“Local Rank Differences” Image Feature Implemented on GPU

Lukáš Polok, Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek, Radovan Jošth

Graph@FIT
Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno, Czech Republic
xpolok00@stud.fit.vutbr.cz
{herout, zemcik, ihradis, ijuranek, ijosth}@fit.vutbr.cz

Abstract. A currently popular trend in object detection and pattern recognition is usage of statistical classifiers, namely AdaBoost and its modifications. The speed performance of these classifiers largely depends on the low level image features they are using: both on the amount of information the feature provides and the executional time of its evaluation. Local Rank Differences is an image feature that is alternative to commonly used haar wavelets. It is suitable for implementation in programmable (FPGA) or specialized (ASIC) hardware, but – as this paper shows – it performs very well on graphics hardware (GPU) as well. The paper discusses the LRD features and their properties, describes an experimental implementation of LRD in graphics hardware, presents its empirical performance measures compared to alternative approaches and suggests several notes on practical usage of LRD and proposes directions for future work.

1 Introduction

Statistical classifiers can very well be used for object detection or pattern recognition in raster images. Current algorithms even exhibit real-time performance in detecting complex patterns, such as human faces [10], while achieving precision of detection which is sufficient for practical applications. Recent work of Šochman and Matas [9] even suggests that any existing detector can be efficiently emulated by a sequential classifier which is optimal in terms of computational complexity for desired detection precision. In their approach, human effort is invested into designing a set of suitable features which are then automatically combined by the WaldBoost [8] algorithm into an ensemble. This approach may significantly reduce the development time of detectors and it may even lead to more computationally efficient detectors – Šochman and Matas report successfully emulating the Kadir-Brady saliency detector [2], while achieving 70× faster detection times over the original implementation.

In practical applications, the speed of the object detector or other image classifier is crucial. Real-time performance is required in many applications such as surveillance, even when processing several input streams. Use of specialized hardware in image

processing and computer vision is nothing new (e.g. [7], [4]). Recent advances in development of graphics processors attract many researchers and engineers to the idea of using GPU's not for their primary purpose – rendering 3D graphics scenes. Different approaches to so-called GPGPU (General-Purpose computation on GPUs) [1] exist and also the field of image processing and computer vision has seen several successful uses of these techniques (e.g. [7], [4]).

Statistical classifiers are built by using low level *weak classifiers* or *image features* and the properties of the classifier largely depend on the quality and performance of the low level features. In face detectors and similar classifiers, Haar-like wavelets [3], [8], [9], [10] are frequently used, since they provide good amount of discriminative information and they provide excellent performance. Other features are used in different contexts, such as the Local Binary Patterns [5]. Recently, designed especially for being implemented directly in programmable or hard-wired hardware, Local Rank Differences [11] have been presented. These features are described in more detail in section 3 of this paper. The main strengths of this image feature are inherent gray-scale transformation invariance, the ability to capture local patterns and the ability to reflect quantitative changes in lightness of image areas.

The following section 2 of this paper briefly presents the Local Rank Differences (see [11] for more detail) image feature. In section 3, the notes on implementation of LRD on a GPU using the Cg high level shading language are given. Section 4 presents the experimental results of the implementation carried out and its comparison to other approaches. Conclusions and suggestions for future research in the area are given in section 5.

2 Local Rank Differences

Let us consider a scalar image $I(x,y) \to \mathbf{R}$. On such image, a *sampling function* can be defined ($x, y, m, n, u, v, i, j \in \mathbf{Z}$)

$$S_{xy}^{mn}(u,v) = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I\big(x + m(u-1) + i, y + n(v-1) + j\big). \quad (\mathbf{1})$$

This sampling function is parameterized by the sampling block dimensions *m, n*, and by the origin of the sampling *(x,y)*, which is a pixel in the image. Note that this function "subsamples" the image by a multiple of pixels in each direction. Note please also that this function can be defined in other manners, namely not by *summing* rectangular blocks of the image but by convolving them with a suitable wavelet filter kernel, etc. Based on this sampling function a rectangular *mask* can be defined:

$$M_{xy}^{mnwh} = \begin{bmatrix} S_{xy}^{mn}(1,1) & S_{xy}^{mn}(2,1) & \cdots & S_{xy}^{mn}(w,1) \\ S_{xy}^{mn}(1,2) & S_{xy}^{mn}(2,2) & \cdots & S_{xy}^{mn}(w,2) \\ \vdots & \vdots & \ddots & \vdots \\ S_{xy}^{mn}(1,h) & S_{xy}^{mn}(2,h) & \cdots & S_{xy}^{mn}(w,h) \end{bmatrix}. \quad (\mathbf{2})$$

The mask is parameterized by sampling block dimensions *m, n* and sampling origin *(x,y)*, just as the used sampling function *S*. Along with these parameters, the mask has its dimensions *w, h* as well. Experiments (see [11]) show that in the context of AdaBoost and WaldBoost object detection, the masks of dimensions 3×3 (*w=3, h=3)* are sufficient. For different classifiers and applications, different sampling block sizes are necessary. For face detectors operating on image windows with resolution of 24×24 pixels, sampling sizes of 1×1 (*m=1, n=1* etc.), 2×2, 2×4, 4×2 are sufficient.

For each position in the mask, its *rank* can be defined:

$$R_{xy}^{mnwh}(u,v) = \sum_{i=1}^{w}\sum_{j=1}^{h}\begin{cases}1, \text{if } S_{xy}^{mn}(i,j) < S_{xy}^{mn}(u,v)\\ 0, \text{otherwise}\end{cases}, \tag{3}$$

id est, the rank is the order of the given member of the mask in the sorted progression of all the mask members. Note that this value is independent on the local energy in the image, which is an important property useful for the behavior of the Local Rank Differences image feature, which is defined as:

$$LRD_{xy}^{mnwh}(u,v,k,l) = R_{xy}^{mnwh}(u,v) - R_{xy}^{mnwh}(k,l) \tag{4}$$

The notation can be slightly facilitated by vectorizing the matrix M by stacking its rows (it is just a convention that row rather than column stacking is used):

$$V_{xy}^{mnwh} = \left[S_{xy}^{mn}(1,1) \quad S_{xy}^{mn}(2,1) \quad \cdots \quad S_{xy}^{mn}(w,h)\right]. \tag{5}$$

The rank of a member of the vector then is (note that for clarity, $V_{xy}^{mnwh}(i)$ denotes the i[th] member of the vector):

$$R_{xy}^{mnwh}(a) = \sum_{i=1}^{w\times h}\begin{cases}1, \text{if } V_{xy}^{mnwh}(i) < V_{xy}^{mnwh}(a)\\ 0, \text{otherwise}\end{cases}. \tag{6}$$

The Local Rank Difference of two positions *a, b* within the vector obviously is:

$$LRD_{xy}^{mnwh}(a,b) = R_{xy}^{mnwh}(a) - R_{xy}^{mnwh}(b). \tag{7}$$

Empirical experiments carried out so far show that one w×h dimension used in a classifier is sufficient (currently we are using 3×3 mask dimension only), i.e. for the purpose of constructing a classifier, no need exists to mix several combinations of mask dimensions, which simplifies the training and evaluation process. Weak LRD classifiers available to the statistical classifier therefore offer varying position *x, y* within the window of interest and varying size *m, n* of the sampling block used.

2.1 The Role of Local Rank Differences in the Object-Detecting Classifier

Fig. 1 shows the simplified flow for evaluating a single LRD classifier. It begins with the detection window (e.g. 31×31 pixels) being classified where rectangular

mask M_{xy}^{mn33} is positioned (considering e.g. 3×3 masks). Each field of the mask spans across several pixels which need to be convolved (see the equation 8 below).

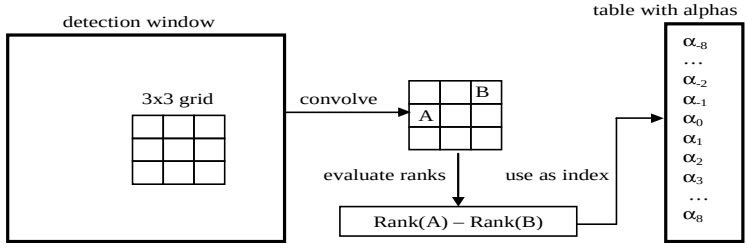

Fig. 1. Use of Local Rank Differences in the classifier

Next, the ranks are evaluated and finally the rank difference is used as index into the alpha table, selecting the weak classifier's result.

2.2 Input Image Pre-Processing

For increasing the performance of the LRD evaluation, the *function* S_{xy}^{mn} defined on the input image can be pre-calculated. As stated above, low number of combinations of $m \times n$ is sufficient for learning an object classifier – experiments show that 1×1, 2×2, 2×4 and 4×2 combinations are enough. The input image I can be convolved with

$$h_{2\times2} = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}, \quad h_{4\times2} = \begin{bmatrix} 1/8 & 1/8 & 1/8 & 1/8 \\ 1/8 & 1/8 & 1/8 & 1/8 \end{bmatrix}, \quad h_{w\times h} = \begin{bmatrix} 1/wh & \cdots & 1/wh \\ \vdots & \ddots & \vdots \\ 1/wh & \cdots & 1/wh \end{bmatrix} \tag{8}$$

and the resulting images at given location (x,y) can contain the values of the sampling function. Such pre-processing of the input images can be done efficiently and the LRD evaluation then only consists of 9 lookups (for the case of 3×3 LRD mask) into appropriate pre-processed image and then evaluation of ranks for two members of the mask. The evaluation then can be done in parallel on platforms supporting vector operations; both GPU and FPGA are strong in such kind of parallelism.

2.3 Local Rank Differences Compared to Haar Wavelets

Comparing LRD with Haar wavelets is only natural as both of these types of features were first intended to be used in detection classifiers. There are two fundamental aspects in respect to the detection classifier which must be addressed. The first aspect is the computational complexity of evaluating the features and the second aspect is the amount of discriminative information the features provide.

Haar wavelets can be computed very rapidly on general purpose CPUs by using the integral image representation [10] which can be created in a single pass through the original image. The simple Haar wavelets of any size can be computed using only six accesses into the integral image, six additions and two bit-shifts. When scanning the

image in multiple scales, this gives the possibility to scale the classifier instead of down-sampling the image. The Haar wavelets are usually normalized by the size of the feature and the standard deviation of pixel values in the classified sub-window. Computation of the standard deviation requires additional integral image of squared pixel values and uses square root.

While the Haar wavelets can be computed relatively efficiently on general purpose CPUs, it may not be the same on other platforms. On FPGAs, the six random accesses into memory would significantly limit performance (only single feature evaluated per every six clock cycles) and the high bit-precision needed for representing the integral images would make the design highly demanding. On the other hand, the nine values needed to compute LRD with grid size 3×3 can be obtained on FPGAs with only single memory access [11] (when preprocessed as shown in Section 2.2) and on GPUs with three or six accesses (see Section 3 for details).

Some detection classifiers evaluate on average very low number of features (even less than 2). In such cases, computing the normalizing standard deviation poses significant computational overhead. Further, the square root which is needed can not be easily computed on FPGAs. The LRD inherently provide normalized results, whose normalization is in fact equivalent to local histogram equalization.

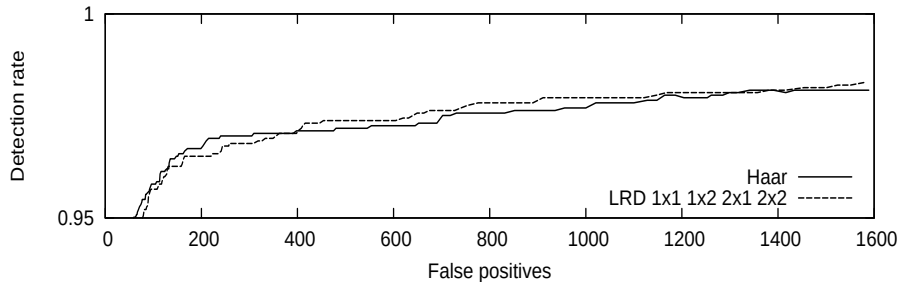

Fig. 2. ROC of two WaldBoost classifiers on a frontal face detection task. Length of the classifiers is 500 and they differ only in type of features which they use (Haar features, LRD).

The detection performance of classifiers with the LRD has been evaluated on the frontal face detection task and it has been compared to the performance of classifiers with the standard Haar wavelets. The results suggest that the two types of features provide similar classification precision. This fact can be clearly seen in Figure 2 which presents receiver operating characteristic (ROC) of two WaldBoost [8] classifiers. One of the classifiers uses the same Haar wavelets as in [10] and the other uses the LRD with block sizes of the sampling function (see Equation 2) restricted to 1×1, 1×2, 2×1 and 2×2 . The classifiers were trained using 5000 hand annotated faces normalized to 24×24 pixels and the non-face samples were randomly sampled from a pool of 250 million sub-windows from more than 3000 non-face images. The results were measured on a set of 89 group photos which contain 1618 faces and total 142 million scanned positions (scale factor 1.2, displacement 2/24). Although the set of LRD features is very limited in this experiment, the detection performance it provides is similar to the full set of Haar wavelets. This is probably due to the localized normalization of the results of the LRD which provides information about local image patterns that goes beyond simple difference of intensity of image patches.

3 LRD Implementation on GPU

As shown in section 2.2, the sampling function for a given sampling block size used by the LRD can be pre-processed by convolving the original input image by a simple convolution matrix. On GPU, built-in texture sub-sampling can be used to achieve this pre-processing efficiently. This is done using very simple fragment shaders and the whole convolution calculation usually takes less than 10% of frame time and was not further optimized.

The step that uses the pre-calculated images is the evaluation of the LRD weak classifiers. Early analysis of the algorithm revealed that its bottleneck would be texture sampling. Therefore, the main goal was to minimize the number of texture samples per pixel and to improve texture sampling coherency in order to achieve the best performance. A trick was used to do this – interleaving the convolution image into different layers of a 3D texture. The dimensions of the texture are:

$$w_t = \frac{w_i}{m} \qquad h_t = \frac{h_i}{n} \qquad d_t = mn \tag{9}$$

Where w_i, h_i are the input image's dimensions, m, n are the sampling block's dimensions and w_t, h_t, d_t is the texture size. The texture organization is illustrated in Fig. 3. Such way of storing image data ensures the texture samples needed to evaluate single LRD classifiers are tightly connected to each other.

To read the 3×3 LRD mask in a naive way, nine texture samples are needed; however, most of today's hardware is not capable of loading nine samples without stalling the pipeline. To avoid this limitation, the (8-bit grayscale) pixels of the convolution texture are packed by four into RGBA vectors stored in the texture memory. Then it takes three or six texture samples, depending on the modulo 4 position, to read all the nine pixels of the mask (in contrast to the nine reads without the use of 3D texture).

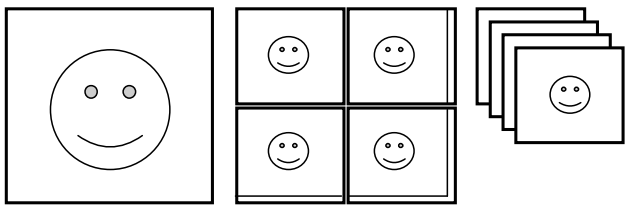

Fig. 3. (from left to right) Original image, interleaved convolution images (for 2x2 kernel) and interleaved images stored as a 3D texture

Pixel unpacking is done in the fragment shader and it needs to choose one of four different branches. It could be solved by a simple *if* statement, but the (expensive) branching instruction can be avoided by rasterizing the image in vertical stripes, one pixel wide and four pixels apart, using a different shader for each modulo 4 position.

Having read the 3×3 grid, the next step is to evaluate the local ranks. The SIMD nature of the GPU can be exploited by keeping the pixels in three 3D vectors. First, the pixels on positions *a* and *b* are picked. Unfortunately, no index parameter can be

used in a shader so the pixels are selected using dot product (which is fairly efficient on GPU). The ranks are calculated using the following code:

```
vec3 accum =  lessThan(vec3(A), row0);
accum      += lessThan(vec3(A), row1);
accum      += lessThan(vec3(A), row2);
accum      -= lessThan(vec3(B), row0);
accum      -= lessThan(vec3(B), row1);
accum      -= lessThan(vec3(B), row2);
float rank_difference = dot(vec3(1,1,1), accum);
```

Fig. 4. Calculation of the local rank difference; *row0*, *row1* and *row2* are vec3 contain the input pixels, *A* and *B* are pixel values on positions *a* and *b*. The *lessThan* function compares its arguments by component and the result is vec3, containing zeros or ones based on comparison. The dot product sums up the Local Rank Difference. This snippet of code evaluates in approximately 14 GPU instructions. Finally, *alpha* is chosen from table (texture).

3.1 The AdaBoost/WaldBoost Object Detection Runtime Framework in GPU

One fragment shader evaluates several LRD's and accumulates them in an accumulated (see above). After accumulating all the weak classifiers in the learned AdaBoost classifier, a decision is made based on a threshold. The overall AdaBoost classifier structure implemented using the shader is in Fig 5.

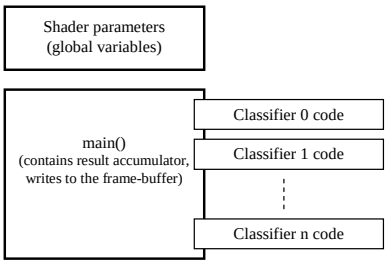

Fig. 5. AdaBoost/WaldBoost object detection GPU runtime shaders with several classifiers

The WaldBoost [8] pipeline is fairly similar to the one of AdaBoost (described above), it only needs facilities to terminate the calculation on individual pixels. This can be done using depth test – the classifier evaluation remains unchanged, but extra rendering passes are added which compare the intermediate accumulated sum with a given threshold and modify the depth-buffer accordingly. That means if output is below the threshold, zero is written into the depth-buffer, otherwise one is written (using `step` to avoid branching). The outputs from the classifier are rasterized on depth 1 so shaders are not executed on positions with zero depth (see Figure 6).

This approach benefits from early depth-test that discards all fragments with the wrong depth (without evaluation). The limitation is that fragments modifying their depth must be evaluated so the number of the stopping decisions must be low. Therefore, training of WaldBoost classifier must include costs of the decisions.

```
uniform sampler3D n_texture_0;
uniform vec2 v_pixel_00;
uniform sampler2D n_alphas;
uniform vec2 v_alpha_pixel;
uniform vec4 v_block_to_slice_00;
uniform vec3 v_selector_a00, v_selector_b00;

void main()
{
    float f_result = .0; // result accumulator
    {
                // classifier 0
    }
…
    {
                // classifier n
    }
    gl_FragColor.r = f_result; // write output fragment
}
```

Fig. 6. AdaBoost shader code; *n_texture_0* is the id of the right texturing unit, *v_pixel_00* is the pixel size of that texture, *n_alphas* is the id of the alphas texturing unit, *v_alpha_pixel* is site for alphas texture, *v_block_to_slice* contains constants required for 3D texture slice from 2D texcoords (width/number of layers, convolution kernel width/number of layers, height/number of layers*convolution kernel width and slight z-offset to aid the right layer sampling), *v_selector_a00* and *v_selector_b00* are vectors selecting the right column from 3×3 grid)

4 Performance Evaluation and Analysis

To evaluate the efficiency of the presented GPU implementation of the LRD, these implementations were compared:

- LRD on GPU (section 3 above)
- Haar on GPU (section 4.1 below)
- LRD on CPU (section 4.2 below)

Evaluation was performed for different resolution of the image, for different sizes of the classified window and for different amount of the weak hypotheses calculated for each classified window. Note that this evaluation is to determine the evaluation speed of the weak classifiers only, not the overall performance of the boosted classifier.

	Win	num	frame-time [milli sec]			time-per-wc [nano sec]		
resol	size	wc	lrdGPU	haarGPU	lrdMMX	lrdGPU	haarGPU	lrdMMX
320x200	16	5	0.244	0.370	17.7	0.872	1.325	55.29
320x200	16	10	0.527	0.469	25.0	0.942	0.839	46.71
320x200	16	50	2.524	3.010	82.0	0.902	1.076	40.04
640x480	16	5	1.173	1.642	101.8	0.810	1.134	58.55
640x480	16	10	2.232	2.159	149.0	0.771	0.745	51.82
640x480	16	50	11.066	15.731	493.0	0.764	1.086	44.05

Table 1. Performance table for LRDonGPU, HAARonGPU and LRDonMMX; the table contains the times of sole evaluation of the classifier, since the pre-processing for the Haar wavelets (integral image calculation), cannot be easily implemented in the GPU (nv7950)

In the following table, a coarse comparison of the performance of the pre-processing stage is given. Note that it is difficult to compare the pre-processing for the Haar wavelets with the LRD convolutions, because the integral image calculation is difficult to implement on the GPU. Note that this is an important advantage of the LRD over the Haar wavelets, especially when in GPU implementation.

resol	LRDonGPU	HAARonCPU	LRDonCPU
320x200	0.72	1.22	2.52
640x480	1.22	10.29	9.13
800x600	3.51	16.41	13.80
1024x768	3.75	27.94	24.80
1280x1024	4.53	45.16	37.45

Table 2. Evaluation of the pre-processing stage (convolutions for the LRD, integral image for Haar wavelets). Times are given in milliseconds.

4.1 Implementation of the Haar-like features on the GPU

Only the simplest (two-fold) Haar wavelet features were used in this testing implementation (though also three-fold features are used in the object detectors, whose evaluation is slightly slower).

```
float GetHaar(float2 p0, float2 p1, float2 p2, float2 p3,
 float2 p4, float2 p5, uniform samplerRECT IntegTexId)
{
  return - texRECT(IntegTexId, p0).a + texRECT(IntegTexId, p1).a * 2.0f
         - texRECT(IntegTexId, p2).a + texRECT(IntegTexId, p3).a
         - texRECT(IntegTexId, p4).a * 2.0f + texRECT(IntegTexId, p5).a;
}
float Horizontal(float2 p0, float2 d, float WIntensity,
 uniform samplerRECT IntegTexId, uniform samplerRECT AlphaTexId, float HaarId)
{
  float2 dx1 = float2(d.x,0.0f); float2 dx2 = float2(d.x+d.x, 0.0f);
  float2 p3 = p0 + float2( 0.0f, d.y);
  float haar = GetHaar(p0, p0+dx1, p0+dx2, p3, p3+dx1, p3+dx2, IntegTexId);
  haar /= d.x*d.y * WIntensity; // Normalization
  haar = clamp((haar+1.0f)*0.5f * 120.0f, 0.0f, 120.0f); // quantization
  return texRECT(AlphaTexId, float2(HaarId, haar)).a;
}
sOutPS FragmentProgram(sVS2PS IN, uniform samplerRECT IntegTexId,
 uniform samplerRECT IntegSqTexId, uniform samplerRECT AlphaTexId)
{
    sOutPS OUT;
    float window_energy = +texRECT(IntegSqTexId, IN.texcoord0).a
               -texRECT(IntegSqTexId, IN.texcoord0 + float2(WND_W, 0.0f)).a
               -texRECT(IntegSqTexId, IN.texcoord0 + float2(0.0f, WND_H)).a
               +texRECT(IntegSqTexId, IN.texcoord0 + float2(WND_W, WND_H)).a;
    float haarid = 0; float sum = 0;
    sum += Horizontal(IN.texcoord0+float2( 0.0f,  0.0f), float2( 8.0f, 8.0f),
                window_energy, IntegTexId, AlphaTexId, haarid); haarid++;
    sum += Vertical(IN.texcoord0+float2( 3.0f,  3.0f), float2( 2.0f, 8.0f),
                window_energy, IntegTexId, AlphaTexId, haarid); haarid++;
    sum += // ...
    OUT.color.r += sum/haarid; OUT.color.a = 1.0f; return OUT;
}
```

Fig. 7. Evaluation of the Haar-like features in the GPU (Cg)

The Haar wavelets require normalization by the energy in the classified window – both to evaluate the energy and to evaluate the features themselves, integral images are used, which is the fastest method available to our knowledge. The calculation of

the integral images constitutes the *preparatory phase* evaluated in the comparison. Please note that (to our knowledge) there is no effective way of calculating the integral image in the shading language, so the preparatory phase is implemented in the CPU. The shader evaluating the classifiers is illustrated in Figure 7.

4.2 Implementation of the LRD on Intel CPU

The performance of the GPU implementation was compared to an implementation on standard Intel CPU using MMX instructions. To simplify feature evaluation as much as possible, the convolutions of image are pre-computed and stored in the memory in such manner that all the results of the LRD grid can be fetched into the CPU registers through two 64-bit loads. This positively affects the evaluation that is performed in MMX CPU instructions (introduced by Intel).

```
row1 = convolution_{w,h}(x, y)
row2 = convolution_{w,h}(x. y+1)
pixelA = (A < 8) ? row1[A] : row2[A-8];
pixelB = (B < 8) ? row1[B] : row2[B-8];
mm0 = expand(pixelA)
mm1 = expand(pixelB)
mm2 = load(row1)
mm3 = load(row2)
mm4 = cmp(mm2, mm0)
mm5 = cmp(mm2, mm1)
mm6 = cmp(mm3, mm0)
mm7 = cmp(mm3, mm1)
mask(mm4, valid0)
mask(mm5, valid1)
mask(mm6, valid0)
mask(mm7, valid1)
mm4 = add(mm4, mm6)
mm5 = add(mm5, mm7)
mm0  = sum_pi8(mm4)
mm0 += sum_pi8(mm5)
return mm0
```

Fig. 8. Pseudocode of the MMX implementation of the LRD

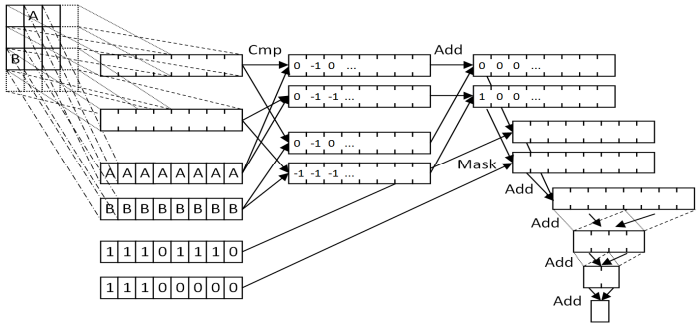

Fig. 9. Block diagram of the MMX implementation of the LRD

A pseudocode of the MMX implementation is shown in Figure 8 and the block diagram of the evaluation is shown in Figure 9. The LRD are parameterized by the feature's position (*x*, *y*) and the block size (*w*, *h*) which determine the convolution image to use. First the data from the subsequent rows of the convolved images are loaded into registers (*row1*, *row2*). The values of the rank pixels are loaded from the data (*pixelA*, *pixelB*) and expanded to the MMX registers. The registers with the data are then compared to the expanded values of *pixelA* and *pixelB* and the result of the comparison is masked (since we are interested in 3×3 grid only and 4×4 pixels were loaded). The comparison's results are summed – the resulting registers, therefore, contain the rank sum of differences of a pixel and vale *A* and *B*. Finally, the 8-bit values in the resulting registers are summed together which corresponds to the LRD response.

The code, compared to CPU without MMX, is more optimal since the values are compared in one step. The slowest step of evaluation is the expansion of 8 bit value to the 64 bit MMX 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. A similar problem is the final sum of rank differences - eight 8 bit values in a register must be summed together. Again, there is no support in instruction set.

5 Conclusions and Future Work

This paper presents an experimental implementation of the Local Rank Differences image feature on a GPU and its comparison to other approaches, specifically to the Haar-like features on the GPU.

The LRD features seem very well suitable for pattern recognition by image classifiers. They exhibit inherent gray-scale transformation invariance, ability to capture local patterns, and the ability to reflect quantitative changes in lightness of image areas. The implementation on the GPU is reasonably efficient, and a great advantage of the LRD over the common Haar wavelets in the GPU environment is the feasibility of the pre-processing stage, which has no obvious efficient solution for the Haar wavelets.

The authors of this paper are currently working on an efficient implementation of the whole WaldBoost engine utilizing the LRD features on the GPU. At the moment, the partial implementation (see section 3.1) is reasonably fast (1.6 ms looking for face in a 256×256 image). However, the authors have several clues how to improve the current implementation and increase its speed possibly several times. Also, for the purpose of efficient implementation in FPGA, some modifications to the LRD feature definition are being prepared that can also have feasible properties on the GPU – by rearranging the memory layout.

Although the implementation of the LRD on CPU, which is used in the comparison (section 4) is efficient (by using recent multimedia instructions of the processor), better implementations and variation of the LRD will also be looked for on the Intel CPU platform.

In any case, this results of the presented work definitely lead in a conclusion that that the Local Rank Differences features present is a vital low level image feature set,

which outperforms the commonly used Haar wavelets in several important measures. Fast implementations of object detectors and other image classifiers should consider the LRD as an important alternative.

Acknowledgements

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics), by the research project "Security-Oriented Research in Informational Technology" CEZMŠMT, MSM0021630528, and by Czech Grant Agency, project GA201/06/1821 "Image Recognition Algorithms".

References

1. General-Purpose Computation on GPUs, (available as of 14.4:2008 at http://www.gpgpu.org)
2. Kadir, T., Brady, M.: Saliency, Scale and Image Description. International Journal of Computer Vision, Volume 45, Number 2 / November 2001.
3. Lienhart, R., Maydt, J.: An extended set of Haar-like features for rapid object detection, ICIP02(I: 900-903).
4. Michel, P. et al: GPU-accelerated Real-Time 3D Tracking for Humanoid Locomotion and Stair Climbing, Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2007
5. Ojala, T., Pietikäinen, M., Mäenpää, T.: Gray scale and rotation invariant texture classification with local binary patterns. In: Computer Vision, ECCV 2000 Proceedings, Lecture Notes in Computer Science 1842, Springer (2000) 404-420.
6. Schapire, R., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. In: Machine Learning, 37(3):297-336, 1999
7. Sinha, S.N., Frahm, J.M., Pollefeys, M., Genc, Y.: GPU-based Video Feature Tracking And Matching, Technical Report TR 06-012, Department of Computer Science, UNC Chapel Hill, May 2006
8. Šochman, J., Matas, J.: WaldBoost — Learning for Time Constrained Sequential Detection. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2
9. Šochman, J., Matas, J.: Learning A Fast Emulator of a Binary Decision Process. In ACCV 2007.
10. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: CVPR, 2001
11. Zemčík, P., Hradiš, M., Herout, A.: Local Rank Differences - Novel Features for Image Processing, In: Proceedings of SCCG 2007, Budmerice, SK, 2007, s. 1-12