus pais ou educadores. Através deste processo, um agente olha para onde o seu interlocutor

dirige o olhar através da produção de um movimento cabeça-olhos que tenta produzir o mesmo foco de atenção. Este trabalho constituirá uma contribuição inovadora para a comunidade científica na área da robótica bio-inspirada.

Propõe-se ainda, para trabalho futuro, uma aplicação prática que conjugue esta implementação com um sistema que possua capacidade de voo e esteja dotado de uma câmara de vídeo, por exemplo, o posicionamento automático desse elemento com capacidade de voo em função de um segundo elemento de referência fixo ou móvel que funcionará como o foco da atenção do sistema (Ver figura 5.2). A partir da utilização do módulo de saliência a executar num computador, e com recurso a uma unidade com capacidade de voo — como por exemplo o Ar.Drone da Parrot, munido de uma câmara de vídeo e equipado com um dispositivo que permite transferência de dados via WiFi — poder-se-á dar realização física à movimentação desta unidade com capacidade de voo para um ponto de referência. Dependendo da situação de aplicação, poder-se-ão definir os pesos a dar a cada mapa de conspicuidade para que se possa adaptar a implementação descrita nesta dissertação à respectiva situação. Foi já feito um trabalho preliminar com um Ar.Drone da Parrot permitindo o controlo remoto através de um *gamepad* e a recepção de imagens da câmara frontal. Para ajudar a resolver o problema da navegação recorreu-se a uma implementação proposta por [25] que estima o fluxo óptico recorrendo às imagens recebidas da câmara de vídeo e aos dados recebidos do sensor inercial.

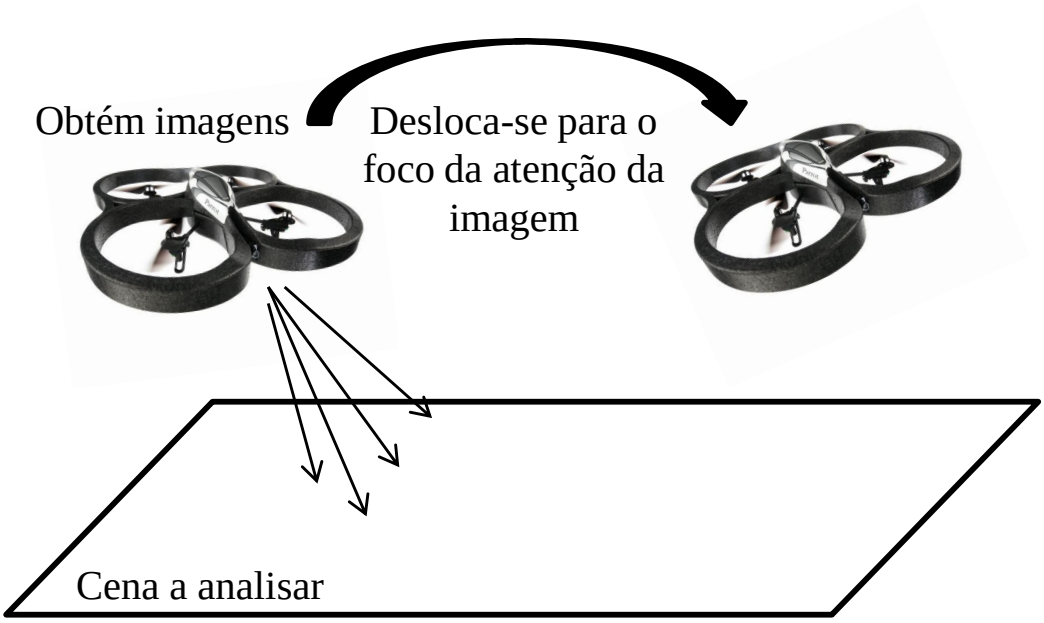

Figura 5.2: Aplicação prática para a implementação do módulo de saliência. Através de um AR.Drone [23] e de um computador equipado com uma placa de comunicação WiFi é possível que autononamente um quadricóptero explore uma área deslocando-se no espaço para os locais mais salientes. São enviadas as imagens recebidas pelo quadricóptero para um computador onde é calculado o mapa de saliência final e esse mesmo computador envia sinais de controlo para que o AR.Drone se mova no espaço para o local mais saliente.

Apêndice A

Guia de utilização do módulo de cálculo de saliência

A.1 Descrição do programa

Primeiramente é perguntado ao utilizador quais os pesos desejados para cada mapa de conspicuidade (intensidade, rivalidade entre cores e orientação). Em seguida o programa carrega uma imagem a cores da câmara de vídeo, cuja resolução é de 640×480 píxeis. Os mapas de conspicuidade e o mapa de saliência final são computados e são apresentados numa janela com um *slider* que permite ao utilizador alternar entre cada mapa de conspicuidade e o mapa de saliência final.

A.2 Estrutura geral

O programa está dividido em duas partes diferentes cada uma com a sua tarefa específica. Está escrito em C/C++ para ser usado em conjunto com biblioteca OpenCV e a API CUDA.

A.2.1 OpenCV

A imagem é carregada para um *array* cuja resolução é de 640×480 píxeis, num sistema de cores RGB com um tipo de dados *unsigned char*, sendo de seguida chamada a função em CUDA para calcular os mapas de conspicuidade e o mapa de saliência final. No final da função os mapas são copiados para variáveis em OpenCV para que sejam apresentados no ecrã.

A.2.2 CUDA

Todos os cálculos necessários para executar o algoritmo são consumados na GPU para que se obtenham os mapas de conspicuidade finais e o mapa de saliência final.

A.3 Hardware necessário

- GPU capaz de processar CUDA com pelo menos *compute capability* > 1.1

A.4 Software necessário

A.4.1 Sistema Operativo

O programa foi desenvolvido no sistema operativo Ubuntu 10.04 32 bits, ou seja, a versão pré-compilada é para ser usada neste ambiente. É possível compilar para uma versão de 64 bits, todavia o ficheiro "CMakeLists.txt" tem de ser alterado para incluir os caminhos das bibliotecas CUDA necessárias para a sua compilação.

A.4.2 Software específico CUDA

- É necessário ter instalado o pacote de desenvolvimento CUDA com uma versão superior à 3.0. Na versão pré-compilada é usada a versão 3.2.

- Biblioteca CUDA Thrust, que se encontra por defeito instalada nas versões CUDA superiores à 4.0.

A.4.3 Software adicional

- OpenCV 2.2;

- CMake > 2.6 - é necessário para efectuar a compilação caso o utilizador pretenda efectuar mudanças ao *software* ou não possua o sistema operativo Ubuntu 10.4 32 bits.

A.5 Guia de utilização do executável

- Se o utilizador não estiver a executar o sistema operativo Ubuntu 10.4 32 bits, o ficheiro "CMakeLists.txt" tem de ser alterado de forma a incluir os caminhos correctos das bibliotecas CUDA.

Executar os seguintes comandos para compilar:

- "cmake";

- "make".

Para correr o programa basta executar o comando "./Saliencia".

A.6 Guia de integração em software de terceiros

Para que seja possível integrar o cálculo dos mapas de conspicuidade e do mapa de saliência final em GPU, deve ter-se em conta os seguintes ficheiros para determinada imagem de entrada: "convolutionFFT2D.cu", "convolutionFFT2D_kernel.cu" e "SalienciaGPU.cu".

O ficheiro de cabeçalho "SalienciaGPU.h" contém a prototipagem das funções necessárias para a integração, que são as seguintes:

- `InitCuda();`

- `ShutdownCUDA();`

- ```
  SalienciaGPU(unsigned char *imagem_entrada,
  unsigned char *Conspicuidade_Intensidade,
  unsigned char *Conspicuidade_Cores,
  unsigned char *Conspicuidade_Orientacao,
  unsigned char *Mapa_Saliencia_Final,float Peso_Intensidade,
  float Peso_Cores_, float Peso_Orientacao);
  ```

A função `SalienciaGPU` recebe como parâmetros: um ponteiro para o início da imagem de entrada RGB de resolução 640×480 píxeis (`*imagem_entrada`) e ponteiros para o início dos mapas de conspicuidade (intensidade, rivalidade entre cores e orientação), que resultam do processo de cálculo do módulo de saliência (`*Conspicuidade_Intensidade`, `*Conspicuidade_Cores e *Conspicuidade_Orientacao`). Recebe também um ponteiro para o mapa de saliência final (`*Mapa_Saliencia_Final`). Os pesos a atribuir a cada mapa de conspicuidade são recebidos através das variáveis `Peso_Intensidade, Peso_Cores_` e `Peso_Orientacao`.
  ```

A.7 Guia de acréscimo de extensões

Na figura A.1 apresenta-se o fluxo de dados do módulo de saliência que inclui as principais funções apresentadas a vermelho.

Para que o utilizador possa acrescentar novas características ao programa ou manipular as já existentes - como adicionar um novo mapa de conspicuidade, existe na implementação proposta um núcleo de funções que têm de ser cumpridas para que seja seguida a linha do modelo proposto por Itti et al. [18]. Estas funções chamam os *kernels* (núcleos de processamento) que foram explicados no capítulo 3 (Implementação).

- `SepararImagens(unsigned char* RGBEntrada, float* intensidade, float* RG,float* BY);`

Nesta função é feita a separação da imagem de entrada RGB tendo em conta a intensidade, a rivalidade RG e a rivalidade BY, e é feita a conversão de *unsigned char* para *float*. No final deste *kernel* obtém-se um mapa de intensidade, um mapa RG e um mapa BY, estando todos a uma resolução 640×480 píxeis.

- `CriarPiramide(float* imgpir640_480, float* imgpir320_480, float* imgpir320_240, float* imgpir160_240, float* imgpir160_120, float* imgpir80_120, float* imgpir80_60, float* imgpir40_60, float* imgpir40_30, float* imgpir20_30, float* imgpir20_15, float* imgpir10_15, float* imgpir10_7, float* imgpir5_7, float* imgpir5_3, float* imgpir2_3, float* imgpir2_1, float* imgpir640_480_RG, float* imgpir320_480_RG, float* imgpir320_240_RG, float* imgpir160_240_RG, float* imgpir160_120_RG, float* imgpir80_120_RG, float* imgpir80_60_RG, float* imgpir40_60_RG, float* imgpir40_30_RG, float* imgpir20_30_RG, float* imgpir20_15_RG, float* imgpir10_15_RG, float* imgpir10_7_RG, float* imgpir5_7_RG, float* imgpir5_3_RG, float* imgpir2_3_RG, float* imgpir2_1_RG, float* imgpir640_480_BY, float* imgpir320_480_BY, float* imgpir320_240_BY, float* imgpir160_240_BY, float* imgpir160_120_BY, float* imgpir80_120_BY, float* imgpir80_60_BY, float* imgpir40_60_BY,`

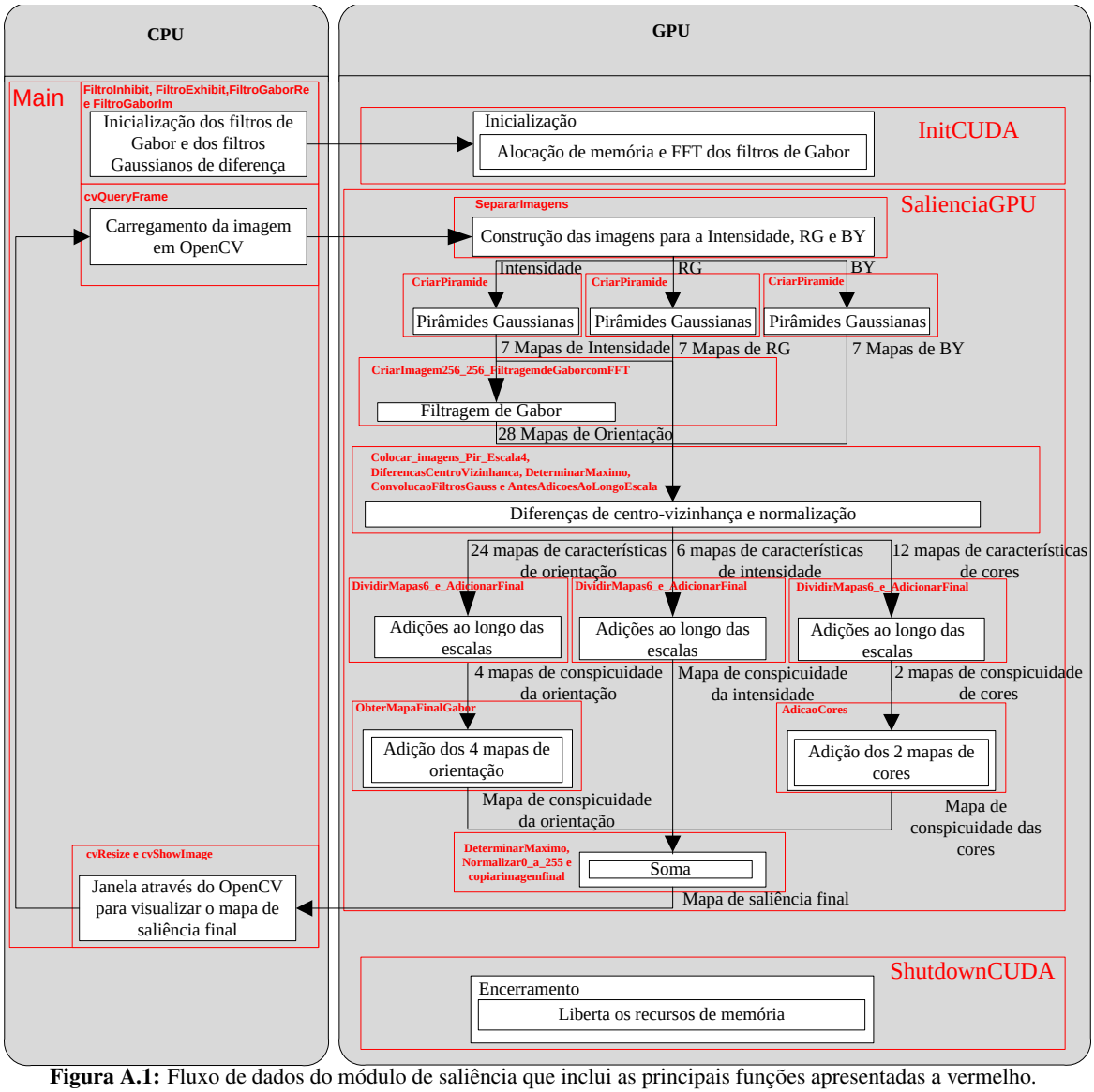

Figura A.1: Fluxo de dados do módulo de saliência que inclui as principais funções apresentadas a vermelho.

```
float* imgpir40_30_BY,float* imgpir20_30_BY,
float* imgpir20_15_BY, float* imgpir10_15_BY,
float* imgpir10_7_BY, float* imgpir5_7_BY, float* imgpir5_3_BY,
float* imgpir2_3_BY,float* imgpir2_1_BY);
```

Através da função "CriarPiramide" são criadas as pirâmides Gaussianas para a intensidade, RG e BY. Recebe como parâmetros: um ponteiro para o mapa de intensidade (`*imgpir640\_480`), ponteiros para os mapas de rivalidade entre cores (`*imgpir640\_480\_RG` e `*imgpir640\_480\_BY`) e ponteiros para as *arrays* de *floats*, com o tamanho das respectivas resoluções, que correspondem às *arrays* necessárias para as sucessivas decimações - na horizontal e na vertical - até se atingir o último nível da pirâmide (resolução 2×1 píxeis).

- `Colocar_Imagens_Pir_Escala4(float *p160_120_0,`

```
float *p80_120_escala4aux_0, float *p80_60_escala4aux_0,
float *p40_60_escala4aux_0, float *p160_120_escala4aux_0,
float *p80_60_0, float *p40_60_escala4aux1_0,
float *p80_60_escala4aux1_0, float *p40_30_0,
float *p40_30_escala4_0, float *p20_15_0,
float *p20_15_escala4_0, float *p10_7_0,
float *p10_7_escala4_0, float *p5_3_0,
float *p5_3_escala4_0, float *p2_1_0,
float *p2_1_escala4_0,
float *p160_120_45, float *p80_120_escala4aux_45,
float *p80_60_escala4aux_45, float *p40_60_escala4aux_45,
float *p160_120_escala4aux_45, float *p80_60_45,
float *p40_60_escala4aux1_45, float *p80_60_escala4aux1_45,
float *p40_30_45, float *p40_30_escala4_45,
float *p20_15_45, float *p20_15_escala4_45,
float *p10_7_45, float *p10_7_escala4_45,
float *p5_3_45, float *p5_3_escala4_45,
float *p2_1_45, float *p2_1_escala4_45,
float *p160_120_90, float *p80_120_escala4aux_90,
float *p80_60_escala4aux_90, float *p40_60_escala4aux_90,
```

```
float *p160_120_escala4aux_90, float *p80_60_90,
float *p40_60_escala4aux1_90, float *p80_60_escala4aux1_90,
float *p40_30_90, float *p40_30_escala4_90,
float *p20_15_90, float *p20_15_escala4_90,
float *p10_7_90, float *p10_7_escala4_90,
float *p5_3_90, float *p5_3_escala4_90,
float *p2_1_90, float *p2_1_escala4_90,
float *p160_120_135, float *p80_120_escala4aux_135,
float *p80_60_escala4aux_135, float *p40_60_escala4aux_135,
float *p160_120_escala4aux_135, float *p80_60_135,
float *p40_60_escala4aux1_135, float *p80_60_escala4aux1_135,
float *p40_30_135, float *p40_30_escala4_135,
float *p20_15_135, float *p20_15_escala4_135,
float *p10_7_135, float *p10_7_escala4_135,
float *p5_3_135, float *p5_3_escala4_135,
float *p2_1_135, float *p2_1_escala4_135,
float *p160_120_int, float *p80_120_escala4aux_int,
float *p80_60_escala4aux_int, float *p40_60_escala4aux_int,
float *p160_120_escala4aux_int, float *p80_60_int,
float *p40_60_escala4aux1_int, float *p80_60_escala4aux1_int,
float *p40_30_int, float *p40_30_escala4_int,
float *p20_15_int, float *p20_15_escala4_int,
float *p10_7_int, float *p10_7_escala4_int,
float *p5_3_int, float *p5_3_escala4_int,
float *p2_1_int, float *p2_1_escala4_int,
float *p160_120_RG, float *p80_120_escala4aux_RG,
float *p80_60_escala4aux_RG, float *p40_60_escala4aux_RG,
float *p160_120_escala4aux_RG, float *p80_60_RG,
float *p40_60_escala4aux1_RG, float *p80_60_escala4aux1_RG,
float *p40_30_RG, float *p40_30_escala4_RG,
float *p20_15_RG, float *p20_15_escala4_RG,
```

```
float *p10_7_RG, float *p10_7_escala4_RG,
float *p5_3_RG, float *p5_3_escala4_RG,
float *p2_1_RG, float *p2_1_escala4_RG,
float *p160_120_BY, float *p80_120_escala4aux_BY,
float *p80_60_escala4aux_BY, float *p40_60_escala4aux_BY,
float *p160_120_escala4aux_BY, float *p80_60_BY,
float *p40_60_escala4aux1_BY, float *p80_60_escala4aux1_BY,
float *p40_30_BY, float *p40_30_escala4_BY,
float *p20_15_BY, float *p20_15_escala4_BY,
float *p10_7_BY, float *p10_7_escala4_BY,
float *p5_3_BY, float *p5_3_escala4_BY,
float *p2_1_BY, float *p2_1_escala4_BY);
```

- `DiferencasCentroVizinhanca(float *p160_120_escala4,`
  `float *p80_60escala4, float *p40_30_escala4,`
  `float *p20_15_escala4, float *p10_7_escala4,`
  `float *p5_3_escala4, float *p2_1_escala4,`
  `float *escala4_cs2_5, float *escala4_cs2_6,`
  `float *escala4_cs3_6, float *escala4_cs3_7,`
  `float *escala4_cs4_7, float *escala4_cs4_8, int flag);`

- `ConvolucaoFiltrosGauss(float *ImagemEscala4,`
  `float *RespostaInhi,float *RespostaExhi);`

- `DeterminarMaximo(float *Escala4, float *maxcs);`

- `AntesAdicoesAoLongoEscala(float *escala4cs2_5,`
  `float *escala4cs2_6, float *escala4cs3_6,`
  `float *escala4cs3_7, float *escala4cs4_7,`
  `float *escala4cs4_8, float *finalcs2_5,`
  `float *finalcs2_6, float *finalcs3_6,`

```
float *finalcs3_7, float *finalcs4_7,
float *finalcs4_8, float *RespostaExhics2_5,
float *RespostaInhics2_5, float *RespostaExhics2_6,
float *RespostaInhics2_6, float *RespostaExhics3_6,
float *RespostaInhics3_6, float *RespostaExhics3_7,
float *RespostaInhics3_7, float *RespostaExhics4_7,
float *RespostaInhics4_7, float *RespostaExhics4_8,
float *RespostaInhics4_8, float maxcs2_5,
float maxcs2_6, float maxcs3_6,
float maxcs3_7, float maxcs4_7, float maxcs4_8);
```

Para se efectuarem as diferenças centro-vizinhança é necessário redimensionar todos os mapas (σ=2...8) para a escala 4 (resolução 4×30 píxeis). As diferenças absolutas são efectuadas através da função "DiferencasCentroVizinhanca" sendo esta chamada 7 vezes no total. De seguida é efectuada a convolução com os filtros de diferenças de Gauss chamando a função "ConvolucaoFiltrosGauss" para cada mapa de características. Esta função é chamada 6 vezes para os mapas de intensidade, 12 vezes para os de rivalidade entre cores e 24 vezes para os de orientação. Para calcular o máximo de cada mapa de características chama-se a função "DeterminarMaximo" para cada um deles. A função "AntesAdicoesAoLongoEscala" é chamada uma vez para os mapas de características da intensidade, duas vezes para os mapas de características da rivalidade entre cores (RG e BY) e quatro vezes para os mapas de características da orientação (θ=0º, 45º, 90º e 135º).

- `DividirMapas6_e_AdicionarFinal(float *final_cs2_5,`
  `float *final_cs2_6, float *final_cs3_6,`
  `float *final_cs3_7, float *final_cs4_7,`
  `float *final_cs4_8, float *Finalcs);`

As adições ao longo das escalas são efectuadas através da função "DividirMapas6_e_AdicionarFinal".

- `Normalizar0_a_255(float *MapaIntensidade, float *MapaCores,`
  `float *MapaOrientacao, float maxint, float maxcores,`
  `float maxori, float pesointensidade,`
  `float pesocor, float pesoorientacao);`

São calculados os máximos de cada mapa de conspicuidade através da função “DeterminarMaximo” sendo os respectivos mapas de conspicuidade normalizados posteriormente para valores entre 0 e 255 através da função “Normalizar0_a_255”.

Bibliografia

[1] J. F. Ferreira, J. Lobo, e J. Dias. Bayesian real-time perception algorithms on GPU — Real-time implementation of Bayesian models for multimodal perception using CUDA. *Journal of Real-Time Image Processing*, 26 de Fevereiro, 2010.

[2] CUDA *CUFFT Library*, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUFFT_Library.pdf

[3] CUDA 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, acedido em Julho de 2011.

[4] V. Podlozhnyuk. *FFT-based 2D convolution*, NVIDIA, 2007. Disponível em: http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/convolutionFFT2D/doc/convolutionFFT2D.pdf, acedido em Julho de 2011.

[5] CUDA GPUs, NVIDIA. Disponível em: http://developer.nvidia.com/cuda-gpus, acedido em Julho de 2011.

[6] Developer Zone, NVIDIA, 2011. Disponível em: http://developer.nvidia.com/cuda-downloads, acedido em Julho de 2011.

[7] Thrust.*A CUDA library of parallel algorithms*, 2011. Disponível em: http://code.google.com/p/thrust/, acedido em Julho de 2011.

[8] R.Tsuchiyama, T.Nakamura, T.Lizuca, A.Asahara e S.Miki. The OpenCL Programming Book. *Parallel Programming for MultiCore CPU and GPU*, 2010.

[9] OpenCV Reference Manual, 2010. Disponível em: https://picoforge.int-evry.fr/projects/svn/gpucv/opencv_doc/2.1/opencv.pdf,acedidoemJulhode2011

[10] OpenCV. *Intel Open Source Computer Vision Library Manual*,2011. Disponível em: `http://sourceforge.net/projects/opencvlibrary`.

[11] MathWorks. *Image Processing Toolbox Manual*,2011. Disponível em: `http://www.mathworks.com`.

[12] James Fung, Steve Mann. Using graphics devices in reverse: GPU-based image processing and computer vision. *2008 IEEE International Conference on Multi Media and Expo ICME 2008 Proceedings*, 2008.

[13] A. Rahman, D. Houzet, D. Pellerin, S. Marat and N. Guyader. Parallel implementation of a spatio-temporal visual saliency model. *Journal of Real-Time Image Processing Volume 6*, Number 1, 3-14, 2011.

[14] R. J. Peters and L. Itti. Applying computational tools to predict gaze direction in interactive visual environments. *ACM Transactions on Applied Perception*, 5(2), 2008.

[15] W. J.Won, S.W. Ban, and M. Lee. Real time implementation of a selective attention model for the intelligent robot with autonomous mental development. *In Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE)*, Dubrovnik, Croatia, volume 3, pages 1309–1314, 2005.

[16] A. Ude, V. Wyart, L. H. Lin, and G. Cheng. Distributed visual attention on a humanoid robot. *In Proceedings of 2005 5-th IEEE-RAS International Conference on Humanoid Robots*, pages 381–386, 2005.

[17] P. Longhurst, K. Debattista, and A. Chalmers. A gpu based saliency map for highfidelity selective rendering. *In Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH)*, pages 21–29, 2006.

[18] L. Itti, C. Koch, and E. Niebur. A model of saliency-based visual attention for rapid scene analysis. *Pattern Analysis and Machine Intelligence*, 20:1254–1259, 1998.

[19] C. Koch. e S.Ullman Shifts in Selective Visual Attention: towards the underlying neural circuitry. *Human Neurobiology*, vol.4, pp. 219-227, 1985.

[20] Dirk Walthera, Christof Koch. Modeling attention to salient proto-objects. *Neural Networks*, 19 (9). pp. 1395–1407, 2006.

[21] l.Itti e C.Koch. Feature combination strategies for saliency-based visual attention systems. *Journal of Electronic Imaging*, 10(1), 161–169, 2001.

[22] Hoi-Man Yip, Ishfaq Ahmad and Ting-Chuen Pong. An Efficient Parallel Algorithm for Computing the Gaussian Convolution of Multi-dimensional Image Data. *Journal of Supercomputing*,14, 233–255,1999.

[23] AR.Drone Parrot, 2011. Disponível em: `http://ardrone.parrot.com/parrot-ar-drone/usa/`

[24] AR.Drone open API platform, 2011. Disponível em: `https://projects.ardrone.org/`

[25] Jun-Sik Kim, Myung Hwangbo, and Takeo Kanade, *Realtime Affine-photometric KLT Feature Tracker on GPU in CUDA Framework*, The Fifth IEEE Workshop on Embedded Computer Vision in ICCV 2009, Sept 2009, pp. 1306-1311.