J Real-Time Image Proc (2011) 6:159–170
DOI 10.1007/s11554-010-0179-0

SPECIAL ISSUE

# Real-time object detection on CUDA

**Adam Herout · Radovan Jošth · Roman Juránek · Jiří Havel · Michal Hradiš · Pavel Zemčík**

Received: 2 September 2009 / Accepted: 17 September 2010 / Published online: 8 October 2010

**Abstract** The aim of the research described in this article is to accelerate object detection in images and video sequences using graphics processors. It includes algorithmic modifications and adjustments of existing detectors, constructing variants of efficient implementations and evaluation comparing with efficient implementations on the CPUs. This article focuses on detection by statistical classifiers based on boosting. The implementation and the necessary algorithmic alterations are described, followed by experimental measurements of the created object detector and discussion of the results. The final solution outperforms the reference efficient CPU/SSE implementation, by approximately 6–8× for high-resolution videos using nVidia GeForce 9800GTX and Intel Core2 Duo E8200.

**Keywords** Object detection · Pattern recognition · Acceleration · CUDA · AdaBoost

A. Herout (✉) · R. Jošth · R. Juránek · J. Havel · M. Hradiš · P. Zemčík
Graph@FIT, Brno University of Technology,
Bozetechova 2, 616 00 Brno, Czech Republic
e-mail: herout@fit.vutbr.cz

R. Jošth
e-mail: ijosth@fit.vutbr.cz

R. Juránek
e-mail: ijuranek@fit.vutbr.cz

J. Havel
e-mail: ihavel@fit.vutbr.cz

M. Hradiš
e-mail: ihradis@fit.vutbr.cz

P. Zemčík
e-mail: zemcik@fit.vutbr.cz

## 1 Introduction

Real-time object detection is an important task with many applications, and it has been studied thoroughly. One very powerful and common approach to this task is using statistical classifiers that classify individual locations of the input image and make a binary decision: the location contains the object or it does not. The result is a set of candidate locations that is further processed, typically by a non-maxima suppression algorithm.

Viola and Jones [19] introduced the very successful face detector, which combines boosting, Haar low-level features calculated on integral image and a focus-of-attention cascade of classifiers. Their design was further developed by many researchers, most importantly for this paper; an optimal decision strategy replacing the original cascade of classification was given in [18] and a set of low-level image features suitable for hardware-accelerated implementations was introduced in [12].

Compute Unified Device Architecture (CUDA) offers a maintainable and portable way of programming general-purpose code for the graphics processing units (GPU). GPUs controlled by CUDA offer a high computational power for tasks that are highly parallel. The mutually parallel threads, however, need to share the same code and as much as possible pass through the same execution path, due to the design of the contemporary GPUs, which is essentially a 32-wide SIMD. The aim of the research presented in this paper is to evaluate the suitability of the CUDA platform for object detection by classifiers and to design efficient versions of object detection algorithms. The algorithmic adjustments and implementation details for the CUDA platform are described and the created implementation is compared to an efficient implementation using the SSE instruction set [10].

Section 2 summarizes the background of the presented work: how boosting is used in real-time object detection and the definition and usability of the Local Rank Patterns as the low-level image feature. Section 3 describes the algorithm and the detection architecture used on the CUDA platform for real-time object detection. The experimental evaluation of the algorithm, experiments carried out to fine tune the algorithm, and evaluation of the CUDA implementation compared to an efficient SSE implementation are given in Sect. 4. The paper is concluded by Sect. 5.

## 2 Background

The face detector proposed by Viola and Jones [19], which for the first time provided acceptable error rates in real-time applications, is a combination of techniques that altogether well minimize the average decision time. The classifier extracts relevant information from the image with Haar-like features, which are computed very fast and in constant time using an intermediate image representation called the integral image. To make the classifier efficient, only the most informative Haar-like features are selected from a highly over-complete set of these features by AdaBoost [6] algorithm, which also creates a classifier. AdaBoost is a general boosting algorithm, which greedily selects weak hypotheses (inaccurate classifiers) and combines them into a strong classifier—a weighted majority of votes. Viola and Jones used AdaBoost for feature selection by keeping the weak hypotheses very simple and each based only on a single Haar-like feature.

The classifier created by AdaBoost consisting of simple weak hypotheses is itself a relatively fast detector. However, it is still not fast enough for real-time applications. To reduce the computations further, Viola and Jones proposed a focus-of-attention mechanism, which they called attentional cascade. This cascade consists of gradually more complex classifiers, of which each rejects a fair portion of background samples while keeping almost all samples of the objects of interest. This reduces the computation time considerably because most of the positions scanned in detection scenarios contain background and thus are rejected very early in the first stages of the cascade.

Although very effective, the attentional cascade is not optimal. It discards all information between the consecutive stages of the cascade and also the lengths and the operating points of the classifiers are not optimal. These issues were addressed by Šochman and Matas [18] who proposed WaldBoost algorithm. WaldBoost is a combination of AdaBoost and Wald's sequential probability ratio test (SPRT). SPRT is an optimal decision strategy in the sense that it provides the fastest possible decision for a given target error rate. As a base for WaldBoost, Šochman and Matas used the real version of AdaBoost by Schapire and Singer [16], which allows confidence-rated predictions of the weak hypotheses and therefore provides faster convergence. The weak hypotheses selected by AdaBoost are regarded as measurements in the context of SPRT. The decision strategy is then a sequence of the weak hypotheses sorted by their significance, each coupled with a condition deciding if the sample can be classified with high enough confidence or further weak hypotheses have to be computed. The decisions themselves are comparisons of the tentative sum of the weak hypotheses with a threshold and as such do not represent significant computational overhead.

Except the WaldBoost algorithm, many other improvements and extensions of the Viola and Jones' detector were proposed. Addressing the non-optimality of the cascade, Brubaker et al. [2] optimize the operating points of cascade stages based on a probabilistic model of the overall cascade's performance. Bourdev and Brandt [1] propose a Soft cascade, which has the same structure as the WaldBoost classifier with monolithic structure of the classifier and a rejection threshold after each weak classifier. The authors propose to restrict the rejection thresholds to be generated by an exponential function with a single parameter. The free parameter is optimized to give the best speed for a given target detection rate and execution time. This calibration is performed after the classifier is learned. Cha and Viola [3] extend the work of Bourdev and Brandt with direct backward pruning, which sets the rejection thresholds so that they give up on those positive examples that are rejected by the complete classifier. The authors also propose multiple instance pruning, which exploits the fact that the detector is free to detect any position that closely overlaps the true position of the object.

Outside the computer vision community, a number of different boosting algorithms were proposed. However, not many of these algorithms were utilized for object detection, as they have different aims than producing a compact and fast classifier. Exceptions are real AdaBoost [16] and Gentle AdaBoost [7, 5]. Several modifications of boosting algorithms were developed specifically for object detection. Totally corrective updates for AdaBoost proposed by Šochman and Matas [17] help to create more compact classifiers by optimizing predictions of all previously selected weak classifiers in each iteration of AdaBoost. FloatBoost [14] performs backtrack steps in which redundant weak classifiers are removed. However, the backtrack steps in FloatBoost violate the weighting scheme of AdaBoost and make the final algorithm unstable.

Also, other different types of image features were proposed. Lienhart and Maydt [15] enrich the set of Haar-like features by features which are rotated by 45°. Li et al. [14] relax the strict adjacency of the rectangular regions of Haar-like features, increasing the number of features to choose from dramatically. Inspired by the work of Li et al., Huang et al. [13] proposed *sparse granular features*, which are linear combinations of sums of pixels in rectangular regions. Heuristic search is used to find suitable compositions of the regions and their mixing weights. Zhang et al. [20] used a variation of $3 \times 3$ local binary patterns, which they call multi-block LBP. Hradiš et al. introduced Local Rank Patterns (LRP) [12] that are based on ranks of intensities instead of pixel values, which makes them invariant to illumination changes. The LRP are described in detail in the next section. Haar-like features and similar simple features have been found to be inadequate for detection of objects such as pedestrians, which are characteristized by strong edges. For such objects, variations of histograms of oriented gradients [4, 11] are more suitable, though they are more expensive to compute.

## 2.1 Local Rank Patterns as the low-level image feature

Local Rank Patterns are low-level image features introduced in [12] and described in detail in [9]. They were designed to constitute an alternative to the commonly used Haar wavelets, which would be suitable for hardware implementations (in FPGA and ASIC chips). Though designed for implementation by circuitry, they perform very well also when implemented on processors and graphics chips.

The Local Rank Patterns (LRP) 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 $2 \times 2$ pixel rectangular areas).

The Local Rank Patterns have several principal advantages over the functions based on the pixel values:

- *Invariance to illumination changes*: LRP are invariant to most of the functions used for brightness and contrast adjustments/normalization in the images. More specifically, LRP are invariant to nearly all monotonic grayscale transformations.
- *Strict locality*: LRP of objects (parts of objects) do not change locally when the object's image is captured under changing conditions (similar to, for example SIFT)
- *Reasonable computational complexity*: computation and memory accesses can be optimized due to regular geometric structure. No explicit normalization is needed, which is specifically important in some classification schemes, such as WaldBoost [18].

## 2.2 Local Rank Patterns' formal definition

The following text gives a formal definition of LRP [9]. Let us consider a scalar image $f : \mathbb{Z}^2 \rightarrow \mathbb{R}$. On such an image, a *sampling function* can be defined as $(\mathbf{x}, \mathbf{u} \in \mathbb{Z}^2, g : \mathbb{Z}^2 \rightarrow \mathbb{R})$ :

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

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

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

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:

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

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.

For each element $k$ in the mask, its *rank* can be defined as:

$$R_k = \sum_{i=1}^{n} \begin{cases} 1 & \text{if } M_i < M_k \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

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 $R_k$ is obtained. Note that the ranks are independent of the local energy in the image.

The Local Rank Pattern image feature is defined as:

$$LRP(a, b) = nR_a + R_b, a, b \in \{1, \ldots, n\}. \tag{5}$$

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 $R_b$. A subset of LRP is the Local Rank Differences (LRD) image feature:

$$LRD(a, b) = R_a - R_b. \tag{6}$$

The previous definition of LRP 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 and which is both informative and efficient to compute. This particular version is used in the reported experiments. We use $3 \times 3$ regular sampling with base neighborhood $\mathbf{U}^{\text{base}}$

$$\mathbf{U}^{\text{base}} = [[0,0][0,1][0,2][1,0][1,1][1,2][2,0][2,1][2,2]], \tag{7}$$

$\mathbf{U}^{mn}$ refers to $\mathbf{U}^{\text{base}}$ where the $x$ coordinates are scaled by $m$ and $y$ by $n$. The $m$ and $n$ are selected according to the size of the sampling function. The current implementation uses

$\mathbf{U}^{11}$, $\mathbf{U}^{12}$, $\mathbf{U}^{21}$ and $\mathbf{U}^{22}$ neighborhoods and rectangular convolution kernels (sampling function) of the corresponding sizes.

## 3 WaldBoost object detection using CUDA

The efficient implementation solves problems of two main domains: the classifier operating on one fixed-size window and parallel execution of this classifier on different locations of the input image. Making the object detector with these two issues separately simplifies the design. However, some extra speed-up could possibly be gained from exchanging information between different classifier instances. The implementation presented in this article keeps the classifier instances as "black boxes" and does not share information between them. Experiments that share the information lay outside the scope of this paper and will be the subject of future research.

The problem of object detection by statistical classifiers (from the point of view of CUDA implementation) can be divided in these steps:

1. loading and representing the classifier data,
2. image pre-processing,
3. object detection,
4. retrieving results.

### 3.1 Loading and representing the classifier data

The constant data containing the classifier (image features' parameters, prediction values of the weak hypotheses summed by the algorithm and WaldBoost thresholds) could be accommodated in texture memory or constant memory of the CUDA architecture. These data are accessed on evaluation of each feature at each position, so the demands for access speed are critical. Although the access would be slightly simpler and faster if the data were stored in the *texturing memory* of the CUDA environment, the experiments showed that the overall detection times were better when the classifier data were stored in the *constant memory*. This is mainly because the image is stored in the texturing memory and is heavily accessed, so offloading the access to the classifier data and to the constant memory relieves a bottleneck of the system. The constant memory (as well as the texturing memory) is cached and the referencing to the classifier data exhibits a large locality of reference—all the threads are typically process the same weak classifier (see below).

### 3.2 Input image pre-processing

The classifier is trained on a training data set of fixed-scale examples. To be able to detect the object in different scales, the image must be scanned in multiple resolutions. The common approach [19] benefits from the ability of the Haar wavelets calculated using the integral image to be evaluated in arbitrary scales in constant time. The LRP features (Sect. 2.2) could be evaluated in a similar manner as well, but experiments (see [8]) showed that, especially on the graphics card, it was notably more efficient to construct a multi-resolution pyramid from the input image and scan it by the detector. See Fig. 1 for an illustration of how the pyramid is built. Note that some pixels of the pyramidal image, which is the actual input of the detection algorithm itself, are left unused. More compact layouts of the images of different resolutions could possibly be found and the amount of the unused pixels could be slightly reduced. However, due to the nature of the WaldBoost algorithm, only a very small number of weak classifiers ($\sim 2$) was evaluated on the unused locations, which were filled with a constant color. The time spent on the evaluation of these areas is a tiny fraction of the whole processing time, and sparing a fraction of this amount would not be worth the relatively complicated and possibly error-prone layout algorithm.

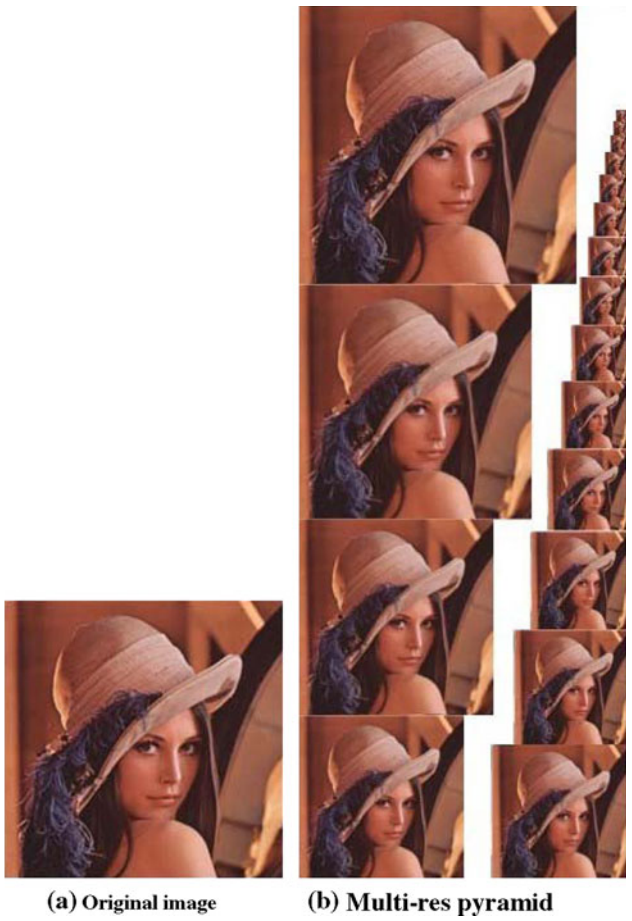


**Fig. 1** Multi-resolution pyramid constructed from the input image

## 3.3 Object detection: overall algorithm design

Programs that are run on the graphics hardware using CUDA are executed as *kernels*; each kernel has a number of *blocks* and each block is further organized into *threads*. The code of the threads consumes hardware resources: registers and shared memory; this limits the number of threads that can be efficiently executed in a block (both the maximal and minimal number of threads).

One thread computes one or more locations of the scanning window in the image. One thread could as well perform a task of smaller granularity—e.g., one or more weak classifiers, but that would imply too much inter-thread communication. The image pixels (or window locations, more precisely) are therefore divided into groups, which are calculated by the threads. The final solution divides the image into rectangular tiles, which are solved by different thread blocks (see Fig. 2). We have experimented with various layouts of the position–thread assignments (some are discussed in Sect. 3.5 below), but this design is simple and achieves no less perofmance than any other design experimented with.

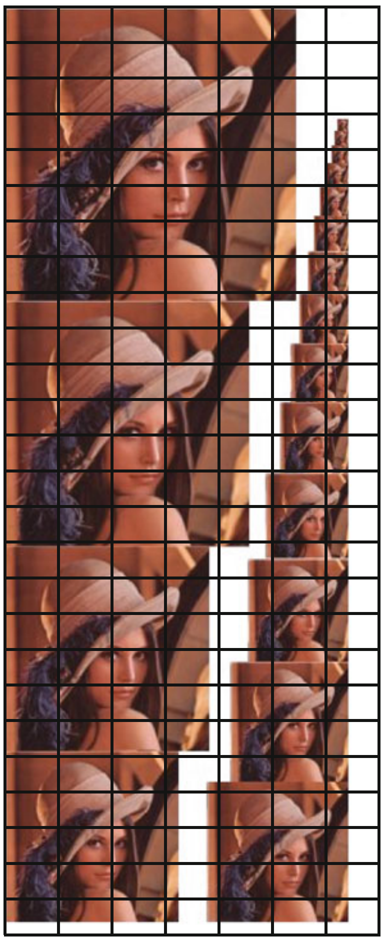


Fig. 2 Blocks of threads

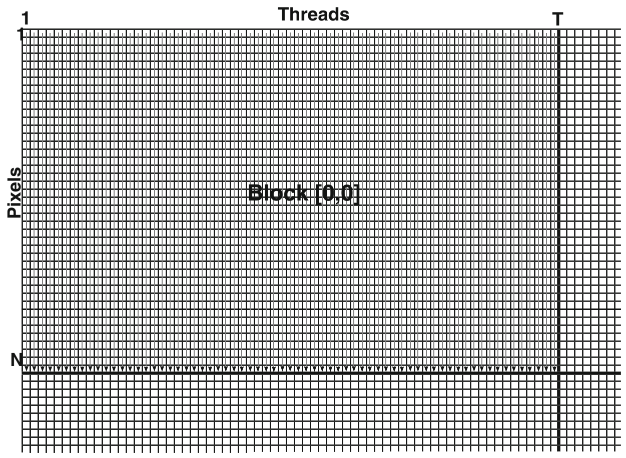


Fig. 3 Threads in a block

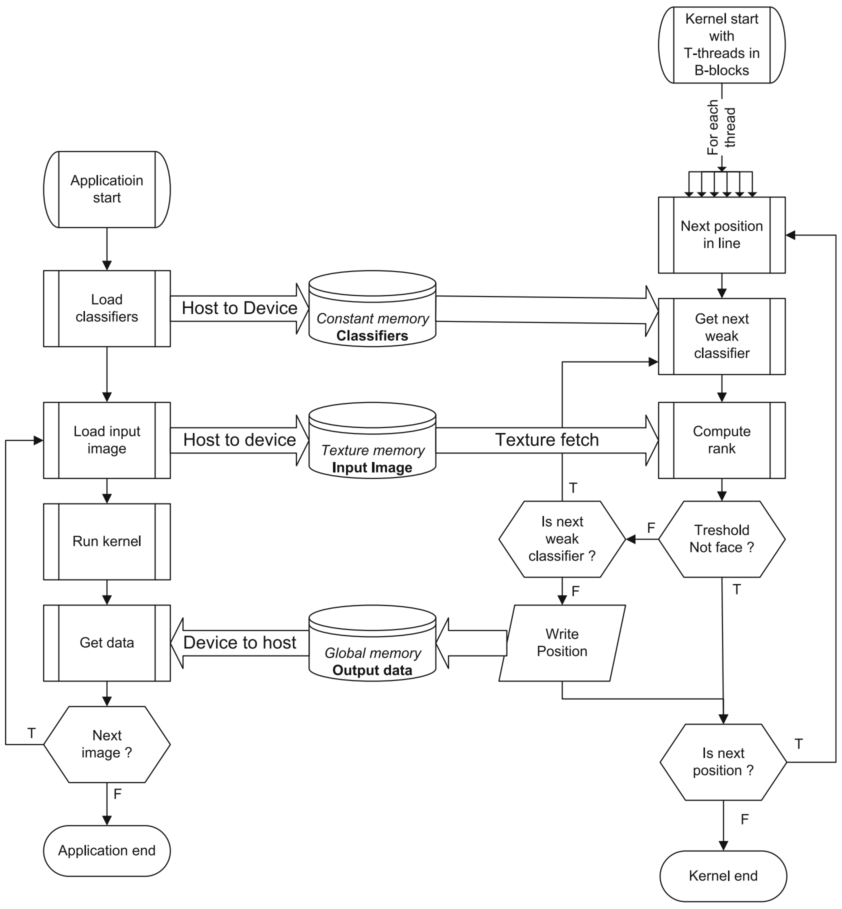
Experiments showed that the suitable number of threads per block was around 128. Executing blocks for only 128 pixels of the image would not be efficient, so we chose that one thread calculated more than one pixel—a whole column of pixels in a rectangular tile (see Fig. 3). A good consequence of this layout is an easy control of the resources used by one block: the number of threads is determined by the height of the tile, and the width controls the whole number of processed window positions by the block. The tile can extend over the whole height of the image or just a part of it. Because of the *thread rearrangement* described below in Sect. 3.4, the total number of pixels processed by one thread block is limited proportionally to the size of the *shared memory* (fast memory in one multiprocessor, which is shared between the threads of one block) and so the image is divided horizontally into several rows of tiles.

When the kernel is started, the image data are referenced by texturing units from the multi-resolution pyramid and the parameters of the classifier are read from the constant memory. When a window position is recognized as the searched object, the coordinates are written to the global memory. To avoid collisions of concurrently running threads and blocks, atomic increment (`atomicInc()`) of one shared word in the global memory is used for synchronization. This operation is rather costly, but the positive detections are so rare that this means of output can be afforded. As a consequence, the results of the whole process are at the end available in one spot of the global memory, which can be easily made available on the host computer. The whole architecture is depicted in Fig. 4.

## 3.4 Thread rearrangement

The CUDA architecture imposes some requirements on the threads to run efficiently. Because of the SIMD (single instruction, multiple data) nature of CUDA, at one time the

**Fig. 4** Object detection architecture. On the *left* side of the figure is the host process, on the *right* is the device kernel



threads must perform identical operations. In the case of branching, the threads are split into groups according to the variant of code they execute and the groups of identical execution paths are run separately. Not all threads in the block are handled in this manner, but the threads are organized into *warps*—groups of threads of fixed count (32 in the current implementations). Organization of the threads into warps is done at kernel start and the threads remain in a warp till their end.

The scanning classifiers indeed execute identical code—they load image data from identical positions (differing only by an additive offset), evaluate identical weak classifiers, compare the intermediate sum to identical thresholds, etc. However, due to the (desired) focus-of-attention capability of WaldBoost, some threads terminate with negative decision earlier than others (see Fig. 5), but the warp continues to evaluate until the very last thread terminates. This leads to relatively low utilization of the hardware resources.

For illustration, Fig. 6 contains the situation in a block after evaluation of ten weak classifiers—white pixels indicate that the classifier evaluation was terminated, and gray pixels indicate positions that are still being evaluated. Note that the threads are arranged into warps of 32 threads and all threads within one warp must evaluate the same

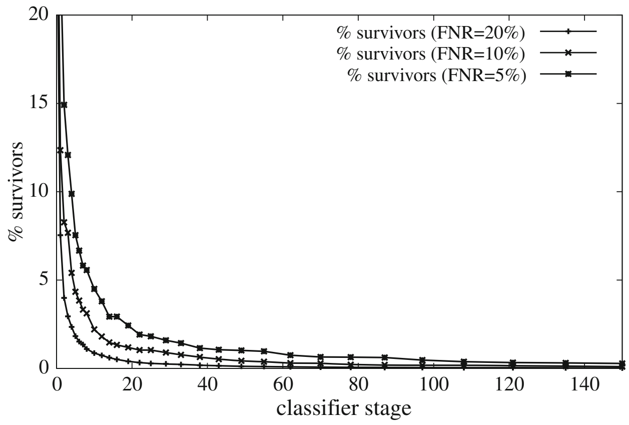


**Fig. 5** Fraction of locations in the image still evaluated after a number of evaluated image features. After the first evaluated feature, 60–70% (depending on the used classifier) locations are eliminated. The classifier is trained with different target false-negative rate (FNR 20, 10, or 5%, respectively)

code path or wait for the others. In this case, it means that the majority of threads wait for several threads exploring a fraction of the image; note that this happens in each row again. However, the situation is not tragic because of the locality of the reference, i.e., the threads evaluate locations

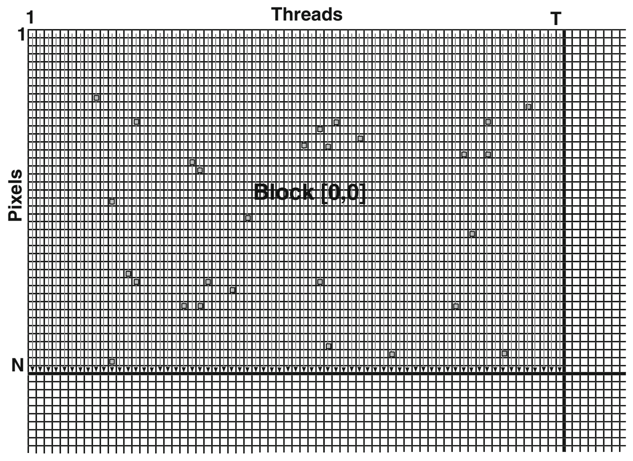
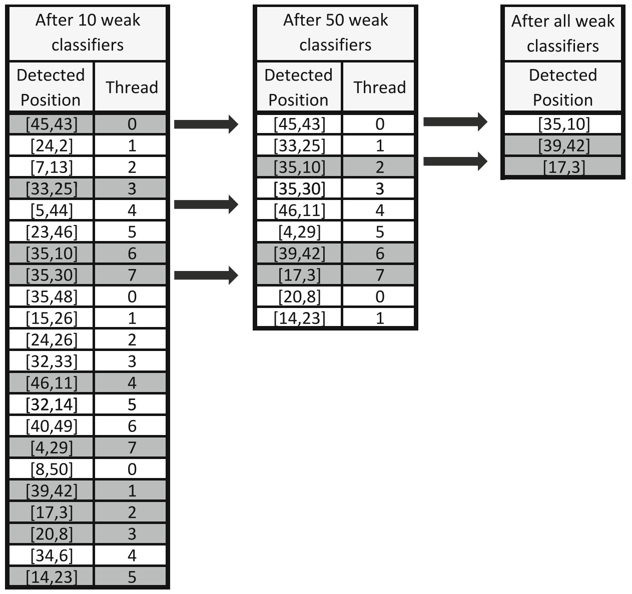


**Fig. 6** Remaining candidates for positive response after ten weak classifiers



| After 10 weak classifiers | |
|---|---|
| Detected Position | Thread |
| [45,43] | 0 |
| [24,2] | 1 |
| [7,13] | 2 |
| [33,25] | 3 |
| [5,44] | 4 |
| [23,46] | 5 |
| [35,10] | 6 |
| [35,30] | 7 |
| [35,48] | 0 |
| [15,26] | 1 |
| [24,26] | 2 |
| [32,33] | 3 |
| [46,11] | 4 |
| [32,14] | 5 |
| [40,49] | 6 |
| [4,29] | 7 |
| [8,50] | 0 |
| [39,42] | 1 |
| [17,3] | 2 |
| [20,8] | 3 |
| [34,6] | 4 |
| [14,23] | 5 |

| After 50 weak classifiers | |
|---|---|
| Detected Position | Thread |
| [45,43] | 0 |
| [33,25] | 1 |
| [35,10] | 2 |
| [35,30] | 3 |
| [46,11] | 4 |
| [4,29] | 5 |
| [39,42] | 6 |
| [17,3] | 7 |
| [20,8] | 0 |
| [14,23] | 1 |

| After all weak classifiers |
|---|
| Detected Position |
| [35,10] |
| [39,42] |
| [17,3] |

**Fig. 7** Thread rearrangement after 10 and 50 weak classifiers

close to each other and the responses of the classifiers are therefore highly correlated.

To address this issue, we propose *thread rearrangement*: at some stage of the classifier, all locations in the image that have not been classified as negative are written into a memory block shared between the threads, and another phase of the classification is started that processes only these locations. This rearrangement can be performed several times during the whole classification process (~500–1,000 stages). See Fig. 7 for an illustration of two rearrangements.

The intermediate positive (more accurately not-yet-negative) samples are stored into the shared memory of the multiprocessor similarly as the final detections are written on the global memory, as described above. The shared memory is very fast (as fast as the registers) and even the instruction of atomic increment in the shared memory is not as costly as in the case of global memory. The scope of accessibility of the shared memory is only within one block of threads, which is only appropriate, because the rearrangement happens within one block.

The instruction of atomic increment in the shared memory is only available in CUDA Compute Capability 1.1—some CUDA-enabled graphics cards would not support it. In urgent need of support of the older hardware, the serialization of the threads can be done using atomic increment in the global memory, which would degrade the performance slightly.

The exact count and locations of the rearrangement steps need to be determined experimentally. Analytical expressions can be sought for, which would determine these from some characteristics of the algorithm and the platform. Such expressions, however, would depend on many variables such as cost of one weak classifier, cost of the rearrangement, speed of the classifier in different phases of the classification process, locality of information in the processed image and many others, and yet would be only crude approximations. Section 4.2 describes experiments carried out to determine the optimal rearrangement locations and discussion on the measurement results. Generally, the major influence of the rearrangements is during the beginning of the classifier, because most of the locations are dropped out very early (see Fig. 5) and only a small fraction of computational load remains in the further stages.

## 3.5 Considerations of alternative algorithm designs

The purpose of this section is to mention several elements of the algorithmic design that were considered for the object-detection architecture, but were found to be inferior to the solution described above.

Many efficient image processing CUDA implementations use the shared memory for storing the processed image. The shared memory is very fast and is dozens of kilobytes large—tiles of the processed image can be loaded into it and processed by thread blocks. We have tried variants of this arrangement and experiments show that using the texture memory is more efficient. The texturing units perform bilinear interpolation between neighboring pixels, which can be used for evaluation of LRP. Most importantly, when using the texturing memory, the execution is as fast as when using shared memory (apparently because the bottleneck is in the calculation, not memory access), and the shared memory remains spared for other helpful purposes, as is the thread rearrangement above.

As discussed in the previous section, one of the factors limiting the performance is that the evaluation of different locations in the image is terminated after varying number of stages of the classifier and, due to the SIMD nature of CUDA, some threads wait idly. We have tried several arrangements, where the threads are assigned the work dynamically, so that when the evaluation at one location terminates, the thread "asks for" another location in the image and processes it. The idea is that the work unit would not be one location in the image, but one weak classifier. The control required by this arrangement and especially the need to synchronize the threads seem to be too complex and these attempts were much slower than the finally achieved solution with the thread rearrangement (although some threads are still idle at times).

We have performed several experiments with the placement and representation of the classifier data (constant for all images and all locations in the images)—Sect. 3.1. A texture can be used for storing it, shared memory or constant memory. Both texture memory and constant memory are cached; shared memory is very fast by itself. Placement of the classifier data into the shared memory requires preloading it upon start of each block from another location and so it is the least efficient solution. The remaining two options (texture memory or constant memory) seem to perform equally well, so storing the classifier in the constant memory is preferred to offloading the texturing units, which are used for accessing the pyramidal image.

## 4 Experimental evaluation

This section summarizes some experiments carried out to optimize and evaluate the object detection architecture defined in the previous sections. Section 4.1 discusses the influence of the image block height on the detection speed, and Sect. 4.2 describes measurements made to optimize the thread rearrangement count and locations (Sect. 3.4). The sections that follow are dedicated to comparison to an efficient SSE implementation of the same algorithm.

### 4.1 Influence of block height

As discussed in Sect. 3.3, the width of the computed block of image defines the number of threads and its height controls the number of locations computed by each thread. The measurements shown in Fig. 8 illustrate two main aspects that need to be taken into account when tuning the implementation for a target application:

- higher block height generally reduces the computation time, because it lowers the number of blocks necessary, and

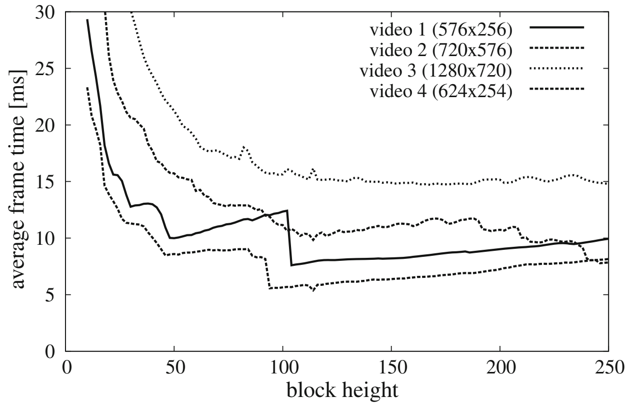


**Fig. 8** Influence of block height on detector's speed

- since the number of blocks is always integer and the blocks must share the same dimensions in CUDA, block heights that are equal or slightly higher than integer fractions of the image height are desired.

For a particular application (described among others by video resolution), a proper block height must be found according to these rules. Note also that for videos of higher resolutions, besides the two mentioned factors, other issues related to the block height influence the performance. These are related to the number of multiprocessors on the board, image dimensions, etc. However, the performance for a given block height is stable, so using a measurement similar to the one illustrated in Fig. 8, the detector can be tuned for a particular application.

### 4.2 Determining optimal thread rearrangement stages

As described in Sect. 3.4, the scanning window locations need to be rearranged several times during the classifications to better use the hardware resources. We have run a number of tests to determine optimal spots for this rearranging. The tests reported that in the current setup, no more than three rearrangements were worth performing. Figure 9 summarizes the detection times for different stages of the first, second and third rearrangement. Each of the tests shows performance of the classifier when one particular rearrangement is positioned to a given stage; the other two rearrangements are positioned optimally. Note that when a rearrangement is performed at the 1,000th stage, the given rearrangement is effectively disabled to see the performance in the case that the rearrangement (and the subsequent ones) was not performed. For each of the individual rearrangements, a minimum (or a set of values which perform similarly well and better than the others) can be found.

The experiments confirm that the first rearrangement matters the most, because it rearranges a large number of

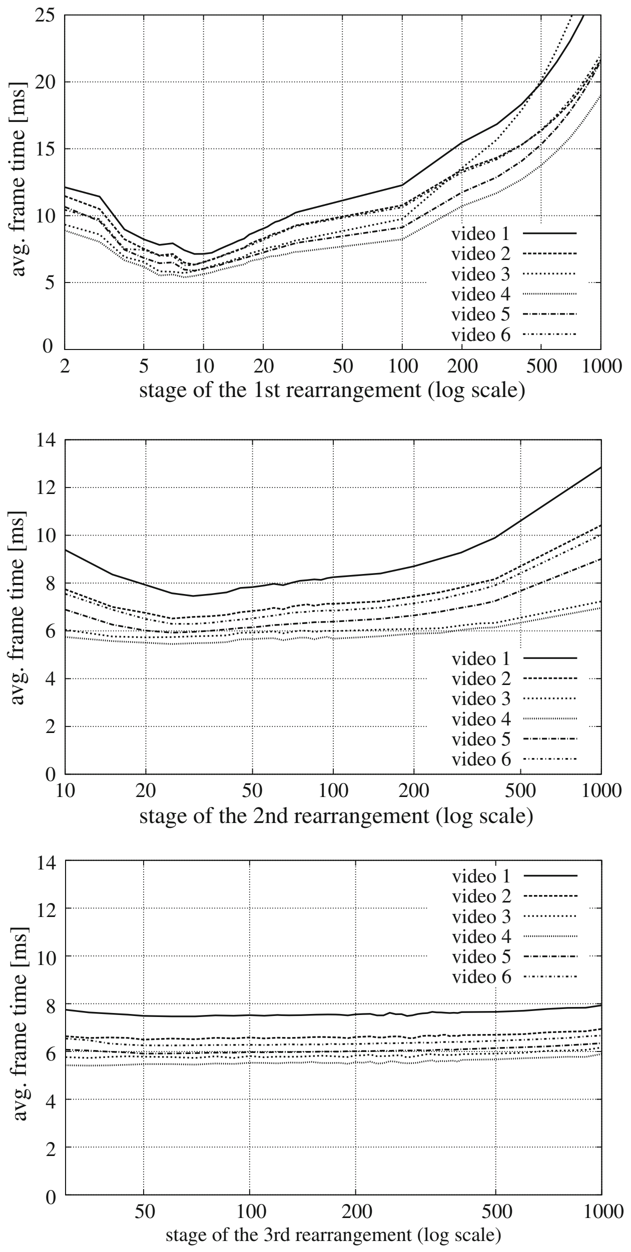


**Fig. 9** Detection time for different stages of rearrangement. The results of such measurement will be different for different classifiers

threads. Note that there can be a lower bound of the first rearrangement stage imposed by the size of the shared memory. The tests were run for six different videos (news broadcasting and movie fragments) resized to standard PAL resolution. Note the difference in the average detection times between different video contents, but rather uniform optima of the rearrangement stage. However, the optimal points for rearrangement are notably different for classifiers trained with different parameters—the shown experiments therefore do not result into fixed rearrangement spots, but rather illustrate the process of optimization for a given classifier.

## 4.3 SSE implementation used as the reference

The performance of the CUDA implementation is evaluated in comparison to an efficient SSE implementation of the same classification principle. For details on the implementation please refer to [10]; these paragraphs will summarize briefly its main characteristics.

This implementation addresses 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 algorithmic evaluation of the local ranks and their differences. It uses the SSE2 instruction set which has extensive support of instructions working with sixteen 8bit values in a single 128bit register.

To simplify feature evaluation as much as possible, the convolutions of the input image with the sampling function (Sect. 2.1) are pre-computed and stored in the memory in such a manner that all the results of the LRP grid can be fetched into the CPU registers through two 64bit loads. Compared to a naive LRP implementation, the described implementation benefits from parallel processing when calculating the ranks. The disadvantage is the limited number of convolution kernels because for each grid size a separate pre-calculated image is required. In our experiments, we used four feature sizes $1 \times 1$px, $1 \times 2$px, $2 \times 1$px and $2 \times 2$px and, therefore, four interleaved convolution images need to be pre-computed.

The evaluation part (Fig. 10) first expands selected values ($A$ and $B$) to full 128 bit length. The value of $A$ (resp. $B$) is then compared to all other values loaded from the sampling function. The comparison result is masked and the result is summed—the number of positive comparisons correspond to the rank of $A$ (resp. $B$). Results for $A$ and $B$ are then combined to produce the LRP value.

## 4.4 Comparison to the SSE implementation

This section gives some measurements done to compare the CUDA implementation with the SSE processor implementation. The time measurements were done using two setups:

- Intel Core i7-920 processor, 6 GB of DDR3 RAM and an NVIDIA GeForce GTX280 with 1 GB of GDDR3 memory.
- Intel Core 2 Duo E8200, 2666 MHz, Dual Channel 2x 2GB DDR2-1066 (533 MHz), NVIDIA GeForce 9800 GTX/9800 GTX+ (512 MB)

Table 1 contains the *pure detection* times per frame for the implementations on six videos of different content and resolution. These detection times do not include any

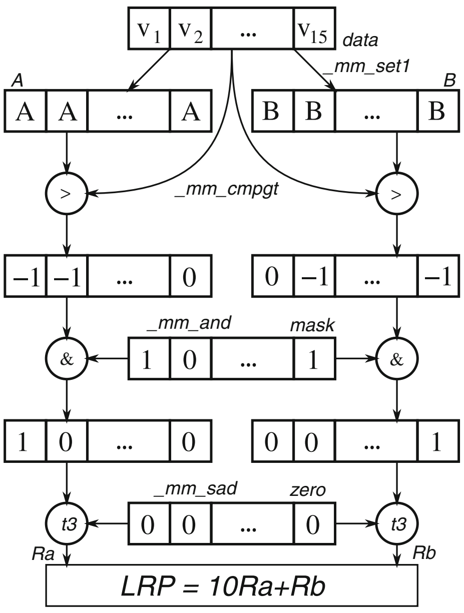


**Fig. 10** Evaluation of LRP using SSE instruction set of Intel CPU. The input is a vector `data` of 16 values, `mask` and indexes *A*, *B* of values from which LRP is calculated

**Table 1** Pure detection times [ms] (i.e., without pre-processing) on different videos, two different hardware setups and CUDA versus SSE

| Video | I7-920 | | C2D 8200 | |
|---|---|---|---|---|
| | GTX280 | | 9800GTX | |
| | CUDA | SSE | CUDA | SSE |
| A: 576 × 256 | 5.4 | 12.2 | 5.1 | 21.7 |
| B: 720 × 540 | 8.6 | 33.0 | 10.3 | 61.3 |
| C: 720 × 576 | 6.3 | 29.9 | 9.4 | 56.1 |
| D: 1,280 × 720 | 13.2 | 69.3 | 14.8 | 125.4 |
| E: 624 × 256 | 4.5 | 11.8 | 4.3 | 21.6 |
| F: 640 × 272 | 5.1 | 13.5 | 7.4 | 25.2 |

preparatory phases (video decompression, pyramid construction, image handing, ...), only the algorithmic detection times. Table 2 contains the total detection times; these are important for the actual use of the detectors. Figure 11 visualizes the pure detection times graphically.

The main observations to make from these figures are:

– CUDA outperforms the processor implementation mainly for large videos (B, C, D). This can be explained by extra overhead connected with transferring the image to the GPU, starting the kernel programs, retrieving the results, etc. These overhead operations consume typically constant time independent of the problem size, so they are better amortized in high-resolution videos.

**Table 2** Total detection times [ms] on different videos, two different hardware setups and CUDA versus SSE

| Video | I7-920 | | C2D E8200 | |
|---|---|---|---|---|
| | GTX280 | | 9800GTX | |
| | CUDA | SSE | CUDA | SSE |
| A: 576 × 256 | 5.5 | 12.6 | 5.7 | 22.1 |
| B: 720 × 540 | 8.9 | 34.0 | 12.1 | 62.4 |
| C: 720 × 576 | 6.7 | 30.9 | 11.2 | 57.3 |
| D: 1,280 × 720 | 14.0 | 71.6 | 18.7 | 128.1 |
| E: 624 × 256 | 4.7 | 12.2 | 5.0 | 22.1 |
| F: 640 × 272 | 5.3 | 14.0 | 8.2 | 25.7 |

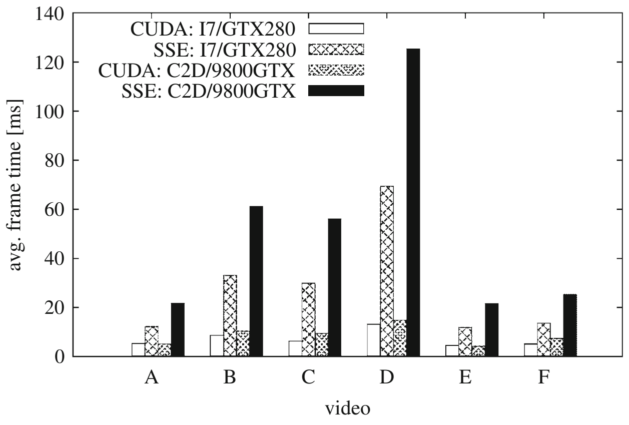


**Fig. 11** Visualization of Table 1

– The Intel I7 920 processor outperforms the Core2 Duo E8200 very significantly—it has twice as many cores and the computational speed is indeed twice as good.

## 4.5 Profiling results

This section reports some profiling results measured for the presented detector. The numerical measurements are given in Table 3. The table reports times spent on the GPU by image pre-processing (pyramid construction, *PreProc*), by the detection itself (*Detect*) and the total time per frame, including the time spent by CPU by video decompression, communication with the GPU and other tasks. Based on these values, the GPU occupancy can be calculated (column *Occupancy*), which perfectly matches the occupancy reported by the NSight profiling tool. This value depends mostly on the parallel cooperation between the CPU and the GPU. The reported occupancies were achieved by using two streams: while one of these streams waits for data, the other one works on face detection. NVIDIA Nsight was used to compute the cache miss rate

**Table 3** Profiling information: GPU occupancy, cache miss rate and partial times

| Video | I7-920 | | | | | C2D E8200 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | GTX280 | | | | | 9800GTX | | | | |
| | PreProc [ms] | Detect [ms] | Total [ms] | Occupancy [%] | CacheMiss [%] | PreProc [ms] | Detect [ms] | Total [ms] | Occupancy [%] | CacheMiss [%] |
| A: 576 × 256 | 0.164 | 5.37 | 7.26 | 76.2 | 6.1 | 0.645 | 5.09 | 6.09 | 94.9 | 5.3 |
| B: 720 × 540 | 0.396 | 8.57 | 10.30 | 87.0 | 10.0 | 1.681 | 10.37 | 12.53 | 96.5 | 7.2 |
| C: 720 × 576 | 0.421 | 6.26 | 7.88 | 84.8 | 7.7 | 1.792 | 9.42 | 11.66 | 96.1 | 5.5 |
| D: 1,280 × 720 | 0.840 | 13.19 | 16.59 | 84.5 | 7.0 | 3.960 | 14.76 | 20.73 | 90.3 | 6.1 |
| E: 624 × 256 | 0.187 | 4.53 | 6.04 | 78.1 | 4.5 | 0.701 | 4.27 | 5.27 | 94.5 | 5.1 |
| F: 640 × 272 | 0.188 | 5.13 | 6.53 | 81.5 | 4.5 | 0.761 | 7.43 | 8.53 | 96.4 | 4.9 |

from the number of cache misses and the number of cache hits.

## 5 Conclusion

This article presents algorithmic modifications and an efficient implementation of object detection by statistical classifiers based on boosting, on the CUDA acceleration platform. The described implementation is measured and the results are compared to an efficient implementation in a standard central processor, using SSE instructions. As illustrated by Fig. 11, the CUDA implementation outperforms the CPU four to eight times for high-resolution videos.

The speed-up is not as high as could be expected from the rough computational power of the GPU compared to the CPU. This is mainly due to the nature of the detection algorithm, which does not match the requirements of the CUDA and generally GPU environment. More algorithmic adjustments may be sought for, to suite the detection algorithms better to the execution platform.

The authors are currently working on an OpenCL implementation, which could bring more portability, ease of programming and software maintenance, etc. However, the CUDA architecture is at the moment more mature and stable, so the performance of the OpenCL implementation may not match the figures given in this paper.

As demonstrated by the measurements carried out, a computer equipped with one or more graphics boards with powerful GPUs can process a multiple of video signals in high resolution in real time. Using the GPU technology may therefore find its applications in surveillance and other real-world industrial tasks.

**Acknowledgments** This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics) and by the research project "Security-Oriented Research in Information Technology" CEZMSMT, MSM0021630528.

## References

1. Bourdev, L., Brandt, J.: Robust object detection via soft cascade. In: CVPR (2005)
2. Charles Brubaker, S., Mullin, M.D., Rehg, J.M.: Towards optimal training of cascaded detectors. In: ECCV06, pp. 325–337 (2006)
3. Cha, Z., Viola, P.: Multiple-instance pruning for learning efficient cascade detectors. In: NIPS (2007)
4. Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 1, pp. 886–893, Washington, DC, USA, 2005. IEEE Computer Society
5. Demirkir, C., Sankur, B.: Face detection using look-up table based gentle AdaBoost. In: Audio- and Video-based Biometric Person Authentication 2005, p. 339 (2005)
6. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: EuroCOLT '95: Proceedings of the Second European Conference on Computational Learning Theory, pp. 23–37. Springer, London, UK (1995)
7. Friedman, J., Hastie, T., Tibshirani, R.: Additive logistic regression: a statistical view of boosting. Ann. Stat. **28**(2):337–407 (2000)
8. Herout, A., Jošth, R., Zemčik, P., Hradiš, M.: GP-GPU implementation of the "Local Rank Differences" image feature. In: Proceedings of International Conference on Computer Vision and Graphics 2008, Lecture Notes in Computer Science, pp. 380–390. Springer, Verlag (2008)
9. Herout, A., Zemčík, P., Hradiš, M., Juránek, R., Havel, J., Jošth, R., Žádník, M.: Pattern recognition, recent advances, chapter low-level image features for real-time object detection, p. 25. IN-TECH Education and Publishing (2009)
10. Herout, A., Zemčík, P., Juránek, R., Hradiš, M.: Implementation of the "Local Rank Differences" image feature using SIMD instructions of CPU. In: Proceedings of Sixth Indian Conference on Computer Vision, Graphics and Image Processing. IEEE Computer Society (2008)
11. Hou, C., Ai, H.Z., Lao, S.H.: Multiview pedestrian detection based on vector boosting. In: Proceedings of the 8th Asian Conference on Computer Vision, Part I. Tokyo, Japan, 18–22 November (2007)

12. Hradiš, M., Herout, A., Zemčík, P.: Local rank patterns—novel features for rapid object detection. In: Proceedings of International Conference on Computer Vision and Graphics 2008, Number 12 in Lecture Notes in Computer Science, pp. 239–248. Springer (2008)
13. Huang, C., Ai, H.Z., Li, Y., Lao, S.H.: High-performance rotation invariant multiview face detection. PAMI **29**(4), 671–686 (2007)
14. Li, S.Z., Zhu, L., Zhang, Z.Q., Blake, A., Zhang, H.J., Shum, H.: Statistical learning of multi-view face detection. In: ECCV '02: Proceedings of the 7th European Conference on Computer Vision Part IV, pp. 67–81. Springer, London, UK (2002)
15. Lienhart, R., Maydt, J.: An extended set of haar-like features for rapid object detection. In: IEEE ICIP 2002, pp. 900–903 (2002)
16. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. Mach. Learn. **37**, 297–336 (1999)
17. Sochman, J., Matas, J.: Adaboost with totally corrective updates for fast face detection. In: AFGR04, pp. 445–450 (2004)
18. Šochman, J., Matas, J.: WaldBoost—learning for time constrained sequential detection. In: CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 2, 20–26 June (2005)
19. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 1, p. 511 (2001)
20. Zhang, L., Chu, R., Xiang, S., Liao, S.C., Li, S.Z.: Face detection based on multi-block LBP representation. In: ICB, pp. 11–18 (2007)

## Author Biographies

**Adam Herout** received his PhD from Faculty of Information Technology, Brno University of Technology, Czech Republic, where he works as an associate professor and leads the Graph@FIT research group. His research interests include fast algorithms and hardware acceleration in computer vision and graphics.

**Radovan Jošth** received his MS degree from the Faculty of Information Technology, Brno University of Technology, Czech Republic. He is currently a PhD student at the Department of Computer Graphics and Multimedia at FIT BUT. His interests include acceleration of various algorithms via GPGPU.

**Roman Juránek** received his MS degree from the Faculty of Information Technology, Brno University of Technology, Czech Republic. He is currently a PhD student at the Department of Computer Graphics and Multimedia at FIT BUT where he works as a member of the Graph@FIT Research Group. His research interests include rapid object detection and acceleration of feature extraction in computer vision.

**Jiří Havel** received his MS degree from the Faculty of Information Technology, Brno University of Technology, Czech Republic. He is currently a PhD student at the Department of Computer Graphics and Multimedia at FIT BUT. His research interests include computer graphics, functional programming and acceleration of various algorithms.

**Michal Hradiš** received his MS degree from the Faculty of Information Technology, Brno University of Technology, Czech Republic. He is currently a PhD student at the Department of Computer Graphics and Multimedia at FIT BUT where he also works as a member of the Graph@FIT Research Group. His research interests include pattern recognition, object detection and semantic image understanding.

**Pavel Zemčík** received his PhD degree from the Faculty of Electrical Engineering and Computer Science, Brno University of Technology, Czech Republic, where he works as an associate professor, vice dean and member of the Graph@FIT Group. His interests include acceleration of computer vision and graphics algorithms, programmable hardware and also applications.