High Level Computer Vision using OpenCV

Maurício Marengoni and Denise Stringhini
Faculdade de Computação e Informática
Universidade Presbiteriana Mackenzie
São Paulo, Brazil
dstring,mmarengoni@mackenzie.br

***Abstract*—This paper presents some more advanced topics in image processing and computer vision, such as Principal Components Analysis, Matching Techniques, Machine Learning Techniques, Tracking and Optical Flow and Parallel Computer Vision using CUDA. These concepts will be presented using the openCV library, which is a free computer vision library for C/C++ programmers available for Windows, Linux MacOS and Android platforms. These topics will be covered considering not only theoretical aspects but practical examples will be presented in order to understand how and when to use each of them.**

***Keywords*-openCV; parallel computer vision; pattern recognition; computer vision;**

I. Introduction

The contents of this article follows from a previous tutorial presented at the 2008 edition of SIBGRAPI as an introductory course in computer vision using openCV. The idea in this article is to study more advanced topics related to pattern recognition, computer vision, and parallel computer vision.

The material presented here will cover topics beyond the basic image processing and low-level vision techniques, so students who already took a first course in computer vision can advance their knowledge and first year graduate students can see more specific computer vision applications. Students will also learn how to design these applications using a free computer vision library (openCV) and how to create an application using processors available in GPU, which comes in many computer desktops and notebooks today.

We start this tutorial presenting a simple review for some important functions in low level vision, mainly to present students with the openCV functions for each of them.

II. Low Level Vision - Spatial Domain

This section presents a summary of some basic functions used in low-level vision processes. These functions are presented without details, so the reader is advised to check more background and details at [1], [2], [7].

A. Threshold

One of the basic fucntions in low-level vision is the threshold function. This function takes an image and a value K for the threshold and computes an output image based on the threshold type being used. The openCV function is:

```
double cvThreshold(src, res, K, max, type);
```

In this case *src* is the source image, *res* is the resulting image, K is the threshold value, *max* is the maximum value expected in the source image and *type* defines the way the output image is computed, if the intensity in the source image is greater than K then:

- CV_THRESH_BINARY: res = max, otherwise 0.
- CV_THRESH_BINARY_INV: res = 0, otherwise max.
- CV_THRESH_TRUNC: res = max, otherwise src.
- CV_THRESH_TOZERO_INV: res = 0, otherwise src.
- CV_THRESH_TOZERO: res = src, otherwise 0.

B. Histograms

A histogram is a function that computes frequency. When considering image histograms the pixel's frequency for each intensity level or intensity range is computed. It can be used for operations, such as comparisons, segmentation, compression, etc. The openCV call for computing a histogram is given by:

```
void cvCalcHist(src, his, add, mask);
```

Here, *src* is a single channel input image, *his* is the histogram computed for the given image, *add* and *mask* are optionals, *add* by default is 0, so it erases *his* before computing the histogram. If *add* is set to 1 the histogram accumulates the values for more than one image. *Mask* is a Boolean matrix used to set the part of the input image where the histogram should be computed.

C. Filtering

Filtering in the spatial domain has several applications and can be used to smooth images and remove noise or even to enhance transitions and help finding edges. OpenCV has two basic functions that can be used for these tasks:

```
void cvSmooth(src, res, type, p1, p2, p3, p4);
void cvFilter2D(src, res, kernel, center);
```

In both functions *src* and *res* are the input and output images, *type* defines the filter type being used, it can be:CV_BLUR (mean filter), CV_BLUR_NO_SCALE (summation), CV_MEDIAN (median value), CV_GAUSSIAN (gaussian filter) and CV_BILATERAL (bilateral 3x3). The parameters *p1* to *p4* are related to the filter type and should be checked at the reference manual [7]. The *kernel* is the filter's mask and *center* is the point used as the mask's center, by default it is *cvPoint(-1,-1)*. The kernel can be defined as:

```
CvMat *filt;
int side  = 3;
int total =256;
double kernel[] = { 1,  4, 6, 4, 1,
```

978-0-7695-4549-3/11 $26.00 © 2011 IEEE
DOI 10.1109/SIBGRAPI-T.2011.11

```
     4, 16,24,16, 4,
     6, 24,36,24, 6,
     4, 16,24,16, 4,
     1,  4, 6, 4, 1,};
...
for(int i=0; i<side*side; i++){
kernel[i]=(1./total)*kernel[i];
}
filt=cvCreateMatHeader(side,side,CV_64FC1);
cvSetData(filt,kernel,side*8);
...
cvFilter2D(src,res,filt,cvPoint(-1,-1));
```

D. Fourier Trasform

Another way to filter an image is performing a convolution operation in frequency domain. The first step is to convert an image to the frequency domain using the Fourier Transform. The Fourier transform requires by itself a set of other operations, the source image has to be embedded in a larger image and padded with zeros in order to avoid boundary effects. The call in openCV for the discrete Fourier transform is:

```
cvDFT(src,res,CV_DXT_FORWARD,
      complexInput->height);
```

Src is the input image already embedded and padded with zeros and *res* is the image converted to frequency domain. The whole code sequence is presented bellow:

```
int dft_Y, dft_X;
CvMat* fft, tmp;
IplImage *im_Real,*im_Imagi,*complexInput;
...
dft_Y=cvGetOptimalDFTSize(src->height-1);
dft_X=cvGetOptimalDFTSize(src->width-1);
fft=cvCreateMat(dft_Y,dft_X,CV_64FC2);
im_Real=cvCreateImage(cvSize(dft_X,dft_Y),
                      IPL_DEPTH_64F,1);
im_Imagi=cvCreateImage(cvSize(dft_X,dft_Y),
                      IPL_DEPTH_64F,1);
complexInput=cvCreateImage(cvGetSize(src),
                      IPL_DEPTH_64F,2);
cvGetSubRect(fft,&tmp,cvRect(0,0,
         src->width,src->height));
cvCopy(complexInput,&tmp,NULL);
if(fft->cols > src->width){
   cvGetSubRect(fft,&tmp,cvRect(src->width,
      0,fft->cols-src->width,src->height));
   cvZero(&tmp);
}
cvDFT(fft,fft,CV_DXT_FORWARD,
        complexInput->height);
```

E. Finding Edges

A typical way to compute edges in an image is to find local variations in intensity levels (gradients). Some of the openCV functions to compute these derivatives and return an image with the possible boundaries are:

```
cvSobel(src, res, xorder, yorder, mask);
cvLaplace(src, res, mask);
cvCanny(src, res, low, high, mask);
```

The *cvSobel* function computes the derivatives in the X and/or Y directions, the *xorder* and *yorder* values determine the derivative's order (at least one of them should be more than 0 and at most 2), *src* is the input image (8-bit) and *res* the output image (16-bit) and *mask* is the filter's size (supported values are 1, 3 , 5 and 7). A special value for the mask size is given by CV_SCHARR which computes the Scharr filter for a 3x3 mask. The *cvLaplace* function computes the second derivatives of an image. The *cvCanny* function implements the Canny edge detector, a well known technique to find edges. In this method if a pixel has a value above the *high* threshold them it belongs to an edge, if the value is bellow the *low* threshold it does not belong to an edge and if it is in between it belongs to an edge if it is connected to a pixel with value above the high threshold. The *mask* here is similar to the *cvSobel*.

F. Basic Segmentation

Segmentation is an important operation in image processing and computer vision because it groups pixels into more meaningful regions which can be used for other tasks such as recognition. There are several ways to segment an image, three openCV methods for segmentation will be presented.

Region Growing

The region growing method is based on similarity among pixels.The idea is that given a seed point (a pixel in the image) the method checks the neighbors of this point. If a neighbor has a similar intensity value the method marks the neighbor as being from the same region and uses the marked pixels as new seeds. The region growing method is implemented in openCV using the *cvFloodFill* function:

```
cvFloodFill(src,seed,value,low,high,comp,flags,
            mask);
```

As usual, *src* is the source image, *seed* is the point where the method will start the region growing process, *value* is the value used to mark the region, *low* and *high* are the limits within the neighbor can be accepted as belonging to the region, *comp* is a connected components structure that holds region's statistics, *flags* defines a set of paramenters for the method (the connectivity, 4 or 8, the relative value for filling the region, etc), and *mask*, which can be used as output image if provided.

Background Subtraction

When one needs to segment an image to check what have changed over time a simple operation is called background subtraction. There are several ways to perform this task, the simplest is just to perform an image difference. In this case all background will be marked with 0 or a low value, which can be set to zero using the threshold function. Opencv performs the subtraction using the *cvAbsDiff* function:

```
cvAbsDiff(src1, src2, res);
```

In this case *src1* and *src2* are the source images and *res* is given by $res = |src1 - src2|$.

Watershed Segmentation

The watershed algorithm is a split-merge region method, it first uses intensity levels to find groups of small regions and

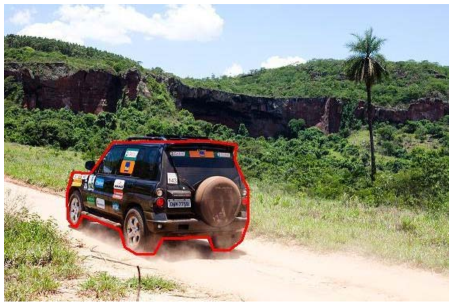

Fig. 1. An ideal contour around an object of interest.

after that uses a set of markers in a mask to group the small regions into areas with similar properties, which are computed from each marker. The watershed method is computed in openCV using the function:

```
cvWatershed(src, markers);
```

Again, *src* is the input image, *markers* is an image with the same size as *src* where the regions are marked, so the method can group smaller regions and segment the image properly.

III. Image Matching

Pattern recognition is an area of computer vision where one tries to find if there is a known pattern in a given image. The process that checks in the image for a pattern's possible location is called image matching. Matching techniques are the simplest way to do pattern recognition. In this section it will present two different approaches for matching: contour matching and template matching.

A. Contour Matching

After segmentation an image is composed by a set of groups of pixels where each group represents a region. This segmented data can be transformed into a compact way that facilitates the region's description and help to compare and match with a given pattern [8]. When thinking about a contour one considers an object of interest and a line surrounding it as an ideal contour (althought not necessarily a closed line), as shown in Figure 1. A contour can be represented in different ways, two of the most common ways to represent a contour are polygons and Freeman chain codes [1].

A boundary can be represented by a connected sequence of straight line segments, each with a specified length and direction. This type of representation is called a Freeman chain code, it uses a 4- or 8-connectivity and the segment's direction is coded following a numbering code. Figure 2 presents these two methods, on top the Freeman chain code with the green marker to set the starting point and the numbering code at the right. On bottom the polygon given by the set of points at the corners of the polygon.

In openCV contours are usually computed from a binary image where it is easier to define contrast. The function used to compute the contours is:

```
int cvFindContours(src,sto,first,header,mode,
                   method);
```

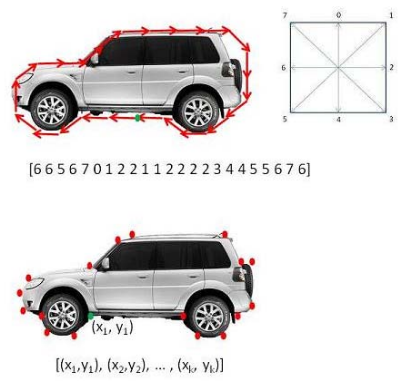

Fig. 2. Contour representation: on top the Freeman chain code and on bottom a set of points for a polygon.

The input image, *src*, is a binary image, *sto* is a memory location where the contours will be written *first* is the first contours in a sequence, *header* gives the size of the object being retrieved and can be either *sizeof(CvContour)* or *sizeof(CvChain)*, depending on the method, *mode* defines the data structure used to store the contours and can be:

- CV_RETR_EXTERNAL: retrieves only the most external contour.
- CV_RETR_LIST: retrieves all contours as a list where the first element is most internal contour.
- CV_RETR_CCOMP: retrieves all contours and organize them as a two level hierarchy, one level with external contours and another with internal contours.
- CV_RETR_TREE: retrieves all contours and organize them as a tree structure with the outer contour as a root.

The *method* parameter is related to the way openCV stores the contour in memory. There are five ways to do this task:

- CV_CHAIN_CODE: uses the Freeman chain code.
- CV_CHAIN_APPROX_NONE: uses the points determined by the Freeman chain code.
- CV_CHAIN_APPROX_SIMPLE: compresses the Freeman chain code and returns the ending points.
- CV_CHAIN_APPROX_TC89_L1 or CV_CHAIN_APPROX_TC89_KCOS: uses a special chain approximation algorithm.
- CV_LINK_RUNS: can only be used with CV_RETR_LIST and represents contours as links horizontal segments.

The returning value of the method is the total number of contours found. One important step before calling *cvFindContours* is the input image binarization. This process requires some filtering afterwards to clean the image, otherwise too many contours might be found. Figure 3 shows a simple and clean binary image (on the left side) and the contours found (on the right) there are a total of 7 contours on the right image. Figure 4 shows the same process applied to a complex (meaning real) image, the contour structure, in this case, has

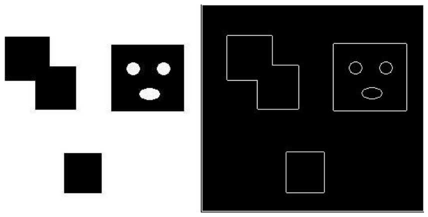

Fig. 3. A simple binary image (left) and the contours found using cvFindContours (right). The function found 7 contours.

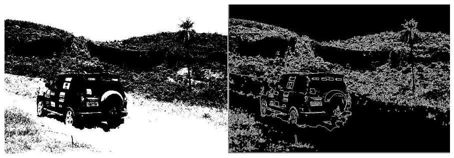

Fig. 4. A complex binary image (left) and the contours found using cvFindContours (right). The function found 3790 contours.

3790 contours, each small point or region in a complex image will have a contour around it.

Once a set of contours are found it is possible to go through each contour using the data structure defined by *CvSeq* (see [1] for details) and select the desired contour for a contour matching. There are several ways to compare two contours using openCV, it depends on the pattern contour and the contour found in an image. For instance, Freeman chain codes are translation invariant, if the length used to compute the Freeman chain is scaled up or down, the chain itself can be also scale invariant, and if the number coding is rotated accordingly, then it can also be rotation invariant. Other ways to compare contours are:

- Compute the contour's perimeter and compare the results: *cvContourPerimeter(contour).*
- Compute the contour's area and compare the results: *cvContourArea(contour, slice).*
- Compute the contour's moments and compare the results: *cvContourMoments(contour, moments).*

In these functions, *contour* is a contour structure computed by *cvFindContours*, a *slice* in the *cvContourArea* is a parameter that allow to compute the area of only part of the contour, otherwise specify *slice* as CV_WHOLE_SEQ. The *moments* argument in the *cvContourMoments* is a data structure type named *CvMoments* and should be previously allocated.

A region moment representation is an interpretation of a binary image as a probability density function of a 2D random variable. This random variable has properties that can be described using statistical characteristics which are called moments [9]. A moment of order (p, q) is depended on scaling, translation and rotation, in digitized images it can be computed as shown in Equation 1

$$\mathbf{m_{pq}} = \sum_{\mathbf{i}=-\infty}^{\infty} \sum_{\mathbf{j}=-\infty}^{\infty} \mathbf{i^p j^q f(i,j)} \tag{1}$$

where i and j are the coordinates of the points inside the region and $f(i, j)$ is the intensity level of the binary image at position i and j. A moment can be translation invariant if it is computed based on orthogonal axis passing in the region's center of mass, these moments are called central moments and they are given by Equation 2

$$\mu_{\mathbf{pq}} = \sum_{\mathbf{i}=-\infty}^{\infty} \sum_{\mathbf{j}=-\infty}^{\infty} \mathbf{(i - x_c)^p (j - y_c)^q f(i,j)} \tag{2}$$

where x_c and y_c are the region's center of mass coordinates.

B. Template Matching

A simple way to do pattern recognition is just to verify if a pattern shows up in the source image. The process makes an exhaustive search of the template in the source image and, in this case, marks each position where the pattern is found. The search might be slow for large source images, but the process is simple and gives good results. The openCV function that performs template matching is:

```
cvMatchTemplate(src,template,result,method);
```

In this function *src* is the source image, *template* is the pattern to be found, *result* is an image showing the results for the matching and *method* describes the way the template matching is performed.

There are six different methods to perform template matching in openCV:

- CV_TM_SQDIFF: this method computes the square difference between the template and the source image, a best match in this case has a 0 value. Equation 3 is used to compute the matching.

$$\mathbf{R_{sdiff}} = \sum_{\mathbf{ij}} \mathbf{[t(i,j) - f(x+i, y+j)]^2} \tag{3}$$

- CV_TM_CCORR: this method makes a correlation between the template and the source image at each position, a best match in this case has a large value (not necessarily the maximum, depending on noise). Equation 4 is used to compute the correlation.

$$\mathbf{R_{ccorr}} = \sum_{\mathbf{ij}} \mathbf{[t(i,j) * f(x+i, y+j)]^2} \tag{4}$$

- CV_TM_CCOEFF: this method is called correlation coefficient matching and it makes the correlation between the template subtracted from its mean and the source image subtracted from its mean at each position, considering only the template's size. Notice that the correlation might give poor results when the image energy, $\sum f^2(x, y)$, varies with position. Equation 5 is used to compute the correlation coefficient, a good match in this case has also a large value.

$$\mathbf{R_{ccoeff}} = \sum_{\mathbf{ij}} \mathbf{[(t(i,j) - \bar{t}) * (f(x+i, y+j) - \bar{f}]^2} \tag{5}$$

The other three methods available for *cvMatchTemplate* are normalized methods, selected by

CV_TM_SQDIFF_NORMED, CV_TM_CCORR_NORMED and CV_TM_CCOEFF_NORMED. These methods work better when there are lighting differences between the template and the source image [10]. In all cases the normalization is performed by dividing the method by the normalization factor presented in Equation 6.

$$\mathbf{NORM} = \sqrt{\sum_{\mathbf{ij}} \mathbf{t^2(i,j)} * \mathbf{f^2(x+i,y+j)}} \qquad (6)$$

Figure 5 shows an application of *cvMatchTemplate*, the template used is shown on the image's top right and the colored squares show where each method found the template. Notice the problem presented by the correlation method.

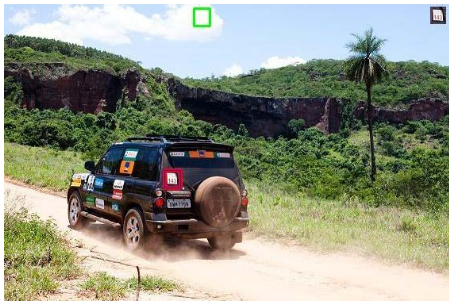

Fig. 5. The result of the cvMatchTemplate using the template at the top right. Notice the problem with the correlation method (the green square).

Figure 6 shows the images obtained by the six methods used by cvMatchTemplate. At the top, from left to right, the squared difference, correlation and correlation coefficient and at the bottom their correspondent normalized results.

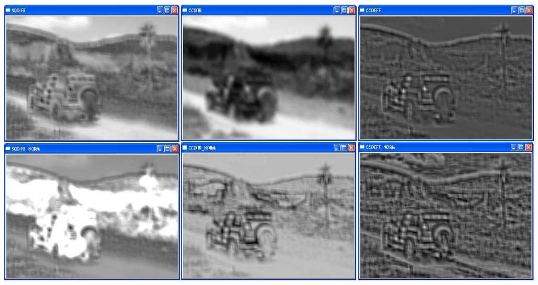

Fig. 6. The output of cvMatchTemplate for each available method. Top row presents squared difference, correlation and correlation coefficient respectively. The bottom row presents the correspondent normalized method.

IV. Pattern Recognition using Machine Learning Techniques

Learning is related to changes in an adaptive system, such that the system can perform similar tasks more efficiently throughout time. When thinking about pattern recognition the adaptive system has to be capable to decide a class that a new image belongs to among the patterns it had seen so far. Figure 7 shows the idea of learning in pattern recognition, if a system receives a set of images having cars, faces and houses, learning means that the system is capable do define regions separating these images and, if a new image is presented the system can classify it into one of the possible classes.

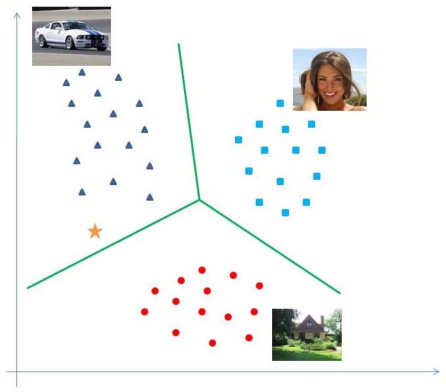

Fig. 7. The idea of machine learning applied to pattern recognition.

Before going through an specific method let's first present some important concepts related to the data and some common functions used for the majority of the learning methods available in openCV.

Learning methods in computer vision can either work with images themselves or they can work with a vector of features extracted from these images. For instance Figure 8 shows the two representations for some digits extracted from license plates. On the top part are the images themselves and the coding used for feature extraction and on the bottom part the features as a vector representation for each image.

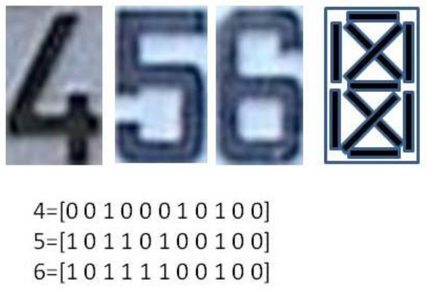

Fig. 8. The two representations for learning, the images themselves and a feature vector extracted from the images.

The data presented for the learning methods in openCV have always to be in a matrix form. If the image is being used to represent the data, all images must have the same dimensions and all images must be converted into vectors, so, if there are K images available and each image has size M by N, the data matrix in openCV will have size $K * MN$. If the images are represented by feature vectors representing P features, then the data matrix will have size $K * P$. Thus the data preparatiom can be decomposed as:

1) Clean the images: if necessary filter the images to remove or reduce noise.
2) Adjust contrast: if necessary have all images with a contrast throughout the whole range, so to avoid dark or light images.
3) Resize the images: all images must have the same dimensions if they will be used as training data.
4) Convert format: images should be converted as vectors.
5) Mix the data: in order to avoid any kind of bias, data from different classes should be mixed in the data matrix.

These steps should also be followed even if the data is a feature vector, eliminating the steps that are not required such as resizing.

The learning methods for openCV have specific calls for each phase of learning and using the system:

- *method→train*: to train the system using the method.
- *method→save*: to save the learned system into a file as an xml file format or yml file format.
- *method→load*: to load the method learned from the file passed.
- *method→predict*:for a new prediction in a new data.

OpenCv has several machine learning techniques used for pattern recognition, a sample will be presented here.

A. K-means

The K-means is not exactly a pattern recognition method but a clustering method that tries to find clusters in the data. One way to perform pattern recognition using K-means is first find the clusters and then look for the pattern in each cluster.

If the number of clusters in the data is not known (K), the user has to make a guess and adjust this number later, if necessary. The method interactively searchs for K centers of mass inside the data. This is one of the most used techniques for grouping [1]. The method itself works as follows:

1) Input: data and the number of groups (K).
2) Ramdonly defines K centers of mass in the data.
3) Associate each entrance with a center of mass.
4) Compute the new center of mass.
5) If the error between the previous centers of masses and the new ones is bellow a certain limit accept and terminate, otherwise return to step 3.

The K-means algorithms has the following problems:

- There is no warranty it will find the best centers of mass for each group, but it will always converge,
- The user has to provide the number of clusters, the method will not give the best number of clusters in the data.
- The method assumes that the covariance in the grouping space is not important or that it was normalized.

Figure 9 shows at left an image with random points (right) the seven clusters found by the K-means method (left).

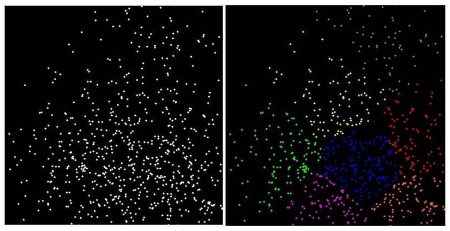

Fig. 9. At left an image with random points and at right the seven clusters found.

The openCV function for the Kmeans algorithms is given by the method:

```
cvKMeans2(data,clusters,result,criteria).
```

In this case *data* is a multidimensional input matrix used for training, *clusters* is the number of clusters given by the user *result* is a matrix where each cluster is marked and *criteria* is a termination criteria passed to the K-means algorithm.

B. Decision Trees

Another machine learning technique is called decision tree. In a decision tree each node represents a random variable and the arcs leaving the node represents the possible outcomes for the random variable. The simplest decision tree is called a binary decision tree where the outcomes for the random variables are true or false. A binary decision tree can be used either for classification or for regression [11].

Usually a binary decision tree uses as data a vector of features extracted from images. Figure 10 shows an idea for digit classification based on features extracted from digit images. The top node checks if the feature *top bar* exists in the image being classified. If it exists then the decision tree checks for the *bottom bar* otherwise it checks for the *middle bar* and so on. The learning process for decision trees tries to find the most discrimanating features in order to create the simplest tree possible.

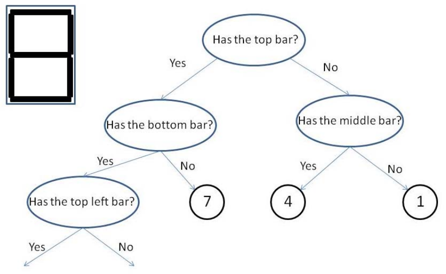

Fig. 10. A model for a decision tree on image classification.

The features are evaluated using three possible measures:

- Entropy:

$$\mathbf{E(feature) = -p * \log_2(p) - (1-p) * \log_2(1-p)} \tag{7}$$

- Gini index:

$$\mathbf{G(feature) = 1 - \sum_{i=1}^{m} f_i^2} \tag{8}$$

- Misclassification:

$$\mathbf{M(feature) = 1 - max(P(w_j))} \tag{9}$$

where p represents the fraction of the data that has the feature. In the Gini index f_i is the fraction of items labelled with value i using the feature. In the misclassification $P(w_j)$ represents the fraction of patterns in class w_j using the feature.

The results interpretation in a decision tree is straightforward which makes it a method used widely. Most implementations for this method allow missing information which is almost always the case when working with feature vectors. The method in openCV for decision trees is called as:

```
decTrees.train(data,type,truevalues,features,
               points,vartypes,missing,
               parameters);
```

In this method *data* is the training data, *type* indicates a row or column matrix, *truevalues* is a vector indicating the expected (true) classification of the data. The remaining parameters are optional, *features* if not 0 is a mask indicating the features that must be used in the training process, *points*, if not 0, indicates the points that must be considered in the training process, *vartypes* indicates if the variables are categorical or ordered and finally *parameters* is used to set tree parameters like depth, missing data, etc (see more details in the openCV's sample directory in the mushroom.cpp file).

C. K Nearest Neighbor

The K Nearest Neighbor (KNN) is a comparative method for pattern recognition. There is actually no learning in this method, the idea, as shown in Figgure 11, is to compare any new image with all images available in the data and take the K nearest images (for any small and possibly odd value of K) and label the new image with the label of the majority among the K. The nearest can be computed using any measurement method, such as Euclidean or Mahalanobis distance.

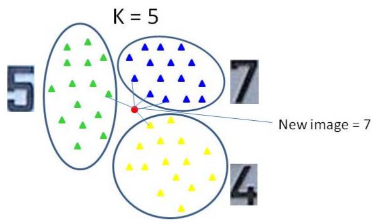

Fig. 11. An example of K nearest neighbor with K=5. The new image (red point) is compared with all points and the 5 closest are kept (three blue, one green and one yellow). The new point is classified as blue which is a 7.

The K nearest neighbor requires only three things: the value of K, the labeled images (data) and the metric to measure closeness. Notice that the KNN method does not process data until a new unlabelled image is presented to the system and once the new image is labelled the computation is discarded. The code in openCV for the KNN method is:

```
CvKNearest knn(traindata,trainclasses,sample,
               regression,K);
response=knn.find_nearest(samples,results,
               neighbors,n_responses,distances);
```

In this case *traindata* is the set of images, the *trainclasses* are the labels for each image, *sample* is a vector with the samples in the data that have to be considered, can be set to 0, *regression* is a boolean variable and indicates if the method is being used for regression or not, and *K* is the number of neighbors to bconsider. In the *find_nearest* method *samples* are the new images to be classified, *results* are the answers for the samples (the labels), *neighbors* is a vector indicating the neighbors for each image in samples, *n_responses* is the class for each neighbor found and *distances* is a vector with the distance for each neighbor found.

D. Boosting

The idea of Boosting is to combine simple decision rules into an accurate decision system. In openCV the method implemented for Boosting is called AdaBoost [1] which uses a simple (weak) classifier (a binary decision tree) to implement the Boosting method. A weak classifier is a decision method which is capable to classify things with a probability just above chance (50%). The combination of several weak classifiers leads to a decision system that typically performs at or near the state of the art [1], performance can be improved only in more specifically designed systems. OpenCV implements four types of boosting for its AdaBoost algorithm:

- DISCRETE: for a discrete data.
- REAL: uses confidence predictions and works well with categorical data.
- LOGIT: works well with regression.
- GENTLE: works better in regression data because it puts less weight on outliers.

The idea in AdaBoost is to find a set of simple binary decision trees where each tree has a weight and the classification of each one of these trees can be combined into a single and strong classification system, as presented in Figure 12. The method in openCV for AdaBoost is called as:

```
boost.train(data,type,trvalues,features,points,
            vartypes,missing,parameters,update);
```

The parameters, as expected, are similar to the parameters used by the binary decision trees. The AdaBoost has an extra parameter, *update*, which is optional and set, by default, as 0, in this case it trains a new set of weak classifiers from scratch. The parameters passed to boost.train are the type of boosting used, the number of weak classifiers used, the limit of percentage that a point should have to be considered, the depth of the tree, etc. Check the example presented in the file letter_recog.cpp in the samples c directory of the openCV foulder and openCV book [1] or the openCV manual [7].

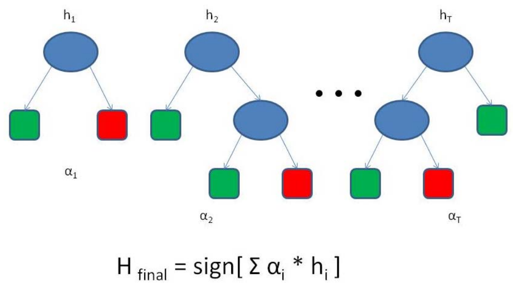

Fig. 12. The idea of AdaBoost and boosting in general. Combine simple classifiers using weights to get a strong classifier.

E. Principal Component Analysis

Principal Component Analysis (PCA) is a technique based on linear algebra which is used to find patterns in high dimensional data. PCA is applied in several areas, such as neuroscience, data compression and computer vision. PCA can also be used to reduce the number of dimensions in data with no (or little) loss of information. The dimensions in this reduced dimensions are called the principal components and they can be used for object recognition like faces [13] in images.

The analysis using principal components is similar to the K Nearest Neighbors. A new image is compared to all images used in the training, but instead of using the whole image the comparison is performed using only the reduced dimensions (principal components) of the images. Again, a measure method is required to make the comparisons. The algorithm for PCA follows these steps:

1) Get the data.
2) Subtract the mean.
3) Compute the covariance matrix.
4) Compute the eigenvectors and eigenvalues of the covariance matrix.
5) Select the principal components, the ones with the highest eigenvalues.

The PCA methods in openCV are listed bellow:

```
cvCalcEigenObjects(nObjs,input,output,ioFlags,
          ioBufSize,userData,limit,avg,eigVals);
cvEigenDecomposite(obj,eig_count,input,ioFlags,
                   userData,avg,coeffs);
cvEigenProjection(input_vecs,eig_count,ioFlags,
                  userData,coeffs,avg,proj);
```

In the *cvCalcEigenObjects nObjs* is the number of objects in the input data, *input* is the input data, *output* is the vector with the eigen objets, *ioFlags* is the input/output flags which indicates if the computation uses callback or not, *ioBufSize* gives the size in bytes for the i/o buffer, use 0 if unknown, *userData* is a pointer for the structure used in the callback mode, 0 otherwise, *limit* is the termination criteria used by the method, *avg* is the averaged object computed by the method and finally *eigVals* returns the eigenvalues computed by the method. The *cvEigenDecomposite* is used to compute the decomposition coefficients for an input object *obj*, *eig_count* gives the number of eigen objects, *input* is the structure with the eigen objects and *coeffs* are the coefficients for the *obj* entered, the other parameters are the same as in the *cvCalcEigenObjects*. In *cvEigenProjection* the object projection over the eigen subspace is computed, *input_vecs* is the input objects *coeffs* are the coefficients computed by *cvEigenDecomposite* and *proj* is the output computed by *cvEigenProjection*. More details about Principal Component Analysis can be found in [12].

F. Neural Networks

An artificial neural network provides a general method for learning different types of functions, such as real or discrete valued functions from examples [14]. Neural networks are among the most effective classifiers available, although the learning phase might take some time when using gradient descent learning method [1].

OpenCV has two methods to implement neural networks, Support Vector Machines (SVM) and Multilayer Perceptron (MLP). Both methods are implemented similarly so we will go through the Multilayer Perceptron and show how to use it, the reader should check in the reference manual [7] for the calls for the SVM methods.

The MLP structure is defined by the number of hidden (intermediate) layers, the number of nodes in each layer and the transition function. Each node in the network (except in the input layer) works by adding up the values that arrive to the node and feed the transition function that, when activated, computes the node's output. These two structures are presented in Figure 13 the top part shows the neural network strucutre and the bottom part shows the node's structure.

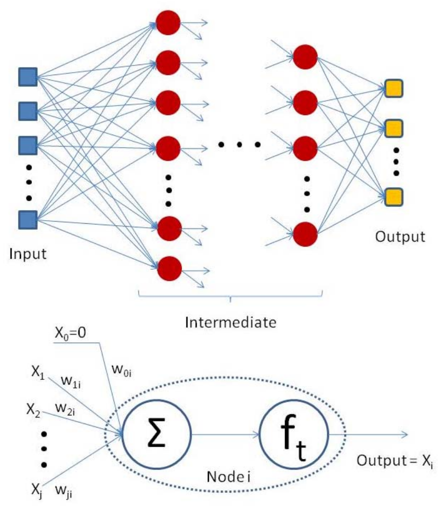

Fig. 13. The Multilayer Perceptron structure is presented on top.The bottom part shows the node's strucutre.

For the MLP openCV implements the backpropagation algorithm using gradient descent for weight updates. Equation 10 shows the error computed at the output layer, which compares the expected value with the computed value. Equation 11 computes the weight's update for each link in the network.

$$\mathbf{E} = \frac{\mathbf{1}}{\mathbf{2}} \sum_{\mathbf{kinOutput}} (\mathbf{O_k} - \mathbf{C_k})^{\mathbf{2}} \tag{10}$$

$$\mathbf{\Delta w_{ij}} = \gamma \frac{\partial \mathbf{E}}{\partial \mathbf{w_{ij}}} \tag{11}$$

In Equation 10, O_k is the expected output for node k and C_k is the computed output for node k. In Equation 11 γ is the learning rate, ($\gamma \leq 1$) and w_{ij} is the weight in the arc connecting node i to node j and Δw_{ij} is the update value for weight ij. The methods in openCV for MLP are described bellow:

```
mlp.create(layer_sizes,function,param1,param2);
mlp.train(input,output,weights,idx,param,flag);
mlp.predict(sample,mlp_response);
```

In *mlp.create layer_sizes* is an array with the number of nodes in each layer, *function* is the activation function type, which can be:IDENTITY, SIGMOID or GAUSSIAN; *param1* and *param2* are related to the activation function selected, see the reference manual [7] for these parameters. In the *mlp.train*, *input* is the input data and *output* is the expected output for each input data, some data in the input might be more significant for training then others, *weights* is an optional floating-point vector of weights for each input data, *idx* is also an optional integer vector that shows if some input data should not be considered, *param* sets parameters for the training termination criteria, check the reference manual [7] for this parameter, finally *flag* also controls the training algorithm and can be any combination of: UPDATE_WEIGHTS, NO_INPUT_SCALE and NO_OUTPUT_SCALE, see the reference manual [7] for details. Finally the parameters for the *mlp.predict* are very simple, *sample* is the new input data and *mlp_response* is a vector with the output computed for the new data.

V. Tracking and Motion

Sometimes when we are interested on computer vision applications we are not just looking for a pattern in a static image, but we want to follow this pattern in an image sequence and learn how it behaves. One option in openCV for tracking objects is the Camshift/Meanshift algorithms. Motion is also an important topic and there is a set of interesting information that can be extracted from an image sequence. Optical Flow is a technique that helps to extract information in an image sequence and openCV has a set of methods to work with optical flow in order to compute image segmentation and find some other 3D information.

A. Tracking

Tracking is the process to follow a pattern in an image sequence, but not necessarily a stream. It would be possible to do pattern recognition in each image (frame) but the process could be slow if information acquired in one image (frame) would not be used in the following images (frames). The tracking process attach a knowledge about the object being followed so the search in the following images (frames) is minimized.

Tracking can be applied in several areas, from security systems to human computer interfaces. There are methods to forecast the object's position frame by frame, from Kalman filters [15] (available in openCV) to particle filtering processes [16]. The first step required is to find object parts that are best to be tracked, these are usually corners.

Corner Finding

This technique assumes a search for an object in an image sequence. The method searchs for feature points in an image that might be easier to find again in another image. OpenCV has a function based on the Harris and Shi and Tomasi corner's definition [1] which is computed using the second derivative, correlation and eigenvalues.

The *cvGoodFeaturesToTrack* is a method which returns the pixel locations that might be easier to find in another image. This technique can also be used in stereo vision [1]:

```
cvGoodFeaturesToTrack(src,eigImage,temp,corners,
        c_count,qual,min_d,mask,bl_sz,harris,k);
```

where *src* is the input image, *eigImage* and *temp* are two scratch images, *corners* are the corners computed and *c_count* gives the number of corners found, *qual* defines the level for the acceptable eigenvalues (always ≤ 1), *min_d* defines the minimum distance between two corners, *mask* has its usual meaning, *bl_sz* defines the region for the correlation, *harris* sets the Harris corner definition (otherwise uses Shi and Tomasi) and *k* is the weight for the Harris correlation matrix.

Mean-Shift and Camshift

MeanShift is a robust method for finding local extreme points in an image as a density distribution function. This is a process where the mean-shift kernel is convolved with the data and the hill-climbing algorithm is applied to find the maximum [1]. The mean-shift algorithm is given by:

1) Select the window with the object of interest.
   - initial location
   - type (uniform, polynomial, exponential or gaussian)
   - shape (symmetric, skewed, rotated, rounded, rectangular)
   - size
2) Finds the window's center of mass.
3) Place the window at its center of mass.
4) Return to step 2 until it converges.

The Camshift (Continuously Adaptive Mean-SHIFT) is based on the Mean-shift algorithm but the window's size is self adjusted depending on the object proximity to the camera. The Mean-shift and Camshift algorithms are capable to track any image that can be viewed as a distribution of features, usually color is used as a feature.

When it is applied, for instance, for face tracking, in each frame, the raw image is converted into a color probability distribution using a skin color histogram. The face size and the face center are computed by the Camshift and this information is used to define the search window for the next frame. Figure 14 shows the camshiftdemo.c program working.

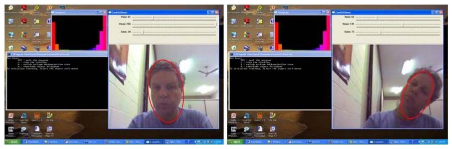

Fig. 14. camshiftdemo.c working (histogram, user interface and image captured and followed).

B. Optical Flow

Optical Flow is a technique used for movement identification in an image sequence without prior knowledge about the image contents. In this technique, typically the movement itself indicates that something is happening in the image.

OpenCV has sparse and dense methods to find movement. Sparse methods require a previous knowledge about the points to be tracked, such as corners described previously. Dense methods associates a speed vector or a pixel displacement in the image, so they don't need to know any specific point in the image. In practical applications, however, the dense techniques have a high processing cost, so, unless they are really required the user should use a sparse method. Table V-B resumes the optical flow methods available in openCV, for more details in each method check the reference manual [7] or [1].

TABLE I
OPTICAL FLOW METHODS IN OPENCV.

Method	Type	Command
Lucas-Kanade	Sparse	$cvCalcOpticalFlowLK$
Lucas-Kanade Piramidal	Sparse	$cvCalcOpticalFlowPyrLK$
Horn-Schunk	Dense	$cvCalcOpticalFlowHS$
Block-Matching	Dense	$cvCalcOpticalFlowBM$

VI. PARALLEL COMPUTER VISION USING CUDA AND OPENCV

The majority of the computers in the market today have a GPU but most of the computer vision and image processing applications are still sequential. The idea behind this topic is to present the students the possibility to use the OpenCV GPU module in order to launch commonly used OpenCV functions in a GPU and improve overall performance.

This topic will cover an introduction to CUDA, the GPU and CUDA architectures and the CUDA programming model, then it will present the basic OpenCV GPU module features, as well as a simple example of how to use the OpenCV GPU main data structure and some functions as well as how to calculate the performance speedup.

A. Introducing CUDA architecture

CUDA (Compute Unified Device Architecture) unifies the programming interface for NVIDIA GPUs. It defines a scalable programming model that could be used to program dozens of different CUDA-enabled NVIDIA GPUs.

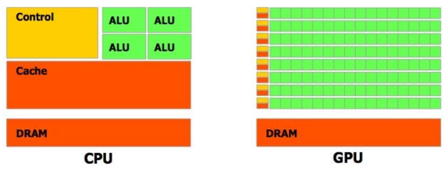

Fig. 15. CPU x GPU (source: NVIDIA).

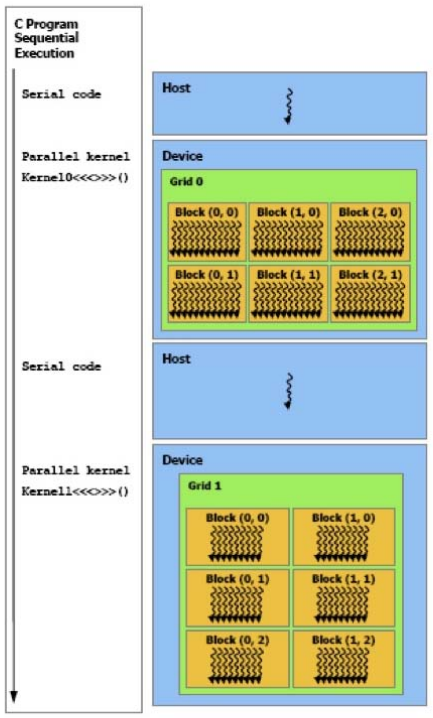

Fig. 16. Heterogeneous computing (source: NVIDIA).

First, it is elementary to understand the differences between CPU and GPU architectures. While the CPU cores dedicate their millions of transistors to a few sophisticated execution units, GPU provides up to hundreds of simple execution units (Figure 15). While exploring parallelism in a multicore CPU requires a few number of independent threads, the CUDA programming model launches hundreds of threads that execute the same code over the GPU cores. This programming model is called SIMT (Single Instruction, Multiple Threads) model and it is derived from the classical SIMD (Single Instruction Multiple Data) model.

Despite their differences, both architectures must coexist through a heterogeneous system, where GPUs are co-processors to CPUs. The GPU is also known as a type of an *accelerator* resource.

On the source code a special function must be developed that executes in the GPU and is launched from the CPU code. In the CUDA programming model [4], the GPU is called *device* while the CPU is called *host*. The special GPU function in CUDA is called *kernel* and it is compiled separately from

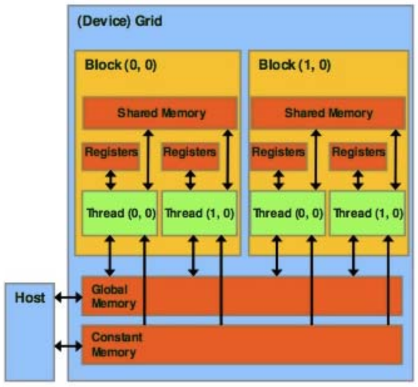

Fig. 17. GPU memory hierarchy (source: NVIDIA).

the host code by the NVIDA C compiler (**nvcc**). Figure 16 shows this heterogenous programming style.

Figure 16 also depicts an example of a kernel configuration. Each kernel is composed by a *grid* of *thread blocks*. A grid is an organized group of blocks that could have up to two dimensions while a block is a group of threads that could have up to three dimensions. These configurations are defined in the source code during the kernel launch.

This organization allows each thread to know its exact unique position in the grid, which is usually important to define the portion of data each thread will work on. This programming model is known as SPMD (Single Program Multiple Data), where all threads execute the exact same code on different portions of data. Thus, while launching a kernel the programmer determines how many threads will execute that code and, more than this, how they are organized.

The memory hierarchy is another important feature to deal with while developing a CUDA application. As a co-processor to the CPU, the GPU is connected to the main memory through a PCI Express bus. The data to be processed must be allocated and then transfered between the main and the GPU memory.

The GPU have a memory hierarchy that must be used in order to hide the memory latency while the threads are accessing the data. Figure 17 presents the GPU memory hierarchy. The main memory components are:

- **Registers**: the compiler allocates for each thread its own set of registers (with local behavior). It is a limited resource, thus an exaggerated use of registers could reduce the performance. Usually, the compiler allocates kernel local variables to registers in order to augment the performance.
- **Shared memory**: all the threads in the same block share this memory. It is explicitly allocated by the programmer through the *__shared__* directive in the kernel code. In the last generation GPU, named *Fermi*, the shared memory could also be configured as a L1 cache memory (in fact they both share a configurable memory space). Both memories help to reduce the latency while accessing the data from the global memory. Like the registers, it is also a limited resource where the allocation is performed basically by dividing the total amount of shared memory available by the total number of threads.
- **Global memory**: it is shared by all threads in the same grid. Its allocation is performed in the host code, as well as the data copy to and from the host. The Fermi architecture has a L2 cache populated with global memory data that helps in reducing the access latency.
- **Constant memory**: it is also shared by all the threads in the same grid, but it is a read only memory. The advantage in using this memory is its reduced memory latency in relation to the global memory.

B. A simple CUDA example

The following example introduces the basic CUDA programming model and its API functions. The example shows a basic matrix *add value* operation: a determined value is added to each element of a matrix.

```
int d = 64; //square matrix dim
int n = d*d; //number of elements
float x_h[d][d]; //host matrix
float *x_d; //pointer to the device matrix
size_t size= n * sizeof(float);
cudaMalloc(&x_d, size);
cudaMemcpy(x_d, x_h, size,
           cudaMemcpyHostToDevice);
```

1) Device memory allocation and data transfer: The CUDA API provides functions to allocate and to transfer data to the device. The example above presents also the two main functions to acomplish this task: *cudaMalloc()* and *cudaMemcpy()*. The first one allocates *size* bytes in the device memory and returns a pointer to it in the first argument (*x_d*). The second one makes a copy of the data (with *size* bytes) from the host memory (*x_h*) to the device memory (*x_d*). The direction of the copy is defined in the fourth argument: *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* (used later to to recover the results from the GPU).

2) Configure and launch the kernel: As mentioned, the threads are organized in a *grid* of *thread blocks*. There is a CUDA type, the *dim3*, that could be used to hold the values of each dimension. The *dim3* variables are used during the kernel launch to define the *grid* and the *blocks* configuration during the kernel execution. In the following example there will be a 4 x 4 grid of 16 x 16 threads each. Thus, the original 64 x 64 matrix will be partitioned into 16 blocks of 256 threads each, totalizing 4096 threads, one per matrix element.

```
int gd = 4; //square grid dim
int bd = 16; //square block dim
dim3 gridDim(gd, gd, 1);
dim3 blockDim(bd, bd, 1);
add_value <<<gridDim,  blockDim>>>
          (x_d, value, d);
```

3) The kernel function: The example bellow shows the *add_value* function that executes an add instruction over hundreds of threads in parallel. The modifier *__global__* indicates that the function must be executed in the device and launched by the host code. Note that there will be 4096 threads executing the kernel at the same time in the GPU.

```
__global__ void add_value
            (float* a,float value,int d){
  int x=threadIdx.x+blockIdx.x*blockDim.x;
  int y=threadIdx.y+blockIdx.y*blockDim.y;
  int offset=x+y*blockDim.x*gridDim.x;
  if ( x < d && y < d ) a[offset]+=value;
}
```

The *cudaMemcpy()* copies a block of data that resides contiguously in the global memory. Hence, the first thing to do is to calculate the exact position in the linearized matrix the thread will work on. This is done using the pre-built variables *threadIdx*, *blockIdx*, *blockDim* and *gridDim*. In the example, the threads configuration is defined by bidimendional grid and blocks, so we can use the values for the *x* and *y* components of these pre-built variables. The goal is to find an *offset* that indicates how many matrix elements there exists before the current thread's matrix element in the linear memory. The formulas in the previous example use the position of the thread in the block, the position of the block in the grid and the values in each dimension to compute the final *offset*. A more detailed example could be found in [6].

After the kernel execution, the *cudaMemcpy()* must be called again to copy the resulting matrix that is in the device global memory to the host memory.

C. Building the OpenCV GPU module

The OpenCV GPU module contains a set of OpenCV C++ classes and functions already adapted to run over any CUDA-enabled NVIDIA GPU. Hence it is possible to start using the GPU performance power even with low experience with CUDA or GPU architecture. On the other hand, using it efficiently requires some basic knowledge. This and the next sections highlights the main OpenCV GPU module classes and functions along with some CUDA explanations.

The first thing to do in order to use the OpenCV GPU module is to build it. It is necessary to have a CUDA-enabled NVIDIA GPU (only the very old ones are not enabled) and the last CUDA and OpenCV versions. At the time of this writing the last versions are OpenCV 2.2 and CUDA 4.0.

There is also a tricky procedure while building with these new versions concerning another pre-requisite: the NVIDIA Performance Primitive (NPP) library. This library contains some vision functions used by the OpenCV GPU module. Before the CUDA 4.0 release the NPP library was separated from the CUDA toolkit, thus the building procedure was different. The last CUDA release (4.0) includes the NPP library as part of the toolkit. Thus, it is important to get the *trunk* version of OpenCV that already considers this change in CUDA 4.0. Building procedure details can be found at [5].

Before building it is usefull to know the *compute capability* of the installed GPU. As the GPU architecture evolves the CUDA software reflects these changes. The CUDA *compute capability*, or just *capability*, identifies the set of features that were added generation after generation of GPU architectures.

TABLE II
DEVICE INFORMATION AND MANAGEMENT SELECTED FUNCTIONS.

Function	Functionality
int getCudaEnabledDeviceCount();	Returns the number of CUDA-enabled devices installed.
void setDevice(int device);	Sets device and initializes it for the current thread.
int getDevice();	Returns the current device index.
string DeviceInfo::name();	Returns the device name.
int DeviceInfo::majorVersion();	Returns the major compute capability version.
int DeviceInfo::minorVersion();	Returns the minor compute capability version.
size t DeviceInfo::freeMemory();	Returns the amount of free memory in bytes.
size t DeviceInfo::totalMemory();	Returns the amount of total memory in bytes.
bool DeviceInfo::isCompatible();	Returns true if the GPU module can be run on the device.

Right now, the newest compute capability is 2.1. The list of features for each capability can be found in [4].

The OpenCV GPU module comes with binaries for capabilities 1.3 and 2.0. For capabilities 1.1 and 1.2 it works with CUDA intermediary code (*PTX*). Using PTX code means that the JIT (Just In Time) compiler will be used in the first execution. This means that the first execution will be slower than the subsequent executions. The OpenCV GPU module does not work if the GPU has capability 1.0 ([7]).

D. The OpenCV GPU module

This section provides basic information on how to use the OpenCV GPU module. The complete set of ported functions are well documented in [7].

1) Device information and management: The functions in this group provide several device information and allow setting the current device if there are more than one installed in the same machine. Part of these functions are presented in Table II, some of them are methods that belong to the C++ class *DeviceInfo*.

2) Data structures: The basic data structure for OpenCV GPU module users is the *GpuMat* that is very similar to OpenCV *Mat*. There are some limitations like no arbitrary dimensions support (only 2D), no functions that returns references to its data (because references on GPU are not valid for CPU) and no expression templates technique support [7]. The example bellow presents a simple code to illustrate the use of *GpuMat* initialization.

```
cv::gpu::GpuMat dst;
cv::gpu::GpuMat src(
cv::imread("ressaca-no-arpoador.jpg",
           CV_LOAD_IMAGE_GRAYSCALE));
```

First it initializes two *GpuMat* structures that will be used in other examples: *dst* and *src*. As it demonstrates, the OpenCV function *imread()*, which returns a *Mat* structure, could be used in the constructor of the *GpuMat* to initialize it with tan image data. This is an implicit conversion that could be used to create *GpuMat* structures to run on the GPU.

Note that there are two *namespaces* used in the code. The *cv::gpu* namespace is used to distinguish the member from the upper level *cv* namespace, that contains all OpenCV components. There are other lower level data structures that could be used to write new CUDA kernels. The related classes are described in [7].

3) Using GPU module functions: The code example bellow shows a call of the *treshold()* function that will run on the GPU. Note that the namespace indicates that the call refers to the GPU version of *treshold()*. Also, the two first arguments are of *GpuMat* type declared and initialized previously. The example shows that a familiarized OpenCV user will not have any dificulty to use the OpenCV GPU module. The example also shows how to use an explicit conversion from *GpuMat* to *Mat*. The function *imwrite()* requires a *Mat* as its second argument. In this case a *GpuMat* structure (*dst*) is typecasted in order to be used by the function.

```
cv::gpu::threshold(src, dst,
  128.0, 255.0, CV_THRESH_BINARY);
cv::imwrite("result.jpg", Mat(dst));
```

E. Performance considerations

As presented previously, porting an application to run over a GPU using the OpenCV GPU module is straightforward, but yet requires some work. The OpenCV GPU samples could be used to obtain some indicators about the performance of the user's specific card.

Here we give an example of how to compute the *speedup* for a given application or portion of a code. The speedup basically is a metric that indicates how much faster an application runs when executed in a better hardware (usually a parallel hardware). Speedup is defined by the formula:

$$\mathbf{S_p} = \frac{\mathbf{T_s}}{\mathbf{T_p}} \tag{12}$$

where T_s is the sequential time and T_p is the parallel execution time.

To calculate the speedup, first it is necessary to get the execution *elapsed time* (or *wall time*) for both versions. The *elapsed time* is obtained by getting a *start time* before the portion of code to be timed and subtracting it from the *end time* obtained after its execution.

The operating systems usually provide several timers that could be used to accomplish this task. The example bellow uses the Linux *gettimeofday* timer to calculate the elapsed time in microseconds. In the example, the *treshold()* function was timed both for CPU (T_s) and GPU (T_p).

```
 struct timeval start, end;
 long mtime, seconds, useconds;
...
gettimeofday(&start, NULL);
cv::gpu::threshold(src, dst,
  128.0, 255.0, CV_THRESH_BINARY);
gettimeofday(&end, NULL);
seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
mtime = seconds * 1000000 + useconds;
```

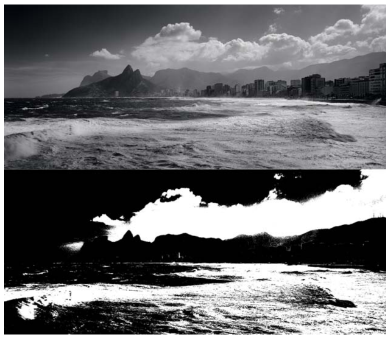

Fig. 18. Original and resulting images for the *treshold()* performance test (original extracted from www.ipanema.blog.br)

```
cout << "Elapsed time: "
  << mtime << " microseconds" << endl;
```

The previous example shows the code used to measure the GPU version of the *treshold()* function. The code is similar: remove the *gpu::* namespace and use the *Mat* structure.

The performance results for an Intel Core I7 950 3.06GHz Quad-Core processor and a Nvidia GeForce GTX 580, 1.5GB GDDR5 card are presented using the Linux Ubuntu 11.04 operating system, the compiler was g++ 4.5.2, with OpenCV 2.2 and CUDA 4.0. Each version was run ten times and the averages were used to calculate the speedup.

The average CPU time was about 2625 microseconds and the average GPU time was about 264 microseconds. After eliminating the upper and the lower values the standard deviation was 9.8 for GPU executions and 12.4 for CPU executions. This gives us a speedup of almost 10x for the GPU execution.

The image used has 2835 x 1225 pixels. The resulting image has the same size in pixels. Figure 18 presents the original image and the resulting image.

F. Porting an existing application

This section shows how to port an existing C++ application to run with the OpenCV GPU module. We use the *dft* sample that comes with the OpenCV package (the original code is in *opencv/samples/cpp/dft.cpp*). The next subsections explain the main procedures to convert the application.

1) Header files: The first step is to add the header file *gpu.hpp* as in the example bellow (the original header files remain the same as in the original *dft.cpp* code). The example also shows, the *cv* and *std* namespaces declared.

```
    #include "opencv2/gpu/gpu.hpp"
    using namespace cv;
    using namespace std;
```

2) GpuMat initialization: As illustrated before, the *GpuMat* is the main data structure an it is used as the main argument in all of the OpenCV GPU functions. In the code example bellow, it is also initialized with the *imread()* function.

```
int main(int argc, char ** argv) {
  const char* filename =
     argc >=2 ? argv[1] : "imagens/lena.jpg";
```

```
gpu::GpuMat img(
    imread(filename,CV_LOAD_IMAGE_GRAYSCALE));
```

3) Using constructors for explicit conversions: In the code sequence that follows we use the GPU versions of *copyMakeBorder()* and *merge()* functions. Note that during the creation of the *planes[]* array the *GpuMat* constructor was used to convert the returning *Mat_* template initialization. Also, the returning *Mat* structure of the *zeros()* function call was converted through the *GpuMat* constructor (there are no GPU versions for these two OpenCV components).

```
int M = getOptimalDFTSize(img.rows);
int N = getOptimalDFTSize(img.cols);
gpu::GpuMat padded;
gpu::copyMakeBorder
     (img, padded, 0, M-img.rows,
      0, N-img.cols, Scalar::all(0));
gpu::GpuMat planes[] =
     {gpu::GpuMat(Mat_<float>(padded)),
      gpu::GpuMat(Mat::zeros(padded.size(),
      CV_32F))};
gpu::GpuMat complexImg;
gpu::merge(planes, 2, complexImg);
```

4) The DFT GPU version: The code bellow shows that there is a difference in the arguments list for this function in relation to the standard *dft()* function. The GPU version requires the size for the DFT and there is also a default flag parameter that is not redefined in the example.

```
Size dft_size(N, M);
gpu::dft(complexImg, complexImg, dft_size);
```

5) More GPU module functions: In the code sequence bellow we continue to use GPU module functions simply by using the *gpu::* namespace.

```
gpu::split(complexImg, planes);
gpu::magnitude(planes[0],planes[1],
     planes[0]);
gpu::GpuMat mag = planes[0];
mag += Scalar::all(1);
gpu::log(mag, mag);
```

The code for the rearrangement of the quadrants of Fourier image will be ommited here because it uses the same porting techniques discussed until now. Basically, we use *GpuMat* structure instead of the original *Mat* structure to declare the subimages for the quadrants.

6) Returning to use the Mat structure: The last step is the *normalize()* function call that is not in the list of the ported functions until the time of this writing. Thus, we constructed a *Mat* from the *GpuMat* in order to passe it to *normalize()*. As a consequence, this function will run in CPU also in the GPU version. The code bellow illustrates the use of *Mat* with *normalize()* and also with the *imshow()* function.

```
Mat magc(mag); //mag is of GpuMat type
normalize(magc, magc, 0, 1, CV_MINMAX);
imshow("spectrum magnitude", magc);
```

7) Performance considerations: The same methodology described in section VI-E is used to measure the elapsed time for the entire application (excepting the *imshow()* call). We obtained a 5x speedup for this simple porting.

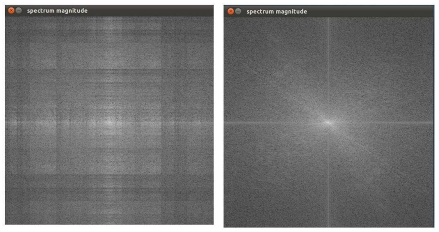

Fig. 19. DFT resulting images.

It was observed a considerable difference in the generated images that are probably a function implementation matter. The Figure 19 presents the spectrum magnitude images generated form OpenCV's *lena.jpg* image (512 x 512), that was used to the performance tests.

REFERENCES

[1] *G. Bradski and A. Kaehler, Learning OpenCV, O'Reilly, 2008.*
[2] *D. Stringhini, I.A. Souza, L.A. da Silva and M. Marengoni, Visão Computacional Usando OpenCV, in Fundamentos de Visão Computacional, editors: M.A. Piteri and J.C. Rodrigues, UNESP 2011*
[3] *Kirk, D. B., Hwu, W.W., Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufman, 2010.*
[4] *NVIDIA Corporation, NVIDIA CUDA C Programming Guide - 4.0, 2011.*
[5] OpenCV Homepage - available at http://opencv.willowgarage.com/ (access in june, 2011)
[6] *Sanders, J.; Kandrot, E. CUDA by Example - An Introduction to General-Purpose GPU Programming. Adison-Wesley, 2011*
[7] *OpenCV Reference Manual, v2.2, December, 2010*
[8] *R.C.Gonzalez and R.E.Woods, Digital Image Processing, Third edition, Prentice Hall, 2008*
[9] *M.Sonka, V.Hlavac and R. Boyle, Image Processing, Analysis, and Machine Vision, 3rd edition, Thomson, 2008.*
[10] *J.P. Lewis, Fast Normalized Cross-Correlation, http://www.idiom.com/ zilla/Papers/nvisionInterface/nip.html*
[11] *Breiman,L., Friedman,J.H., Olshen,R.A., and Stone,C.J.Classification and Regression Trees. Wadsworth International, 1984, Belmont, Ca.*
[12] *Smith, L.I., A tutorial on Principal Components Analysis, http://www.cs.otago.ac.nz/cosc453/student_tutorials/ principal_components.pdf*
[13] *Turk,M. and Pentland,A., Eigenfaces for Recognition, Journal of Cognitive Neurosicence, Vol. 3, No. 1, 1991, pp. 71-86*
[14] *Mitchell, T.M., Machine Learning, McGrawHill, 1997.*
[15] *Kalman R. E., A new approach to linear filtering and prediction problems, Transactions of the ASME: Journal of Basic Engineering, Vol. 82, 1960, pp. 35-45*
[16] *Isard M. e Blake A.,Condensation-conditional density propagation for visual tracking, International Journal in Computer Vision, IJCV, 29(1), 1998, pp. 5-28.*