UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

MARCELO TEIDER LOPES

HISTOGRAMA DE GRADIENTES ORIENTADOS UTILIZANDO PROCESSAMENTO PARALELO EM GPU

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA

2015

MARCELO TEIDER LOPES

HISTOGRAMA DE GRADIENTES ORIENTADOS UTILIZANDO PROCESSAMENTO PARALELO EM GPU

Trabalho de Conclusão de Curso de Engenharia da Computação, apresentado ao Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de "Engenheiro em Computação".

Orientador: Prof. Bogdan Tomoyuki Nassu

CURITIBA

2015

AGRADECIMENTOS

Agradeço ao professor orientador Dr. Bogdan Tomoyuki Nassu pelo apoio e incentivo em me fazer concluir esse trabalho, e por ser paciente e compreensivo com os meus atrasos.

Agradeço também aos colegas de curso Lucas Campiolo Paiva, Francisco Delmar Kurpiel e Luis Guilherme Machado Camargo, por ajudas pontuais com a implementação e com a monografia.

"*I love deadlines. I like the wooshing sound they make as they fly by.*"

Douglas Adams

RESUMO

LOPES, Marcelo T.. HISTOGRAMA DE GRADIENTES ORIENTADOS UTILIZANDO PROCESSAMENTO PARALELO EM GPU. 53 f. Trabalho de Conclusão de Curso – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná. Curitiba, 2015.

Extração de descritores de características é uma tarefa fundamental para o processo de reconhecimento de padrões. Nesse projeto é apresentada uma implementação em GPU de um descritor de características de imagem baseado na distribuição local de orientação de gradientes, o descritor *Histograms of Oriented Gradients* (HOG), e esta implementação é comparada com implementações de referência pré-existentes em CPU e GPU, além de uma implementação em CPU desenvolvida para este trabalho. Foi obtida uma aceleração de 14,3 vezes em comparação com a implementação em CPU para imagens de resolução *Full HD* (1920×1080 *pixels*) e um ganho de performance superior a 30% em relação à implementação em GPU de referência.

Palavras-chave: GPU, HOG, visão computacional, extração de características

ABSTRACT

LOPES, Marcelo T.. HISTOGRAM OF ORIENTED GRADIENTS USING GPU PARALLEL PROCESSING. 53 f. Trabalho de Conclusão de Curso – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná. Curitiba, 2015.

Feature extraction is fundamental to the process of pattern recognition. In this project a GPU implementation of an image feature descriptor based on local distribution of gradient orientation, the Histograms of Oriented Gradients (HOG) descriptor, and it is compared to pre-existing reference implementations in both CPU and GPU, together with a CPU implementation developed in this project. A speedup of 14,3, compared to the CPU implementation, is achieved for Full HD (1920×1080 pixels) resolution images and a performance gain of over 30%, compared to the reference GPU implementation.

Keywords: GPU, HOG, computer vision, feature extraction

LISTA DE TABELAS

LISTA DE SIGLAS

GPU	Graphical Processing Unit
GPGPU	General Purpose Computing on Graphics Processing Units
ASIC	Application-Specific Integrated Circuit
HOG	Histograms of Oriented Gradients
CUDA	Compute Unified Device Architecture
API	Application Programming Interface
IDE	Integrated Development Environment
SM	Stream Multiprocessor
SMX	Next Generation Stream Multiprocessor
SMM	Maxwell Stream Multiprocessor
ISA	Instruction Set Architecture
SoC	System on Chip
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A área de visão computacional tem aplicações nas mais diversas áreas, como por exemplo automação de processos industriais, diagnóstico médico, preservação de documentos históricos, vigilância, identificação biométrica, digitalização automática de documentos, indexação de bancos de dados de imagens e controle e monitoração de tráfego urbano. Com o desenvolvimento da área, ela se torna cada vez mais pervasiva no cotidiano.

Técnicas de visão computacional tendem a ser computacionalmente pesadas, envolvendo a aplicação sequencial de diversas operações sobre um grande volume de dados, especialmente considerando o processamento de vídeos de alta resolução. Essas técnicas também tendem a ser altamente paralelizáveis, por operarem de forma localizada nos dados, ou seja, uma operação em uma região da imagem é independente da imagem como um todo.

Com o aumento do poder de processamento de GPUs (*Graphical Processing Units* - unidades de processamento gráfico), aliado ao desenvolvimento do uso desse tipo de processador para aplicações de propósito geral, tem-se um interesse em desenvolver sistemas de visão computacional utilizando GPUs, de forma a alcançar desempenho em tempo real em aplicações com vídeo de alta resolução. Em algumas aplicações também é interessante obter baixo consumo de potência, para a aplicação em sistemas embarcados.

Uma tarefa muito comum em visão computacional é a obtenção de *descritores de características* de regiões da imagem, de maneira a reduzir a quantidade de dados a serem processados posteriormente e também generalizar informação, de forma a aumentar a robustez do sistema à variações.

Neste trabalho é apresentada uma implementação de um descritor de imagens baseado em distribuição de orientações de gradiente de imagem, o HOG (*Histograms of Oriented Gradients* - histogramas de gradientes orientados), em uma GPU de baixo custo e baixo consumo. A implementação foi feita como uma biblioteca, de maneira a

facilitar a utilização em trabalhos posteriores. Foi obtida uma taxa de processamento superior a 20 quadros por segundo para imagens com resolução *Full HD* (1920×1080 *pixels*). Foi feita uma comparação de desempenho com uma implementação pré-existente de referência (Implementação do HOG do OpenCV), executada no mesmo processador, conseguindo uma performance consideravelmente superior.

1.1 APRESENTAÇÃO DO DOCUMENTO

O restante deste trabalho se organiza da seguinte forma. No capítulo 2 é realizado um levantamento de trabalhos relacionados extração de características utilizando GPUs. O capítulo 3 apresenta a teoria de processamento de imagens e dos algoritmos utilizados no desenvolvimento do trabalho, e o capítulo 4 a arquitetura da GPU e o modelo de programação utilizado para processamento paralelo nesta. No capítulo 5 se encontra uma descrição da biblioteca implementada e no capítulo 6 é detalhada a implementação da GPU, seguido de um relatório dos testes realizados no capítulo 7. Finalmente, encontram-se algumas considerações finais sobre o projeto, incluindo indicações de trabalhos futuros, no capítulo 8.

2 TRABALHOS RELACIONADOS

2.1 USO DE GPU PARA EXTRAÇÃO DE CARACTERÍSTICAS

Zolynsky, Braun e Berns descrevem a implementação de um descritor baseado em *Local Binary Patterns* (1), usando o *framework* Cg da NVIDIA, um precursor de GPGPU(*General-Purpose computing on Graphics Processing Units* - computação de propósito geral em unidades de processamento gráfico) focado em reutilizar o processamento gráfico de outras maneiras. Comparando a performance de uma CPU Core2Quad 2.4GHz com as GPUs GeForce 7600 GT, GeForce 8600 GT e GeForce 8800 GTS, foram obtidos tempos de execução de aproximadamente $\frac{1}{14}$ a $\frac{1}{18}$ do tempo gasto na CPU.

Park compara o desempenho de velocidade do algoritmo de detecção de bordas de Canny aplicado em conjunto com o algoritmo *Vector Coherence Mapping*, utilizado para detecçao de movimentos em vídeos (2). Os algoritmos foram executados em uma CPU, 2.4 GHz Intel Core 2, e duas GPUs, uma GeForce 8800GTS e uma placa de vídeo *onboard* 8600MGT do Apple MacBookPro. Tanto a implementação na CPU como na GPU utilizaram os mesmos parâmetros e estrutura para que as comparações fossem as mais corretas possíveis. Os resultados demonstram que a GPU *onboard* obteve uma velocidade de execução 3.15 vezes maior do que a CPU. A GeForce 8800GTS obteve uma velocidade de execução 22.96 vezes maior do que a CPU.

2.2 IMPLEMENTAÇÕES PARALELAS DO HOG

Prisacariu e Reid descrevem uma implementação multiescala do HOG em GPU (3). A implementação foi testada em uma placa de vídeo GeForce GTX 285, obtendo velocidades de execução até 67 vezes maiores do que numa CPU. Para fazer o redimensionamento necessário para a análise multiescala, eles copiam a imagem para o cache de textura da placa, usando as funções de interpolação em *hardware*. A normalização de cor é executada em um bloco de tamanho 16×16, com cada *thread* processando um *pixel*. Os gradientes são computados usando dois *kernels*, o primeiro computa os gradientes

horizontais e o segundo calcula os gradientes verticais, a magnitude e a orientação dos gradientes. No cálculo dos histogramas, no agrupamento em blocos e na normalização em blocos, cada bloco do HOG é mapeado para um bloco de *threads*.

Bauer, Brunsmann e Schlotterbeck-Match descrevem uma implementação híbrida usando FPGA, CPU e GPU (4), alcançando tempo de execução de cerca de 300 microssegundos. O pré-processamento, cálculo de gradientes e histogramas é realizado na FPGA, a normalização em blocos na CPU e o classificador SVM na GPU.

Suleiman e Sze apresentam o desenvolvimento de um ASIC(*Application-Specific Integrated Circuit* - circuito integrado de aplicação específica) capaz de executar classificação multiescala usando HOG, a uma taxa de 60 quadros por segundo em vídeo de alta definição (*Full HD*) (5).

3 HISTOGRAMAS DE GRADIENTES ORIENTADOS

Neste capítulo são primeiramente definidos alguns conceitos básicos. Em seguida, o algoritmo implementado nesse projeto é apresentado, o HOG (*Histogram of Oriented Gradients* - Histogramas de Gradientes Orientados) (6), um descritor de imagens que representa a distribuição local das orientações dos gradientes.

3.1 IMAGEM E *PIXEL*

Para os propósitos desse trabalho, uma *imagem* é definida como uma matriz bidimensional I, sendo que $I[x,y]$ representa o elemento contido na linha y e coluna x da matriz. Para respeitar a convenção normalmente utilizada, os índices x e y iniciarão em 0, e o *pixel* $I[0,0]$ será o canto superior esquerdo da imagem.

Cada elemento $I[x,y]$ de uma imagem é denominado de *pixel*, sendo que este pode conter informações de um ou mais *canais de cor*. Cada canal de cor contém um valor na faixa $[0,1]$, representando intensidade luminosa naquele canal de cor. Caso a imagem possua apenas um canal de cor, será uma imagem em escala de cinza. Neste trabalho, serão usadas imagens em escala de cinza, contendo apenas um canal, e imagens coloridas RGB, contendo os canais de cor vermelho, verde e azul.

3.2 GRADIENTES DE IMAGEM

Para uma imagem em escala de cinza I, o gradiente de imagem é uma matriz G, com as mesmas dimensões de I, onde cada ponto $G[x,y]$ representa a variação de luminosidade que ocorre em $I[x,y]$. Como a variação é bidimensional, cada ponto do gradiente de imagem será um vetor de duas dimensões, podendo ser representado tanto na forma cartesiana quanto polar. Com base nisso, temos as seguintes definições:

Gradiente horizontal de uma imagem I é o componente horizontal de G, sendo denotado por G_x e calculado seguindo a equação 3.1:

$$G_x[x,y] = I[x+1,y] - I[x-1,y] \tag{3.1}$$

Gradiente vertical de uma imagem I é o componente vertical de G, sendo denotado por G_y e calculado seguindo a equação 3.2:

$$G_y[x,y] = I[x,y+1] - I[x,y-1] \tag{3.2}$$

Magnitude de gradiente de uma imagem I é a magnitude de G, sendo denotada por G_ρ e calculada seguindo a equação 3.3:

$$G_\rho[x,y] = \sqrt{G_x[x,y]^2 + G_y[x,y]^2} \tag{3.3}$$

Orientação de gradiente de uma imagem I é a orientação (ou fase) de G, sendo denotada por G_θ e calculada seguindo a equação 3.4:

$$G_\theta[x,y] = \arctan\left(\frac{G_y[x,y]}{G_x[x,y]}\right) \tag{3.4}$$

A magnitude do gradiente de um *pixel* terá valores próximos de zero em regiões homogêneas da imagem e valores diferentes de zero nas regiões de transição de intensidade, ou seja, nas regiões de borda.

Existem várias maneiras de obter o gradiente para uma imagem colorida. Será utilizada a mesma escolhida por Dalal e Triggs no artigo que apresenta o HOG originalmente (6). Para cada *pixel* será calculado o gradiente de cada canal de cor e será escolhido o que tiver a maior magnitude.

3.3 HISTOGRAM OF ORIENTED GRADIENTS - HOG

O descritor de características HOG, descrito em (6), é composto por diversos histogramas das orientações dos gradientes da imagem, calculados em uma partição da imagem em diferentes regiões, chamadas de *células*. O cálculo do HOG é dividido em uma série de etapas: normalização de cor da imagem, cálculo dos gradientes, cálculo dos histogramas e normalização de contraste local.

3.3.1 NORMALIZAÇÃO

Primeiramente é realizada uma normalização de cor, substituindo o valor de cada canal de cor de cada *pixel* pela sua raiz quadrada.

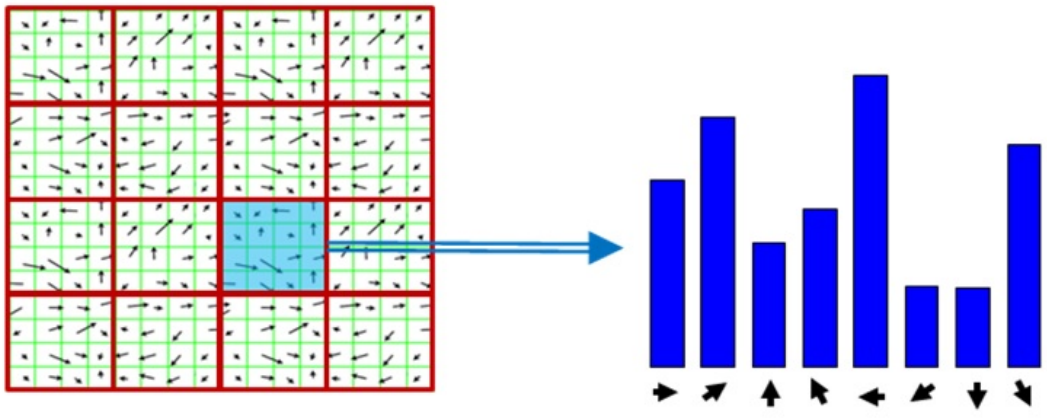
3.3.2 CÁLCULO DOS GRADIENTES

Para cada um dos três canais de cor da imagem, o gradiente horizontal e o gradiente vertical são calculados, e, a partir destes, são calculadas a magnitude e a orientação de gradiente. Para cada *pixel*, são armazenadas a magnitude e orientação do canal de cor que tiver a maior magnitude naquele *pixel*.

3.3.3 CÁLCULO DOS HISTOGRAMAS

A região a ser descrita (ou, de maneira equivalente, as matrizes de magnitude e orientação de gradiente) é particionada em uma malha de células de mesmo tamanho, e um histograma é calculado para cada célula, como demonstra a Figura 1. Em (6) são usadas 128 células de 8×8 *pixels*, em uma janela de 128×64 *pixels*, porém esse valor pode ser modificado de acordo com as necessidades do problema a ser resolvido.

Figura 1: Representação da partição em células.

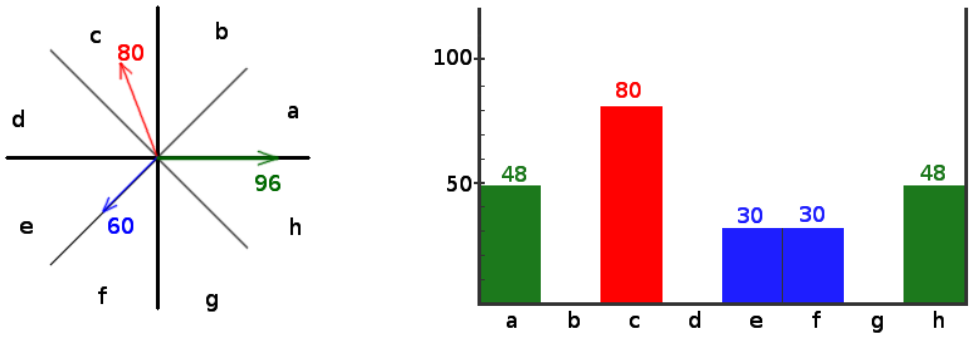


Fonte: Gil's *CV blog*, disponível em: <https://gilscvblog.wordpress.com/2013/08/18/a-short-introduction-to-descriptors/> Acesso em jun. 2015.

As classes do histograma representam faixas de ângulos da orientação do gradiente e são ponderadas pela magnitude dos mesmos. Para evitar artefatos de quantização no histograma obtido, cada gradiente é dividido entre as duas classes cujo limiar é o mais próximo do ângulo do gradiente, ponderando pela diferença entre o ângulo do gradiente

e o limiar. A Figura 2 representa três exemplos básicos da divisão em classes, quando os gradientes estão exatamente no centro de uma classe ou exatamente na divisão entre duas classes.

Figura 2: (2a) Três gradientes, onde o valor na seta representa magnitude, e (2b) o histograma respectivo obtido.



(a) Gradientes **(b)** Histograma

Fonte: Autoria Própria

Na Figura 3 está representada a divisão entre duas classes quando a orientação do gradiente está em uma posição qualquer.

As posições onde um gradiente será colocado podem ser obtidas em tempo constante, associando a cada classe um identificador e utilizando relações calculadas a partir do ângulo do gradiente e do número de classes do histograma. A largura $|C|$ da faixa de valores de uma classe do histograma, sendo n o número de classes do histograma, é dada pela equação 3.5.

$$|C| = \frac{2\pi}{n} \tag{3.5}$$

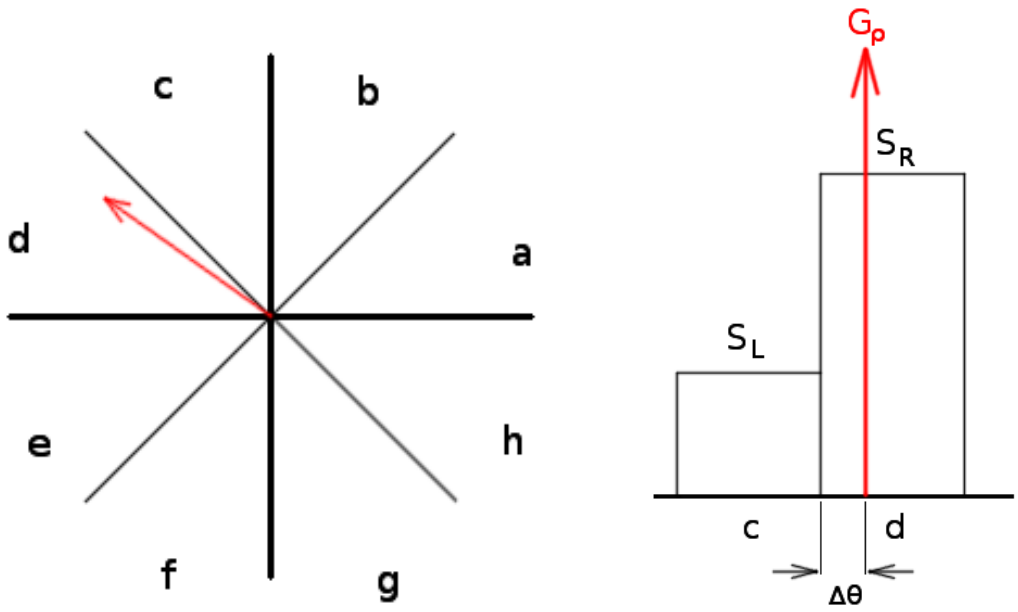
Associando a cada classe um identificador entre 0 e $n-1$ ($a=0$, $b=1$, $c=3$,...), pode-se calcular em quais classes $id_{left}(G_\theta)$ e $id_{right}(G_\theta)$ através das equações 3.6 e 3.7.

$$id_{left}(G_\theta) = \left\lfloor \frac{G_\theta}{|C|} - 0.5 \right\rfloor \bmod n \tag{3.6}$$

$$id_{right}(G_\theta) = \left(id_{left}(G_\theta) + 1\right) \bmod n \tag{3.7}$$

O valor de ângulo limiar θ_{limiar} que divide as duas classes é obtido pela equação

Figura 3: (3b) Um gradiente e (3a) as duas classes do histograma onde ele será adicionado.



(a) Gradientes **(b)** Histograma

Fonte: Autoria Própria

3.8

$$\theta_{limiar} = \left\lceil \frac{G_\theta}{|C|} - 0.5 \right\rceil \cdot |C| \tag{3.8}$$

A diferença entre G_θ e $\theta limiar$, representado por $\Delta\theta$, é obtida pela equação 3.9. Note que o valor pode ser negativo.

$$\Delta\theta = G_\theta - \theta_{limiar} \tag{3.9}$$

Finalmente, os valores somados às classes $id_{left}(G_\theta)$ e $id_{right}(G_\theta)$, respectivamente denominados de S_L e S_R, podem ser calculados pelas equações 3.10 e 3.11

$$S_L = \left(0.5 - \frac{\Delta\theta}{|C|}\right) \cdot G_\rho \tag{3.10}$$

$$S_R = \left(0.5 + \frac{\Delta\theta}{|C|}\right) \cdot G_\rho \tag{3.11}$$

Se o valor de I_θ for menor do que θ_{limiar}, o valor de $\Delta\theta$ vai ser negativo e a maior

parte do gradiente vai ser colocada na classe $id_{left}(G_\theta)$. Caso contrário, a maior parte será colocada na classe $id_{right}(G_\theta)$. O valor de $\Delta\theta$ sempre vai estar entre -0.5 e +0.5, senão o limiar mais próximo estaria entre outras duas classes.

3.3.4 GERAÇÃO DE DESCRITORES HOG

Os histogramas obtidos são agrupados em blocos de tamanho fixo e então normalizados dentro de cada bloco. O agrupamento em blocos funciona como uma janela deslizante sobre as células, com cada histograma calculado possivelmente aparecendo múltiplas vezes no descritor final.

Em (6) são apresentadas três maneiras de normalizar os blocos, todas com desempenho semelhante (para a tarefa de de localização de pedestres). Dessas, foi escolhida a *L1-sqrt*, por ser computacionalmente mais leve. Sendo H um vetor contendo os valores de cada um dos histogramas que compõem o bloco, H_i um elemento desse vetor e ε uma constante pequena maior do que zero, a equação 3.12 demonstra o cálculo da norma L1, denotada por $\|H\|$ e a equação 3.13 a normalização final resultante.

$$\|H\|_1 = \sum_i H_i \tag{3.12}$$

$$H_{normalizado} = \sqrt{\frac{H}{\|H\|_1 + \varepsilon}} \tag{3.13}$$

O descritor final é a concatenação do resultado de cada bloco, resultando em um vetor de *número de blocos* $\times$ *número de células por bloco* $\times$ *número de classes do histograma* dimensões. Para a detecção de pessoas, em (6), foi utilizada uma janela de 64×128 *pixels*, com células de 8×8 *pixels*, resultando numa malha de 8×16 células. Para o agrupamento em blocos, foram utilizados blocos de tamanho 2×2, deslocando uma célula por vez, o que resulta numa malha de $7 \times 15 = 105$ blocos. O descritor final tem, então, 105 blocos, com 4 células por bloco, e um histograma de 9 classes, totalizando em $105 \times 4 \times 9 = 3780$ dimensões.

4 CUDA: ARQUITETURA E PROGRAMAÇÃO

CUDA (*Compute Unified Device Architecture*) é um *framework* de programação paralela desenvolvido pela NVIDIA[1], envolvendo várias ferramentas necessárias para tal, como arquitetura de *hardware*, API (*Application Programming Interface* - Interface de programação de aplicações), bibliotecas, compilador e IDE (*Integrated Development Environment* - Ambiente de desenvolvimento integrado). Ele permite realizar programação de propósito geral para GPU (*Graphical Processing Unit* - Unidade de processamento gráfico) utilizando linguagens de alto nível, como C/C++, Fortran e Python (7).

Como uma GPU não é ideal para várias aplicações de propósito geral, esta é usada junto a uma CPU (*Central Processing Unit* - unidade central de processamento), que irá chamar funções para serem executadas na GPU. Essa CPU é chamada de *host*, enquanto a GPU é chamada de *device*. O fluxo padrão de execução é representado na Figura 4.

4.1 ARQUITETURA DE GPUS

A arquitetura de uma GPU difere de uma CPU principalmente por ser voltada a executar uma instrução em um conjunto de dados, ao invés de um único dado. Para isso, as GPUs fazem uma troca entre estruturas avançadas de controle de execução de código por estruturas de processamento lógico-aritimético, como demonstra a Figura 5. Um conjunto de núcleos que compartilha a mesma estrutura de controle é denominado *Stream Multiprocessor* (SM). [2]

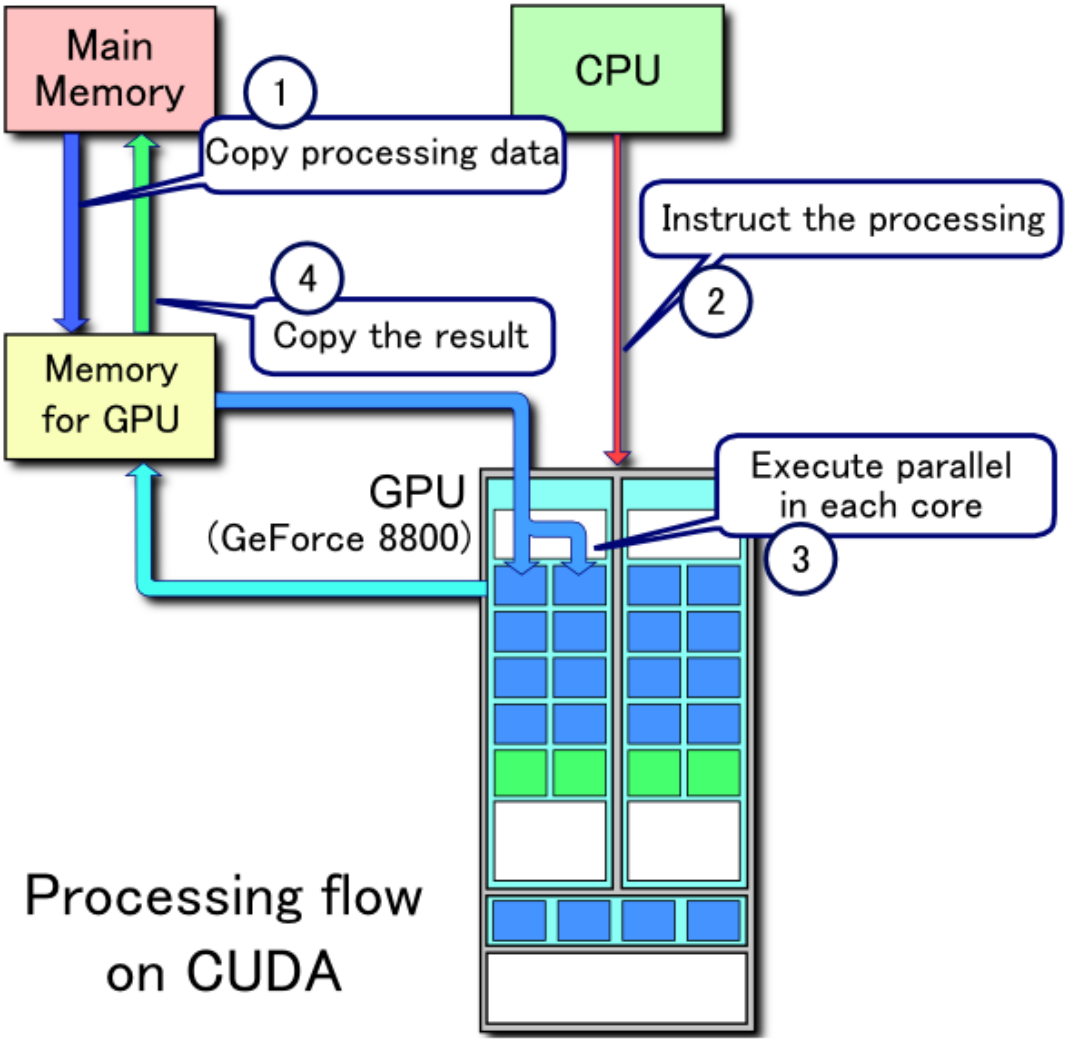
Cada SM possui um banco de registradores, que é particionado entre todos os seus núcleos, e três regiões de memória que são compartilhadas: memória compartilhada, cache de textura e cache de constantes. A Figura 6 demonstra essas regiões de memória.

Cada tipo de memória tem vantagens e desvantagens. O acesso a memória com-

[1] https://developer.nvidia.com

[2] Na arquitetura Kepler é chamado de *Next Generation Stream Multiprocessor* (SMX) e na Maxwell de *Maxwell Stream Multiprocessor* (SMM).

Figura 4: Fluxo de execução padrão usando CUDA.



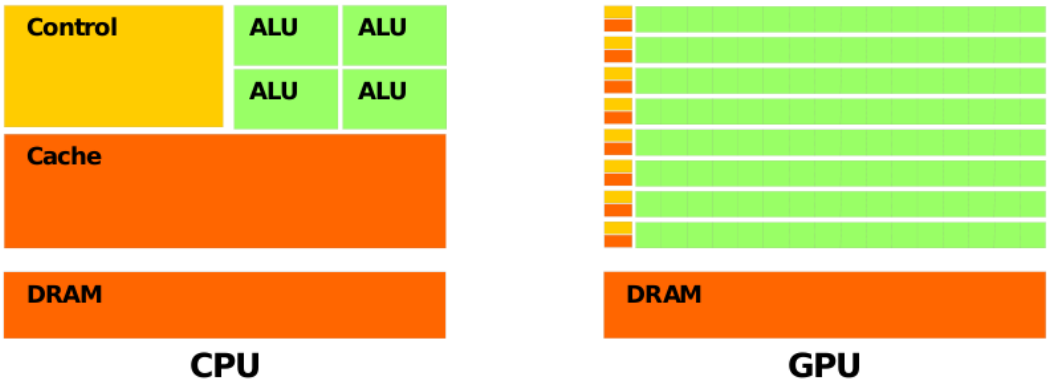
Fonte: Wikipedia, disponível em: <https://commons.wikimedia.org/wiki/File:CUDA_processing_flow_(En).PNG> Acesso em jun. 2015

partilhada é mais rápido do que o acesso a outros tipos de memória, as caches de constantes e de texturas são somente-leitura (para o processador da GPU) e a última tem estruturas especiais para fazer interpolação dos dados contidos nesta.

GPUs *dedicadas* são fisicamente separadas da CPU *host* e, portanto, possuem memória global (*Device Memory* na Figura 6) separada da memória RAM do *host*. GPUs *integradas* possuem acesso direto à memória RAM do *host*, e não é necessário copiar memória entre a CPU e a GPU, acelerando a execução.

Cada SM tem, também, uma ou mais *warps*. Dentro de um *warp* não é possível executar duas instruções diferentes ao mesmo tempo. Caso isso ocorra (por causa de execução condicional no código), o multiprocessador serializa as instruções. Em todas as

Figura 5: Diferença na arquitetura de uma CPU, à esquerda e uma GPU, à direita. As estruturas de controle de execução de código são compartilhadas entre vários núcleos.



Fonte: *Personal Blog of* Mohamed F. Ahmed, disponível em: <https://mohamedfahmed.wordpress.com/2010/05/03/cuda-computer-unified-device-architecture/> Acesso em jun. 2015

GPUs da NVIDIA capazes de realizar processamento de propósito geral, o tamanho do *warp* é 32 núcleos (8).

4.1.1 ACESSO OTIMIZADO DE MEMÓRIA

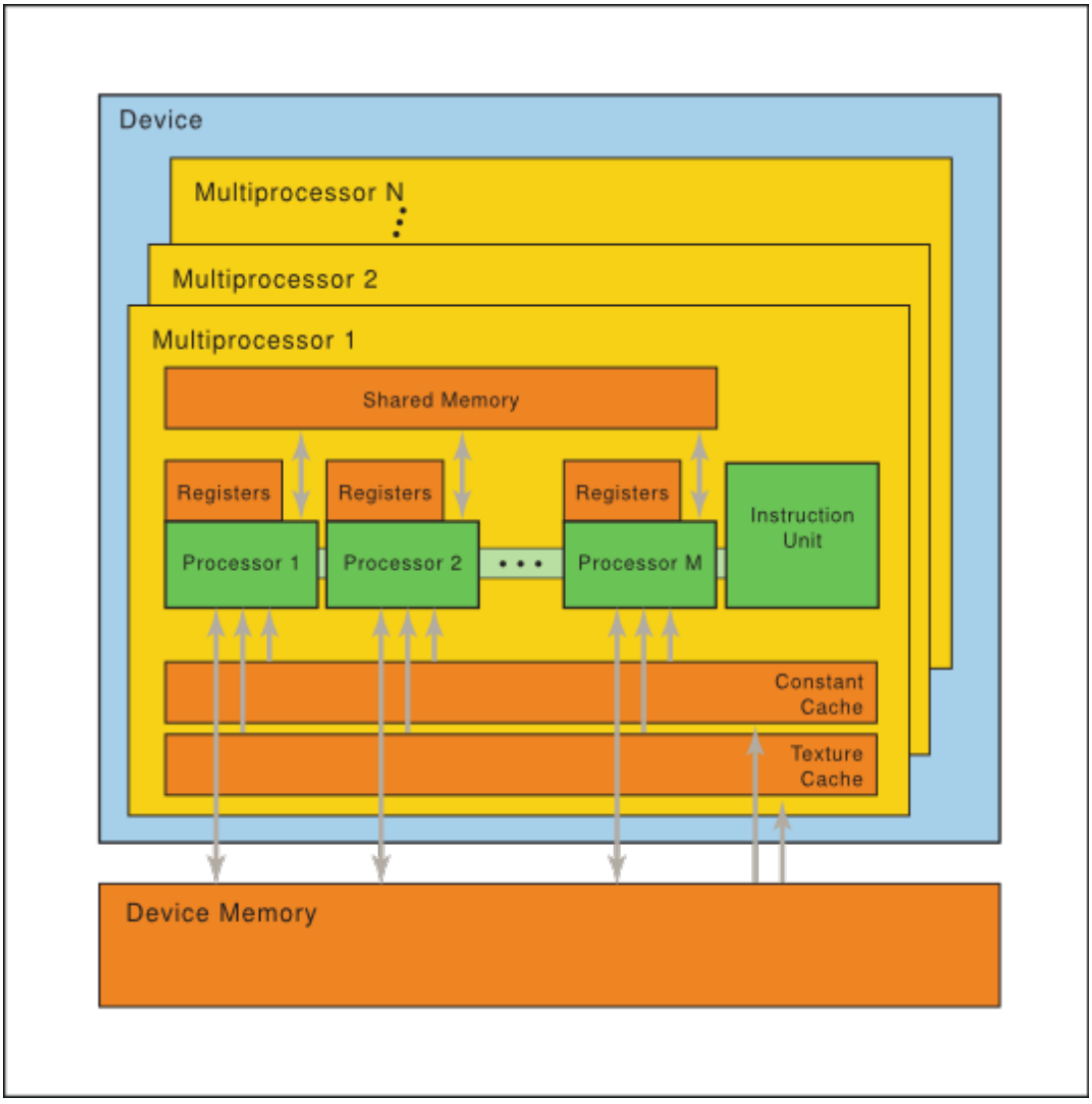
Para otimizar o acesso a memória é necessário ter em mente algumas restrições. Cada tipo de memória possui uma maneira específica de otimizar o acesso. A seguir são apresentadas as restrições necessárias para o acesso à memória global e memória compartilhada, usadas nesse trabalho.

4.1.1.1 Memória Global

A memória global é lida para o dispositivo em transações de 32, 64 ou 128 *bytes* alinhadas, ou seja, o primeiro endereço deve ser múltiplo do número de *bytes* lidos. Caso as *threads* consecutivo dentro de um *warp* acessem endereços consecutivos da memória, essa leitura é *coalescida* em uma única transação, desde que o tamanho do elemento acessado por cada *thread* seja 1, 2, 4, 8 ou 16 *bytes* (8).

Acesso não-alinhado de memória resulta em transações extras, lendo mais grupos de 32, 64 ou 128 *bytes*, causando uma perda de desempenho. Acesso de memória com endereços distantes uma da outra, dentro de um *warp*, causa grandes perdas de desempenho no acesso à memória.

Figura 6: Representação da arquitetura de memória de uma GPU.



Fonte: NVIDIA's *Parallel Thread Execution* ISA (*Instruction Set Architecture* - arquitetura de conjunto de instruções) *Documentation*, disponível em: <http://docs.nvidia.com/cuda/parallel-thread-execution/#set-of-simt-multiprocessors-with-on-chip-shared-memory> Acesso em jun. 2015

4.1.1.2 Memória Compartilhada

A memória compartilhada é particionada em 32 bancos de memória intercalados, de maneira que palavras consecutivas de 32 ou 64 *bits* (dependo da configuração do modo da memória compartilhada) são mapeados para bancos consecutivos. Em outras palavras, a primeira palavra está no primeiro banco, a segunda no segundo, e assim por diante até o trigésimo segundo banco, depois do qual o processo recomeça. Quando duas *threads* dentro de um *warp* acessam duas palavras diferentes dentro do mesmo banco, ocorre um *conflito de banco*. Caso um conflito de banco ocorra, o acesso a memória é serializado

entre as *threads* correspondentes. Duas *threads* acessando exatamente o mesmo endereço de memória não causa um conflito, resultando em um *broadcast* no caso de uma leitura e apenas uma das *threads* realizando a gravação, no caso de uma gravação de memória (qual *thread*, exatamente, é indeterminado).

4.2 PROGRAMAÇÃO EM CUDA

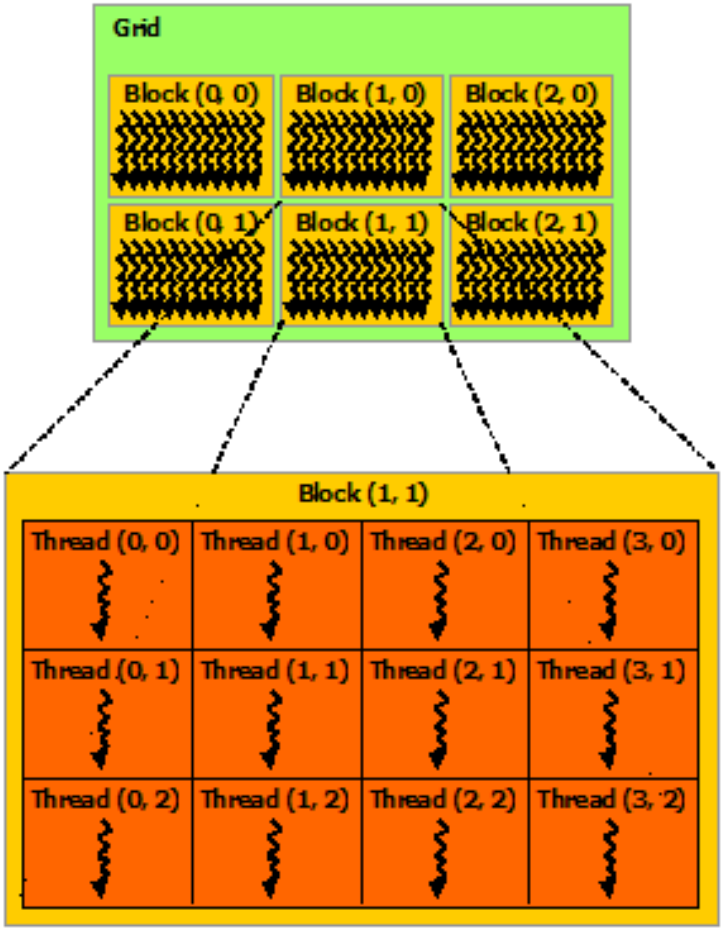
O modelo de programação usando a API de programação CUDA é baseado na arquitetura física da GPU, com sua separação em *Stream Multiprocessors* e núcleos. Uma função da GPU é executada várias vezes, dependendo dos parâmetros de configuração usados.

As funções a serem executadas na GPU são chamadas de *kernels*. O *host* configura e inicia a execução assíncrona dos *kernels*, usando dois parâmetros para escolher o número de vezes que o *kernel* será executado: número de blocos e *threads* por bloco. Cada *thread* é executada em um único núcleo e cada bloco é executado em um único SM. A divisão dos blocos em *warps* e o escalonamento destes são feitos automaticamente pelo SM, sem garantia da ordem de execução. Caso ocorra serialização dentro de um *warp*, também não há garantia da ordem de execução. A configuração do número de blocos para a execução de um *kernel* é chamada de *grid*.

De maneira a facilitar a programação para aplicações onde os dados são organizados logicamente de maneira 2D (como, por exemplo, processamento de imagens) ou 3D (como, por exemplo, simulações físicas), as *threads* e blocos podem ser configurados com uma estrutura especial representando um escalar inteiro de três dimensões (duas dimensões são alcançadas fazendo com que a terceira tenha tamanho 1). A Figura 7 representa a organização em blocos de *threads* em duas dimensões.

Cada *thread* em um bloco tem um *id* único e cada bloco em um *grid* tem, também, um *id* único. Ambos os *ids* são representados usando um escalar de três dimensões. Cada *thread* tem acesso, em tempo de execução, a quatro constantes escalares tridimensionais representando: *id* da *thread* atual dentro do bloco, *id* do bloco a qual a *thread* pertence, dimensões do tamanho do bloco e dimensões do tamanho do *grid*. Essas constantes podem ser usadas para localizar em quais posições na memória a *thread* deve operar.

Figura 7: *Grid* de blocos de *threads* de duas dimensões.



Fonte: ***CUDA C Progamming Guide***, **disponível em:**
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/#thread-hierarchy>
Acesso em jun. 2015

5 ORGANIZAÇÃO DA BIBLIOTECA

O método HOG foi implementado em uma biblioteca, de maneira a facilitar a utilização em projetos futuros. Foram incluídas na biblioteca uma implementação em GPU e outra em CPU, com uma interface comum.

No restante deste capítulo primeiramente serão apresentadas as ferramentas de *software* e *hardware* utilizadas, assim como a organização da biblioteca. Após isso, o capítulo traz uma documentação breve de cada um dos módulos.

5.1 FERRAMENTAS UTILIZADAS

5.1.1 KIT DE DESENVOLVIMENTO JETSON TK1

A placa embarcada Jetson TK1, produzida pela NVidia, foi escolhida para o projeto devido ao seu poder de processamento, baixo consumo e custo relativamente baixo se comparado com as alternativas disponíveis no mercado.

A placa utiliza o SoC (*System on Chip*) Tegra K1, um processador de 4 núcleos ARM Cortex A15, e possui uma GPU integrada com a arquitetura Kepler contendo 192 cores CUDA separados fisicamente em três blocos. Cada bloco tem sua própria cache, que é compartilhada entre seus 64 cores e pode ser utilizada para sincronizar as *threads* em execução. A placa possui uma memória RAM de 2GB com largura de 64 bits, sendo ela compartilhada entre a CPU e a GPU, não sendo necessário copiar os dados entre as partes, diferente de uma GPU discreta que possui sua própria memória dedicada separada da CPU. A memória é grande o suficiente para nos permitir trabalhar com imagem e vídeo.

Juntamente com a placa, foi utilizado o ambiente de desenvolvimento NSight Eclipse Edition da NVidia, uma versão modificada do Eclipse, preparada para compliação de código CUDA.

5.1.2 OPENCV

OpenCV[1] é uma biblioteca de código aberto para aplicações de processamento de imagem. A biblioteca é disponibilizada gratuitamente e recebe suporte por parte de sua comunidade. Ela possui um módulo com algumas funcionalidades implementadas para aplicações em GPU, e a NVidia disponibiliza uma versão compilada com otimizações especifícas para o processador Tegra K1, como parte do *framework* CUDA.

Foi utilizada a versão 2.4.9 da biblioteca para desenvolvimento e a versão otimizada da NVidia para execução no processador Tegra K1.

5.1.3 TINYXML-2

Para realizar a permanência de configuração do sistema, foi utilizada a biblioteca TinyXML-2[2], incluída como dois arquivos-fonte no projeto.

5.1.4 BOOST

Alguns detalhes de implementação foram supridos pelo uso de algumas bibliotecas do conjunto de bibliotecas Boost[3]. Mais especificamente, foram usadas as bibliotecas *thread*, *chrono*, *random* e *filesystem* para, respectivamente, fazer chamadas assíncronas de função, medir o tempo de execução de trechos de código, geração eficiente de números aleatórios e automatização de acesso aos arquivos do sistema para realização dos testes.

5.1.5 GITHUB

Foi utilizada a plataforma GitHub para realizar o versionamento de código do projeto. A biblioteca pode ser encontrada em: <https://github.com/EAPVA/GHoGLib>, o código usado para testes em <https://github.com/EAPVA/GHoGLib_Tests> e um código de exemplo utilizando a biblioteca em <https://github.com/EAPVA/GHoGLib_Example>.

5.2 ORGANIZAÇÃO DA BIBLIOTECA

A biblioteca possui as seguintes classes e interfaces:

- **HogDescriptor** - Implementação na CPU do HOG.

[1] http://opencv.org
[2] http://grinninglizard.com/tinyxml2
[3] http://boost.org

- **HogGPU** - Implementação na GPU do HOG, é uma subclasse de HogDescriptor.
- **Settings** - Funcionalidade de permanência de parâmetros de configuração.
- **Utils** - Classe estática com métodos utilitários.
- **IClassifier** - Interface para implementar um classificador. Para ser utilizada em trabalhos futuros.

Além disso, a biblioteca possui dois *headers* especiais:

- **HogCallbacks.inc** - Possui métodos de *callback* para serem usados nos métodos assíncronos.
- **GHoGLibConstants.inc** - Contém constantes usadas no sistema.

A implementação da classe HogGPU está dividida em quatro arquivos:

- **HogGPU.h** - Possui a definição da classe
- **HogGPU.cu** - Possui a implementação da parte *host* da classe.
- **HogGPU_impl.cuh** - Define as funções *kernel* usadas na classe.
- **HogGPU_imph.cu** - Implementa as funções *kernel* usadas.

5.2.1 HOG CALLBACKS

Algumas classes de *callback* são definidas para serem utilizadas em métodos assíncronos:

- **ImageCallback** - Retorna uma image processada.
- **GradientCallback** - Retorna duas matrizes, uma com as magnitudes dos gradientes e outra com as orientações.
- **DescriptorCallback** - Retorna um vetor conténdo o descritor de características.
- **ClassifyCallback** - Retorna o resultado de uma classificação (se a região de uma imagem pertence ou não a uma classe.) Para ser usado em trabalhos futuros.
- **LocateCallback** - Retorna uma lista de retângulos de objetos indentificados em uma imagem. Para ser usado em trabalhos futuros.

5.3 DESCRIÇÃO DA BIBLIOTECA

Foi dada preferência pela utilização de *containers* do OpenCV, de maneira a facilitar a integração com o OpenCV em trabalhos futuros.

5.3.1 ENUMERAÇÕES

5.3.1.1 GHOG_LIB_STATUS

Define os códigos de erro usados na biblioteca.

- **GHOG_LIB_STATUS_OK** - Nenhum erro ocorreu.
- **GHOG_LIB_STATUS_INVALID_PARAMETER_NAME** - Foi realizada a leitura ou gravação de um parâmetro inexistente.
- **GHOG_LIB_STATUS_UNKNOWN_ERROR** - Um erro desconhecido ocorreu.

5.3.2 CLASSES

5.3.2.1 HogDescriptor

Uma das classes principais do sistema, fornece as funcionalidades usadas para o cálculo do descritor, permite configurar o algoritmo, e fornece algumas funções para facilitar as tarefas de classificação e descrição, caso seja fornecido um classificador. A classe HogGPU é subclasse desta, sobrescrevendo apenas os métodos que usam a GPU, de maneira a evitar duplicação de código.

A seguir uma breve descrição de cada um dos métodos públicos da classe.

```
void alloc_buffer(cv::Size buffer_size,
        int type,
        cv::Mat& buffer);
```

Reserva uma região de memória para ser usada pela biblioteca. Não possui muita utilidade para a implementação em CPU.

```
GHOG_LIB_STATUS image_normalization(cv::Mat& image,
        ImageCallback* callback);

void image_normalization_sync(cv::Mat& image);
```

Versões assíncrona e síncrona para realizar a normalização da imagem.

```
GHOG_LIB_STATUS calc_gradient(cv::Mat input_img,
        cv::Mat& magnitude,
        cv::Mat& phase,
        GradientCallback* callback);

void calc_gradient_sync(cv::Mat input_img,
        cv::Mat& magnitude,
        cv::Mat& phase);
```

Versões assíncrona e síncrona para calcular a magnitude e a orientação de gradiente da imagem.

```
GHOG_LIB_STATUS create_descriptor(cv::Mat magnitude,
        cv::Mat phase,
        cv::Mat& descriptor,
        DescriptorCallback* callback);

void create_descriptor_sync(cv::Mat magnitude,
        cv::Mat phase,
        cv::Mat& descriptor);
```

Versões assíncrona e síncrona para obter o descritor a partir dos gradientes.

```
GHOG_LIB_STATUS classify(cv::Mat img,
        ClassifyCallback* callback);

bool classify_sync(cv::Mat img);
```

Versões assíncrona e síncrona para realizar a classificação de uma imagem. Chama os três métodos acima (normalização, cálculo de gradientes e cálculo do descritor), passando o resultado para o classificador. Requer que um classificador tenha sido passado para a classe e treinado.

```
GHOG_LIB_STATUS locate(cv::Mat img,
        cv::Rect roi,
        cv::Size window_size,
        cv::Size window_stride,
        LocateCallback* callback);
```

```
std::vector< cv::Rect > locate_sync(cv::Mat img,
        cv::Rect roi,
        cv::Size window_size,
        cv::Size window_stride);
```

Versões assíncrona e síncrona para localizar objetos de interesse em uma imagem. Não implementado no momento.

```
void load_settings(std::string filename);
```

Carrega configurações a partir de um arquivo XML(*Extensible Markup Language* - Linguagem de marcação extendível).

```
void set_classifier(IClassifier* classifier);
```

Muda o classificador utilizado.

```
GHOG_LIB_STATUS set_param(std::string param,
        std::string value);
std::string get_param(std::string param);
```

Respectivamente modifica e lê configurações da biblioteca.

5.3.2.2 HogGPU

A classe reimplementa alguns métodos da HogDescriptor. Detalhes sobre a implementação são discutidas no próximo capítulo.

```
void alloc_buffer(cv::Size buffer_size,
        int type,
        cv::Mat& buffer);
```

Reserva uma região de memória compartilhada entre a GPU e a CPU.

```
GHOG_LIB_STATUS image_normalization(cv::Mat& image,
        ImageCallback* callback);
void image_normalization_sync(cv::Mat& image);

GHOG_LIB_STATUS calc_gradient(cv::Mat input_img,
```

```
                cv::Mat& magnitude,
                cv::Mat& phase,
                GradientCallback* callback);
        void calc_gradient_sync(cv::Mat input_img,
                cv::Mat& magnitude,
                cv::Mat& phase);

        virtual GHOG_LIB_STATUS create_descriptor(cv::Mat
           magnitude,
                cv::Mat phase,
                cv::Mat& descriptor,
                DescriptorCallback* callback);
        virtual void create_descriptor_sync(cv::Mat magnitude,
                cv::Mat phase,
                cv::Mat& descriptor);
```

Reimplementação das funções de normalização, cálculo de gradientes e geração dos descritores. Chamam funções *kernel* implementadas no arquivo HogHPU_impl.cu.

5.3.2.3 HogGPU_impl.cu

Nesse arquivo estão implementados os métodos *kernel* chamados pela classe HogGPU.

```
__global__ void gamma_norm_kernel(float* img,
        int image_height,
        int image_width,
        int image_step);
```

Cada bloco de *threads* calcula a normalização de até 64 *pixels* de uma linha da imagem de entrada.

```
__global__ void gradient_kernel(float* input_img,
        float* magnitude,
        float* phase,
        int image_height,
        int image_width,
        int input_image_step,
        int magnitude_step,
        int phase_step);
```

Cada bloco de *threads* calcula o gradiente de maior magnitude de até 64 *pixels* de uma linha da imagem de entrada.

```
__global__ void histogram_kernel(float* magnitude,
        float* phase,
        float* histograms,
        int input_width,
        int input_height,
        int magnitude_step,
        int phase_step,
        int cell_row_step,
        int cell_width,
        int cell_height,
        int num_bins);
```

Cada bloco de *threads* calcula, para 64 *pixels* de uma linha da imagem de entrada, a soma parcial dos histogramas correspondentes a esses *pixels*. Essa divisão foi feita dessa maneira para otimizar o acesso à memória global.

```
__global__ void gamma_norm_kernel(float* img,
        int image_height,
        int image_width,
        int image_step);
```

Cada bloco de *threads* realiza o agrupamento e normalização de até 8 blocos do descritor, em uma linha de blocos (ou seja, ele não pega mais do que uma linha de blocos da malha).

5.3.2.4 Settings

Possui métodos para ler e escrever parâmetros de ocnfiguração do sistema em um arquivo XML, usando a biblioteca TinyXML para isso.

5.3.2.5 Utils

Criada para conter funções estáticas de utilidade para o sistema. No momento só possui um método, para fazer divisões entre números com duas dimensões:

```
        static cv::Size partition(cv::Size numerator,
                cv::Size denominator);
```

O resultado dessa operação é $\left(\frac{a_1}{a_2}, \frac{b_1}{b_2}\right)$, sendo (a_1, b_1) o numerador e (a_2, b_2) o denominador.

5.3.2.6 IClassifier

Uma interface para uma classe capaz de realizar a tarefa de classificação de um descritor de características. Define duas classes de *callback*, para serem usadas em métodos assíncronos:

```
class ClassificationCallback
{
public:
        virtual ~ClassificationCallback() = 0;
        virtual void result(cv::Mat inputs,
                cv::Mat output) = 0;
};

class TrainingCallback
{
public:
        virtual ~TrainingCallback() = 0;
        virtual void finished(cv::Mat train_data) = 0;
};
```

Além disso, define os seguintes métodos:

```
        virtual GHOG_LIB_STATUS train_async(cv::Mat train_data,
                cv::Mat expected_outputs,
                TrainingCallback* callback) = 0;
        virtual GHOG_LIB_STATUS train_sync(cv::Mat train_data,
                cv::Mat expected_outputs) = 0;
```

Versão assíncrona e síncrona para realizar o treinamento. O parâmetro *train_data* é uma matriz onde cada linha representa a entrada de uma instância de treinamento e o parâmetro *expected_outputs* é um vetor com cada elemento sendo a saída esperada correspondente a uma das entradas.

```
  virtual GHOG_LIB_STATUS classify_async(cv::Mat input,
                ClassificationCallback* callback) = 0;
        virtual cv::Mat classify_sync(cv::Mat input) = 0;
```

Versão assíncrona e síncrona para realizar a classificação. O parâmetro *input* é um vetor contendo o descritor a ser classificado e a função retorna um vetor resultante,

para o caso de classificadores que retornam um resultado multidimensional (como, por exemplo, redes neurais).

```
        virtual GHOG_LIB_STATUS load(std::string filename) = 0;
        virtual GHOG_LIB_STATUS save(std::string filename) = 0;

        virtual GHOG_LIB_STATUS set_parameter(std::string
           parameter,
                std::string value) = 0;
        virtual std::string get_parameter(std::string parameter)
           = 0;
```

Métodos para fazer a permanência de configuração e do treinamento do classificador.

6 IMPLEMENTAÇÃO DO HOG EM GPU

Neste capítulo será detalhada a implementação e invocação de cada um dos quatro *kernels* utilizados para o cálculo do descritor HOG.

Cada *kernel* foi projetado considerando blocos de *threads* de tamanho fixo, procurando realizar acesso coalescido da memória e uso de memória compartilhada para acesso a dados gerados por outras *threads*. Para a divisão do problema em blocos foi aplicada uma técnica comumente utilizada em programação CUDA, onde se fixa o tamanho do bloco de *threads* e se calcula o tamanho do *grid* a partir do tamanho do bloco e do tamanho da entrada. Foram usados blocos com uma, duas ou três dimensões conforme apropriado para a abstração da organização do problema e exclusivamente *grids* de duas dimensões. Para exemplificar o cálculo do tamanho do *grid*, será usado um tamanho de entrada de 1280×720 *pixels*

6.1 NORMALIZAÇÃO DA IMAGEM

Este é o *kernel* mais simples. Cada linha da imagem é particionada em grupos de até 64 *pixels*, cada um alocado para um bloco de *threads* diferente. Cada bloco tem tamanho $(3, 64)$, com o *id* da primeira dimensão indicando o canal de cor e o *id* da segunda dimensão indicando o *pixel* no qual a *thread* irá operar (relativo ao começo do grupo de *pixels* do bloco. O *id* do bloco indica a linha e o grupo em qual o bloco irá operar. Cada *thread* calcula a raiz quadrada do valor de entrada.

Para uma entrada de 1280×720 é necessário um *grid* de 20×720 blocos para completar o processo.

6.2 CÁLCULO DOS GRADIENTES

Neste *kernel*, novamente, o tamanho do bloco de *threads* é $(3, 64)$, com a primeira dimensão representando canal de cor e a segunda a posição relativa do *pixel*.

A execução de um bloco ocorre em duas etapas. Na primeira etapa cada *thread* calcula a magnitude e a fase do canal de cor do *pixel* correspondente, armazenando a resposta na memória compartilhada. Após todas as *threads* fazerem isso, as *threads* com *id* na primeira dimensão igual a zero verificam qual a maior magnitude entre os três canais de cor do seu *pixel* e armazenam o gradiente correspondente.

Para armazenar os valores das magnitudes e orientações dos gradientes, cada bloco precisa de 2 *buffers* $\times$ 64 *pixels* $\times$ 3 *canais de cor* $\times$ 4 *bytes por float* = 1536 *bytes* = 1.5Kb de memória compartilhada por bloco.

Novamente, para uma entrada de 1280×720 é necessário um *grid* de 20×720 blocos para completar o processo.

6.3 CÁLCULO DOS HISTOGRAMAS

Os *buffers* da magnitude e orientação do gradiente são particionados em grupos de 64 elementos, resultando num bloco de tamanho 64. Cada bloco de *threads* irá processar a soma parcial de todos os seus elementos nos histogramas correspondentes e cada *thread* em um gradiente.

A execução de um ocorre em três etapas. Na primeira cada *thread* calcula em quais classes do histograma o gradiente será dividido, e quanto será adicionado em cada classe. Todos esses resultados são armazenados em *buffers* na memória compartilhada.

Após todas as *threads* terminarem a primeira etapa, as primeiras n *threads* são selecionadas para fazer a soma parcial dos histogramas, de maneira a não causar conflitos de gravação ou leitura. Esta é a parte mais lenta da implementação, por dois motivos: o número de *threads* ativas é muito reduzido e o acesso a memória é irregular. O resultado dessa soma é armazenado num *buffer* de tamanho *número de classes do histograma* $\times$ *número de células processadas*. O número de células processadas é obtido dividindo 64 pela largura da célula.

Quando a soma parcial é concluída o *buffer* com as somas parciais é particionado entre as *threads*, que chamam uma função *thread-safe* para adicionar esses valores nas classes do histograma. Se existirem muitos conflitos de acesso existirá perda de desempenho.

Considerando a largura da célula como 8 *pixels* e 9 classes no histograma, são necessários, para armazenar os valores usados internamente pelo bloco, 4 *buffers* $\times$ 64 *elementos* $\times$ *bytes por float* + $\frac{64}{8} \times 9$ = 1328 *bytes* = 1.28Kb de memória compartilhada

por bloco.

Mais uma vez, para uma entrada de 1280×720 é necessário um *grid* de 20×720 blocos para completar o processo.

6.4 NORMALIZAÇÃO EM BLOCOS

Para fazer a normalização é usado um bloco de *threads* com dimensões (*número de classes do histograma*, *número de células por bloco do HOG*, 8). Para 9 classes no histograma e 4 células por bloco do HOG, tem-se uma dimensão de bloco de *threads* de $(9,4,8)$ *threads*. A primeira dimensão indica qual é a classe onde a *thread* vai operar, a segunda qual o histograma dentro do bloco do HOG e a terceira em qual bloco do HOG de um conjunto de 8 blocos a *thread* irá operar.

A execução de um bloco de *threads* ocorre em três etapas. Primeiramente cada *thread* carrega o valor de uma classe do histograma na memória compartilhada. Em seguida oito *threads* são escolhidas para calcular a soma de todos os valores dentro de um bloco do HOG, armazenando o resultado em um *buffer* na memória compartilhada. Finalmente, cada *thread* calcula a normalização da classe que ele tinha previamente carregado na memória e armazena no descritor final.

Para 4 células por bloco do HOG e 9 classes no histograma, é necessário um *buffer* na memória compartilhada com tamanho $9 \times 4 \times 8 + 8 = 296$ *bytes* por bloco de *threads*

Considerando uma imagem de entrada de 1280×720, com o tamanho da célula sendo 8×8 *pixels*, o tamanho do bloco sendo 2×2 células, deslocando o bloco uma célula por vez e com 9 classes no histograma, serão calculados 160×90 histogramas, agrupados em 159×89 blocos. Para realizar a normalização dessa quantidade de blocos do HOG, é necessário um *grid* de 20×89 blocos de *threads*.

7 AVALIAÇÃO EXPERIMENTAL

Foram realizadas duas baterias de testes. A primeira comparando o desempenho de tempo de execução do código escrito para GPU, o código escrito para CPU, a implementação disponível no OpenCV para CPU e a implementação do OpenCV para GPU.

Para garantir a confiabilidade da solução utilizando GPU, também foi criada uma bateria de testes comparando o resultado obtido na CPU e na GPU.

Todos os testes foram realizados utilizando otimizações, passando o parâmetro $-O3$ para o compilador *nvcc* e foram executados na placa Jetson TK1.

7.1 TESTES DE DESEMPENHO DE TEMPO DE EXECUÇÃO

Cada teste de desempenho de tempo consiste em, para uma das implementações testadas, carregar uma imagem em alta resolução na memória, escolher aleatoriamente 1000 janelas de tamanho fixo da imagem, medir e armazenar o tempo gasto para executar a implementação em cada uma das janelas e finalmente calcular a média, desvio padrão, máximo e mínimo dos valores medidos. Devido a limitações de memória e tempo de execução dos testes, o número de iterações foi reduzido para 500 nas janelas maiores do que 1280×720. Para a implementação deste trabalho é medido também o tempo gasto para executar cada uma das três funções da biblioteca necessárias para obter o descritor: normalização de imagem, cálculo dos descritores e cálculo do descritor.

Esse teste foi executado para cada uma das quatro implementações, com vários tamanhos de janela, de maneira a medir a escalabilidade de cada uma. foram escolhidas 15 janelas com tamanhos múltiplos de 16×9 células (ou seja, tamanhos múltiplos de 128×72 *pixels*).

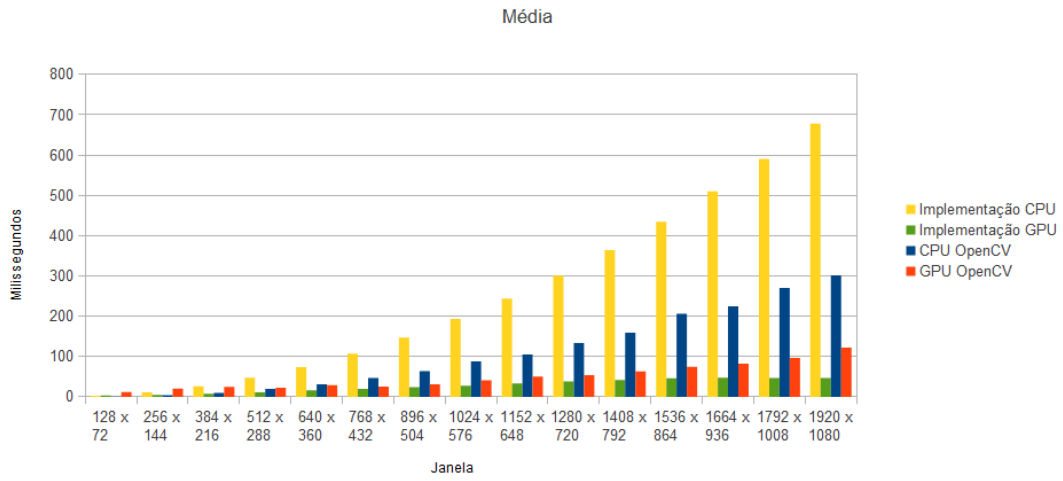
Tabela 1: Média de tempo de execução para vários tamanhos de janela, em milissegundos.

Janela	CPU	GPU	CPU OpenCV	GPU OpenCV
128 x 72	2,97	3,82	1,05	12,28
256 x 144	11,71	5,15	4,31	20,80
384 x 216	26,52	7,81	10,25	25,36
512 x 288	47,56	11,60	19,82	22,90
640 x 360	74,32	16,47	31,20	29,40
768 x 432	107,85	20,02	46,88	25,59
896 x 504	147,37	24,01	64,45	31,71
1024 x 576	193,96	27,98	88,10	41,05
1152 x 648	244,01	33,42	105,22	50,80
1280 x 720	301,37	38,47	133,53	54,32
1408 x 792	364,07	41,79	159,66	63,15
1536 x 864	435,18	46,30	206,18	74,48
1664 x 936	510,34	47,83	225,29	82,69
1792 x 1008	590,94	46,81	270,39	96,65
1920 x 1080	678,01	47,41	301,78	122,82

7.1.1 RESULTADOS

A Tabela 1 contém os valores da média de tempo gasto em uma janela para cada uma das quatro implementações e a Figura 8 um gráfico de barras correspondente.

Figura 8: Comparação de tempo de execução entre as quatro implementações

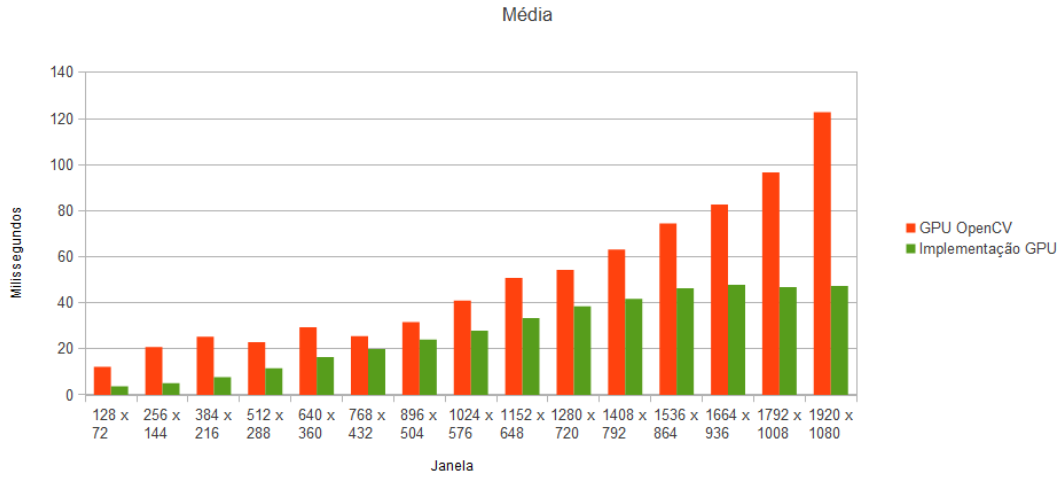


Fonte: Autoria própria.

A Figura 9 contém um gráfico de barras obtido apenas com os valores das implementações em GPU, de forma a facilitar a comparação entre as duas.

A Tabela 2 mostra a aceleração obtida (quantas vezes é possível executar um dos algoritmos na mesma quantidade de tempo que se executa o outro, ou, em outras

Figura 9: Comparação entre as implementações do OpenCV e deste trabalho



Fonte: Autoria própria.

palavras, quantos quadros por segundo podem ser processados por um enquanto o outro executa um quadro) pela implementação em GPU em comparação com a implementação em CPU deste trabalho.

A Tabela 3 mostra a aceleração obtida em comparação com o OpenCV.

A Tabela 4 contém os valores da média, desvio padrão, máximo e mínimo para a implementação em CPU deste trabalho.

A Tabela 5 contém os valores da média, desvio padrão, máximo e mínimo para a implementação em GPU deste trabalho.

A Tabela 6 contém os valores da média, desvio padrão, máximo e mínimo para a implementação em CPU do OpenCV.

A Tabela 7 contém os valores da média, desvio padrão, máximo e mínimo para a implementação em GPU do OpenCV.

A Tabela 8 contém as médias das medidas de tempo para cada uma das etapas (normalização, cálculo dos gradientes, cálculo do descritor) para a implementação da CPU e a tabela 9 essas medidas para a implementação da GPU. As tabelas 10 e 11 contém, respectivamente, os valores de desvio padrão para essas medidas.

Tabela 2: Aceleração em comparação com a implementação em CPU

Tamanho	CPU	GPU	Aceleração
128 x 72	2,97	3,82	0,78
256 x 144	11,71	5,15	2,27
384 x 216	26,52	7,81	3,40
512 x 288	47,56	11,60	4,10
640 x 360	74,32	16,47	4,51
768 x 432	107,85	20,02	5,39
896 x 504	147,37	24,01	6,14
1024 x 576	193,96	27,98	6,93
1152 x 648	244,01	33,42	7,30
1280 x 720	301,37	38,47	7,83
1408 x 792	364,07	41,79	8,71
1536 x 864	435,18	46,30	9,40
1664 x 936	510,34	47,83	10,67
1792 x 1008	590,94	46,81	12,62
1920 x 1080	678,01	47,41	14,30

Tabela 3: Aceleração em comparação com o OpenCV

Tamanho	OpenCV	GPU	Aceleração
128 x 72	12,28	3,82	3,21
256 x 144	20,80	5,15	4,04
384 x 216	25,36	7,81	3,25
512 x 288	22,90	11,60	1,97
640 x 360	29,40	16,47	1,78
768 x 432	25,59	20,02	1,28
896 x 504	31,71	24,01	1,32
1024 x 576	41,05	27,98	1,47
1152 x 648	50,80	33,42	1,52
1280 x 720	54,32	38,47	1,41
1408 x 792	63,15	41,79	1,51
1536 x 864	74,48	46,30	1,61
1664 x 936	82,69	47,83	1,73
1792 x 1008	96,65	46,81	2,06
1920 x 1080	122,82	47,41	2,59

Tabela 4: Estatísticas de tempo para a implementação em CPU deste trabalho

Tamanho	Média	Desvio Padrão	Mínimo	Máximo
128 x 72	2,97	1,03	1,86	31,52
256 x 144	11,71	1,68	9,49	45,09
384 x 216	26,52	2,55	22,69	77,88
512 x 288	47,56	3,56	41,32	104,21
640 x 360	74,32	4,47	65,30	86,86
768 x 432	107,85	5,78	97,51	124,53
896 x 504	147,37	7,83	134,46	169,86
1024 x 576	193,96	9,47	179,69	224,82
1152 x 648	244,01	10,77	226,86	281,23
1280 x 720	301,37	12,58	280,92	343,70
1408 x 792	364,07	13,52	342,80	408,08
1536 x 864	435,18	15,24	411,13	487,64
1664 x 936	510,34	16,63	483,80	564,13
1792 x 1008	590,94	18,80	562,46	564,13
1920 x 1080	678,01	17,91	649,04	742,36

Tabela 5: Estatísticas de tempo para a implementação em GPU deste trabalho

Tamanho	Média	Desvio Padrão	Mínimo	Máximo
128 x 72	3,82	0,16	3,64	5,96
256 x 144	5,15	0,23	4,81	7,04
384 x 216	7,81	0,59	2,59	22,43
512 x 288	11,60	0,58	3,26	15,73
640 x 360	16,47	0,39	3,26	20,26
768 x 432	20,02	4,94	4,88	27,06
896 x 504	24,01	7,22	6,36	42,82
1024 x 576	27,98	9,95	10,58	46,06
1152 x 648	33,42	12,59	12,79	57,82
1280 x 720	38,47	14,80	14,52	73,69
1408 x 792	41,79	15,57	17,17	72,18
1536 x 864	46,30	16,73	20,38	79,30
1664 x 936	47,83	17,28	22,98	118,59
1792 x 1008	46,81	14,96	24,48	99,85
1920 x 1080	47,41	13,41	25,62	111,37

Tabela 6: Estatísticas de tempo para a implementação em CPU do openCV

Tamanho	Média	Desvio Padrão	Mínimo	Máximo
128 x 72	1,05	0,03	1,03	1,65
256 x 144	4,31	0,85	4,15	24,62
384 x 216	10,25	0,30	10,07	18,71
512 x 288	19,82	0,10	19,46	20,55
640 x 360	31,20	0,10	30,96	31,87
768 x 432	46,88	0,39	46,60	58,27
896 x 504	64,45	4,61	62,36	102,90
1024 x 576	88,10	0,40	87,59	98,92
1152 x 648	105,22	0,53	104,77	116,40
1280 x 720	133,53	0,57	133,00	146,45
1408 x 792	159,66	0,73	159,12	172,36
1536 x 864	206,18	0,79	205,59	217,89
1664 x 936	225,29	0,85	224,63	238,68
1792 x 1008	270,39	0,94	269,62	284,55
1920 x 1080	301,78	0,98	300,97	315,64

Tabela 7: Estatísticas de tempo para a implementação em GPU do openCV

Tamanho	Média	Desvio Padrão	Mínimo	Máximo
128 x 72	12,28	1,00	7,12	20,17
256 x 144	20,80	5,18	7,44	53,32
384 x 216	25,36	5,51	9,75	49,26
512 x 288	22,90	7,00	9,91	62,19
640 x 360	29,40	10,17	13,50	90,31
768 x 432	25,59	4,32	16,00	47,27
896 x 504	31,71	6,72	22,01	79,91
1024 x 576	41,05	6,41	28,90	59,90
1152 x 648	50,80	9,98	36,43	97,16
1280 x 720	54,32	5,80	45,48	74,55
1408 x 792	63,15	7,72	54,73	87,95
1536 x 864	74,48	1,98	64,16	94,58
1664 x 936	82,69	3,95	74,59	99,38
1792 x 1008	96,65	1,91	86,63	138,56
1920 x 1080	122,82	15,10	99,05	158,41

Tabela 8: Tempo gasto em cada uma das etapas na CPU.

Tamanho	Normalização	Gradientes	Descritor	Total
128 x 72	0,18	1,72	1,07	2,97
	5,99%	57,91%	36,10%	100,00%
256 x 144	0,72	6,63	4,37	11,71
	6,11%	56,59%	37,30%	100,00%
384 x 216	1,62	14,83	10,07	26,52
	6,10%	55,93%	37,97%	100,00%
512 x 288	2,88	26,18	18,50	47,56
	6,06%	55,05%	38,89%	100,00%
640 x 360	4,49	40,71	29,12	74,32
	6,05%	54,77%	39,18%	100,00%
768 x 432	6,46	58,66	42,72	107,85
	5,99%	54,39%	39,61%	100,00%
896 x 504	8,79	79,89	58,69	147,37
	5,96%	54,21%	39,83%	100,00%
1024 x 576	11,46	103,87	78,63	193,96
	5,91%	53,55%	40,54%	100,00%
1152 x 648	14,50	130,86	98,66	244,01
	5,94%	53,63%	40,43%	100,00%
1280 x 720	17,89	161,79	121,69	301,37
	5,94%	53,69%	40,38%	100,00%
1408 x 792	21,64	194,86	147,57	364,07
	5,94%	53,52%	40,53%	100,00%
1536 x 864	25,74	232,82	176,62	435,18
	5,92%	53,50%	40,59%	100,00%
1664 x 936	30,21	273,19	206,95	510,34
	5,92%	53,53%	40,55%	100,00%
1792 x 1008	35,09	316,22	239,63	590,94
	5,94%	53,51%	40,55%	100,00%
1920 x 1080	40,31	361,99	275,71	678,01
	5,95%	53,39%	40,66%	100,00%

Tabela 9: Tempo gasto em cada uma das etapas na GPU.

Tamanho	Normalização	Gradientes	Descritor	Total
128 x 72	0,87	0,88	2,07	3,82
	22,68%	23,12%	54,21%	100,00%
256 x 144	1,00	1,10	3,05	5,15
	19,36%	21,40%	59,24%	100,00%
384 x 216	1,33	1,53	4,95	7,81
	17,01%	19,60%	63,40%	100,00%
512 x 288	1,82	2,14	7,64	11,60
	15,68%	18,46%	65,85%	100,00%
640 x 360	2,43	2,92	11,12	16,47
	14,77%	17,75%	67,48%	100,00%
768 x 432	2,98	3,56	13,48	20,02
	14,88%	17,79%	67,33%	100,00%
896 x 504	3,61	4,29	16,10	24,01
	15,02%	17,89%	67,09%	100,00%
1024 x 576	4,16	4,96	18,86	27,98
	14,86%	17,73%	67,41%	100,00%
1152 x 648	4,91	5,99	22,51	33,42
	14,71%	17,94%	67,35%	100,00%
1280 x 720	5,37	6,85	26,25	38,47
	13,96%	17,80%	68,24%	100,00%
1408 x 792	5,42	7,40	28,97	41,79
	12,98%	17,70%	69,32%	100,00%
1536 x 864	5,78	7,83	32,69	46,30
	12,48%	16,92%	70,61%	100,00%
1664 x 936	5,85	8,04	33,95	47,83
	12,22%	16,80%	70,98%	100,00%
1792 x 1008	5,94	7,91	32,95	46,81
	12,70%	16,90%	70,40%	100,00%
1920 x 1080	6,21	7,99	33,21	47,41
	13,11%	16,85%	70,04%	100,00%

Tabela 10: Desvio padrão para cada etapa da implementação em CPU

Tamanho	Normalização	Gradientes	Descritor
128 x 72	0,01	0,38	0,93
256 x 144	0,02	1,42	0,58
384 x 216	0,04	2,46	0,26
512 x 288	0,23	3,32	0,36
640 x 360	0,07	4,25	0,37
768 x 432	0,11	5,45	0,57
896 x 504	0,12	7,26	0,86
1024 x 576	0,12	8,57	1,34
1152 x 648	0,14	9,79	1,44
1280 x 720	0,15	11,39	1,64
1408 x 792	0,34	12,14	1,86
1536 x 864	0,19	13,55	2,13
1664 x 936	0,20	14,65	2,43
1792 x 1008	0,21	16,57	2,64
1920 x 1080	0,22	15,54	2,82

Tabela 11: Desvio padrão para cada etapa da implementação em GPU

Tamanho	Normalização	Gradientes	Descritor
128 x 72	0,14	0,05	0,16
256 x 144	0,11	0,06	0,14
384 x 216	0,14	0,09	0,44
512 x 288	0,22	0,13	0,37
640 x 360	0,12	0,15	0,23
768 x 432	0,65	0,93	3,50
896 x 504	0,96	1,33	5,08
1024 x 576	1,33	1,82	6,94
1152 x 648	1,70	2,32	8,94
1280 x 720	1,84	2,66	10,98
1408 x 792	1,79	2,96	11,93
1536 x 864	1,76	2,96	12,95
1664 x 936	1,50	2,97	13,76
1792 x 1008	1,30	2,70	11,83
1920 x 1080	1,40	2,45	10,52

Tabela 12: Velocidade de execução de cada implementação.

Tamanho	CPU	GPU	CPU OpenCV	GPU OpenCV
128 x 72	3,10	2,41	8,75	0,75
256 x 144	3,15	7,16	8,56	1,77
384 x 216	3,13	10,62	8,09	3,27
512 x 288	3,10	12,71	7,44	6,44
640 x 360	3,10	13,99	7,39	7,84
768 x 432	3,08	16,57	7,08	12,97
896 x 504	3,06	18,81	7,01	14,24
1024 x 576	3,04	21,08	6,70	14,37
1152 x 648	3,06	22,34	7,09	14,69
1280 x 720	3,06	23,95	6,90	16,97
1408 x 792	3,06	26,68	6,98	17,66
1536 x 864	3,05	28,66	6,44	17,82
1664 x 936	3,05	32,56	6,91	18,83
1792 x 1008	3,06	38,59	6,68	18,69
1920 x 1080	3,06	43,74	6,87	16,88

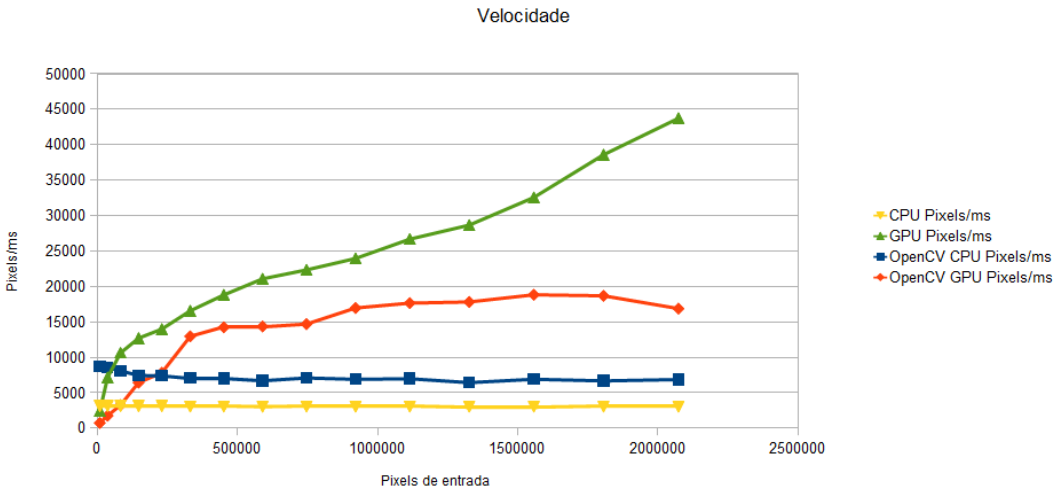
7.1.2 AVALIAÇÃO DOS RESULTADOS

Em primeiro lugar pode-se notar que a aceleração é maior para janelas de tamanho maior, tanto para a implementação deste trabalho quanto para a implementação em GPU do OpenCV. Para observar melhor esse resultado foi calculado um valor de *megapixels* processados por segundo, a partir dos valores da tabela 1, de forma a ter um indicativo de velocidade de execução para cada tamanho de janela. Esses valores estão presentes na Tabela 12 e um gráfico correspondente na Figura 10. Nota-se que a velocidade da implementação em CPU deste trabalho é aproximadamente constante, a da implementação em CPU do OpenCV decresce com o tamanho da entrada, provavelmente devido a *overheads* no processamento de muitos histogramas ou muitos blocos, a velocidade da implementação em GPU do OpenCV cresce até estabilizar em torno dos 19 *megapixels* por segundo, e a da implementação em GPU deste trabalho cresce continuamente até o maior tamanho de janela usado.

Um resultado que pode ser obtido através dos dados das tabelas 4, 5, 6 e 7 é que as implementações em GPU tendem a apresentar uma variação maior de tempo de execução, especialmente considerando os valores máximo e mínimo.

Comparando as implementações em CPU e GPU deste trabalho, nota-se que a etapa mais custosa na CPU é o cálculo dos gradientes, que é a etapa que apresenta maior aceleração na implementação da GPU e que a etapa mais custosa para a GPU é a terceira, cálculo dos histogramas e normalização em blocos. Tamém pode-se notar que a terceira

Figura 10: Comparação de velocidade entre as quatro implementações



Fonte: Autoria própria.

etapa é a que apresenta maior variação de tempo de execução na GPU. Isso se deve a problemas relacionados com acesso à memória compartilhada no cálculo dos histogramas e com acesso à memória global para agrupar os histogramas em blocos.

Quando é realizada a soma parcial no *kernel* do cálculo dos histogramas o acesso à memória compartilhada é bem irregular, porque depende da orientação dos gradientes sendo processados. Sendo assim, o acesso causa muitos conflitos de banco imprevisíveis, serializando a execução do *kernel* e reduzindo a eficiência do *kernel*.

No agrupamento em blocos o acesso da memória não é localizado, cada bloco precisa de histogramas que estão localizados em posições distantes um do outro. Caso as transações fossem organizadas pela entrada, com cada *thread* sendo responsável por copiar uma classe do histograma para várias posições do descritor final, ao invés de ser responsável por calcular um elemento do descritor, a leitura poderia ser coalescida, mas a gravação teria o problema de acessar posições distantes de memória.

7.2 TESTES DE COMPARAÇÃO DE RESULTADO ENTRE CPU E GPU

Para comparar as duas implementações, foi realizado um procedimento semelhante ao usado nos testes de desempenho de tempo. Uma imagem é carregada na memória, são escolhidas 1000 janelas de tamanho fixo (1280×640 *pixels*), e, para cada janela utilizada, são obtidos dois descritores, um utilizando a implementação em CPU e outro usando a implementação em GPU e é calculada a distância euclidiana entre os dois, nor-

Tabela 13: Distancia euclidiana normalizada entre os descritores da CPU e da GPU

Tamanho	Normalização	Magnitude	Orientação	Descritor
Média	0,000	0,000	0,070	0,000
Desvio Padrão	0,000	0,000	0,021	0,000
Mínimo	0,000	0,000	0,027	0,000
Máximo	0,000	0,000	0,468	0,000

malizada pela soma das magnitudes dos descritores, considerando cada descritor como um vetor n-dimensional. Isso gera um valor de erro igual a 0 caso os dois descritores sejam iguais e um valor igual a 1 caso eles tenham uma orientação simétrica, independente da magnitude de cada um. Finalmente, a média, desvio padrão, mínimo e máximo são calculados para os 1000 valores obtidos. O mesmo procedimento é repetido para a normalização de imagem e para o cálculo dos gradientes, considerando a concatenação dos valores dos *pixels* como um vetor, no caso da normalização, e obtendo dois vetores, um com a concatenação das magnitudes e outro com a concatenação das orientações, para os gradientes. A Tabela 13 contém os resultados desses testes.

Com esse teste, se comprovou um erro de 0% nos resultados, ou seja, os dois descritores são equivalentes. As diferenças obtidas para o cálculo da orientação do gradiente foram investigadas, e chegou-se a conclusão de que a função usada para obter o arco tangente retorna, em alguns casos, $+\pi$ na CPU e $-\pi$ na GPU e em alguns outros o contrário. Esse resultado parcial é indiferente para o cálculo dos histogramas.

8 CONCLUSÕES

Métodos para obter o descritor HOG foram implementados na CPU e na GPU, obtendo uma aceleração na GPU de até 14,3 vezes se comparado com a implementação sequencial feita na CPU. Também foi realizada uma comparação com uma implementação de referência já existente, a do OpenCV, obtendo um ganho de desempenho de 28% a 159% para janelas de tamanho 512×288 ou maiores, e superiores a 200% para janelas menores do que isso.

A implementação foi projetada para e testada em um processador de baixo custo e baixo consumo, conseguindo velocidades de execução de aproximadamente 21 quadros por segundo para janelas de tamanho 1920×1080 *pixels* e aproximadamente 26 quadros por segundo para janelas de tamanho 1280×720 *pixels*.

Para chegar a tais ganhos, foi essencial projetar a implementação da GPU considerando os diversos tipos de memória disponíveis para uso e seus padrões de acesso otimizados. Isso resultou em ganhos muito expressivos para as tarefas de normalização da imagem e cálculo dos gradientes, sendo menos expressivos para o cálculo dos histogramas e normalização em blocos. Isso se deve à necessidade de acessos irregulares de memória no cálculo dos histogramas, e também a acessos de memória não contínua no agrupamento em blocos. Mesmo assim a implementação em GPU dessas etapas do método continua a ser mais eficiente do que a de CPU para janelas de tamanho grande.

8.1 TRABALHOS FUTUROS

Aqui são citadas algumas propostas de continuidade do trabalho realizado.

Implementar um classificador

No escopo desse trabalho não foi implementado um classificador para ser usado junto com o descritor. Muitos classificadores são altamente paralelizáveis e podem aproveitar a arquitetura da GPU para acelerar a sua velocidade de execução. Seria interessante

implementar a etapa de classificação na biblioteca, tanto para tornar a implementação mais completa quanto para reproduzir os testes de tempo por etapa, para verificar qual parcela de tempo é gasta na tarefa de classificação.

Adicionar opções de configuração da biblioteca

Alguns parâmetros da biblioteca podem ser configurados. No momento podem ser escolhidos o tamanho da célula, tamanho do bloco, deslocamento do bloco, tamanho da janela de detecção (em dimensões da malha de células) e número de classes no histograma. Seria interessante permitir ao usuário escolher outras opções, como, por exemplo, método de normalização do bloco, método de obtenção do gradiente de uma imagem colorida, método de normalização de cor da imagem, permitir escolher o método de alocação do *buffer* de memória para permitir a utilização com outras GPUs, entre outros.

Tornar a implementação mais flexível

Alguns parâmetros da biblioteca foram fixados em valores específicos, e seria interessante que eles fossem configuráveis de acordo com a necessidade. Em especial os tamanhos dos blocos de *threads* foram escolhdos com o processador Tegra K1 em mente, e deveriam poder ser modificados mais facilmente. Também seria interessante permitir outros formatos de imagem de entrada, como imagens em escala de cinza. Essas mudanças possivelmente exigiriam um reprojetamento de alguns dos *kernels*.

Melhorar a API da biblioteca

A API da biblioteca precisa ser atualizada para levar em conta alguns desenvolvimentos da implementação. Em especial a função de cálculo do descritor pode ser separada em cálculo de histogramas e agrupamento/normalização em blocos.

REFERÊNCIAS

1 ZOLYNSKY, G.; BRAUN, T.; BERNS, K. Local binary pattern based texture analysis in real time using a graphics processing unit. **VDIBERICHT**, VDI; 1999, v. 2012, p. 321, 2008.

2 PARK, S. I. et al. Low-cost, high-speed computer vision using nvidia's cuda architecture. In: IEEE. **Applied Imagery Pattern Recognition Workshop, 2008. AIPR'08. 37th IEEE**. [S.l.], 2008. p. 1–7.

3 PRISACARIU, V. A.; REID, I. **fastHOG - a real-time GPU implementation of HOG Technical Report No. 2310/09**. 2009.

4 BAUER, S.; BRUNSMANN, U.; SCHLOTTERBECK-MATCH, S. Fpga implementaion of a hog-based pedestrian recognition system. In: **Tagungsband zum Workshop der Multiprojekt-Chip-Gruppe**. [s.n.], 2009. p. 49–58. Disponível em: <http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2009/Bauer09-FIO.pdf>.

5 SULEIMAN, A.; SZE, V. Energy-efficient hog-based object detection ad 1080hd 60 fps with multi-scale support. In: **Signal Processing Systems (SiPS), 2014 IEEE Workshop on**. [S.l.: s.n.], 2014. p. 1–6.

6 DALAL, N.; TRIGGS, B. Histograms of oriented gradients for human detection. In: IEEE. **Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on**. [S.l.], 2005. v. 1, p. 886–893.

7 LANGUAGE Solutions. 2015. Disponível em: <https://developer.nvidia.com/language-solutions>. Acesso em: 30 de julho de 2015.

8 CUDA C Programming Guide. 2015. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Acesso em: 30 de julho de 2015.