GP-GPU Implementation of the "Local Rank Differences" Image Feature

Adam Herout, Radovan Josth, Pavel Zemcik, Michal Hradis

Graph@FIT, Brno University of Technology, Bozetechova 2, Brno, CZ
{herout, ijosth, zemcik, ihradis}@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 processor 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) used in general purpose manner (GPGPU, namely CUDA in this case) as well. The paper discusses the LRD features and their properties, describes an experimental implementation of the LRD in graphics hardware using CUDA, presents its empirical performance measures compared to alternative approaches, suggests several notes on practical usage of LRD and proposes directions for future work.

1 Introduction

Current algorithms of statistical classification for object detection or pattern recognition exhibit real-time performance in detecting complex patterns, such as human faces [1], while achieving precision of detection which is sufficient for practical applications. Recent work of Sochman and Matas [2] 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 [3] algorithm into an ensemble.

In practical applications, the speed of the object detector or other image classifier is crucial. 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) [4] exist and also the field of image processing and computer vision has seen several successful uses of these techniques (e.g. [5], [6]).

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 [7], [3], [2], [1] 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 [8]. Recently, designed especially for being implemented directly in programmable or hard-wired hardware, Local Rank Differences [9] 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.

Prior to this GP-GPU [4] in CUDA [10] implementation and related research, the authors of this paper implemented the LRD features in the GPU as shaders [11]. The Cg implementation is fairly efficient, the main disadvantage was the need of complicated control of the rendering pipeline from the CPU (by issuing commands to render quads, lines or other primitives in a complex pattern that covered the searched area of the image). This disadvantage is minimized by the properties of the GP-GPU philosophy. The CUDA implementation presented here, compared to the Cg one [11] benefits also from some memory arrangement improvements, from improved training process and other minor advances.

The following Section 2 of this paper briefly presents the Local Rank Differences (see [9] for more detail) image feature. In Section 3, the notes on implementation of LRD using CUDA 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 $I(x, y) \rightarrow \mathbf{R}$. On such image, a sampling function can be defined ($x, y, m, n, u, v, i, j \in \mathbf{Z}$)

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

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:

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

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. 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, and also the set 1×1, 1×2, 2×1 and 2×2 provides good performance, enabling efficient implementations.

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):

$$V_{xy}^{mnwh}(u, v) = \left[S_{xy}^{mn}(1, 1) \;\; S_{xy}^{mn}(1, 1) \;\; \ldots \;\; S_{xy}^{mn}(1, 1) \right]. \tag{5}$$

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 $w \times h$ 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, there is no need 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 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

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

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 look-ups to the memory (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.

2.2 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 computational complexity of evaluating the features and 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 [1] 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 the 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 accesses [9] (when preprocessed as mentioned in Section 2.2) and on GPUs with three or six accesses [11].

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

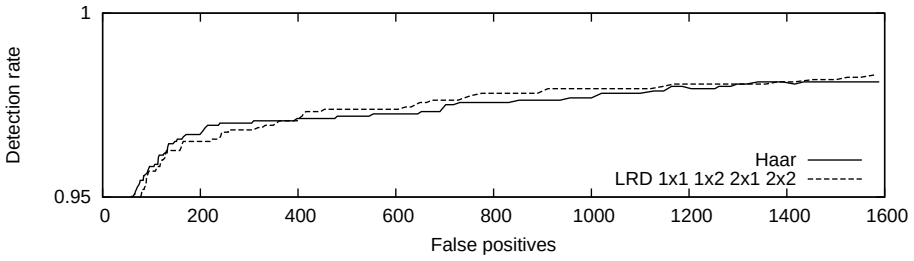

Fig. 1. ROC of two WaldBoost classifiers on a frontal face detection task. Length of the classifiers is 500 and they differ only in type of features which they use (Haar features, LRD).

normalized results, whose normalization is in fact equivalent to local histogram equalization.

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 features. The results suggest that the two types of features provide similar classification precision. One of the two classifiers compared in Figure 1 uses the same Haar wavelets as in [1] 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 using CUDA

The implementation of the LRD using CUDA corresponds with the theoretical description of the LRD in a straightforward way. It appears that a wise choice is relying on the combinations 1×1, 1×2, 2×1 and 2×2 of the LRD sampling function. Such sampling limits the descriptive power of the features slightly, but allows nice performance improvements. Thanks to the built-in texture sampling with bilinear interpolation on the usable graphics cards, sums of 2 neighboring pixels in vertical or horizontal direction or sum of four neighboring pixels consume the same amount of time as sampling just one source pixel. The scanned image can be used in such way without any pre-processing stage. The following Figure 2 contains the central part of the CUDA code implementing the LRD evaluation.

```
__device__ int GetRankDiff(unsigned int posX, unsigned int posY,
                        unsigned int BlockSizeId, unsigned int BlockABId)
{
  unsigned int mempos = threadIdx.x*9;         // address to the temp mem
  float uiBlockWidth  = _uiBlockSizeId >> 3;   // mask size
  float uiBlockHeight = _uiBlockSizeId & 7;    // mask size

  float px = posX + AbsX1 + float(BlockSizeId >> 3)/2.0f; // curr pixel X
  float py = posY + AbsY1 + float(BlockSizeId & 7)/2.0f;  // curr pixel Y

  // get sums of each matrix block (1x1, 1x2, 2x1, 2x2)
  s_fBlockSum[mempos+0] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+1] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+2] = tex2D(tImage1, px, py).x;
  px -= 2.0f*uiBlockWidth;  py+=uiBlockHeight;      // shift to next line
  s_fBlockSum[mempos+3] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+4] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+5] = tex2D(tImage1, px, py).x;
  px -= 2.0f*uiBlockWidth;  py+=uiBlockHeight;      // shift to next line
  s_fBlockSum[mempos+6] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+7] = tex2D(tImage1, px, py).x; px+=uiBlockWidth;
  s_fBlockSum[mempos+8] = tex2D(tImage1, px, py).x;

  // compute the rank difference between blockA and blockB
  int iRank = 0;
  unsigned int uiBlockA = _blockABId >> 4;
  unsigned int uiBlockB = _blockABId & 15;
  for (unsigned int bi = 0; bi < 9; bi++) {
    if (s_fBlockSum[mempos+bi] < s_fBlockSum[mempos+uiBlockA]) iRank--;
    if (s_fBlockSum[mempos+bi] < s_fBlockSum[mempos+uiBlockB]) iRank++;
  }
  return iRank;
}

__device__ unsigned char LRD()
{
  float ret = 0.0f;
  for (unsigned cid=0; cid < WCCount-1; cid++) { // loop over weak class.
    uint4 w0 = tex1D(tWeakParam, cid);   // get WeakClassifier parameters
    // Compute WeakClassifier rank and convert it to predictor value
    ret += tex2D(PredValues, GetRankDiff(w0.x, w0.y, w0.z, w0.w)+8, cid);
  }
  return (unsigned char)ret;
}
```

Fig. 2. The central part of the CUDA implementation code. The LRD() function loops over all the weak classifiers in the boosted cascade (stored in a 1D texture), gets the rank difference (by calling GetRankDiff(...) and uses the difference as an index to the table of *alpha* values obtained by training the classifier.

Compared to the previously published Cg implementation of the LRD [11], the CUDA offers some advantages. The biggest problem of the shader version was the need for rather complicated drawing of geometric primitives on the "screen" to control the object detection process. The whole of the input image needs to be covered by the primitives, but for efficiency reasons, simple drawing of one rectangle of the same size as the input image was not possible. In the GP-GPU version, all the coding and control is simpler and more straightforward. As shown later in the performance evaluation section 4, the price to pay for such feasibility of programming is the performance, or rather performance distribution depending on the input size and on the count of the weak classifiers.

4 Performance Evaluation and Analysis

To evaluate the efficiency of the presented GP-GPU implementation of the LRD, these implementations were compared: LRD on GPU using CUDA (section 3 above), LRD on GPU using Cg, Haar on GPU using Cg, and LRD on CPU/MMX. 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.

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

		frame-time [milli sec]				time-per-wc [nano sec]			
	num	lrd	lrd	haar	lrd	lrd	lrd	haar	lrd
resol	wc	CUDA	GPU	GPU	MMX	CUDA	GPU	GPU	MMX
320×200	5	13.90	0.244	0.370	17.7	43.44	0.872	1.325	55.29
320×200	10	13.63	0.527	0.469	25.0	21.29	0.942	0.839	46.71
320×200	50	13.50	2.524	3.010	82.0	2.20	0.902	1.076	40.04
640×480	5	56.87	1.173	1.642	101.8	37.03	0.810	1.134	58.55
640×480	10	53.82	2.232	2.159	149.0	17.52	0.771	0.745	51.82
640×480	50	32.95	11.066	15.731	493.0	2.14	0.746	1.086	44.05

In Table 2, a coarse comparison of the performance of the pre-processing stage is given. 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. The actual CUDA implementation works without the pre-processing, because it relies on the 1×1, 1×2, 2×1 and 2×2 set of mask dimensions. As indicated by

the graph in Figure 1, such limited set of sampling function dimensions is still sufficient and well comparable with the commonly used Haar features.

Table 2. Evaluation of the pre-processing stage (convolutions for the LRD, integral image for Haar wavelets); the pre-processing needs to be performed on every frame. Times are given in milliseconds. Note that pre-processing for the LRD is notably cheeper, even on CPU and performs excellently on GPU. Note also, that the presented CUDA implementation requires no pre-processing stage.

resol	LRDonCUDA	LRDonGPU	HAARonCPU	LRDonCPU
320×200	—	0.72	1.22	2.52
640×480	—	1.22	10.29	9.13
800×600	—	3.51	16.41	13.80
1024×768	—	3.75	27.94	24.80
1280×1024	—	4.53	45.16	37.45

Table 1 includes such regimes of evaluation, that were designed to correspond to real-time operation even on slower platforms, as is the C code for the CPU (it is considered slow compared to the parallel architectures as FPGA or GPU). In that table, the CUDA code does not perform excellently, but a tremendous increase of performance is observed when the number of weak classifiers is increased (towards 50 in the table).

Further exploration showed that the CUDA platform (at its current version – 2.0beta) exhibits relatively slow and constant load-time of the code to be executed. Also the current implementation of the boosted classifier, as indicated in Table 3, consumes constant run time for wide range of increasing number of weak classifiers – though the computational load should be linearly proportial to it. This anomaly must be further explored and may be related to some characteristic of the GPU architecture or a flaw in the compiler. However, if the boosted classifier would be a standard AdaBoost [1] or similar, the number of weak classifiers would be constantly high (hundreds). In such case the CUDA implementation outperforms tremendously any other solution available to our knowledge.

Several notes about the different alternative implementations used in the comparison follow.

GPU Implementation using Cg Shading Language An efficient memory layout is used (utilizing 3D textures and other techniques) to allow the shader to access all the nine values of the LRD mask in 3 or 6 texture look-ups. The pixel data are stored as components of the `.rgba` vector, and vector operations can be used in the calculation.

For the pre-processing task, which is constituted by several passes of subsampling by an integer fraction (see section 2.1), built-in hardware means of texture sampling are used on the GPU – see Table 2 for results.

Haar Features 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).

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, and the implementation in CUDA is also not straightforward and efficient, so the preparatory phase is implemented in the CPU. The shader code evaluating the classifiers can be found in [11].

Table 3. Behavior of the CUDA implementation for a range of image sizes and number of weak classifiers per scanned window. Two parts of the table show the time consumed per frame and this measure divided per the number of weak classifiers in the frame. The times are structured into Load time of the program, Execution time and the sum of these both.

		Frame time			Feature time		
	num of	Load	Exec	Total	Load	Exec	Tot
resol	WC	[ms]	[ms]	[ms]	[ns]	[ns]	[ns]
128×128	5	2.18	4.26	6.44	26.69	52.00	78.69
256×256	5	2.29	11.94	14.23	7.00	36.43	43.44
512×512	5	2.35	46.17	48.53	1.79	35.23	37.02
1024×1024	5	3.37	169.67	173.04	0.64	32.36	33.00
1600×1600	5	4.86	403.88	408.74	0.38	31.55	31.93
128×128	40	2.18	4.85	7.03	3.32	7.40	10.73
256×256	40	2.26	11.38	13.64	0.86	4.34	5.20
512×512	40	2.30	42.62	44.93	0.22	4.06	4.28
1024×1024	40	2.79	165.58	168.38	0.06	3.94	4.01
1600×1600	40	3.47	399.88	403.35	0.03	3.90	3.93
128×128	160	2.01	4.37	6.38	0.76	1.66	2.43
256×256	160	2.34	11.34	13.69	0.22	1.08	1.30
512×512	160	2.36	42.36	44.73	0.05	1.01	1.06
1024×1024	160	2.65	165.29	167.95	0.01	0.98	1.00
1600×1600	160	3.60	411.19	414.80	0.01	1.00	1.01
128×128	640	1.98	12.31	14.30	0.19	1.17	1.36
256×256	640	2.24	25.54	27.79	0.05	0.60	0.66
512×512	640	2.36	98.16	100.52	0.01	0.58	0.59
1024×1024	640	2.70	385.54	388.25	0.00	0.57	0.57
1600×1600	640	4.25	1028.21	1032.46	0.00	0.62	0.63

LRD on the CPU using MMX instruction set The performance of the GPU implementation was compared to an implementation on standard Intel

CPU using MMX instructions. To simplify the feature evaluation as much as possible, the convolutions of the image with the sampling function kernel are pre-computed and stored in the memory in such manner that all the results of the LRD grid can be fethed into the CPU registers through two 64-bit loads. This positively affects the evaluation that is performed in MMX CPU instructions (introduced by Intel). The authors are currently finishing experiments with the SSE instruction set, which improves the results of the MMX set slightly and whose source code is shorter and clearer. The SSE implementation's performance seems promising on the CPU but still does not compare to the GPU/shader or GP-GPU performance.

5 Conclusions and Future Work

This paper presents an experimental implementation of the Local Rank Differences image feature using the CUDA GP-GPU environment and its comparison to other approaches, specifically to a CPU implementation and to the (commonly used) Haar 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, or even ability of excellent performance without any input image pre-processing.

The authors of this paper are currently working on an efficient implementation of the whole WaldBoost engine utilizing the LRD features on the GPU, both written as shaders and using a GP-GPU environment. At the moment, the partial implementation 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. Another direction of future research is improving the training process of WaldBoost detectors using different low-level features or even combinations of several kinds of features in a single detector.

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, the 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" CEZMSMT, MSM0021630528, and by Czech Grant Agency, project GA201/06/1821 "Image Recognition Algorithms".

References

1. Viola, P., Jones, M.: *Rapid object detection using a boosted cascade of simple features.* In: CVPR, 2001
2. Sochman, J., Matas, J.: *Learning A Fast Emulator of a Binary Decision Process.* In ACCV 2007.
3. Sochman, 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
4. *General-Purpose Computation on GPUs,* `http://www.gpgpu.org`, avail.: 2008-07
5. 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
6. 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
7. Lienhart, R., Maydt, J.: *An extended set of Haar-like features for rapid object detection,* ICIP02(I: 900-903).
8. Ojala, T., Pietikainen, M., Maenpaa, 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.
9. Zemcik, P., Hradis, M., Herout, A.: *Local Rank Differences - Novel Features for Image Processing,* In: Proceedings of SCCG 2007, Budmerice, SK, 2007, s. 1-12
10. *CUDA,* `http://www.nvidia.com/cuda`, available: 2008-07
11. Polok, L., Herout, A., Zemcik, P., Hradis, M., Juranek, R., Josth, R.: *"Local Rank Differences" Image Feature Implemented on GPU,* accepted for publication in: Proceedings of Advanced Concepts for Intelligent Vision Systems (ACIVS 2008), 2008
12. Schapire, R., Singer, Y.: *Improved boosting algorithms using confidence-rated predictions.* In: Machine Learning, 37(3):297-336, 1999