Low-Level Image Features for Real-Time Object Detection

Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek,
Jiří Havel, Radovan Jošth and Lukáš Polok
Graph@FIT, Brno University of Technology, Faculty of Information Technology
Czech Republic

1. Introduction

Object detection in still images and in video sequences has a wide range of applications and while it is a very costly task from the computational resources point of view, very high demand exists for efficient object detection methods and implementations. One of the frequently used techniques of fast object detection is usage of classifiers to scan the image and attempt classification of every potential object position or even every potential position in the image being searched. The classifiers can be implemented as statistical classifiers based on supervised machine learning and can take as their input *low-level features* (sometimes called weak classifiers) extracted from the window being classified. In principle, such features can be immediately the image pixels, but by using more complex feature extractors, the classifiers can achieve better performance – both in the detection rate and in the speed.

This chapter describes several image feature extractors used in real-time object detection and in detail discusses the novel features based on local ranks. The features have been designed so that they have equal descriptive and generalization power as their state-of-the-art alternatives, but at the same time to be efficiently implementable in hardware. These features prove to be efficient, not only in the hardware implementations (tested in FPGA chips), but also when implemented using the SSE instruction set of the contemporary CPU's and implemented in the graphics processors (GPU's).

The classification background and specification of requirements on the low-level image feature extractors is given in section 2. Formal definition of the features based on local ranks is given in section 3. The performance of the LRP image feature extractors was evaluated from the point of view of the classifier construction and the results are also given in section 3. Section 4 describes efficient implementations of the feature extractor on different hardware platforms, namely the SSE instruction set; FPGA (Field-Programmable Gate Arrays) defined in the hardware definition language VHDL; and GPU implementations, using both the shading language GLSL and the CUDA programming environment. Section 4 also contains performance evaluation of the different implementations of the low-level feature extractors.

2. Background

Object detection presented in this chapter is based on supervised machine learning, namely statistical classifiers. The basic idea used in this approach is usage of several classifiers combined in such way that the output of their combination, which can also be seen as classifier, has better results than each of the input classifiers (Windridge & Kittler, 2003). Initially, the approach of combining the classifiers was exploited with ad-hoc selection of relatively well working classifiers, but later on it was found that the relatively small set of well working input classifiers can be replaced with a large set of very simple image features. If such a large set of simple image features is well combined (the supervised learning process well performed), the resulting classifier, often called a strong classifier, can be even better than with the previously known approach with, in fact, less effort as the simple image features do not require, from the design point of view, as much effort as the relatively well working weak classifiers.

2.1 Object Detection by Boosting

In 2001 Viola and Jones presented the first real-time frontal face detector which provided a precision of detection high enough for practical applications. This performance was achieved by combining ideas which together very well minimize the average computation time. The individual parts are the Haar-like features used to efficiently extract discriminative information from images; the AdaBoost learner which combines simple hypotheses into a powerful decision rule; and the attention-cascade structure of the detector which greatly reduces the average decision time. Additionally, bootstrapping was used when training the detector to achieve very low false positive rates needed when detecting objects in images. The significant success of the Viola and Jones face detector consequently encouraged further research in similar approaches and resulted in a great number of modifications to this original detector.

The performance of the detection classifiers largely depends on the type of features they use. The ideal features should be computationally inexpensive, and to some degree, invariant to geometry and illumination changes, and should provide high discriminative power – all at the same time. High discriminative power is needed to achieve high precision of detection and it also implies more compact and faster classifiers as lower number of features is needed to be computed for the classifier to make a decision. In general, the ideal type of features can differ for different types of objects (Šochman & Matas 2007). However, simple image filters have been proven to generalize well across various types of objects (Schneiderman, 2004). These filters decorelate the neighboring pixel values; utilize knowledge about frequency properties of images; and they also provide low tolerance to geometric transformations. Most of the filters which are used for object detection do not respond to the zero-frequency component, and they can be also normalized to compensate lighting changes.

When using simple filters, it is possible to transform the data in such a way that all the information in the original data is represented with the same number of coefficients (wavelet transformation). However, it is more efficient to consider all the possible filters and choose only the most discriminative for the classifier. This way, the most relevant information is extracted in the least amount of time and the classifier can be simpler. For example, Viola Jones (2001) used a highly over-complete set of Haar-like features totaling 180,000 for samples 24×24.

Viola and Jones (Viola & Jones, 2001) used AdaBoost (Freund & Schapire, 1995) algorithm to both select informative features and create the classifier. AdaBoost (shown in Fig. 1) is one of the boosting algorithms. It combines simple (*weak*) classifiers into a very accurate prediction rule (*strong classifier*). If each of the weak classifiers is based on only a single feature, the boosting algorithm then effectively performs feature selection. The weak classifiers are selected in a greedy fashion and combined to minimize an exponential loss function. AdaBoost creates large-margin classifiers in the weak classifier space.

The AdaBoost algorithm has certain properties which makes it especially useful for real-time detection. The strong classifier is a linear combination of the weak classifiers which makes it very efficient to compute. Also, the algorithm rapidly converges to a good solution on training data which minimizes the size of the strong classifier. Finally, the AdaBoost algorithm has been proven to reach an arbitrarily low classification error rate on the training data as long as the weak classifiers provide at least some useful information. This can be generalized in that the AdaBoost algorithm is guaranteed to reach a specific error at any operating point. In the Viola & Jones detector, this fact is exploited when creating classifiers for the cascade stages, where the reaching of a specified error at a specific operating point is used as the stopping criterion. This way, the complexity of the classifier is kept low while maintaining the required error rate.

```
Given $S = \langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$ ,  $x_i \in \mathrm{X}, y_i \in \mathrm{Y} = \{-1, +1\}$
Initialize $\mathrm{D}_1(i) = 1/m$ .
for $t = 1, \ldots, T$ :
    Train weak learner using distribution $\mathrm{D}_t$ .
    Get weak hypothesis $h_t : \mathrm{X} \to \{-1, +1\}$ .
    Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \varepsilon_t}{\varepsilon_t}\right)$
    Update: $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$
    where $Z_t$ is a normalization factor.
Output the final hypothesis: $H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$
```

Fig. 1. The original version of AdaBoost (Freund & Schapire 1995) with notation modified according to (Freund & Schapire 1999).

The ensemble classifier created by AdaBoost can be itself a powerful and efficient classifier capable of detecting objects in images. However, such a classifier would have to still be composed of hundreds of weak hypotheses. Such a large classifier would certainly not provide real-time performance in most of the desired scenarios. To reduce the computational complexity of the detector, Viola and Jones exploited the fact that the vast majority of samples classified when scanning images for desired objects belong to background.

They created an *object-specific focus-of-attention mechanism* which they called *cascade* and which is essentially a degenerated decision tree (see Fig. 2), where each of the nodes is a

strong classifiers created by AdaBoost. The individual stages of the cascade either reject the processed sample as background or they send the sample to the next classifier.
As the decision task becomes harder for the later stages, the classifiers become longer. The cascade is the first mechanism which allows creation of such focus-of-attention mechanisms at least partially automatically.

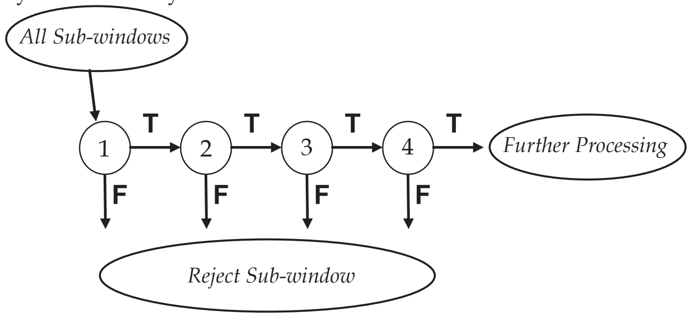


Fig. 2. The detection cascade. The cascade is composed of a series of increasingly more complex classifiers which either reject the classified sub-window as background or pass it to the subsequent stage. The object is detected only if the corresponding sub-window successfully passes through all of the stages. (Viola & Jones, 2001)

The detection cascade can be created according to the desired false positive rate and false negative rate of each stage. In such a case, AdaBoost increases the size of the strong classifier until the required rates are reached. However, in (Viola & Jones, 2001b), the authors set the lengths of the individual stages manually. Moreover, the cascade is in many aspects sub-optimal. First, all information between the consecutive stages is lost, even though the previous stage already provides a very good solution to the problem of the next stage. Second, the operating points of the classifiers and their lengths are set ad-hoc and not optimally. These two problems were addressed many times (Brubaker et al., 2006; Šochman & Matas, 2004; Xiao et al., 2003), most notably, Šochman and Matas (2005) presented *WaldBoost* algorithm which solves these two problems in a natural way.
The WaldBoost algorithm is a combination of real AdaBoost (Schapire & Singer, 1999) and Wald's (1945) *sequential probability ratio test*. In WaldBoost, rejection thresholds are set after each iteration of the AdaBoost algorithm. The thresholds are set as Wald proposes in the sequential probability ratio test, which he proves is the fastest possible classification strategy for a given target error rate. Also, as the resulting classifier is monolithic, no information is lost.

2.2 Image Features Based on Haar Wavelets

The Haar features were introduced by Papageorgiou et al. (1998), who used them as an input for support vector machine to create a very accurate classifier. Viola and Jones (2001) used the Haar features for rapid object detection in a framework with an AdaBoost classifier and thresholding weak hypotheses. The features, in their basic form, are based on the difference of adjacent rectangular regions of the input image. They respond strongly on edges and line segments of the image.

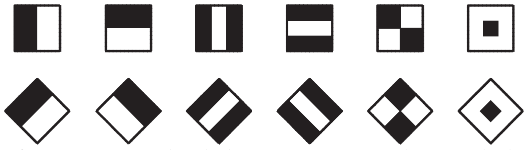


Fig. 3. Shapes of Haar features. Standard shapes on top and extended set on the bottom

The shapes of the wavelets typically used in pattern recognition are displayed in Fig. 3. The Haar features are very popular for their extremely low computational cost when evaluated on integral image and for providing good amount of information at the same time. The extended Haar feature set was introduced by Lienhart and Maydt (2002). The difference from the commonly used features is that new features are rotated by 45 degrees.

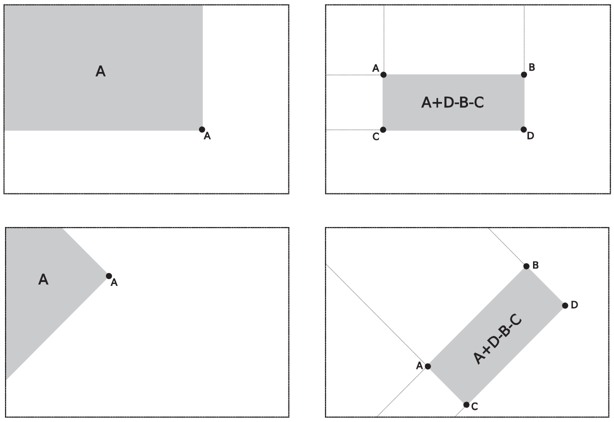


Fig. 4. Integral images. Standard integral image (top) and integral image required to evaluate 45 degree rotated Haar features (bottom).

Efficient evaluation of Haar features is achieved by using integral image (Fig. 4, top). The integral image stores in each pixel the sum of all pixels above and to the left of it. As a consequence, the sum of pixels of an arbitrary axis-aligned rectangular region in the image can be obtained by referencing only the corner pixels. For the extended set, a different type of integral image is required (Fig. 4, bottom).
An important advantage of the features is that the response can be obtained in constant time regardless of the size of the feature in the image. A preprocessing stage is required to create the integral images, though. Similar to other convolution-based features, the Haar features need to be normalized to achieve (at least partial) invariance to lighting conditions, which can significantly increase computational demands. The typical choice of the normalization value is the standard deviation of local intensity for which another integral image is required.

2.3 Local Binary Patterns

The Local Binary Patterns (LBP) are widely used in texture processing. They were introduced by Ojala et al. (2000) and some improvements have been proposed since then. LBPs in their basic form capture information about local textural structures by thresholding samples from a local neighborhood by its central value and forming the pattern code (Fig. 5). The code is calculated as a weighted sum of the thresholded samples. The weights correspond to powers of 2, so each sample sets a single bit in the pattern value.

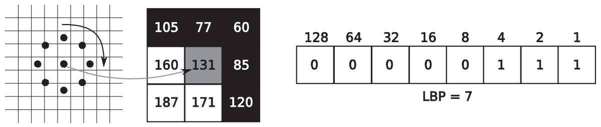


Fig. 5. Example of LBP evaluation: sampling of the neighborhood (left), thresholding sampled values by the central value (middle) and forming of the LBP code (right).

Typically, the circular neighborhood with 8 samples is used (8 bit pattern), but other variants are also possible. LBP are most frequently used in combination with local histograms to describe a local image area and segment the image.

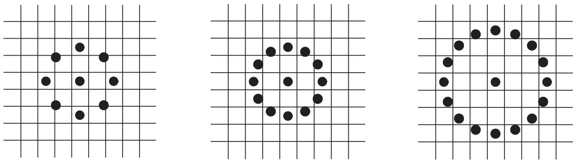


Fig. 6. Different sizes of Local Binary Patterns.

The LBP is not rotationally invariant, it is dependent on which sample is considered first when forming the code. Rotational invariance can be achieved by normalization of the pattern by shifting the bits – the lowest value is selected as the LBP result. The LBPs exhibit very good performance when used as features in object detection (Zhang et al., 2007).

3. Local Rank Functions

The experience with known features, such as Haar features and Local Binary Patterns, suggests that in many cases the classification benefits from the intensity information. On the other hand, the intensity information is subject to changes due to brightness and contrast adjustments of the images while invariance to these changes is very often wanted. This fact causes the applications using features directly based on intensity, such as Haar features, to normalize the image window being classified (e.g. through equalization of its histogram to have a constant energy and zero mean value or through other comparable techniques). However, regardless of the normalization method, the normalization can be very costly from the computational point of view especially comparing it to the cost of, for example, the computation of Haar features evaluation itself.

The novel Local Rank Functions (LRF) are based on the idea that the intensity information in the image can be well represented by the order of the values (intensities) of the pixels or small pixel regions (e.g. summed 2×2 pixel rectangular areas). This idea is backed by the fact that calculation of the values of features based on the order of pixels is equivalent to (or based on the exact evaluation method at least very close to) normalizing the image through histogram equalization (Acharya & Ray, 2005) and then evaluation of the feature value based on the pixel or small regions intensities.

The Local Rank Functions – functions based on the order of pixel values rather than the values of pixels themselves – have several principal advantages over the functions based on the values themselves:

- Invariance to illumination changes – the Local Rank Functions are invariant to most of the functions used to brightness and contrast adjustments/normalization in the images. More specifically, Local Rank Functions are invariant to nearly all monotonic gray-scale transformations.
- Strict locality – Local Rank Functions of objects (parts of objects) do not change locally when the object's image is being captured under changing conditions (similar to for example SIFT)
- Reasonable computational complexity – computation and memory accesses can be optimized thanks to regular geometric structure. No explicit normalization is needed, which is specifically important in some classification schemes, such as WaldBoost (Šochman & Matas, 2005).

3.1 Local Rank Functions' Formal Definition

Let us consider a scalar image $f : Z^2 \to R$. On such an image, a *sampling function* can be defined $(\mathbf{x}, \mathbf{u} \in Z^2, g : Z^2 \to R)$

$$S_{\mathbf{x}}^{g}(\mathbf{u}) = (f * g)(\mathbf{x} + \mathbf{u}) \tag{1}$$

This sampling function is parameterized by the convolution kernel g, which is applied before the actual sampling, and by the vector $\mathbf{x}$ which is the origin of the sampling. Next, let us introduce a vector of relative coordinates ($n \in N$)

$$\mathbf{U} = [\mathbf{u}_1\ \mathbf{u}_2\ \ldots\ \mathbf{u}_n], \mathbf{u}_i \in Z^2 \tag{2}$$

This vector of two-dimensional coordinates can define an arbitrarily shaped neighborhood and it will be used together with the sampling function to obtain a vector of values describing the neighborhood of this shape on position $\mathbf{x}$ in the image

$$M = \left[S_{\mathbf{x}}^{g}(\mathbf{u}_1) S_{\mathbf{x}}^{g}(\mathbf{u}_2) \ldots S_{\mathbf{x}}^{g}(\mathbf{u}_n) \right]. \tag{3}$$

This n-tuple of values will be referred to as the *mask* in the following text. The term mask is reasonable as the vector was created by "masking" global information from the image and leaving only specific local information. Note that in general, the sampling function does not have to be uniform over the mask

$$M = \left[S_{\mathbf{x}}^{g_1}(\mathbf{u}_1) S_{\mathbf{x}}^{g_2}(\mathbf{u}_2) \ldots S_{\mathbf{x}}^{g_n}(\mathbf{u}_n) \right], \tag{4}$$

but the implementations described in this text all use the uniform sampling function.
For each element *k* in the mask, its rank can be defined as

$$R_k = \sum_{i=1}^{n} \begin{cases} 1, \text{if } M_k < M_i \\ 0, \text{otherwise} \end{cases} \tag{5}$$

i.e., the rank is the order of the given member of the mask in the sorted progression of all the mask members. This way an n-tuple of ranks *R* is obtained. Note that the ranks are independent on the local energy in the image.
On the n-tuple of ranks *R*, a variety of functions which extract discriminative information can be defined. These *Local Rank Functions* (LRF), as we call them, have the form

$$LRF : Z^n \rightarrow Z \ . \tag{6}$$

One of the possible variants of LRF is the *Local Rank Pattern* image feature (LRP) (Hradiš et al., 2008), which selects two specific ranks and encodes their values. The LRP is defined as

$$LRP(a,b) = R_a \cdot n + R_b \ , \ a,b \in \{1,\ldots,n\} \ . \tag{7}$$

Note that n is the number of samples taken in the neighborhood and therefore the result of LRP is unique for each combination of values of the two ranks R_a and R_b. This fact suggests an alternative definition of the LRP when we allow the results of LRP to be pairs of values instead of a single value

$$LRP(a,b) = [R_a \ R_b] \ . \tag{8}$$

The LRP have some interesting properties which make them promising for image pattern recognition. Mainly, LRP are invariant to monotonous gray-scale changes such as changes of illumination intensity. This invariance results from using ranks instead of absolute values to compute the value of the feature. In fact, using the ranks has the same effect as locally equalizing the histogram of the convolved image $f * g$.

Further, LRP are strictly local – their results are not influenced by image values outside the neighborhood defined by **U**. This is a clear advantage over wavelet features (e.g. Haar-like features) which, in the way they are commonly used, need global information to normalize their results. This locality makes the LRP highly independent, for example, on changes of background and on changes of intensity of directional light.

The meaning of the values produced by the LRP can be understood in two ways. First and most naturally, the results give information about the image at the locations of the two ranks $\mathbf{x} + \mathbf{u}_a$ and $\mathbf{x} + \mathbf{u}_b$ and information about their mutual relation. On the other hand, the results also carry information about the rest of the neighborhood, especially if the neighborhood is small. In such cases the results of LRP carry good information about the local pattern in the image.

In the previous text, the LRP have been defined for two-dimensional images. However, the notation allows very a simple generalization for higher-dimensional images by changing the dimensionality of **x**, **u** and of the relative coordinates in **U** to Z^3 for 3D or Z^k for general dimensionality. Furthermore, it is possible to use more than two ranks to compute the results of the LRP. For example:

$$LRP(a,b,c) = R_a \cdot n^2 + R_b \cdot n + R_c \tag{9}$$

The LRP from their nature produce a large set of possible results, which can in the context of recognition/detection cause problems when only small training datasets are available and when the memory available on the target computational platform is limited. One way to deal with this issue – and to shrink the output set of the image features – are the Local Rank Differences (Polok et al., 2008), which can be defined as

$$LRD(a,b) = R_a - R_b \ . \tag{10}$$

The LRD computes the difference of two ranks which is very similar to the Haar-like features (Fig. 3) with added local image contrast normalization.

The definition of the LRP (and LRD) which was given in the previous text is very general. It allows arbitrary sizes and shapes of the neighborhoods and arbitrary convolution kernels. However, we can define a set of LRP which is suitable for creating classifiers for detecting objects in images – which is both informative and efficient to compute. This particular version is used in the reported experiments.

First, let us define a suitable set of neighborhoods. The number of samples taken in the neighborhood should be high enough for the features to have good discriminative power. However, the number of samples must not be unreasonably high because the computational complexity of the LRP grows linearly with the number of samples. A good first choice is local rectangular sub-sampling which takes nine samples arranged in a regular 3×3 grid. Such neighborhoods can be defined by a base neighborhood $\mathbf{U}^{base}$ which is then scaled to generate all the required sizes, e.g. for 3×3, $\mathbf{U}^{base} = [[0\ 0][1\ 0][2\ 0][0\ 1][1\ 1][2\ 1][0\ 2][1\ 2][2\ 2]]$.
In the further text, $\mathbf{U}^{(mn)}$ will refer to a neighborhood which is created from $\mathbf{U}^{base}$ by scaling the x-coordinates by m and scaling the y-coordinates by n.
The type and purpose of the convolution kernel g in the LRP can differ. It could be a derivation filter or a wavelet, but the most basic purpose of g is to avoid aliasing when scaling the neighborhood. Aliasing could be avoided by using any low-pass filter such as the Gaussian filter. For efficiency reasons, we use rectangular averaging convolution kernels. In the following text, $g^{(kl)}$ will stand for a rectangular averaging filter of dimensions k by l:

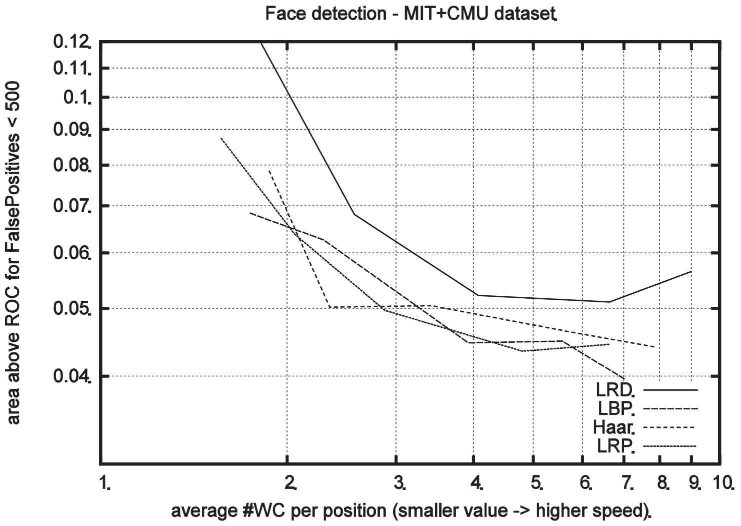
$$g^{(kl)}(\mathbf{x}) = \begin{cases} 1, \text{if } 0 \le \mathbf{x}_1 < k \text{ and } 0 \le \mathbf{x}_2 < l \\ 0, \text{otherwise} \end{cases}. \tag{11}$$

When scaling the neighborhood, it is reasonable to keep the size of the averaging filter the same as the distance between the samples. In such a case, m in $\mathbf{U}^{(mn)}$ is equal to k in $g^{(kl)}$ and n *is* equal to l. In (Polok et al., 2008), a set of LRD with only four neighborhoods $\mathbf{U}^{(1\,1)}$, $\mathbf{U}^{(1\,2)}$, $\mathbf{U}^{(2\,1)}$ and $\mathbf{U}^{(2\,2)}$ for samples of dimension 24×24 pixels is used with success.

3.3 Detection Performance

In the context of real-time object detection, the main measurable criterion which should be used to compare individual types of features is how much useful information they can extract in a certain amount of time. The second criterion is how much are they invariant to irrelevant information. Both of these criteria have to be evaluated with respect to a certain learning algorithm. The first criterion can be directly evaluated on a training set and the second corresponds to generalization on a test set. When using some focus-of-attention mechanism, the amount of extracted useful information determines the speed of the classifier which can be then related to the precision of detection on a testing set.
We have used WaldBoost (Šochman & Matas 2005) as the learning algorithm and tested the features on two detection tasks – face detection and eye detection. We have compared the Haar-like features, LBP, LRD and LRP (all neighborhoods $\mathbf{U}^{(mn)}$ which completely fit into the samples are used). For each type of the features, classifiers for five different target error rates (1%, 2%, 5%, 10% and 20%) were created. The five target error rates resulted in five gradually faster classifiers which allowed us to explore the speed/precision tradeoff provided by the features on the particular detection task. Ideally, the speed of the classifiers should be measured using some efficient implementation of the features. However, such an approach distorts the results with a different level of optimality of the individual feature

implementations. To remove these, we report here the speed in average number of evaluated features per classified position.



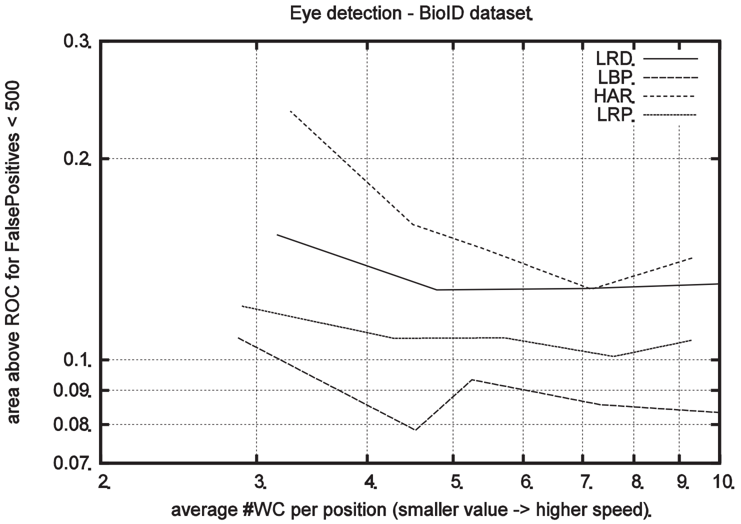


Fig. 7. Comparison of performance of image features on face detection (top) and eye detection (bottom) tasks. The graphs show the area above ROC (integrating miss-rate over false positives) as a function of average classifier speed (lower is more precise and to the left is faster). The classifiers were created by the WaldBoost algorithm for five different target error rates (1%, 2%, 5%, 10% and 20%) for each type of feature-set. The five target error rates resulted in five gradually faster classifiers – shown as a single line. The graphs can be also used to evaluate the precision/speed tradeoff for each type of feature-set for the particular task.

As can be seen in Fig. 7, Haar-like features, LBP and LRP all perform very similarly on the face detection task followed by the LRD. On the other hand, clear differences can be seen on the eye detection task where LBP are the best, second are the LRP which are followed by the LRD, while the Haar-like features are the worst. These results show that it is not possible to select a single best feature set for a variety of detection tasks. The performance of the features can be influenced by the number of the training samples, the type of distinguishing information and by the amount of intra-class variance. However, the experiments show that LRP and LRD provide in general similar detection performance as Haar-like features and LBP. Also, LRP should perform better than LRD on most tasks.

4. Fast Implementations of Selected Feature Sets

The image classification and detection tasks, as discussed in section 2, can be used as a base for various image processing and computer vision applications. Inevitably, this fact causes a situation (and it happens in many applications), where the classification and detection tasks become time critical and possibly their performance also becomes an enabling factor of various applications. Therefore, fast implementation of the classifiers is very important.

Obviously, computation of the image features is the most time-consuming part of the detectors derived from the Viola & Jones (2001) face detector. While its complexity varies with the type of the features, it is at least an order of a magnitude more demanding than the actual classifier itself (consisting of merely the sum of the feature responses). Therefore, the extraction of features seems to be the most critical part of the detection applications.

4.1 Object Detection Using the SSE Instruction Set

This section presents a high performance implementation of the LRP feature extraction on today's standard CPU's. The implementation uses pre-convolved images to obtain values of the sampling function and SIMD instruction set (of Intel CPU and compatible) for actual response computation.

The implementation must address two crucial issues: memory accesses performed by the algorithm (minimizing the number of memory accesses and ensuring their speed by aligning the operands) and the actual computation of the local ranks. Current CPUs provide SIMD capabilities that are interesting for efficient LRP evaluation. The SSE instruction set (and SSE2 in particular) has extensive support of instructions working with sixteen 8bit values in a single 128bit register.

Compared to naive LRP implementation the described implementation benefits from parallel processing when calculating the ranks. Its disadvantage is the limited number of convolution kernels $g^{(kl)}$ and neighborhoods $\mathbf{U}^{(mn)}$ (see section 3.1), because for each grid size a separate pre-calculated image is required. Minimizing the number of these images thus reduces computational cost of the preprocessing stage. In this work we use four sizes – $\mathbf{U}^{(1\,1)}$, $\mathbf{U}^{(1\,2)}$, $\mathbf{U}^{(2\,1)}$ and $\mathbf{U}^{(2\,2)}$, so four convolution images need to be calculated during the preprocessing stage.

Storing of the Image Convolutions

To simplify the feature evaluation as much as possible the convolution of the input image with a rectangular kernel (corresponding to feature block shape) is pre-computed and stored in the memory so that all the results of the LRP grid can be fetched into CPU registers by two 64-bit loads.

The convolved image is divided into separate blocks which correspond to different modulo shifts of the convolution kernel in the image – e.g. for 4×2px convolution kernel eight possible shifts exist: 0,0; 0,1; … 0,3; 1,0; … 1,3. This structure of the image is important for loading adjacent convolution values with the same modulo shift of the kernel. Fig. 8 shows the situation where the 2×2px kernel is used and four blocks are formed in the convolved image.

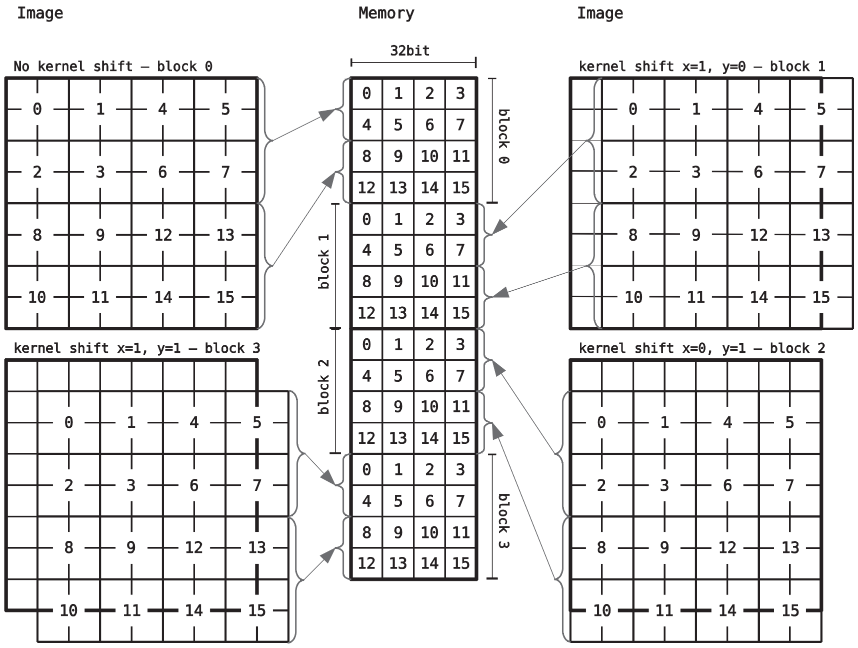


Image
Memory
Image
32bit
No kernel shift – block 0
kernel shift x=1, y=0 – block 1
kernel shift x=1, y=1 – block 3
kernel shift x=0, y=1 – block 2
block 0
block 1
block 2
block 3

Fig. 8. Example of storage of 2×2px convolution of 8×8px image in the memory.

The convolution image $I^{(k,l)}$ represents pre-calculated sampling function $S^{(g)}$ with $g^{(k,l)}$. I is divided into a set of blocks B, where each block corresponds to an image convolved with a differently shifted convolution kernel.

$$I^{(k,l)} = \{B_{0,0}, B_{1,0} \ldots B_{k-1,l-1}\}. \tag{12}$$

Each block is divided into stripes P representing two rows of pixels and each stripe is further divided into 32-bit words corresponding to four adjacent results of the sampling function. $P_{u,v}$ refers to the v-th word of the u-th stripe (Fig. 9).

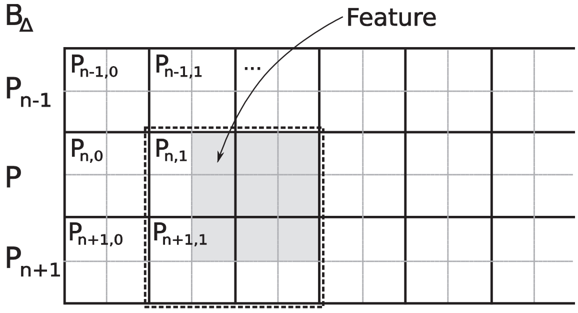


Fig. 9. Stripes in a convolution image in block B_Δ .

The LRP feature is always located in two consecutive stripes and in two consecutive words in each of them which allows them to be read by a small number of read operations.

LRP Evaluation

For evaluation of feature $F_{x,y,w,h}$ with position (x, y) and size of sampling function (w,h), we use convolved image $I^{(w,h)}$. The feature is then located in block B_Δ corresponding to the shift of the feature sampling function in the image:

$$\Delta = (x \bmod w, y \bmod h) . \tag{13}$$

The first word in which the feature's values are placed in the block is $P_{u,v}$.

$$u = \frac{y}{2h}, v = \frac{x}{2w} . \tag{14}$$

By loading words $P_{u,v}$, $P_{u,v+1}$, $P_{u+1,v}$ and $P_{u+1,v+1}$ we obtain 16 responses of the sampling function in a 4×4 grid. The sub-grid of 3×3 values aligned to Δ_M corresponds to the actual feature data.

$$\Delta_M = \left(\frac{x}{w} \bmod 2, n = \frac{y}{h} \bmod 2 \right) . \tag{15}$$

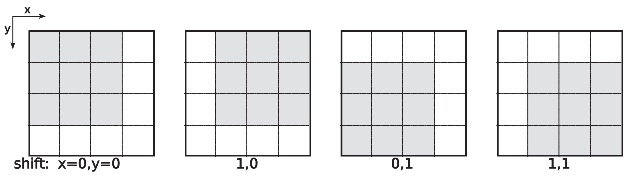


Fig. 10. Four masks used for a 4×4 grid.

Note that the previous equations stand only when the width and height of the sampling function are powers of 2. In that case, the computations are reduced to simple bit manipulations which can be very efficiently optimized. The block and mask indexing can be pre-calculated in look-up tables to further reduce the computations in the run-time to simple table indexing.

The code of the evaluation using SIMD instructions (by using Intel's intrinsic functions in C language) is shown in Fig. 11 and the block diagram of the evaluation is in Fig. 12. The LRP are parameterized by the feature's position within the classified image *fx,fy*, the block size *fw,fh* which determines the convolution image to use, and indexes of the rank pixels *idxA* and *idxB*.

```
// Inputs:
//   fx,fy,fw,fh,idxA,idxB - feature parameters
//   conv - two dimensional array of four convolution images
//   mask - array with masks stored linearly
//////////// PREPARATORY PHASE ////////////////
// Pointers to image convolved with kernel corresponding to the size of
feature blocks
Convolution & c = conv[fx][fy];
signed char * base = c.block[fx % fw][fy % fh]; // Ptr to block data
// get ptr to proper stripe and word in it
signed char * data0 = base + c.row_step * (fy / (2 * fh)) + 4 * (fx / (2 *
fw));
signed char * data1 = data0 + c.row_step;
// Position dependent mask
int mask_shift_x = (fx / fw) % 2;
int mask_shift_y = (fy / fh) % 2;
// Get values of rank pixels
char valA = (idxA < 8) ? data0[idxA] : data1[idxA-8];
char valB = (idxB < 8) ? data0[idxB] : data1[idxB-8];
//////////// LRP EVALUATION ////////////////
// Load the LRP grid to register
__m128i data = _mm_set_epi64(*(__m64*)(data0), *(__m64*)(data1));
// Zero register
__m128i zero = _mm_setzero_si128();
// Expansion of values of rank pixels
__m128i A = _mm_set1_epi8(valA);
__m128i B = _mm_set1_epi8(valB);
// Count values greater or equal to A
union {
  __m128i q;
  signed short ss[8];
} P1 = { _mm_sad_epu8( // Sum the results
           _mm_and_si128( // Mask the comparison result
             _mm_cmpgt_epi8(A, data), // compare the data to value A
             masks[mask_shift_x][mask_shift_y]),
           zero)
};
// Count values greater or equal to B
union {
  __m128i q;
  signed short ss[8];
} P2 = { _mm_sad_epu8( // Sum the results
           _mm_and_si128( // Mask the comparison result
             _mm_cmpgt_epi8(B, data), // compare the data to value B
             masks[mask_shift_x][mask_shift_y]),
           zero)
};
// calc the LRP results as sum of top and bottom part of SAD result.
int pattern1 = P1.ss[0] + P1.ss[4];
int pattern2 = P2.ss[0] + P2.ss[4];
// LRD is then pattern1 - pattern2
```

Fig. 11. LRP feature evaluation code

The basic step of the evaluation is to compare all data to the values in the rank positions. The comparison of two registers results in `-1` (`0xFF`) when the condition is satisfied. The masking discards values that are not in the feature mask and also converts `0xFF` values to `0x01` so the sum of all items of the register corresponds to the number of positive comparisons. The results are summed together by an instruction which calculates the sum of absolute differences (SAD) of two registers. The instruction operates over the higher and lower 64 bits separately, so the results need to be summed together to obtain the actual sum.

The sums for A and B rank values are then subtracted to obtain the LRD response or used as a 2D vector – LRP.

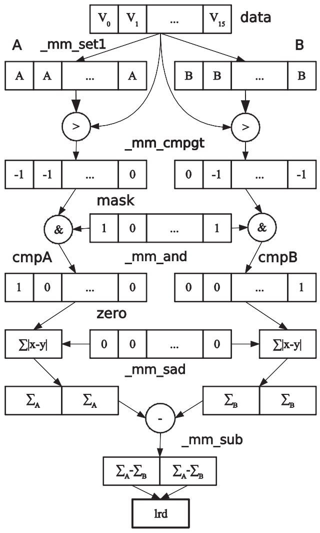


Fig. 12. Block scheme of the code from the previous figure (evaluation part only).

The LRD evaluation is described in Fig. 12. First, the data are compared to A and B vectors and masked (temporary results *cmpA*, *cmpB*). The sums of absolute differences of *cmpA* and *cmpB* are subtracted and the results for high and low parts are summed together producing the LRD value.
The evaluation is much more efficient compared to CPU code without SSE since all the values are processed in parallel. The slowest step of the evaluation is the expansion of an 8-bit value to a full 128-bit SSE register. Since the instruction set lacks a single instruction to do this, the expansion must be done by a sequence of *shift-left* and *or* instructions.

4.2 Object Detection Using GP-GPU (CUDA)

Today's GPUs provide a large amount of brute-force computational power and can be used for General Purpose usage of GPU (GP-GPU), among others for image processing and pattern recognition. CUDA is an architecture which allows one to easily use the GPU for GP-GPU algorithms with high parallelization capabilities. The programming is done in a language that strongly resembles the C language and is therefore easily understandable. The CUDA code is closely coupled with the host computer C/C++ code and their mutual communication is straightforward.
Various image-processing tasks execute large numbers of identical operations on different pieces of data that makes the highly parallel CUDA suitable for them. In the context of object detection by statistical classifiers, different positions of the sliding window are the mutually parallel tasks which perform an identical operation: the statistical classifier. In cases of the classification cascade or WaldBoost (mentioned above), the evaluation of the classifier at various locations is identical, but can be interrupted by the focus-of-attention mechanism used.

The CUDA implementation is structured to several separate operational blocks:

- First, the constant data is prepared which is mainly the data of the classifiers for object detection. Two possibilities exist for their location: the texture memory or the constant memory. The texture memory is very fast, but as a resource is shared with other parts of the algorithm, so the constant memory (cached in CUDA) was finally preferred.
- To detect in multiple scales, an image pyramid is constructed from the input image (see Fig. 13). The pyramid is constructed using OpenGL and uses the hardware-accelerated texturing available in today's graphics hardware. Thus constructed OpenGL frame-buffer is converted into a CUDA texture.
- The CUDA part of the program is executed in *kernels*, which are divided into *blocks* and further into *threads*, which are organized into *warps.* To use the execution environment most efficiently, the implementation is structured to totally use the shared memory (memory shared among threads within a block) and hardware registers. Each scanning window position is evaluated independently of the others; the positions, therefore, can be evaluated in different threads. To use the resources efficiently, at least 128 threads should be used. Our solution executes one thread for one scan-line within a rectangular part of the input image; the length of the scan-line is limited by the size of the shared memory. This arrangement is the result of a set of experiments – it organizes the threads into a sufficient number of blocks (to use all multiprocessors on contemporary GPU's), the number of threads is suitable, the shared memory is maximally used. See Fig. 13 for illustrations of arrangement into blocks and threads.
  Because of the nature of the WaldBoost evaluation which terminates the evaluation of weak classifiers at different stages of the classifier, a significant fraction of threads (grouped into warps in CUDA platform) can be idle at various times of the execution. The tasks assigned to the threads are rearranged repeatedly – the strategy of rearrangement of threads exceeds the scope of this text.

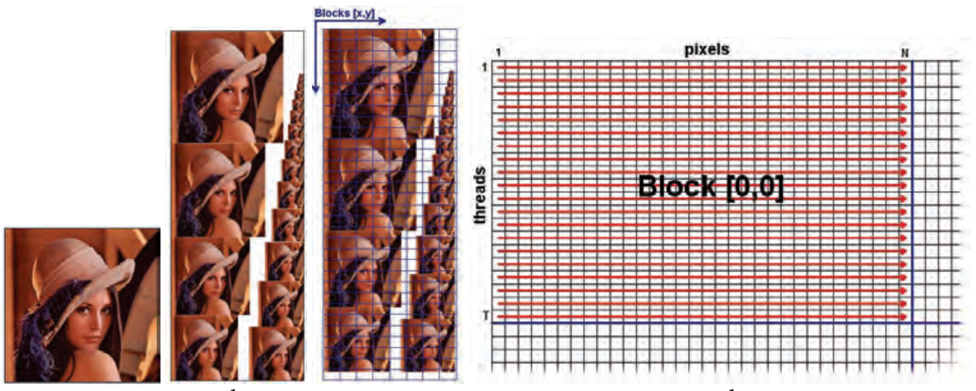


Fig. 13. Image pyramid (b); assignment to blocks (c) and threads (d) for an input image (a)

- The output data is stored in the global memory and is retrieved by device-to-host memory copying. Memory mapped pointers could be used instead which would provide the advantages of automatic asynchronous copy between the device and the host memory, thread/stream synchronize would have to be performed in that case.

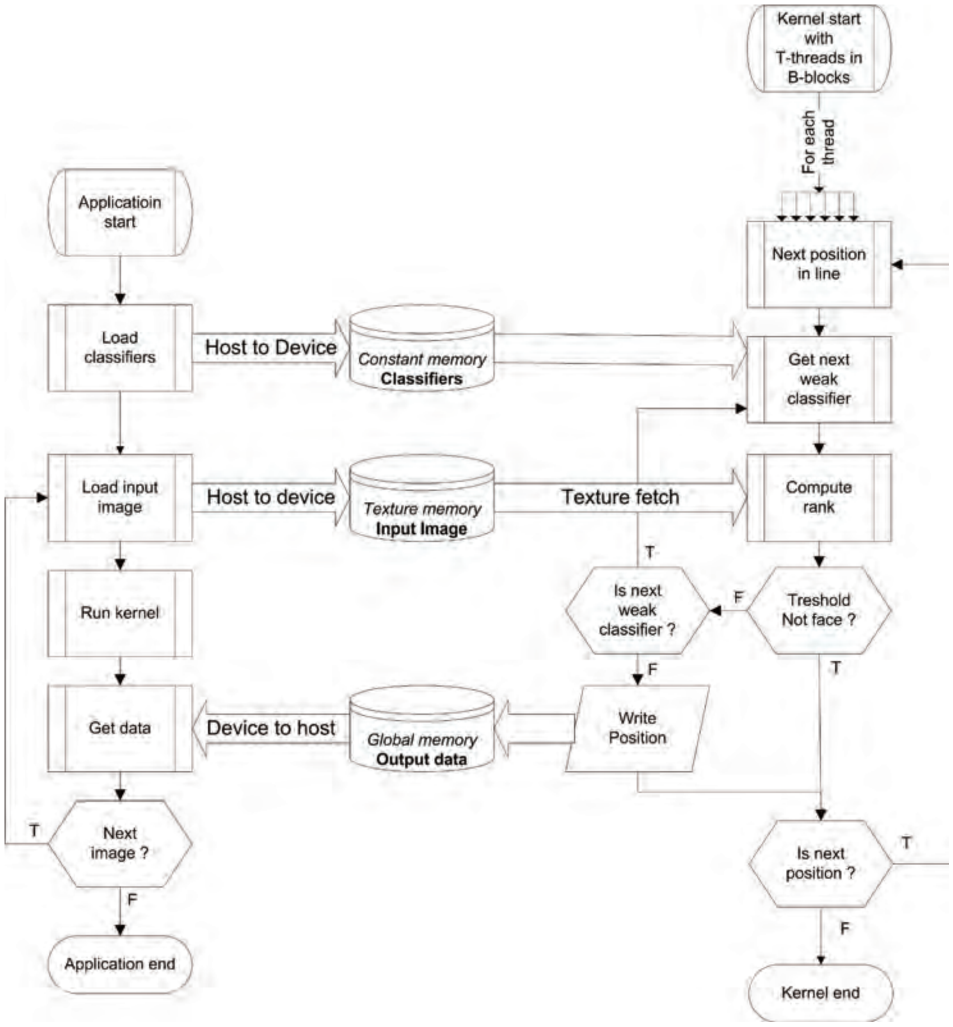


Fig. 14. Overall CUDA implementation structure

4.3 Object Detection Using GPU (GLSL)

This section presents our experiments with an OpenGL implementation of the LRD detector, consisting of the *convolution precalc* module and a *feature extractor*. It can work on most of today's common GPU's which support OpenGL 2.0. To achieve better compatibility and portability, our implementation prefers the frame-buffer objects (FBO) above platform-dependent P-buffers and GLSL shading language above the Cg language.
The implementation takes a raster image in the system memory as input, then it needs to upload it to an OpenGL texture in the GPU memory, feature evaluation shaders get executed and a raster with detector responses is downloaded back to the system memory.

There was no attempt for asynchronous data transfers to hide transport delay, but earlier work proved that such transfers are possible on GPU.
One implementation is already described in (Polok et al. 2008) which relies on complex, optimized image data storage. The implementation measured here is more straightforward because it is limited to sampling function dimensions 1x1, 1x2, 2x1 and 2x2. Such a limitation does not notably harm the information content extracted by the features, but significantly improves the performance. The bilinear filter (implemented in the texturing hardware of GPU) samples four pixels and assigns them weights, based on fractional texture coordinates. It is possible to simulate 1x1, 1x2, 2x1 and 2x2 pixel sums just by a texture coordinate offset:

$$
\begin{aligned}
&(s_f, t_f) = (s - \lfloor s \rfloor, t - \lfloor t \rfloor)\\
&w_{0,0} = (1 - s_f)(1 - t_f)\\
&w_{1,0} = s_f(1 - t_f)\\
&w_{0,1} = (1 - s_f)t_f\\
&w_{1,1} = s_f t_f
\end{aligned}
$$

where *s*, *t* are texture coordinates (in pixel scale, not OpenGL normalized coordinates), s_f and t_f are fractional texture coordinates and finally $w_{i,j}$ is the weight for texel with offset *(i, j)*.
It is now possible to illustrate the creation of some simple convolution kernels. In case that the texture coordinates are integers, fractional coordinates are zero and all weights are zero, except $w_{0,0}$ which is one, a 1x1 kernel is created. Adding offset ½ to *s* yields s_f = ½ and therefore $w_{0,0}$ and $w_{1,0}$ are ½ while $w_{0,1}$ and $w_{1,1}$ remain zero, acting as a 1×2 convolution kernel. Other kernels can be achieved analogously, as illustrated in Fig. 15.

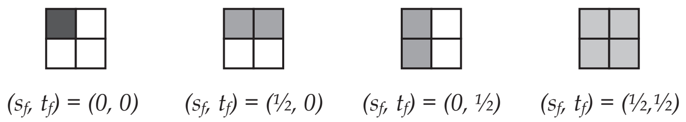


Fig. 15. Simple equal-weights convolution kernels using bilinear filtering

This introduces some interesting consequences. There is no need for a pre-calculation phase; also, we just need a single texture to evaluate all weak classifiers in the WaldBoost classifier, which is important for two reasons:
First – there is no need for branching in the classifier to select the proper convolution texture for a particular weak classifier and, therefore, there is no need to split the classifier evaluation into multiple rendering passes as in (Polok et al. 2008).
And second – all textures required to evaluate the WaldBoost classifier can be bound simultaneously to available texturing units. This issue could be actually solved using 3D textures to contain more convolution images, which can be indexed by a texture coordinate; that however severely limits the maximal image resolution (to 512×512 on nVidia cards). Another similar approach could be using custom-generated mipmap levels, where the third texture coordinate would be the texture LOD bias, which somewhat decreases the precision but could be used in practical implementations. Finally, the proper solution is using a texture array object, which contains multiple convolution images, occupies a single texturing unit and can be indexed from within the shader; but this increases the hardware requirements as this extension is not implemented in all common graphic cards today.

As we can now evaluate all the weak classifiers in a loop in a single pass, we need a way to store the classifier properties. These are the sampling function parameters – the origin of the sampling **x** and convolution kernel ***g*** (represented by fractional texture coordinate offset - which is added to **x** and stored with it, and the kernel dimensions). Next, there are positions of blocks *a* and *b*, and finally, WaldBoost thresholds (negative and positive). These values can be fit into two RGBA pixels in a floating-point texture, as illustrated in Fig. 16.

$$\begin{bmatrix} x.x + g.s_f \\ x.y + g.t_f \\ g.width \\ g.height \end{bmatrix} \begin{bmatrix} a \\ b \\ negThresh \\ posThresh \end{bmatrix}$$

Fig. 16. Weak classifier properties, stored in two pixels of a RGBA texture

We chose to store the classifier alphas in one texture and the rest of the classifier properties in another. This separation of otherwise related data is justified by their different formats and different access patterns into these textures. While the alpha texture needs a single channel (LUMINANCE) and is sampled rather randomly, the classifier properties need four channels (RGBA) and they all are read in a sequential manner.
Once the textures described above are generated, it is possible to evaluate the features in the fragment shader. The shader requires the data textures and the image texture as its input. For each weak classifier, the properties texture is read first so the mask can be read from the source image texture. Then it is necessary to get values of blocks *a* and *b* from the mask. In the fragment shader it is not possible to use an array referencing operator to select values from the matrix, so these need to be masked-out using dot products. Once the values of blocks *a* and *b* are known it is straightforward to evaluate their ranks R_a and R_b. All that remains is to read the alpha texture, accumulate the classifier response and compare it with the WaldBoost thresholds. The complete shader code is in Fig. 17.

```
#extension ARB_texture_rectangle : enable

uniform sampler2DRect n_alphas, n_cl_data, n_src_image;
// texture samplers: alphas, classifier properties and source image

uniform float f_final_thresh;
// final threshold

void main()
{
    float f_accum = 0.0;
    // classifier response accumulator

    for(int i = 0; i < classifier_count; ++ i) {
        vec4 v_data_a = texture2DRect(n_cl_data, vec2(i, 0.0)); // off.x, off.y, step.x, step.y
        vec3 v_data_b = texture2DRect(n_cl_data, vec2(i, 1.0)).xyz; // a, b, negThreshold
        // get classifier properties (as described in Fig. 16, positive threshold not implemented)

        vec3 lrd0, lrd1, lrd2;
        {
            vec4 v_tc01 = gl_TexCoord[0].xyxy + v_data_a.xyxy;
            v_tc01.z += v_data_a.z;
            vec2 v_tc2 = v_tc01.zw;
            v_tc2.x += v_data_a.z;
            // get texcoords for first three pixels of LRD grid

            lrd0.x = texture2DRect(n_src_image, v_tc01.xy).x;
            lrd1.x = texture2DRect(n_src_image, v_tc01.zw).x;
            lrd2.x = texture2DRect(n_src_image, v_tc2).x;

            v_tc01.yw += v_data_a.ww;
            v_tc2.y += v_data_a.w;
            // shift texcoords to the next pixels of LRD grid

            lrd0.y = texture2DRect(n_src_image, v_tc01.xy).x;
            lrd1.y = texture2DRect(n_src_image, v_tc01.zw).x;
            lrd2.y = texture2DRect(n_src_image, v_tc2).x;

            v_tc01.yw += v_data_a.ww;
            v_tc2.y += v_data_a.w;
            // shift texcoords to the next pixels of LRD grid

            lrd0.z = texture2DRect(n_src_image, v_tc01.xy).x;
            lrd1.z = texture2DRect(n_src_image, v_tc01.zw).x;
            lrd2.z = texture2DRect(n_src_image, v_tc2).x;
        }
        // read LRD grid 3x3 pixels (convolutions hidden inside texture sampling units)

        float a, b;
        {
            vec4 ax_ay_bx_by;
            ax_ay_bx_by.xz = mod(v_data_b.xy, 3.0); // get x-coords of a, b
            ax_ay_bx_by.yw = v_data_b.xy / 3.0; // get y-coords of a, b

            vec4 v_first_col = vec4(lessThan(ax_ay_bx_by, vec4(1.0)));
            vec4 v_third_col = vec4(greaterThanEqual(ax_ay_bx_by, vec4(2.0)));
            vec4 v_second_col = vec4(1.0) - v_first_col - v_third_col;
            // compare coords to < 1, >= 1 && < 2, >= 2
            // (index to x-y conversion and multiplexing, using fp arithmetic)

            vec3 v_a_row = vec3(v_first_col.y, v_second_col.y, v_third_col.y);
            vec3 v_b_row = vec3(v_first_col.w, v_second_col.w, v_third_col.w);
            a = dot(lrd0 * v_first_col.x + lrd1 * v_second_col.x + lrd2 * v_third_col.x, v_a_row);
            b = dot(lrd0 * v_first_col.z + lrd1 * v_second_col.z + lrd2 * v_third_col.z, v_b_row);
            // mask-out values of blocks a and b
        }
        // demultiplex values of a and b blocks

        vec3 lrd_vec = vec3(greaterThan(vec3(a), lrd0));
        lrd_vec += vec3(greaterThan(vec3(a), lrd1));
        lrd_vec += vec3(greaterThan(vec3(a), lrd2));
        lrd_vec -= vec3(greaterThan(vec3(b), lrd0));
        lrd_vec -= vec3(greaterThan(vec3(b), lrd1));
        lrd_vec -= vec3(greaterThan(vec3(b), lrd2)); // three comparisons in parallel
        float lrd = dot(vec3(1.0), lrd_vec); // sum-up lrd vector components
        // calculate rank difference

        f_accum += texture2DRect(n_alphas, vec2(8.0 + lrd, i)).x;
        // fetch response from alpha texture, accumulate

        if(f_accum < v_data_b.z) // compare with WaldBoost negative threshold
            discard; // framebuffer is pre-filled with zeros (using glClear())
        // early cutoff, as simple as that
    }
    // loop trough all classifiers

    gl_FragColor.xyz = vec3(f_accum > f_final_thresh);
    // perform threshold here, write result
}
```

Fig. 17. LRD detector shader in GLSL

Note that the shader actually evaluates the LRD. An axtension to LRP is rather simple, rank differences for *a* and *b* are calculated separately and are then combined to be used as an index to the alphas texture. The other approach could be using a 3D texture for alphas and using both rank differences as texture coordinates, the third coordinate being the classifier index.

There is one more issue to mention: loops in the fragment shaders are limited to 255 iterations (at least in nVidia implementations), after that they are interrupted (as if a break instruction was called). WaldBoost classifiers used in the performance evaluation were about 1,000 weak classifiers long, so the shader needs to contain four identical loops of 250

iterations each. The only modification then is using the classifier index instead of the loop counter to address the classifier properties texture and the alphas texture.
Even though branching on GPU is not very efficient because the program-flow control is shared between multiple processor cores, this implementation is faster than the previous attempts to control the shader execution using occlusion queries and tile-based rendering. The implementation is now, thanks to bilinear filter convolutions, very simple and can be executed without modifications on as old hardware as GeForce 6600.

4.4 Object Detection Using FPGA

The primary criteria of the FPGA design considered were high speed and using also a small consumption of resources. An additional design criterion was an adaptability to various modifications of the detectors based on AdaBoost. As a result, the proposed architecture is similar to a specialized processor. The program of the processor is composed of a feature description calculation field and a feature result processing field to enable implementation of the feature evaluation and feature result processing part of the AdaBoost classifier. Note that the number of the evaluated weak classifiers is especially in the case of early termination modifications of AdaBoost (e.g. WaldBoost in Šochman, Matas, 2005), very small (typically below 20) and variable as the early termination is based on the intermediate result of the first weak classifiers.
The maximum classification window size of 31×31 pixels was chosen while the size of the scanned image is not explicitly limited. The actual processing is performed in 128×31 image stripes that should be selected from the image of interest.

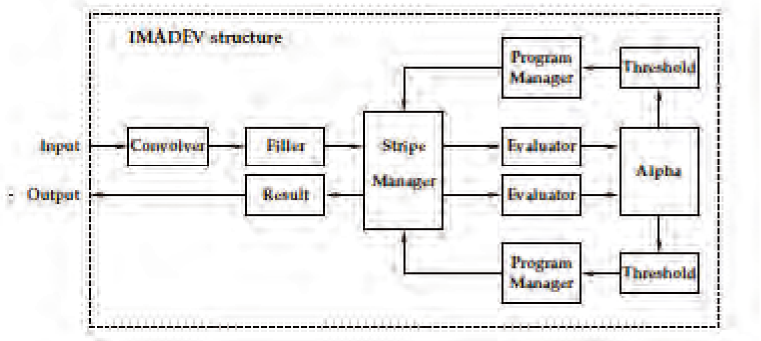


Fig. 18. The overall structure of the FPGA classifier

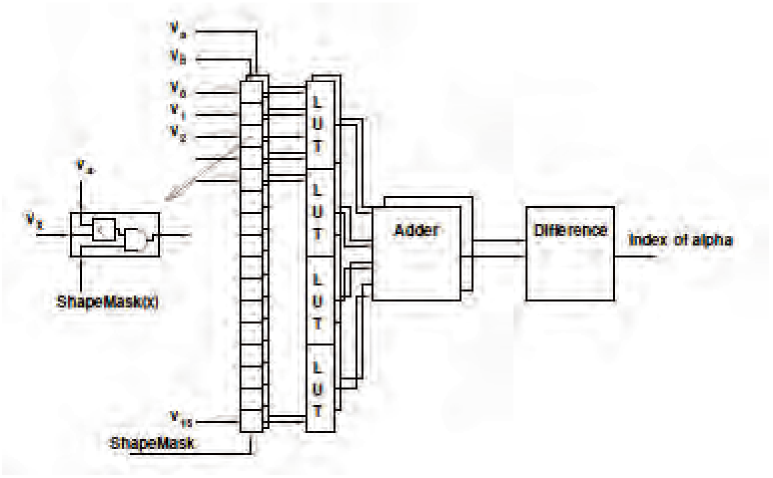


Fig. 19. The block diagram of the implementation of the LRD/LRP features implementation.

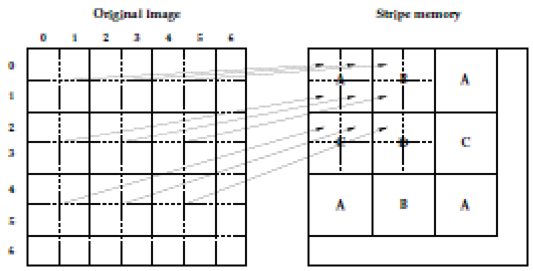


Fig. 20. Organization of the data in the BlockRAM.

It is efficient to use the pipeline in such a manner that several virtual instances of the classifier (for several locations of the window within the image stripe) are allocated. An attempt to speculatively calculate future terms in the sum is much less efficient, because if early termination is performed in this case, all the pipeline content is rendered useless and must be flushed – having an adverse effect on performance. Efficiency of the implementation can also be improved by building more instances of the engine around the memory that stores the convolved picture elements while the memory can sufficiently supply more engines with data. This approach increases the efficiency of exploitation of the hardware resources (in the case of, for example, Xilinx Virtex FPGA series it is efficient to use two ports of the BlockRAMs to supply the two engines with data).
The engine (see block structure in Fig. 18) is designed to process stream of incoming data using a FIFO-like interface. While the input part keeps filling the memory of Stripe Manager by the data, the processing part performs the classification using valid data in the Stripe Manager. If the classification is successful (the result is evaluated to 1 - current position contains the object of interest) then the position is returned as a result. If the evaluation is -1, no output is performed. Pixels of the original image are pre-processed by the convolution

unit (Convolver). The concept of the processing pipeline is based on an efficient hardware resource utilization by several virtual instances of the AdaBoost classifier (time multithreading). The total number of virtual instances is equal to the number of stages in the processing pipeline. In our case, five virtual instances circulate through the pipeline to overlap the execution time of the features (i.e., there is always one virtual instance in every stage of the pipeline). The engine exploits the parallelism at the level of the sliding windows used for object detection; each instance corresponds to one position of the sliding window in the image stripe. It means that in the beginning the instance acquires its sliding window and then evaluates all the features till program termination. Finally, it again acquires a new sliding window position if available in the stripe. Additional parallelism is gained using a memory technology which allows connecting two independent processing pipelines to one dual-port memory. The pipeline starts in Program Manager which stores the program common to all virtual instances. The program consists of 64-bit instructions, each defining one weak classifier. The most important fields of the instruction are: convolution index, X and Y position of the 3x3 grid in the window, and threshold. The instruction is sent to the Stripe Manager, which stores the valid stripe of the image in four individually addressable banks/BlockRAMs. Each bank has two reading ports, each one allocated for one of the pipelines.

The evaluator block shown in Fig. 19 is designed to compute the feature in a parallel manner. Note that the mask is applied on the results of the comparators using logic ANDs, thus invalid picture elements are not included in the ranking, although they are compared. The result of LRD is transformed with an arbitrary normalization function (based on LUT) into an index to the memory containing the desired coefficients based on machine learning. The Threshold module that calculates the sums of such coefficients holds a separate sum for each virtual instance, which is compared with a threshold (stored in the instruction). The result of the comparison determines how the evaluation of the classifier should continue. It can either continue with the next feature or end the evaluation of current position with a result (detected, not detected). The result is sent to the Program Manager and the pipeline is closed.

The engine was synthesized in a small FPGA Virtex-II 250 which is placed on the PCI board together with a DSP. The DSP uses DMA transfers to move data in and out of FPGA using the EMIF. Face detection was chosen for performance evaluation as it is considered to be a hard and widely known detection problem in machine vision community. A dataset containing 5,396 faces and 70,820 non-faces with resolution of 26x26 pixels was used to train and evaluate the classifier. The engine is able to evaluate two classifiers in one clock cycle (10ns) due to two processing pipelines. The AdaBoost with twenty weak classifiers was chosen for its acceptable error rate and still low computational demands. The performance comparison is done using the number of evaluated windows per second or frames per second. In our case, eight million evaluated windows per second have been achieved. The design is written in VHDL and synthesized for Xilinx Virtex-II technology. It takes about 1,490 Slices and 14 BlockRAMs.

4.5 Performance evaluation

Comparing the performance of these diverse implementations is not trivial. The most significant performance metric is probably the detector throughput in frames per second for a sufficiently long video. The processing time for one frame does not reflect the case where more frames are processed in parallel or pipelined. This is the case of FPGA implementation, for example. There, processing is divided into two pipeline stages - transfer to/from the card and detection. Also with four detection engines on the Uni1p card, up to eight frames can be processed in one moment; this situation also occurs on the GPU implementation. On the other hand, the time for one frame is an important metric in situations where separate frames are processed.

The processing time can be split into several phases. The crudest division is on preprocessing and scanning. The preprocessing can be further divided into contruction of the image pyramid and calculation of the convolutions. In some implementations, some of these phases do not exist at all or are interleaved. In that case, the time is measured for all interleaved phases together, since separate measurement would seriously affect the performance.

The tests were performed on a computer with CPU Intel Core2 Duo E8200 at 2.66 GHz, 3 GB DDR3 RAM and ASUS NVidia ENGTX280/HTDP graphics card. The table shows all three partial times for one frame, together with the total frame processing time. These times are in milliseconds. The times for missing or interleaved phases are left blank, meaning the time is equal to zero. The last column shows the theoretical throughput in frames per second (only the detection phases were measured, no video reading/decoding, waiting for the camera or image displaying were counted in).

A recording of television news was used as the test data. Three experiments with differently sized video were executed: low resolution video (640×350px, Table 1), broadcasting quality video (720×576px, Table 2) and high resolution HD video (1920×1080px, Table 3). *Simple* refers to straightforward implementation of LRD evaluation with no special optimizations, *Haar* is the same case as *Simple* but Haar-like features are used in the classifiers. The *SSE*, *CUDA* and *GPU* correspond to the implementations described in section 4. Note that the percentage of participation of the preprocessing and scanning phases do not have to sum up to 100 %; the rest small amount of time is overhead spent in the auxiliary parts of the program.

	Preprocessing		Scanning		Total	Throughput
	[ms]	%	[ms]	%	[ms]	[fps]
Simple	3.2	1.6	191.2	98.0	195.0	5.2
SSE	0.5	1.4	31.3	96.8	32.3	31.1
CUDA	0.2	1.1	12.1	94.5	12.8	78.7
GPU	0.1	1.0	10.0	87.3	11.5	86.9
Haar	7.6	3.9	187.7	95.8	195.9	5.1

Table 1. Results for low resolution video (640×350px)

	Preprocessing		Scanning		Total	Throughput
	[ms]	%	[ms]	%	[ms]	[fps]
Simple	8.5	1.8	448.0	97.8	458.0	2.2
SSE	1.4	1.7	78.4	96.5	81.2	12.3
CUDA	0.5	2.8	17.2	89.7	19.2	52.1
GPU	0.3	1.4	20.4	85.0	24.0	41.6
Haar	20.4	3.5	551.8	96.2	573.8	1.7

Table 2. Results for broadcasting quality video (720×576px)

	Preprocessing		Scanning		Total	Throughput
	[ms]	%	[ms]	%	[ms]	[fps]
Simple	20.2	2.5	764.3	97.0	787.9	1.3
SSE	3.2	2.0	153.1	96.0	159.6	6.3
CUDA	1.1	3.4	28.2	86.2	32.7	30.6
GPU	0.5	1.4	25.4	77.3	32.8	30.4
Haar	48.2	4.3	1059.9	95.3	1111.4	0. 9

Table 3. Results for full HD video (1920×1080px)

The FPGA implementation was measured separately, because it significantly differs from the other implementations. Most importantly, it does not support classifiers longer than 256 stages, because of limited on chip memory. Therefore, direct comparison would be misleading.
Each of the four DX64 modules on the Uni1p board is able to process 21 frames 320×240 px or 6 frames 640×480 per second. In both cases, the image pyramid has 15 levels. Because of the FPGA size, only one evaluator and one convolving unit was employed and a part of the data-filling functionality is done by the DSP instead of the FPGA.

5. Conclusions

This contribution presents the Local Rank Differences/Patterns low-level image feature extractor and its efficient implementations on several hardware architectures. This image feature set was not only developed to provide equal classification performance as its state-of-the-art alternatives, but to be executed much more efficiently in hardware implementations – either programmable hardware (FPGA) or custom specialized chips (ASIC). However, the feature set performs well also on more conventional platforms based on processors.
The measurements given in section 4 show that the speed achieved by using Local Rank Functions – namely Local Rank Differences (the special case) – is interesting. The baseline implementation outperformed the state-of-the-art Haar wavelets (especially in case of higher resolutions), and the hardware-accelerated implementations speeded-up the baseline LRD implementations more than by order of magnitude. Measurements show that the performance on the GPU's is equal for CUDA and GLSL programming. Considering that CUDA is much more intuitive and compatible to standard C language programming, the conclusion can be drawn that CUDA (or possibly OpenCL in near future) is a good selection for exploiting graphics hardware for non-rendering tasks, such as object detection.
Future work should include exploiting more general Local Rank Functions and conducting further investigation of the combinations of the features with more traditional ones.

6. References

Acharya, T. & Ray, A. (2005) *Image Processing: Principles and Applications*, Wiley-Interscience ISBN 0-471-71998-6

Brubaker, S. C.; Mullin, M. D. & Rehg, J. M. (2006) Towards Optimal Training of Cascaded Detectors, *In ECCV06,pp.* 325-337

Freund, Y. & Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting, EuroCOLT '95: *Proceedings of the Second European Conference on Computational Learning Theory, Springer-Verlag*, pp. 23-37

Freund, Y. & Schapire, R. (1999). A short introduction to boosting, *Japonese Society for Artificial Intelligence, 14*, pp. 771-780

Hradiš, M.; Herout, A. & Zemčík, P (2008) Local Rank Patterns - Novel Features for Rapid Object Detection, *Proceedings of International Conference on Computer Vision and Graphics 2008*, pp. 1-12

Lienhart, R., Maydt, J. (2002). *An Extended Set of Haar-Like Features for Rapid Object Detection*, IEEE ICIP 2002, pp. 900-903

Ojala, T., Pietikäinen, M., Mäenpää, T. (2000). *Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns*, pp. 971-987, IEEE Trans. Pattern Anal. Mach. Intell., Vol. 24., ISSN 0162-8828

Papageorgiou, C. P.; Oren, M. & Poggio, T. (1998). A General Framework For Object Detection, *Proceedings of the Sixth International Conference on Computer Vision*, ISBN 81-7319-221-9, Washington, DC, USA, 1998

Polok, L.; Herout, A.; Zemčík, P.; Hradiš, M.; Juránek, R. & Jošth, R. (2008). "Local Rank Differences" Image Feature Implemented on GPU, *Proceedings of the 10th International Conference on Advanced Concepts for Intelligent Vision Systems*, pp. 170-181, ISBN 978-3-540-88457-6, Juan-les-Pins, France, Springer, Berlin, Heidelberg, DE

Schapire, R.E. & Singer, Y. (1999) Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37: pp. 297–336

Šochman, J. & Matas, J. (2004) Inter-Stage Feature Propagation in Cascade Building with AdaBoost, *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1, IEEE Computer Society,* pp. 236-239

Šochman, J. & Matas, J. (2005). WaldBoost – Learning for Time Constrained Sequential Detection, *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2, IEEE Computer Society,* pp. 150-156

Viola, P. & Jones, M. (2001) Rapid Object Detection using a Boosted Cascade of Simple Features, *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on, IEEE Computer Society, 1*, pp. 511

Wald, A. (1945), Sequential Tests of Statistical Hypotheses, *The Annals of Mathematical Statistics*, 16, 117-186

Windridge, D & Kittler, J. (2003) A Morphologically Optimal Strategy for Classifier Combination: Multiple Expert Fusion as a Tomographic Process, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Volume 25 , Issue 3 (March 2003), pp. 343 – 353, ISSN:0162-8828

Xiao, R.; Zhu, L. & Zhang, H.-J. (2003) Boosting Chain Learning for Object Detection. ICCV '03: *Proceedings of the Ninth IEEE International Conference on Computer Vision*, IEEE Computer Society, 2003, pp. 709

Zhang, L.; Chu, R.; Xiang, S.; Liao, S. & Li, S. Z. (2007) Face Detection Based on Multi-Block LBP Representation, *ICB,*11-18 Xiao, R.; Zhu, L. & Zhang, H.-J. (2003) Boosting Chain Learning for Object Detection. ICCV '03: *Proceedings of the Ninth IEEE International Conference on Computer Vision*, IEEE Computer Society, 709