Sliding-Windows for Rapid Object Class Localization: A Parallel Technique

Christian Wojek, Gyuri Dorkó, André Schulz, and Bernt Schiele

Computer Science Department
TU Darmstadt
{wojek, dorko, schulz, schiele}@cs.tu-darmstadt.de

Abstract. This paper presents a fast object class localization framework implemented on a data parallel architecture currently available in recent computers. Our case study, the implementation of Histograms of Oriented Gradients (HOG) descriptors, shows that just by using this recent programming model we can easily speed up an original CPU-only implementation by a factor of 34, making it unnecessary to use early rejection cascades that sacrifice classification performance, even in real-time conditions. Using recent techniques to program the Graphics Processing Unit (GPU) allow our method to scale up to the latest, as well as to future improvements of the hardware.

1 Introduction

In recent literature, densely sampled local descriptors have shown excellent performance, and therefore have become more and more popular for object class recognition. As the processing power of computers increases, sliding window-based techniques become more and more feasible for real-time applications. While interest point detectors offer a smart way for pre-sampling possible locations and therefore provide a sparser set for learning and recognition, the advantage of dense random sampling, or sampling on a regular lattice has been shown [1,2] to outperform sparse representations. Many of the best object class detectors use sliding window techniques (e.g, [3,4,5,6,7,8,9]), i.e., extract overlapping detection windows at each possible position, or on a regular lattice, and evaluate a classifier. The sliding window technique is, in general, often criticized as being too resource intensive, and consequently, it is often seen as unfeasible for real-time systems. However, many high dynamic automotive applications are interested in detecting pedestrians using this technique in a fast and yet robust manner [6,10,11]. In general, gradient based methods [5,6,7,8,9] perform very well, but most of them are computationally expensive.

Existing real-time solutions include incorporating simple features, that are computed rapidly, such as Haar-like wavelets in [3,4], and improving the speed via early rejection. This is typically achieved by a cascade of classifiers [3,12], or alternatively arranging features from coarse to fine for multi-resolution processing [13]. While these techniques [12,13] can make a state-of-the-art detector [5]

G. Rigoll (Ed.): DAGM 2008, LNCS 5096, pp. 71–81, 2008.

faster, their early rejection comes with drawbacks. Zhang *et al.* [13] misclassifies "harder examples", i.e. detections having lower confidence, more easily. This performance loss is compensated by running the detector with more expensive features at higher resolution. Zhu *et al.* [12] select a subset of features from a detection window using AdaBoost. Since the number of features is fixed, their method becomes computationally expensive for scanning a large number of windows, which is a typical requirement for detecting objects on small scales.

The ideal solution is to avoid rejection phases relying on coarser features, downscaled images, or other approximations, and to process the entire detection window with a strong, high-resolution classifier. In this paper we argue that methods that sacrifice classification performance in order to achieve speedups, do not stand in the long term. We show that by using parallel architectures that can be found in many recent PC's graphics processors (GPUs) we can easily obtain a speedup of 30 and more. As a case study, we present an implementation of Dalal & Triggs' Histograms of Oriented Gradients (HOG) approach using a technology called general-purpose computation on graphics processing units (GPGPU). Our performance analysis shows guidelines for better optimization, and how to avoid unnecessary overhead using GPGPU technology.

HOG descriptors are features developed for object class detection, and when combined with SVM classification it is one of the best detectors available [14]. To the best of our knowledge there is no published GPU-based HOG implementation available in the literature. However, there are a few related ongoing vision projects, that take advantage of the GPU. Examples include Lowe's Scale Invariant Feature Transform (SIFT) [15], a very similar type of descriptor, which has been implemented on GPUs in OpenVIDIA [16] and recently by Mårten Björkman [17].

2 Object Class Detection Using HOG

This section provides a brief overview of object class detection using HOG [5] features. All provided parameters correspond to experiments on *people* detection, and are similar to [5].

Detection Phase. A given test image is scanned at all scales and locations. The ratio between the scales is 1.05, and the overlapping detection windows are extracted with a step size of 8 pixels, both horizontally and vertically. HOG features are computed for each detection window, and a linear SVM classifier decides upon the presence of the object class. Finally, a robust mode estimator, a mean shift algorithm, fuses multiple detections over position and scale space (3D), and the system returns bounding boxes marked by their confidence.

Figure 1 illustrates the computation of a rectangular HOG feature for a given detection window. After image normalization and gradient computation, each detection window is divided into adjacent cells of 8×8 pixels. Each cell is represented by a 9-bin histogram of gradient orientations in the range of $0° - 180°$, weighted by their magnitudes. A group of 2×2 cells is called a block. Blocks are overlapping, and are normalized using L_2-Hys, the Lowe-style clipped L_2 norm. A block is

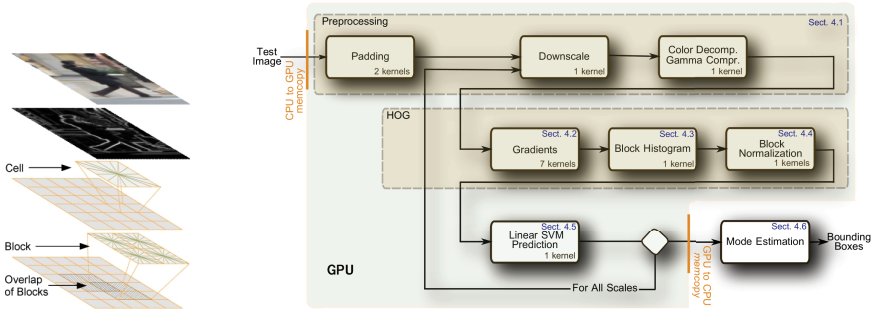


Fig. 1. A HOG descriptor (left). Steps of localization using HOG descriptors (right).

represented as a concatenation of all cell histograms, and a HOG feature as a concatenation of all blocks. For people, a detection window is 64×128 pixels. When blocks overlap 50%, i.e., 1 cell – which is a typical choice for efficient CPU implementations – a detection window consists of $7 \times 15 = 105$ blocks, and therefore the length of a HOG descriptor is $105 \times 2 \times 2 \times 9 = 3780$. To be robust to small translations, cell histograms are computed with trilinear interpolation. Gradient magnitudes are weighted by a Gaussian ($\sigma = 8.0$) centered at the middle of the given block. In case of color images, channels are separated, and orientation histograms are built using the maximum gradient of the channels.

Learning Phase. The HOG descriptors are computed similarly to detection. The learning phase differs in that there is neither need to compute the full scale-space for all images, nor to scan the images with a sliding window. Using the given annotations, normalized crops of fixed resolution are created and fed into SVM training. Negative crops are first chosen at random, or given by the dataset. After SVM training they are resampled (false positives) to create "hard examples" and retrain the SVM – a typical technique to improve the classifier by one order of magnitude [10].

3 Programming on the GPU

The term GPGPU refers to a technique that uses the graphics chip as a co-processor to perform scientific computations. The architecture of GPUs allows highly parallel computations at high speed, and thus provides an excellent platform for computer vision. GPU manufacturers have realized the need for better support of non-graphics applications, and therefore they have been working on novel architectures. In this paper our implementation is based on NVIDIA's CUDA architecture and programming model. Consequently, we use a CUDA capable card, GeForce 8800 Ultra, for our experiments. All numbers and speed measurements in this paper reflect this model. While CUDA allows us to use typical computer graphics procedures, such as vertex and fragment shaders, algorithms still need to be adapted to achieve high data level parallelism, and efficient memory access.

The graphics card GeForce 8800 Ultra, a highly multi-threaded device, consists of 16 multi-processors, each one made up of 8 processors, and therefore, capable of running 128 *threads* simultaneously. Programs running on the GPU, called *kernels*, are compiled with NVIDIA's C compiler. Kernels are launched with a user specified *grid* and *thread block* configuration. Thread blocks group up to 512 threads together, and are arranged in a grid to help complex addressing. Each block runs on the same multi-processor and therefore may share data, via on-chip *shared memories*. Each multi-processor has 8192 registers and 16384 bytes of shared memory that are dynamically allocated to threads and thread blocks. Due to these limitations and the configuration of threads, not all processors can be active all the time. The ratio that reflects how a kernel occupies the GPU is called the *occupancy* and is 100% at best. In general, higher occupancy hides the latency of global memory accesses better, and therefore often leads to better performance. Besides the on-chip shared memory there are three other types of off-chip memories. The *global memory* (768MB), also called device memory, has high latency and is not cached. *Constant memory* (65536 bytes) is typically used if all threads are accessing the same pre-computed value, and *texture memory* (65536 bytes) is optimized for 2D spatial locality. Constant and texture memories are transparently cached (8KB on-chip). Each type of memory has different access patterns, and thus programmers have to decide where the data is stored for best performance. E.g., the high-latency global memory is best accessed in continuous chunks that are aligned w.r.t. thread blocks. This is the so-called *coalesced memory access*.

4 HOG on the GPU

Figure 1 shows the steps of our implementation. First, the image is transferred from the CPU's main memory to the GPU's global memory. After initial padding, the test image is gradually downscaled, and for each scale the HOG descriptor is computed on the color normalized channels. A linear SVM is evaluated and the scores are transferred back to the CPU's memory for non-maximum suppression. Training of the SVM is done on the CPU with fixed image crops (Sect. 6), but using the GPU implementation to extract HOGs. In the following we detail the steps of our detector.

Preprocessing. Preprocessing consists of four steps. In order to detect objects that are partially cropped or near the image boundaries, extra padding is added to each side of the image. Then, the image is gradually downscaled, color channels are separated, and on each channel a color normalization is performed. In the following we discuss the implementation of each step.

Padding. After a test image is transferred to the global memory of the GPU, extra pixels are added to each side of the image. Each new pixel is computed by averaging the color of the closest 5 pixels in the previous row/column.

The implementation is split into 2, vertical and horizontal, kernels while for each 2 thread blocks are launched. Due to the pixel dependencies from previous computations, kernels compute the missing pixels in a row/column-wise manner.

Table 1. Maximum occupancy per kernel is determined by the number of registers, amount of shared memory (in bytes), and the thread block configuration. Bold numbers indicate the current limitation. Padding needs additional shared memory D, see text for details. The last column shows whether the kernel has fully coalesced memory access.

Kernel	Registers	Sh.Mem.	Thrd/Blk	Occupancy	Coal. mem.
Padding	**22**	$80 + D$	320	max.42%	Only vert.
Downscale	9	40	16×16	100%	Yes
C. Decomp., Gamma com.	7	72	16×16	100%	Yes
Horizontal Convolution	6	556	**145**	83%	Yes
Vertical Convolution	**15**	**3244**	16×8	67%	Yes
Grad.Ori.Mag. - Max	**13**	60	16×16	67%	Yes
Block Histograms	13	**2468**	16×4	50%	Yes
Block Normalization	5	312	**36**	67%	No
Linear SVM Evaluation	**15**	1072	128	67%	Yes

Our implementation loads an entire row/column into the shared memory (max. 16KB), imposing a reasonable limit on target image dimensions, 2038 pixels. Due to the limitation on the number of registers the kernel occupancy is at most 42% as indicated in Tbl. 1.

Downscale. Our downscale kernel takes advantage of the texturing unit to efficiently subsample the source image by a factor of 1.05 using linear interpolation. The target image is "covered" by thread blocks which consist of 16×16 threads. Each thread computes one pixel of the downscaled image.

Color Decomposition & Gamma Compression. This kernel's purpose is to separate the color channels of a 32-bit color interleaved image to red, green, and blue. The target pixels of the decomposed channels are also converted to floats, for further processing. Each thread corresponds to a pixel, and for efficient memory access they are grouped into 16×16 thread blocks. Since gamma compression also is a pixel-wise operation, it is integrated into this kernel for best performance, i.e., to save unnecessary kernel launches.

Color Gradients. Separable convolution kernels (from the SDK examples) compute x and y derivatives of each color channel ($3 * 2$ kernel launches). According to the guidelines, thread block sizes are fixed to 145 and 128 threads for horizontal and vertical convolutions, respectively. The occupancy is bounded by these numbers, and is 83% for horizontal and 67% for vertical processing (cf. Tbl. 1).

The next kernel computes gradient orientations and magnitudes. Each thread is responsible for computing one pixel taken as a maximum of the gradient on the three channels. For efficiency threads are grouped in 16×16 blocks.

Block Histograms. Our implementation is inspired by the `histogram64` example [18]. The basic idea of parallel histogram computation is to store partial results, so-called sub-histograms, in the low-latency shared memory. If the number of histogram bins per cell, h_c is 9, our algorithm requires $h_c * sizeof(\text{float}) = 9 * 4 = 36$ bytes of shared memory per thread. There are two pre-computed tables, Gaussian weights and bilinear spatial weighting, transferred to the texture

memory. Interpolation between the orientation bin centers is computed in the kernel. Assuming HOG block size of 2×2 cells, and 8×8-pixel cell sizes, the Gaussian weights require $16 * 16 * 4 = 1024$ bytes, and the bilinear weighting table needs $16 * 16 * 2 * 2 * 4 = 4096$ bytes.

Each thread block is responsible for the computation of one HOG block. Threads within a block are logically grouped, such that each group computes one cell histogram, and each thread processes one column of gradient orientation and magnitude values corresponding to the HOG block. Given the above mentioned cell and block sizes, in our case a thread block has 16×4 threads. This arrangement reflects the cell structure within a HOG block, and therefore provides easier indexing to our pre-computed tables.

The second part of the kernel fuses the sub-histograms to a single HOG block histogram using the same technique as `histogram64` [18]. Our configuration runs with 50% GPU occupancy, due to size limits on shared memory (cf. Tbl. 1).

Block Normalization. HOG blocks are normalized individually using L_2-Hys by a kernel, where each thread block is responsible to normalize one HOG block, and consists of the number of histogram bins per block, $h_b = 36$, threads. Squaring of the individual elements as well as the sum of the squares are computed in parallel. Keeping a full HOG block in shared memory avoids the latency of global memory accesses. The kernel runs with 67% occupancy (cf. Tbl. 1).

Linear SVM Evaluation. This kernel is similar to the block normalization kernel, since both are based on a dot product, and therefore inspired by the example `scalarProd` [18]. Each thread block is responsible for one detection window. Each thread in a block computes weighted sums corresponding to each column of the window. Partial sums are added in a pairwise element fashion, at each time using half of the threads until only one thread is left running. Finally, the bias of the hyperplane is subtracted and the distance from the margin is stored in global memory. The number of threads per block is 128.

During computation, the linear weights of the trained SVM are kept in texture memory. Keeping all values of a detection window in shared memory would occupy nearly all available space ($7 * 15 * 36 * 4 = 15120$ bytes), therefore we have decided to store one partial result of the dot product for each thread, $128 * 4 = 512$ bytes. The kernel runs with 67% GPU occupancy (cf. Tbl. 1).

Non-Maximum Suppression. The window-wise classification is insensitive to small changes in scale and position. Thus, the detector naturally fires multiple times at nearby scale and space positions. To obtain a single final hypothesis for each object, these detections are fused with a non-maximum suppression algorithm, a scale adaptive bandwidth mean shift [19].

This algorithm is currently running on the CPU. Our current time estimates suggest that it is not yet worth to run it on the GPU. However, parallelization of kernel density estimates with mode searching could itself be a research topic. In the future, we plan to run the estimation on the CPU asynchronously and simultaneously to the other computations on the GPU.

5 Discussion on GPU Implementations

This section summarizes our general experience for porting existing computer vision techniques to the GPU. The following guidelines should give an impression for what is worth, and what is hard to realize on GPU architectures.

Port Complete Sequence of Operations to the GPU. Due to the transfer overhead between the CPU and the GPU, it is not profitable to port only small portions of a complete framework to the GPU. E.g., just to run convolution on the GPU and do the rest on the CPU involves an overhead twice as much as the effective computation on the GPU. It is better to keep the data on the GPU for further processing, in particular, if we can further compress it. E.g., our transfer time of all SVM results currently takes 0.430ms even for a large image of 1280×960, however, transferring back all HOG descriptors would have taken 2 to 3 orders of magnitude more time.

Group Subsequent Steps Together. Our experience has shown that integrating kernels that access the data in the same fashion leads to significant speed improvements due to the reduced number of kernel launches. E.g., if we split the decompose colors & gamma compression, or the gradient orientation & maximum selection kernels into two, our algorithm slows down by 2ms for each. Figure 2 (left) shows that the GPU computation time for an image of size 320×240 is 13.297ms, and the program actually spends 20.179ms in the driver software, which includes the effective GPU time and the additional overhead of kernel launches and parameter passing.

Larger Data, Higher Speedup. Consequently, the more data we process, the larger is the expected speedup compared to a CPU implementation. Notice that the overhead is independent of the GPU time, and in case of longer computations, it could be relatively small. Figure 2 (right) shows the real GPU computation in relation to the kernel running time, including overhead, on different image sizes. While for a smaller image the overhead is 34% for a larger image it is only 17%.

Choose the Right Memory Type. Different memory types have different access patterns. It is important to choose the right one. E.g., the SVM evaluation may store the SVM weights in constant memory. However, since each thread accesses a different weight, it is better to use the texture memory. In our case SVM evaluation speeds up by a factor of 1.6, i.e., by 3ms for a 320×240 image using

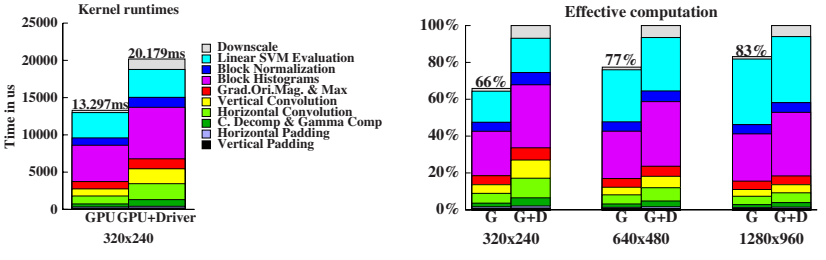


Fig. 2. Effective GPU times and calling overheads. See text for details.

the texture memory. Similarly another 3ms is won by storing the pre-computed Gaussian weights in the texture memory for the histogram computation.

Address Aligned Data. Alignment guidelines are essential for global memory access. In simple cases this usually means additional padding of images. For more complicated cases, when the same data is accessed multiple times using different patterns, the threads have to be aligned on the data, e.g., by launching more threads, and according to the alignment some of them do nothing. Our experience has shown that non-coalesced global memory access may cause a slowdown of kernels of up to 10 times.

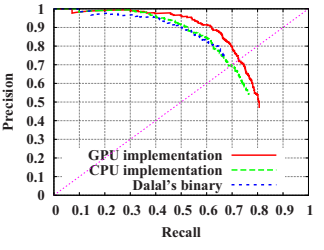
Flexibility has High Impact on Speed. Due to the above guidelines, flexibility, i.e., using not hard-coded parameters can cause significant slowdown by, e.g; non-coalesced memory access, or by increasing kernel launch overhead due to more parameters, or by more variables and computations that increase the number of registers and the amount of required shared memory, and consequently reducing occupancy.

Launch Many Threads to Scale for the Future. Finally, to scale well for future improvements of hardware a good implementation launches thousands of threads simultaneously, even if only 128 run physically parallel on the current cards.

Due to the above overheads, the sub-optimal memory access, and the rest of the computation, loading/saving, etc., one can only expect an actual speedup of a magnitude less than 128. In the following section we report real WALL times for our experiments, and measure the actual speedup of our HOG implementation.

6 People Detection Experiments

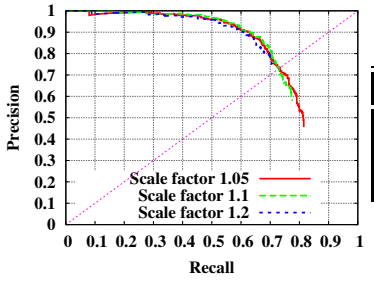
In order to verify both performance and runtime of our implementation we conducted several experiments on the INRIA Person test set [5]. The dataset contains people in different challenging scenes. For training, the dataset contains 2416 normalized positive (i.e., people cropped from 615 images) and 1218 negative images. For testing the dataset has 453 negative images, a set of 1132 positive crops, and their corresponding 288 full size images.



Implementation	WALL time
Dalal's binary	39 min 28s
Our CPU	35 min 1s
Our GPU	**1 min 9s**

Fig. 3. Performance on the INRIA Person test set

For evaluation we use precision-recall curves, which provide a more intuitive and more informative report than FPPW (false positives per window) on the performance of object localization. FPPW plots do not reflect the distribution of false positives in scale and location space, i.e., how the classifier performs in the vicinity of objects, or on background that is similar to the object context. As described earlier our system has a non-maximum suppression step to merge nearby detections to one final hypothesis, and therefore providing a clear way for evaluation. Consequently, our results are computed using only the full-sized



WALL times (per image)

Downsc. Factor	320x240	640x480	1280x960
1.05 (orig.)	29ms	99ms	385ms
1.1	20ms	59ms	216ms
1.2	15ms	38ms	133ms

Fig. 4. Increasing downscale factor on the INRIA person test set

288 positive images, and not the crops. Detections are counted as true positives if they match the ground truth annotations with less than 50% overlap error, and double detections are counted as false positives, according to the PASCAL [14] criteria.

Even though we have done our best to implement the original algorithm as close- as possible, due to restructuring the algorithm and using different precision for computations, small changes in recognition performance are expected. For this reason, our first set of experiments compares our localization results to CPU implementations in terms of recall and precision. Figure 3 (top) shows three curves. The blue dotted curve corresponds to results obtained by running the publicly available binary written by the original author; the dashed curve, performing similar to the dotted, is our CPU based reimplementation of [5]; the solid red curve is our GPU implementation, which obtains slightly better results. The improvement probably comes from floating point precision on interpolated histogram computation, since the CPU implementations use integers with rounding errors at several points, presumably for speedups. Figure 3 (right) reports total run times[1] for the test. Our implementation runs 34 times faster than Dalal's binary, and 30 times faster than our CPU reimplementation.

How can we make our detector even faster? First, one can try to improve the performance by reducing the overhead, e.g., by transferring more images at a time to the GPU, or by reducing kernel calls. Employing several GPUs at a time allows pipelining and the expected throughput can be further increased up to 4 times with currently available GPU configurations. If we are ready to trade our performance for speed, small modification on the parameters may also be sufficient. Figure 4 shows an example, when the algorithm uses a coarser scale-space than before. Speed results are reported in a more intuitive way, on a per image basis. The experiment shows, that a small adjustment of the scale factor does not influence the precision of our detector, but causes a small drop in recall. On average, on a 320×240 image the localization speeds up from 34 fps to 67 fps, i.e., by a factor of 2.0.

[1] WALL times always indicate total running time, i.e., the "real" time reported by the `time` utility on the binary.

7 Conclusions

In this paper we have shown a parallel implementation of an object class detector using HOG features. Our implementation runs 34 fps on 320×240 images, and is approximately 34 times faster than previous implementations, without any tradeoff in performance. Our experiments used one single GPU only, but due to the flexible programming model, it scales up to multi-GPU systems, such as the Tesla Computing Systems with an additional expected speedup of 2 to 4. We have also analyzed the overhead created mainly by data transfers and system calls, which defines the current limitation of these architectures.

Experiments on adjusting sliding-window parameters have shown the tradeoff between classification performance and speed: we have shown a detector that runs at 67 fps with similar precision, but a small drop in recall.

In the future, we plan to further improve our current implementation by reducing kernel launches and test on multi-GPU systems, as well as to adopt other features and classifiers to GPU-based architectures.

Acknowledgements. This work has been funded, in part, by the EU project CoSy (IST-2002- 004250).

References

1. Nowak, E., Jurie, F., Triggs, B.: Sampling strategies for bag-of-features image classification. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3954. Springer, Heidelberg (2006)
2. Tuytelaars, T., Schmid, C.: Vector quantizing feature space with a regular lattice. In: ICCV (October 2007)
3. Viola, P.A., Jones, M.J.: Robust real-time face detection. IJCV 57(2), 137–154 (2004)
4. Papageorgiou, C., Poggio, T.: A trainable system for object detection. IJCV 38(1), 15–33 (2000)
5. Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: CVPR, pp. 886–893 (2005)
6. Shashua, A., Gdalyahu, Y., Hayun, G.: Pedestrian detection for driving assistance systems: Single-frame classification and system level performance. In: International Symposium on Intelligent Vehicles, pp. 1–6 (2004)
7. Laptev, I.: Improvements of object detection using boosted histograms. In: BMVC, vol. III, pp. 949 (September 2006)
8. Tuzel, O., Porikli, F., Meer, P.: Human detection via classification on Riemannian manifolds. In: CVPR (June 2007)
9. Sabzmeydani, P., Mori, G.: Detecting pedestrians by learning shapelet features. In: CVPR (June 2007)
10. Munder, S., Gavrila, D.M.: An experimental study on pedestrian classification. PAMI 28(11), 1863–1868 (2006)
11. Gavrila, D.M., Philomin, V.: Real-time object detection for smart vehicles. In: ICCV, pp. 87–93 (1999)
12. Zhu, Q., Avidan, S., Yeh, M., Cheng, K.: Fast human detection using a cascade of histograms of oriented gradients. In: CVPR (June 2006)

13. Zhang, W., Zelinsky, G., Samaras, D.: Real-time accurate object detection using multiple resolutions. In: ICCV (October 2007)
14. Everingham, M., Zisserman, A., Williams, C., van Gool, L.: PASCAL visual object classes challenge results (2006)
15. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. IJCV 60(2), 91–110 (2004)
16. OpenVIDIA: GPU accelerated CV library, `http://openvidia.sourceforge.net/`
17. Björkman, M.: CUDA implementation of SIFT (2007)
18. NVIDIA: NVIDIA CUDA SDK code samples
19. Comaniciu, D.: An algorithm for data-driven bandwidth selection. PAMI 25(2), 281–288 (2003)