Fast Human Detection with Cascaded Ensembles

by

Berkin Bilgiç

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACUHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author...
Department of Electrical Engineering and Computer Science
January 6, 2010

Certified by...
Ichiro Masaki
Principle Research Associate
Thesis Supervisor

Certified by...
Berthold K.P. Horn
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by...
Terry P. Orlando
Chairman, Department Committee on Graduate Theses

Fast Human Detection with Cascaded Ensembles

by

Berkin Bilgiç

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Detecting people in images is a challenging task because of the variability in clothing and illumination conditions, and the wide range of poses that people can adopt. To discriminate the human shape clearly, Dalal and Triggs [1] proposed a gradient based, robust feature set that yielded excellent detection results. This method computes locally normalized gradient orientation histograms over blocks of size 16×16 pixels representing a detection window. The block histograms within the window are then concatenated. The resulting feature vector is powerful enough to detect people with 88% detection rate at 10^{-4} false positives per window (FPPW) using a linear SVM. The detection window slides over the image in all possible image scales; hence this is computationally expensive, being able to run at 1 FPS for a 320×240 image on a typical CPU with a sparse scanning methodology.

Due to its simplicity and high descriptive power, several authors worked on the Dalal-Triggs algorithm to make it feasible for real time detection. One such approach is to implement this method on a Graphics Processing Unit (GPU), exploiting the parallelisms in the algorithm. Another way is to formulate the detector as an attentional cascade, so as to allow early rejections to decrease the detection time. Zhu *et al.* [2] demonstrated that it is possible to obtain a 30× speed up over the original algorithm with this methodology.

In this thesis, we combine the two proposed methods and investigate the feasibility of a fast person localization framework that integrates the cascade-of-rejectors approach with the Histograms of Oriented Gradients (HoG) features on a data parallel architecture. The salient features of people are captured by HoG blocks of variable sizes and locations which are chosen by the AdaBoost algorithm from a large set of possible blocks. We use the integral image representation for histogram computation and a rejection cascade in a sliding-windows manner, both of which can be implemented in a data parallel fashion. Utilizing the NVIDIA CUDA framework to realize this method on a Graphics Processing Unit (GPU), we report a speed up by a factor of 13 over our CPU implementation. For a 1280×960 image our parallel technique attains a processing speed of 2.5 to 8 frames per second depending on the image scanning density, with a detection quality comparable to the original HoG algorithm.

Thesis Supervisor: Ichiro Masaki
Title: Principle Research Associate

Thesis Supervisor: Berthold K.P. Horn
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank Prof. Berthold K.P. Horn and Dr. Ichiro Masaki for their advice on this thesis. I feel very lucky to be a graduate student at MIT, and I am sure that without a certain amount of luck I would not be able to be here. I am grateful to the MIT–DuPont Fellowship program for their support during the first year of my studies.

I also would like to thank my parents Handan and Mustafa Bilgiç, my brother Anıl Bilgiç, and Ayşegül Mutluay for all the times they were there when I needed them, and for their unconditional love.

Contents

List of Figures

Chapter 1

Introduction

This thesis addresses the problem of object detection from images, in particular the detection of people. Even though this is a challenging task, the existence of various practical applications of person localization has made this field a well studied computer vision topic. As digital cameras become more wide spread, the volume of available data to digital camera owners reach such a point that digital content management presents itself as a problem. In order to automatically add tags to images to help classify the content thus has become an important research goal. Further, in a surveillance setting, it is of interest to detect and track people in order to monitor their activities. From yet another point of view, locating people in digital images might be a useful feature in a digital camera, which can enable automatic adjustment of imaging parameters of the device. Considering that many digital camera brands now include face detection routines as a built-in feature, it is conceivable that a reliable person detector can also be incorporated in future devices.

More importantly, a person detector can be employed in a computer vision based collision prevention system in a vehicle. Such a system can either warn the driver when a collision is imminent, or may directly control an automatic breaking system. However, there are several obstacles that may stand in the way of this idea from being implemented in vehicles. Even though a computer vision based realization would consist of cheap components such as digital cameras and image processing chips, it would eventually come at some additional cost to the manufacturer. If the collision prevention / warning system does not become an official requirement, then it might be a difficult for it to find wide spread use. Also, if the aforementioned system cannot satisfy a certain detection versus false positive rate requirement, it might result in unwanted warnings that can distract the driver. More importantly, if the vision system is employed in a collision prevention framework that can actively control the vehicle's brakes, false positives may even cause traffic accidents, rather than preventing them. Therefore, if such a system is to be used in an on-board application, it should achieve low false positive rates at given detection rates.

As drivers have limited response speed to stimuli, they need to have ample time to react when a dangerous situation arises. Therefore, a collision warning system that relies on computer vision needs to be able to operate in real time to provide a reaction time margin to the driver. The same consideration applies to a prevention system, since it has to make decisions on time based on the situation at hand.

We can summarize the previous criteria by stating that a person detector needs to be reliable (e.g. it should have low false positive and high detection rates) and should be able to operate in real time. The gradient based, robust feature set that Dalal and Triggs [1] proposed manages to discriminate the human shape clearly, and yields excellent detection results. Thus, it satisfies the first criterion. Their method uses a sliding-window strategy to sequentially consider each detection window within the image. The descriptor used for classification is based on computing locally normalized gradient orientation histograms over small blocks. When the block histograms are concatenated to form a feature vector for each window, a linear SVM classifier is sufficient to achieve detection rate of 88% at 10^{-4} false positives per window (FPPW). To search for all people across scales, the image is gradually downscaled and all scales are evaluated. As common to sliding-window based detectors, the Histograms of Oriented Gradients (HoG) method suffers from high computation times. Even with a sparse scanning methodology, it can process a 320×240 image at a rate of only 1 FPS on a standard CPU, hence failing to satisfy the second criterion.

In this thesis, we investigate a data parallel approach to speed up the HoG algorithm. Our method is inspired by the seminal face detector of Viola-Jones [4] and the cascade detector of Zhu *et al.* [2]. The common point of these works is to formulate the classification task as a combination sub-tasks that get progressively more complex. Each sub-task focuses on evaluating correctly the examples that the combination of all previous sub-tasks fails to classify. This cascade representation decreases the overall detection time significantly by spending very little time on detection windows that are easy to evaluate (which constitute the majority of the image), and by using more computational resources on harder detection windows (which are a small minority in the input).

The underlying data structure that makes feature evaluation for the cascade-of-rejectors type detectors very efficient is the notion of the integral image. The Viola-Jones detector uses Haar-like wavelets that depend on sums of rectangular pixel areas. When HoG features are employed in the cascade, the integral image can be again used, by converting each histogram bin to an integral image referred to as the integral histogram.

In addition to the speed up in detection time obtained with the cascade formulation and the integral image idea, we propose to exploit a second level of performance gain: parallelism. We can identify two sources of parallelism in the case of the cascade detector that utilizes HoG features. First, since the detector evaluates each detection window independently of the others, these windows can be classified in parallel. Second, it is possibly to implement integral image computation by using the work-efficient, parallel algorithm called *parallel prefix sum.* This algorithm can be utilized to sum the rows of the input image rapidly. Since we can sum each row separately, we can add a second dimension of parallelism to our implementation.

In our work, we use the NVIDIA CUDA framework to realize these ideas. CUDA is the computing platform in NVIDIA graphics processing units that enables the developers to code parallel algorithms through industry standard languages. The CUDA programming model acts as a software platform for massively parallel high-performance computing by providing a direct, general-purpose C language interface 'C for CUDA' to the programmable multiprocessors on the GPUs. When implemented on this platform, we observed a significant speed up in our cascade detector's performance.

The contributions of this thesis include:

- A quantitative comparison of different versions of the Dalal-Triggs algorithm in terms of detection speed and accuracy.
- Demonstrating that integral images can be computed efficiently on graphics processing units. We present computation performances faster by a factor of 3 to 4 relative to similar previous results.
- We present a detailed study of the first (to our knowledge) parallel implementation of cascade-of-rejectors type detectors. Even though this type of detectors does not fit well

in the GPU memory model, we do manage to obtain a 13× speed-up relative to our CPU implementation.

The organization of this thesis is as follows. In the following chapter we provide a brief background on person detection and review previous work. In Chapter 3 we detail the Dalal-Triggs algorithm and present example detection results obtained with our implementation of this method. We continue by considering two different versions of this approach, and observe how they differ from the original one. In Chapter 4, we describe the published GPU implementations of the Dalal-Triggs algorithm. Chapter 5 provides details about our cascaded HoG detector, and compares results with [1] and [2]. Efficient parallel algorithms for integral image computation are presented in Chapter 6. The real-time realization of the cascade detector on the GPU is discussed in Chapter 7, and we conclude with Chapter 8 by summarizing our results.

Chapter 2
Background

Thanks to its numerous application fields, object detection has been a well studied research topic. There is an extensive literature on object detection, and [5] gives an elegant survey on this topic. The proposed techniques can be classified according to

- Generation of initial object hypothesis,
- The features and descriptors they extract from the input, and
- The classifiers that they employ to evaluate theses features.

By following [13] and [5], we review the literature relevant to our work according to these guidelines. Section 2.1 presents object detection methods based the way they obtain the initial object location hypothesis. Section 2.2 discusses some of the important features used in object detection. Section 2.3 covers the object detectors according to the classification frameworks they utilize.

2.1 Region of Interest (ROI) Selection

One way to generate the initial object location hypotheses is by sliding a detection window over the image. To account for objects of different sizes, each image scale needs to be considered separately. Depending on the scanning density in space and scale, the number of hypotheses generated will differ. Since a typical image would typically contain thousands of detection windows, the computational costs are often too high to allow real time processing [1, 6, 17] for methods of this type. It is possible to obtain significant performance benefits by expressing the detector as an attentional cascade [2, 8, 14]. Another way to alleviate the high computational costs is to reduce the search space by imposing certain constraints on the detector, such as using the object aspect ratio or the knowledge about camera geometry [18, 19].

It is also possible to generate initial object hypotheses by extracting features from the whole input image. A common way to achieve this is to include object motion in the detection scheme. In a surveillance setting, it is possible to model the background as a mixture of Gaussians and obtain the foreground by subtraction [20]. However, the static modeling of the background cannot be employed in situations where the camera itself is in motion. To generalize to these cases, optical flow techniques can be used [21] to compute the deviation from the estimated ego-motion field [18].

2.2 Features and Descriptors

There is a vast literature on using salient points or regions to obtain a sparse representation of a given image. Such approaches extract certain local image features, which are usually called *key points* and form feature vectors on the basis of these key points for object detection. As the detection methods depend solely on these points, it is crucial to choose the key points such that they are reliable, accurate, and repeatable [5]. That is, we would like to always find the same point on an object, regardless of the changes to the image. Hence, an ideal key point detector would locate points that are insensitive to changes in scale, lightning, perspective [22]. Some of the commonly used key point detectors include the Harris corner detector (invariant to rotation and additive intensity changes, but not to scaling), Harris-Laplacian [23] (a scale invariant version of the Harris detector), and the SIFT descriptor [24], which is probably the most popular key point locator. In the SIFT method, key locations are defined as maxima and minima of the result of Difference of Gaussians (DoG) function applied in scale-space to a series of smoothed and resampled images. Then the local scale and dominant orientation given by the key point detector is used for voting into orientation histograms with weights based on gradient magnitudes. This way the SIFT descriptor attains rotation and scale invariance.

Another notable focus in object detection is using a parts-based approach. These methods involve decomposing the pedestrian appearance into semantically meaningful parts (such as head, torso, legs etc.) [25, 26]. By constraining the spatial relations among the parts in the ensemble, parts-based approaches integrate the local part responses into a final detection [5]. In more detail, for each semantic part a discriminative classifier is trained, and a model is employed to represent the geometric relations among the parts.

Finally, we discuss the detectors that encode image regions with operators similar to Haar wavelets. Among these, [6] proposes to use the absolute values of Haar wavelet coefficients at different orientations and scales as descriptors. The features employed in the detector suggested in [4] are reminiscent of Haar basis functions and form an overcomplete set for image representation. These features are computed as differences between sums of the intensity values in pixel regions. In this work, the authors also introduced the integral image, which makes it possible to calculate the sum of the intensities within any rectangular region by using only four array references, once the integral image is computed. Additionally, they expressed the detector as a progressive rejection based chain, in which the strong classifiers at each stage become more complex with depth in the chain. By adding temporal information to their detector for person detection in video, the authors further improved detection accuracy [8].

2.3 Classification

After extracting a set of features from the image with any of the aforementioned detection methods, the image regions corresponding to evaluated features need to be assigned to one of the classes, positive or negative. The mapping from the features to the labels can be constructed either by using a generative or a discriminative method. Generative methods model the pedestrian appearance in terms of its class-conditional density function. Combining this with the class priors, it is possible to infer the posterior probabilities for both classes in a Bayesian framework. On the other hand, discriminative methods focus on directly modeling the decision boundary between the object and non-object classes from training examples.

2.3.1 Generative Models

We would like to touch upon several generative detection schemes in this part. A part of them includes 2D shape cues, which are useful since they tend to reduce variations in pedestrian appearances. Some methods aim to model the shape space by a set of exemplar shapes [19, 27]. Such approaches require a large amount of examples to represent the in-class variations of the objects. It has been demonstrated that these methods can operate in real time by using efficient matching techniques based on distance transforms within pre-computed hierarchical structures.

Combining shape and texture information within a parametric appearance model has also been investigated [29]. This approach involves separate statistical models for shape and intensity variations. The intensity model is learned from shape-normalized examples and iterative error minimization schemes are used for estimation model parameters for shape and texture.

In [30], a probabilistic representation is used for all aspects of object under consideration: shape, scale, appearance and occlusion. To learn the parameters of this scale invariant object model, the expectation-maximization (EM) algorithm is used in a maximum likelihood setting. This model is used for calculating the likelihood ratios for classification.

2.3.2 Discriminative Models

One of the most popular ways to build discriminative classifiers for object detection includes Support Vector Machines (SVMs) [31]. SVMs find the separating boundary between the two classes that attains the maximum margin in the feature space. This feature space is implicitly determined by the kernel that is employed in training. In [1] and [17] linear SVMs are used for classifying the dense descriptors computed with gradient orientation histograms. Mapping the input features to higher (possibly infinite) dimensional spaces by using polynomial or radial basis functions may also yield an increase in detection quality [6, 33]. These methods employ nonlinear SVMs as classifiers within a parts-based detection framework. However, using more complex kernels relative to a linear one results in higher computational costs.

A second powerful tool in forming discriminative classifiers is Boosting [26]. This procedure generates a linear combination of a set of weak classifiers (base learners) in order to produce a strong classifier (ensemble). The linear combination is formed sequentially, thus each added base learner is chosen such that it minimizes the weighted error on the training set. The weights on the training examples are determined by evaluating the training set by the current strong classifier. Hence, if an example is classified incorrectly, it is more important for the current base learner to classify this point correctly. Boosting is used particularly in building rejection cascades in the computer vision context. In this chain of rejectors, each stage consists of a strong classifier trained on the false positives of the total detector up to and excluding the present stage. These types of classifiers are slow to train (with training times in the order of days), but they are capable of rapid execution because of the way the cascade is built. Also, special data structures such as the integral image significantly speed up the performance.

Among the detectors that rely on Boosting, [14] uses covariance matrices as object descriptors, [4] utilizes Haar-like wavelets, and [8] combines these wavelets with motion information. In [2], base learners are chosen as linear SVMs of gradient orientation histogram blocks, and AdaBoost is put to use to train strong classifiers of the rejection cascade.

As an example of detection frameworks that employ other types of discriminative classifiers, [34] can be considered. This method uses a feed-forward multi-layer neural network with adaptive local receptive field features as nonlinearities in the hidden network layer.

Chapter 3

Histograms of Oriented Gradients Algorithm

In this part, we describe the Histograms of Oriented Gradients (HoG) detection algorithm, originally proposed by Dalal and Triggs [1]. As this method comprises the foundation of the cascaded HoG detector which is of central interest to our work, we provide detail about this algorithm. In particular, we relate the steps of forming the HoG descriptor in a detection window, explain why this method is capable of achieving excellent detection results, and present exemplary detection results obtained with our implementation of the algorithm. In addition, we discuss two different flavors of the HoG method which are suggested by [9]. These two approaches are able to execute faster than the original Dalal & Triggs method, however they omit certain steps while building the window descriptors. For this reason, they are expected to be inferior to the original algorithm. We conclude with a performance comparison of our implementations of the three HoG based detectors.

3.1 The Dalal & Triggs Algorithm

The method starts by applying square root gamma correction to the input image. Color information is used when available. If the input is restricted to be grayscale, detection performance is reported to be worsened by 1.5% at 10^{-4} False Positives per Window (FPPW). The detector is sensitive to the way that the image gradients are computed. Among the methods tested (uncentered [−1, 1], centered [−1, 0, 1], cubic corrected [1, −8, 0, 8, 1] kernels, Sobel filter, and 2x2 diagonal masks), the centered gradient kernel turned out to give the best results. Also, smoothing with a Gaussian kernel prior to gradient computation is reported to significantly damage the detection quality. When the inputs are in RGB color space, gradients are computed for each channel, and the one with the largest norm is retained at each pixel.

Next, each pixel computes a weighted vote for an edge orientation histogram based on the orientation of the gradient element centered on it. These votes are accumulated into spatial bins that are called *cells*, which can either be rectangular or log-polar regions. The gradient orientations are evenly discretized into several orientation bins, lying between either 0° and 180° (unsigned gradient) or 0° and 360° (signed gradient). To reduce the aliasing, the gradient votes are trilinearly interpolated between neighboring bin centers in orientation and space. The votes that the pixels cast are a function of the gradient magnitude belonging to those pixels. In their experiments, Dalal & Triggs report that taking the square root of the magnitude or using binary edge presence voting decreases the detection quality compared to simply using the gradient magnitude itself.

It is further reported that fine orientation coding is crucial for good performance, however increasing the number of orientation bins beyond 9 does not affect the results significantly. Also using unsigned gradients as opposed to signed ones is seen to be more descriptive.

The essential part in the Dalal & Triggs algorithm is the normalization of the orientation histograms. Since gradient strengths vary over a wide range due to local illumination variations, effective local contrast normalization becomes important. The authors evaluated several local normalization schemes, all of which include grouping cells into larger structures called *blocks* and normalizing these independently. The HoG descriptor is then built by concatenating all the block responses in a detection window. Allowing overlapping between histogram blocks is also reported to improve detection performance.

The investigated block geometries are square or rectangular blocks partitioned into square or rectangular cells, and circular blocks partitioned into cells with log-polar arrangement. The rectangular formulation is denoted as R-HoG and the circular one is called C-HoG. When the R-HoG blocks are chosen to be squares, they are divided into grids of $\varsigma \times \varsigma$ cells, and each cell is made up of $\eta \times \eta$ pixels and contains β orientation bins. Authors state that optimum performance is obtained when ς is 2 or 3, and η is between 6 and 8. Additionally, downweighting the gradients near the borders of the blocks with using a Gaussian window was seen to improve detection results.

Dalal & Triggs relate that using rectangular blocks instead of square ones does not turn out to be as effective. More importantly, they have tried to employ multiple block types with different block and cell sizes. This improved the detection performance, but the size of the window descriptor increased significantly, and resulting in higher computational costs. We will later see that it is possible to take one more step along this idea by choosing which blocks sizes shall be used with Boosting.

C-HoG blocks can be thought as a form of center-surround coding, as they consist of cells in a log-polar arrangement. Authors investigated two different forms of C-HoGs, ones with a single circular cell, and ones whose central cell is divided into angular sectors (Figure 3-1). The two types of circular blocks were seen to yield the same performance in practice. At least two radial bins (a center and a surround) and four angular bins are required for good performance. Increasing the number of radial bins does not change the detection results, but adding more angular bins reduces the detection quality. The optimum radius of the central bin is 4 pixels, and Gaussian weighting does not affect the performance. Authors' results are very similar for R-HoG and C-HoG block strategies.

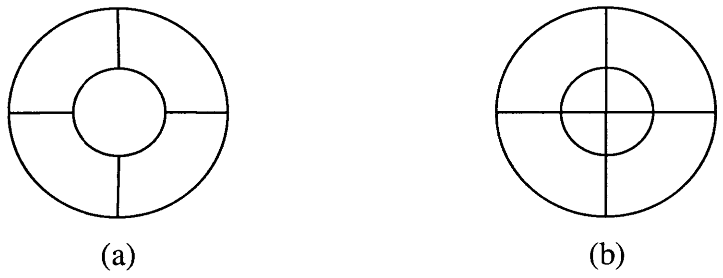


Figure 3-1: A C-HoG block with a single circular cell (a), and a block whose central cell is divided into sectors (b).

The cells that are grouped into blocks are normalized together using one of the four proposed normalization schemes. Let us denote the block descriptor vector with $\mathbf{v}$, its k-norm with $\|\mathbf{v}\|_k$ and a small positive constant with ε. The schemes of interest are then:

(a) L2-norm: $\mathbf{v} \to \frac{\mathbf{v}}{\|\mathbf{v}\|_2 + \varepsilon}$

(b) L2-Hys: L2-norm followed by limiting the maximum element in the descriptor to 0.2, then renormalizing

(c) L1-norm: $\mathbf{v} \rightarrow \dfrac{\mathbf{v}}{\|\mathbf{v}\|_1 + \varepsilon}$

(d) L1-norm followed by square root: $\mathbf{v} \rightarrow \left(\dfrac{\mathbf{v}}{\|\mathbf{v}\|_1 + \varepsilon} \right)^{1/2}$

Except for using L1-norm, all methods were seen to yield similar results. Omitting normalization altogether decreases the detection rate by 27% at 10^{-4} FPPW, which again points to the importance of local normalization. It is also noted that results are insensitive to the choice of ε over a wide range.

3.2 Data Set and Training

In their work, Dalal & Triggs also introduce a new and significantly challenging data set, 'INRIA'. This set contains 2478 training images of people in upright position with the size 64×128, as well 1218 full-size ($\geq$ 320×240) negative images for training. Also a test set of 288 high resolution images is provided to evaluate detection performance. People in the training set appear in any orientation against a wide variety of cluttered and complex backgrounds (Figure 3-2).

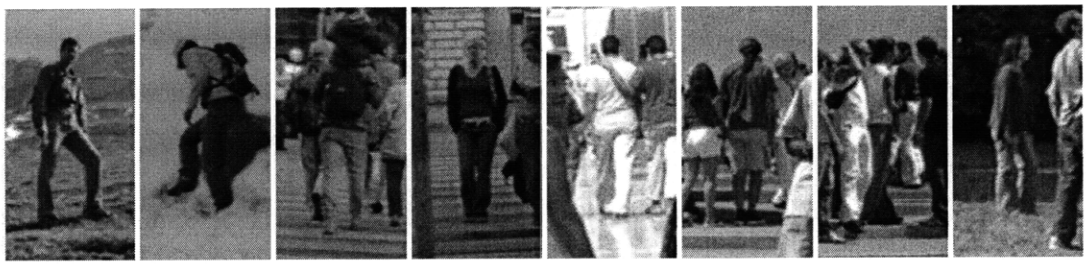


Figure 3-2: Some sample images from the INRIA positives. The data set includes images with partial occlusions, complex backgrounds with crowds, and wide range variations in pose, illumination and clothing.

Using the default R-HoG detector (RGB color space without gamma correction, centered [−1, 0, 1] gradient kernel without smoothing, $\beta = 9$ orientation bins in $0° - 180°$, blocks consisting of $\varsigma \times \varsigma = 2\times2$ cells with dimensions $\eta \times \eta = 8\times8$ pixels, Gaussian window weighting for votes using $\sigma = 8$ pixels, L2-Hys block normalization, overlapping blocks with a stride of 8×8 pixels), we can define 7×15 = 105 blocks in each detection window of size 64×128. Since each block vector has 2x2x9 = 36 dimensions, the window descriptors are 105x36 = 3780 dimensional feature vectors. Using 2478 positive samples from the INRIA data set and randomly sampling 12180 patches of size 64×128 from the full-size negatives, Dalal & Triggs trained a linear SVM classifier with the extracted window descriptors. In order to decrease the false positive rate, they further ran this classifier on the full-size negatives and added the results to the initial negative training patches. This bootstrapping process was seen to improve the detection rate by 5% at 10^{-4} FPPW. Authors also investigated the benefits of using a radial basis kernel, and observed that it improved the detection rate by 3%. However, this came at a high computational cost, which decreased the feasibility of the detector in a real time setting.

3.3 Implementation Considerations

Our software implementation of the Dalal-Triggs algorithm produces similar results to the original implementation offered by the authors. Our realization of the default R-HoG detector utilizes the functions present in the OpenCV libraries [12] and the linear SVM classifier used in classification is trained with SVMLight[10]. The steps of object localization are presented in Figure 3-3. We start by loading the RGB image into CPU's main memory and converting it 32 bit floating point format. Next, we convolve the input with the centered gradient kernels and obtain the magnitude and orientation images. The detection window scans the image with vertical and horizontal strides of 8 pixels. Since this is the same as the block stride within each detection window, we do not need to recompute the block histograms for each window separately. Thus, we precompute the block histograms and register them, then share them among neighboring detection windows to significantly decrease the detection time. Since we use trilinear interpolation for the magnitude votes, a pixel in a block can contribute to up to 8 different histogram bins (2 in orientation, times 4 in space). While locally normalizing the block histograms, L2-Hys scheme is used with clipping the maximum value of the entries to 0.2.

After obtaining all window descriptors by sharing block histograms, we evaluate them by taking the dot product between the descriptors and the SVM weights, and subtracting the hyperplane constant. We register the detection result to a binary array for later use. To search for objects in larger scales, the image is subsampled by a ratio of 1.05. We finish downscaling when at least one dimension of the image is smaller than or equal to the detection window size 64×128. We go on to process the registered binary detection results obtained from our multi-scale search. As common to sliding-window based detectors, it is possible for the detector to fire multiple times around to the vicinity of an object in both space and scale. We use a mode estimator based on the mean shift algorithm to fuse these results (non-maxima suppression).

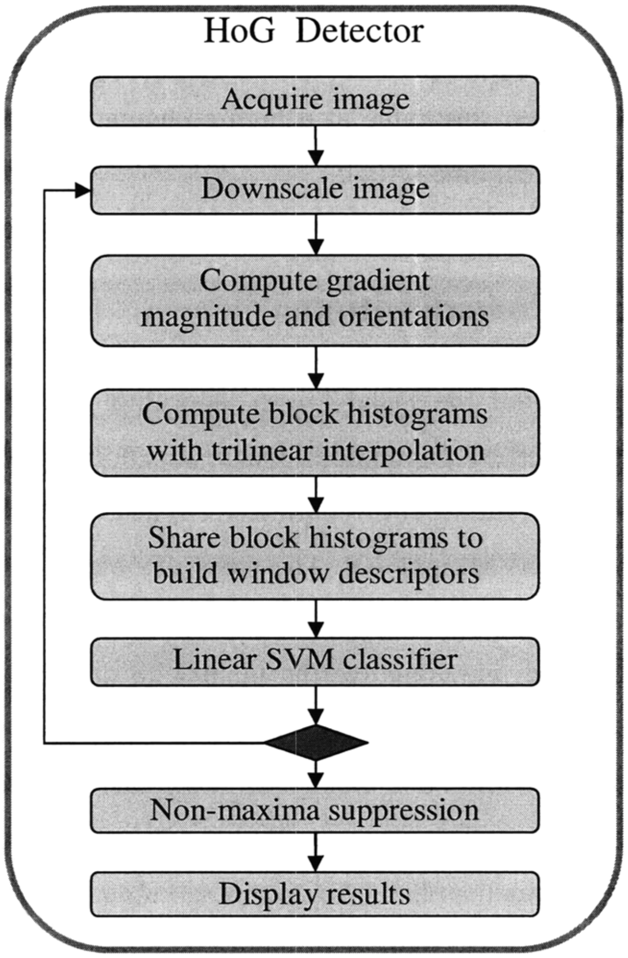
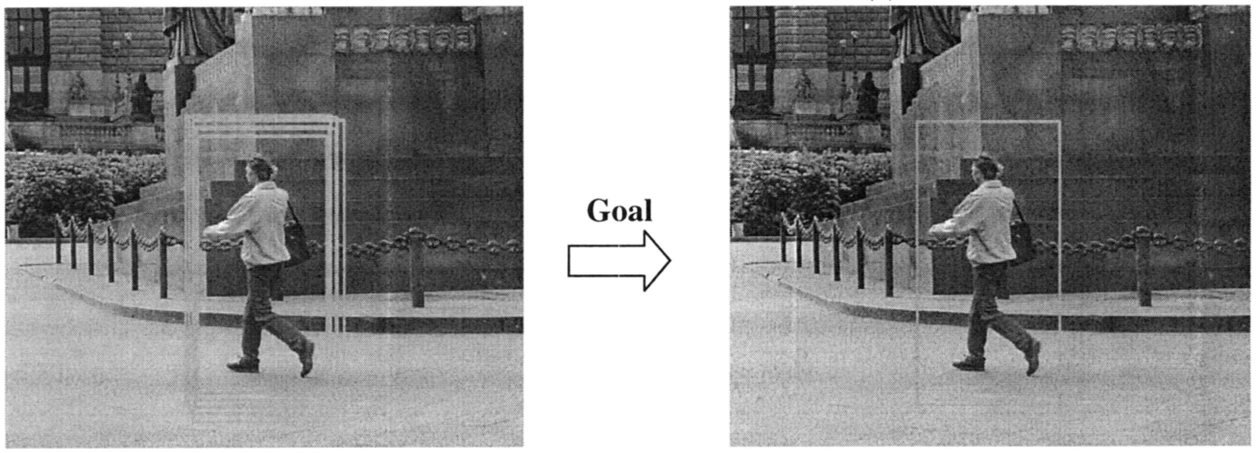

HoG Detector
Acquire image
Downscale image
Compute gradient magnitude and orientations
Compute block histograms with trilinear interpolation
Share block histograms to build window descriptors
Linear SVM classifier
Non-maxima suppression
Display results

Figure 3-3: Steps of object localization with the Dalal-Triggs algorithm [2].

Non-Maxima Suppression: To fuse overlapping detections as in Figure 3-4, we make use of the guidelines in [13]. This involves representing detections using kernel density estimation (KDE) in the 3-D position and scale space. KDE evaluates continuous densities by applying a smoothing kernel over observed data points, where the detection scores computed by evaluating the detection window with the linear SVM classifier are incorporated by weighting the detection points with these scores while computing the density estimate. The modes of the density estimate correspond to the final scales and locations of the detections. In this estimation setting, the width of the kernel that is used for smoothing should not be less than the spatial and scale strides used in detection. Also, it should not be wider than the objects to be detected so as not to confuse two nearby samples.

(a)



(a) (b)

Figure 3-4: (a) A typical detection result obtained with the Dalal-Triggs detector, without applying any post-processing. (b) Final image obtained with fusing the detections in the space and scale domain with a mean shift algorithm.

We detail the mode estimation process by following [13] and [35]. We let $\mathbf{y}_i$ denote the detections in the 3-D position-scale space for $i = 1, \ldots, n$ and define a 3×3 covariance matrix $\mathbf{H}_i$ that represents the smoothing widths for each detection. We note that the scale dimension of the detection vectors is expressed in the log domain, to account for the exponential increments in this dimension coming from the HoG detector. To fuse the detections, we search for the local modes of the density estimate by taking the smoothing kernel as a Gaussian. In this case, the kernel density estimate at point $\mathbf{y}$ is given as

$$f(\mathbf{y}) = \frac{1}{n(2\pi)^{3/2}} \sum_{i=1}^{n} |\mathbf{H}_i|^{-1/2} t(w_i) \exp\left(-\frac{D^2(\mathbf{y},\mathbf{y}_i,\mathbf{H}_i)}{2} \right) \tag{3.1}$$

where

$$D^2(\mathbf{y},\mathbf{y}_i,\mathbf{H}_i) \equiv (\mathbf{y}-\mathbf{y}_i)^T \mathbf{H}_i^{-1} (\mathbf{y}-\mathbf{y}_i) \tag{3.2}$$

is the Mahalanabois distance between **y** and **y**$_i$ and

$$t(w) = \begin{cases} 0, & if \ w < 0 \\ w, & otherwise \end{cases} \tag{3.3}$$

stands for the transformation function used for evaluating the detection weights. In the case of SVM classifiers, the weights w_i are simply obtained by computing the discriminant for the window descriptor vectors. Let us also define the following weights

$$\overline{w}_i(\mathbf{y}) = \frac{|\mathbf{H}_i|^{-1/2} t(w_i) \exp(-D^2(\mathbf{y},\mathbf{y}_i,\mathbf{H}_i)/2)}{\sum_{i=1}^{n} |\mathbf{H}_i|^{-1/2} t(w_i) \exp(-D^2(\mathbf{y},\mathbf{y}_i,\mathbf{H}_i)/2)} \tag{3.4}$$

and let

$$\mathbf{H}_h^{-1}(\mathbf{y}) = \sum_{i=1}^{n} \overline{w}_i(\mathbf{y}) \mathbf{H}_i^{-1} \tag{3.5}$$

be the data weighted harmonic mean of the covariance matrices **H**$_i$ evaluated at **y**. Using these, the mode can be iteratively estimated by the update equation

$$\mathbf{y}_m = \mathbf{H}_h(\mathbf{y}_m) \sum_{i=1}^{n} \overline{w}_i(\mathbf{y}_m) \mathbf{H}_i^{-1} \mathbf{y}_i \tag{3.6}$$

by starting from an initial **y**$_i$ until convergence. The final **y**$_m$ is the estimated fusion of the detection results.

To run this algorithm, we need to specify the amount of uncertainty expressed by **H**$_i$ for each detection vector. Again following [13], we constrain these matrices to be diagonal ones according to

$$\mathbf{H}_i = \begin{bmatrix} (\exp(s_i)\sigma_x)^2 & 0 & 0 \\ 0 & (\exp(s_i)\sigma_y)^2 & 0 \\ 0 & 0 & \sigma_s^{\ 2} \end{bmatrix} \tag{3.7}$$

where s_i is the scale of the i^{th} detection in log scale and σ_x, σ_y, and σ_s are smoothing parameters supplied by the user. We use parameter setting $\sigma_x = 8$, $\sigma_y = 16$ and $\sigma_s = \log(1.3)$ in our implementation. By noting that the final detections are displayed in terms of bounding boxes based on the estimated modes, we conclude the discussion of the Dalal-Triggs algorithm and present several detection results.

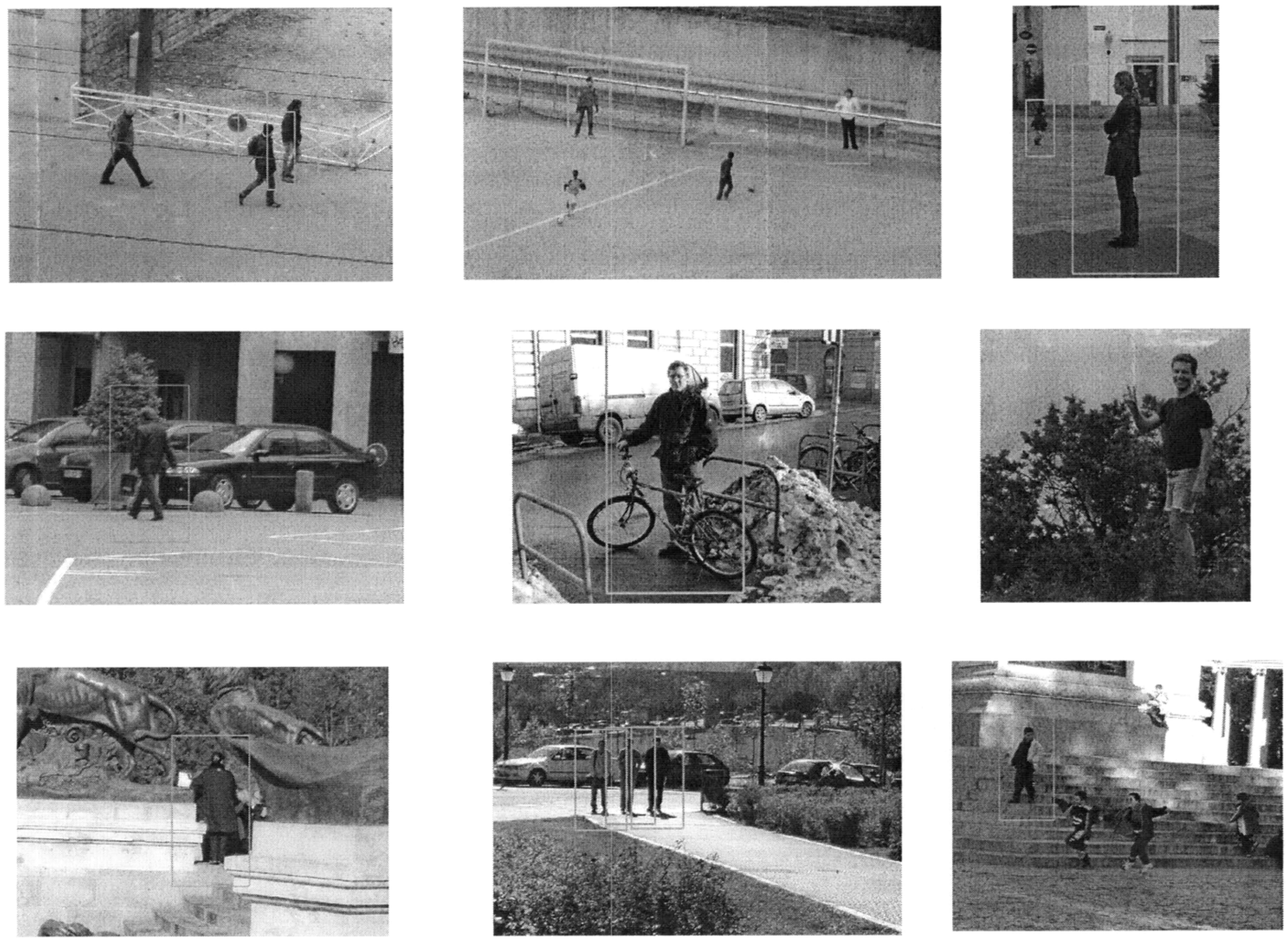


Figure 3-5: Several detection results obtained with our implementation of the Dalal-Triggs algorithm. The downscaling ratio is 1.05 and the window stride is 8×8.

3.4 Modifications on the Dalal-Triggs Algorithm

Apart from the original HoG algorithm proposed by Dalal & Triggs, we consider two different flavors of this method. These modified versions are proposed by [9], and they differ from the original mainly in the way that the block histograms are computed. In this section, we briefly detail the source of this difference, and based on our CPU implementations of these methods, we relate how they compare against the Dalal & Triggs algorithm in terms of detection speed and accuracy (detection rate vs. false positive rate).

3.4.1 Detector with Histograms Precomputing and Caching

This method corresponds to the CPU-HC detector in [9]. Its basic differences from the original algorithm are that, trilinear interpolation of magnitude orientation votes and the Gaussian weighting of histogram blocks are omitted. Secondly, it focuses on grayscale images without including color information. We have seen that when the block strides within a detection window and the stride of the detection window itself inside the image are equal, it is possible to share block histograms among detection windows. The same idea is utilized in the case of CPU-HC, however the shared data structures are the cell histograms now. This is made possible by the fact that the cell histograms are not coupled via the Gaussian weights and interpolation anymore.

More formally, let us denote the strides of the detection windows in the horizontal and vertical directions by d. Moreover, let the cell sizes, denoted with r, be equal to d such that $d = r = 8$ pixels. In this setting, CPU-HC computes histogram *cells* independently of the scan windows, and shares them among different windows during classification, as opposed to sharing histogram *blocks* as the Dalal-Triggs algorithm does.

3.4.2 Detector with Integral Images

This variant of the HoG detector is denoted as CPU-II in [9]. As its counterpart CPU-HC does, this method omits the usage of Gaussian weights, vote interpolation and RGB color space information. The major difference of this approach is that, it uses the integral histogram idea [2] to improve efficiency. For each histogram bin, an integral image of gradients is computed. In our case, we have 9 integral images to store the histogram bins. As the sum of the pixels within any rectangular region inside the integral image can be computed with four array access operations (a point we will further elaborate in Chapter 4), the window descriptor for any window can be computed using $4 \cdot 9 \cdot (w/r) \cdot (h/r)$ references. In this relation, w and h are the width and the height of the detection window, and r is again the cell size.

An important advantage of this approach is that it does not require the assumption of having equal window and block strides. Hence, it is likely to present a significant speed up relative to the Dalal-Triggs and CPU-HC algorithms in the case where this assumption is not present. However, it is less advantageous to use CPU-II when $d = r$.

The omission of Gaussian weighting and vote interpolation should make the CPU-HC and CPU-II methods inferior to the original one. In Figure 3-6 we quantify this point by comparing the ROC curves of our implementations of these two methods with the Dalal-Triggs algorithm. In the low FPPW region, the Dalal-Triggs approach has higher accuracy, but in the high FPPW region the other two methods catch up with the original algorithm. Table 3-1 presents detection time measurements of all three approaches, based on our implementations. We note that the CPU-HC method can run almost twice as fast as the Dalal-Triggs algorithm.

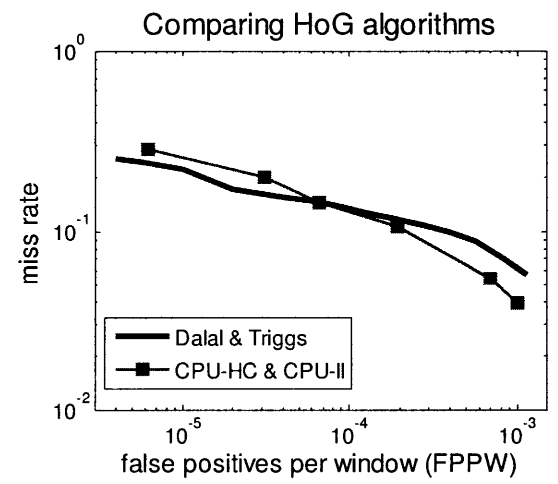


Figure 3-6: ROC curves for the Dalal-Triggs algorithm and its modified version that omits trilinear interpolation and Gaussian weighting.

Method	Scaling : 1.05	Scaling : 1.1	Scaling : 1.2
Dalal & Triggs	24.24 s	13.07 s	7.61 s
CPU-II	21.73 s	11.65 s	6.73 s
CPU-HC	13.06 s	7.08 s	4.09 s

Table 3-1: Comparing processing times of HoG based person detector based on our implementation using a 1280×960 input image. The CPU-HC method outperforms the other two, running almost twice as fast.

Chapter 4

Histograms of Oriented Gradients on the GPU

The object localization algorithm proposed by Dalal & Triggs [1] is one of the most popular detectors in the computer vision literature. Apart from being easy to implement, it is among the state of the art detectors in terms of detection rates at given false positive ratios [5]. Unfortunately, it suffers from the fact that it is very costly to evaluate dense HoG descriptors in a sliding window fashion, even when sharing blocks among detection windows is possible. This chapter discusses a method to decrease the detection times of this detector, by formulating the problem so as to expose the maximum amount of parallelism.

Even though sliding-window type object hypothesis generation is costly, it is prone to be expressed as a parallel process. This is because the detection windows can be evaluated independently of each other. Additionally, image processing applications such as convolution, integral image generation, and subsampling are implicitly parallel operations, since they allow certain regions (e.g. rows and columns) in the input image to be processed independently of the others. Luckily, there is a hardware platform that enables the developers to exploit such parallelisms.

Here, we investigate the possibility of implementing parallel algorithms on Graphics Processing Units (GPUs). In particular, we examine three different GPU implementations of the Dalal-Triggs algorithm we have compiled from the literature [3, 9, 36] and report on their performances.

4.1 The GPGPU Paradigm

The term GPGPU (*G*eneral *P*urpose computing on *G*raphics *P*rocessing *U*nits) refers to using graphics processing units to accelerate non-graphics problems. The many-core architecture of the new generation GPUs enables them to execute thousands of threads in parallel. These threads are managed with zero scheduling overhead and are lightweight compared to CPU threads. To fully utilize the great computational horsepower of the GPUs, thousands of threads need to be launched within each parallel routine. The potential benefit of employing a graphics card can be quantified by the theoretical floating point performance of the device. Whereas modern CPUs have peak performances on the order of 10 GFLOPs, commercial GPUs can exceed the 1 TFLOP limit. The CUDA framework is a widely adopted by for programmers that develop GPGPU applications.

CUDA (*C*ompute *U*nified *D*evice *A*rchitecture) is the computing platform in NVIDIA graphics processing units that enables the developers to code parallel algorithms thorough industry standard languages. The CUDA programming model acts as a software platform for massively parallel high-performance computing by providing a direct, general-purpose C language interface 'C for CUDA' to the programmable multiprocessors on the GPUs. According to this model, parallel portions of an application are executed as *kernels*. CUDA allows these kernels to be executed multiple times by multiple *threads* simultaneously. A typical application would use thousands of threads to achieve efficiency. In the CUDA terminology, the GPU is referred to as the *device* while the CPU is called the *host*.

At the core of the model lie three abstractions – a hierarchical ordering of thread groups, on-chip shared memories, and a barrier instruction to synchronize the threads active on a GPU multiprocessor. In order to scale to future generation graphics processors, multiple threads are grouped in *thread blocks* and multiple blocks reside in a *grid* that has user specified dimensions. Thread blocks may contain up to 512 threads, and the threads inside a block can communicate via low latency, on-chip *shared memory*. To prevent read-after-write, write-after-read, and write-after-write hazards, `__syncthreads()` command can be used to coordinate communication between the threads of the same block. A group of 32 threads that are executed physically simultaneously on a multiprocessor is called a *warp*.

There are six different memory types in the CUDA model that provide flexibility to the programmer. Apart from the shared memory (16kB) that is visible to all threads within a block, each thread has access to a private local memory and registers. Additionally, there are three types of off-chip memory that all threads may reach. The *global memory* (1792MB) has high latency and is not cached. The *constant memory* (64kB) is cached and particularly useful if all threads are accessing the same address. The *texture memory* is also cached and optimized for spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. Textures can be bound to either linear memory or CUDA arrays; hence their maximum sizes depend on the particular data structure they are used with. Textures also provide hardware interpolation, which has very small performance cost.

The fact that shared memory resides on the multiprocessors where computations are performed whereas the global memory types are off-chip is reflected by the vast difference in their access speeds. It takes about 400 to 600 clock cycles to issue a memory instruction for the global memory, but the same operation occurs about 150 times faster in the shared memory. Therefore, if the same addresses need to be accessed multiple times, it would be beneficial to reach them via the shared memory.

Apart from the flexibilities it offers, the CUDA model has also certain drawbacks that limits its usefulness. Among these, memory coalescence criterion for accessing global memory, bank conflicts that may arise when using shared memory, branch divergence that can be caused by using conditional statements and the large overhead required for memory transfer between the CPU and the GPU can be counted. We briefly touch upon these points, barrowing from [37].

Coalescence: Since the global memory is not cached, the way that it is accessed becomes important. There are two key points in accessing the memory in a coalesced manner. First, the GPU is capable of reading 4, 8, and 16-byte words from global memory into registers in a single instruction. Second, global memory bandwidth is used most efficiently when the memory accesses of the threads in a half-warp can be coalesced into a single memory transaction of 32, 64, or 128 bytes. Thus, if threads of a half-warp access 4, 8, or 16-byte words, where all 16 words lie in the same segment of the memory, and if the threads access the words in sequence, coalescence will be achieved regardless of the generation of the GPU that is used.

Bank conflicts: To achieve high memory bandwidth, shared memory is arranged in 16 equally sized modules called *banks*, which can be accessed simultaneously. So, memory requests made by half-warps to *n* different addresses that reside in *n* different banks suffer no conflicts. However, if more than one read or write requests are made for a single address, the accesses need to be serialized, and this situation is referred to as a bank conflict. To overcome this problem, padding the shared memory is a common method in practice. However, managing bank conflicts is an optimization step that is of secondary importance, since shared memory is already a very low latency structure.

Branch divergence: At each instruction issue time, the instruction unit on a multiprocessor selects a warp that is ready to execute and issues the next instruction to this warp. Since a warp executes one common instruction at a time, full efficiency is reached when all threads agree on their paths. But if some of the threads diverge due to a data dependent conditional statement, this branch is serially executed. This increases the warp execution time and causes inefficiencies.

Memory transfer: The data transfer bandwidth is significantly higher for device to device transactions compared to host to device ones. Therefore, it is advisable to reduce the data transfer between the host and the device. There are several ways to increase the CPU to GPU transfer bandwidth, which include using page locked memory or sending large chunks of data rather than using many smaller ones.

4.2 HoG Algorithm on the GPU

Here, we briefly consider three different implementations of the Dalal-Triggs algorithm on the GPU.

Zhang and Nevatia's implementation [9]: This method is based on realizing the CPU-HC algorithm (Ch. 3.4.1) on a graphics processor. It consists of four modules; scaling, feature extraction, classification and reduction. The scaling module is used for downsampling the input image, which resides on a texture in the GPU. The feature extraction step calculates the cell histograms over the whole image, then shares them among windows to build the window descriptors. In the classification part, inner products of the SVM weights and descriptors are computed along with normalization. The reduction step transfers classification results to the CPU. In terms of detection accuracy (miss rate versus false positive rate), they report similar results as the Dalal-Triggs method. Regarding detection performance, they note that an input of size 384×288 takes 73 ms to evaluate, where the downsampling ration is taken as 1.05. This about an order of magnitude speed up relative to the CPU-HC implementation.

Wojek *et al.*'s implementation [3]: This approach does not omit any of the steps proposed by Dalal & Triggs. After padding the sides of the input to be able to detect objects near the borders, they decompose it into 32-bit floating point color channels and apply gamma compression. Via convolution, color gradients are computed and gradient magnitude and orientations are obtained. For block histogram computation, they use both trilinear interpolation and Gaussian weighting. They let one thread block be responsible for a HoG block. In this setting, each thread works on one column of gradient orientation and magnitudes, and since there are 4 cells within each block, this calls for a thread block of size 16×4. Each of the block normalization and SVM evaluation steps are done by a separate kernel. Non-maxima suppression is done on the CPU side. In terms of accuracy, their ROC curve is very similar to the original HoG detection results. They report a speed up by a factor of 34 over the CPU implementation. For a 1280×960 image with 1.05 downscaling ratio, processing time is 385 ms.

Priscariu and Reid's implementation [36]: Similar to Wojek *et al.*'s work, this approach includes all the steps in the original Dalal & Triggs algorithm. One difference between the two GPU implementations is the thread block configuration in histogram computation. Priscariu and Reid divide each thread block into 4, so that each logical division of threads is responsible for one histogram cell. They employ 8 threads per cell, thus have a 8×4 block configuration. They note that their implementation gives the same results as the original Dalal-Triggs binaries. In terms of speed, they do not differ much from Wojek *et al.*'s implementation. For a 1280×960 image, reported processing time is 353 ms. The difference in detection time might also be stemming from using a different graphics card.

We finalize our discussion with a table that summarizes the detection performances of these implementations.

Implementation	**320×240**	**640×480**	**1280×960**
Zhang and Nevatia [9]	≈ 70 ms	≈ 250 – 300 ms	≈ 1 – 1.2 s
Wojek *et al.* [3]	29 ms	80 ms	385 ms
Priscariu and Reid [36]	22 ms	99 ms	353 ms

Table 4-1: Detection times of three different HoG implementations. The downscaling ratio is 1.05 and window strides are taken to be 8 pixels. The results of [9] are based on our extrapolations, as they report only for images of size 384x288.

Chapter 5

Cascaded Ensembles with Histograms of Oriented Gradients Features

We have noted that the Dalal-Triggs algorithm suffers from being computationally expansive, making it unfeasible to run in real-time. Chapter 4 discussed one way of alleviating this drawback. Here, we discuss another approach to this problem, originally proposed by Zhu *et al.* in [2].

5.1 The Cascade-of-Rejectors Formulation

The HoG algorithm suggested by Dalal and Triggs computes locally normalized gradient orientation histograms over blocks of size 16×16 to represent a detection window. The fact that the detection window slides over the image in all possible image scales makes this approach computationally expensive, and unfeasible for real-time applications such as surveillance, active driving assistance and tracking.

One of the most important works in the object detection literature of the last decade is the cascade detector of Viola and Jones [4]. Apart from being reliable, it demonstrated more than an order of magnitude speed up relative to its competitors at the time it was proposed. Their work has three key contributions. The first one is the introduction of the data structure *integral image* which made the computation of features that involve summation of image regions extremely efficient. The second contribution is using AdaBoost [37] to build robust classifiers. The third key point is the method of combining classifiers trained by AdaBoost in a cascade which allows background regions of the image to be quickly discarded while spending more time on promising object-like regions.

We have noted that the Dalal-Triggs gives outstanding detection accuracy. The success of this algorithm depends on two key factors: a dense window descriptor based on small histogram blocks and local block normalization that emphasizes their relative behavior. Firstly, even though this dense formulization gives an excellent description power, it also results in redundant computations for image regions that clearly do not resemble a human. Using a coarser descriptor would prune these computations, and enable us to focus our resources on detection windows that are harder to classify. Secondly, these small histogram blocks of size 16×16 might miss the "big picture", being unable to correspond to a semantic part of the human body. These mappings might be recovered if blocks of larger sizes and different aspect ratios could be employed. To address these points, Zhu *et al.* [2] proposed to form an attentional cascade consisting of stages that get progressively more complex (Figure 5-1). This suggested method brings together the key points of the Viola-Jones' work that provides efficiency and Dalal-Triggs' HoG features that give reliability.

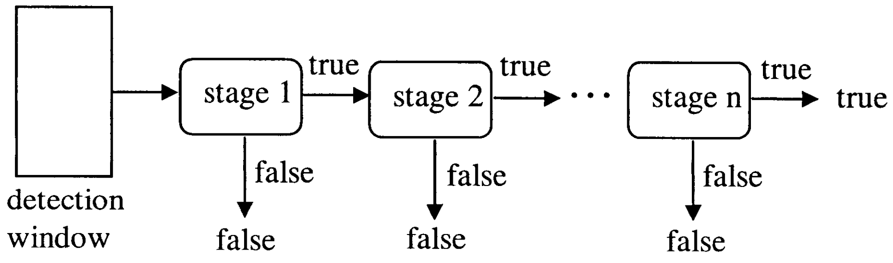


Figure 5-1: Cascade-of-rejectors. Detection window is passed to the stage 1 which decides true or false. A false determination stops further computation, and the window is classified to contain non-person. A true determination triggers the computation at the following stage. Only if the window passes through all the stages, it is classified to contain a person.

The cascade approach is based on early rejection of detection windows which clearly do not contain a person by evaluating a small number of features, and focusing the computational resources on windows that are harder to classify. We would like to find out which combinations of blocks can be used together to capture the "big picture" and make fast rejections in the early stages of the cascade, as well as the combinations of blocks that can provide detail about a detection window in the later stages. A systematic way of finding complementary features out of a feature pool is using the AdaBoost algorithm [37].

By using HoG blocks of different sizes, locations and aspect ratios as features, it is possible to run the AdaBoost algorithm to determine which features to evaluate in each stage of the cascade. Thus, the complete detector is formed by a cascade of ensemble classifiers, each of which uses base learners that are linear SVMs based on the chosen HoG block features. In Zhu *et al.*'s approach [2], the pool of all possible features contains over 5000 HoG blocks, as opposed to the 105 fixed size blocks that define a window in the Dalal-Triggs algorithm. As expected, they report that the first stages of the cascade employ large blocks that attempt to summarize the windows effortlessly, while the later stages also include smaller blocks that express the details.

By this cascade formulation, the method in [2] was reported to yield 4 to 30 FPS performance on a 320×240 image, depending on the scanning density. Hence, it achieves the real-time performance we are looking for, albeit with small sized inputs. Although not reported in their work, it is conceivable that this would correspond to about 0.3 to 2 FPS for a 1280×960 image, not being able to run in real-time for such high resolution images.

Zhu *et al.* also report ROC curves that are comparable with the original Dalal-Triggs algorithm, which makes their work even more significant. Using Haar-like wavelet features with the Viola-Jones style detector has been seen to give successful results rates for face detection [4]. However, this set of features was reported not to be able to perform as good as the HoG algorithm when applied to human detection, especially in a cluttered and complex dataset [5, 2]. But Viola and Jones demonstrate that it is possible to improve detection accuracy by employing motion information as well as wavelets in [8].

As we have noted that the Viola-Jones detector has influenced the field thanks to its three key contributions. The method proposed by Zhu *et al.* also uses one of these, the integral image idea, to compute the orientation histograms efficiently.

5.2 Integral Histograms of Oriented Gradients

By setting the horizontal and vertical window strides to 8 pixels, which is equal to the strides of histogram blocks within the detection windows, the Dalal-Triggs algorithm eliminates the redundant computations. Computing and caching the block histograms over the whole image and sharing them among the detection windows makes it possible to work around the problem of recomputing data for overlapping windows. However, in the case of the cascade algorithm, it is not possible to cache the histograms and share them among the windows since the relative locations of blocks have no order and a "block stride" cannot be defined. This leads us to using the integral image idea.

The integral image, as detailed in Figure 5-2, enables us to compute the sum of the elements within a rectangular region by using 4 image access operations. In our CPU implementation of the cascaded HoG detector, we discretize each pixel's gradient orientation into 9 bins, then compute and store an integral image for each histogram bin, as suggested in [2]. The HoG for any rectangular region then can be computed by $9 \times 4 = 36$ image access operations, 4 for each of the 9 bins.

This histogram computation method differs from the Dalal-Triggs algorithm because of the omissions of the Gaussian mask used for weighting the votes of histogram blocks, and the trilinear interpolation (in space and orientation) used for histogramming.

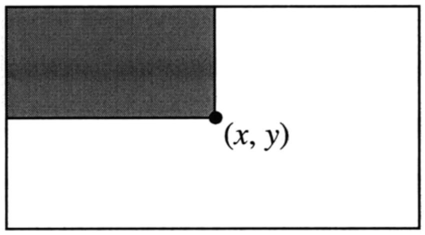


Figure 5-2: The value of the integral image I_{int} at point (x, y) is the sum of all the pixels above and to the left in the input image I;

$$I_{\text{int}}(x, y) = \sum_{i \le x, j \le y} I(i, j)$$

5.3 Histogram Cells and Blocks

In our implementation of the cascaded HoG detector, we consider block sizes ranging from 12×12 to 64×128, with the constraint that the block width and height must be divisible by two. Also, we consider block aspect ratios of (1 : 1), (1 : 2) and (2 : 1). Depending on the block size, we choose a step size which can take values {4, 6, 8}. This way, we can define a feature pool of 5029 distinct blocks inside a detection window of size 64×128. Each of these blocks is further divided into a grid of 2x2 histogram *cells*, over which the orientation histograms are computed. Each cell gives rise to a 9 dimensional histogram vector, and these are concatenated to form a 36 dimensional block histogram. Computing a single block histogram thus requires 9 × 9 = 81 integral histogram accesses (Figure 5-3).

Choosing the histogram blocks out of a large pool of features gives us the power to represent semantic parts in the human body in an explicit way (some parts may correspond to torso, legs, etc.). By placing these features in a cascade that progressively gets more complex, we also avoid making unnecessary computations for objects that do not resemble a person, since the blocks in the early stages are usually large, and they capture the "big picture" effortlessly. In the original Dalal-Triggs algorithm, we make the same amount of computation for each window, regardless of the complexity of the classification task we are trying to solve.

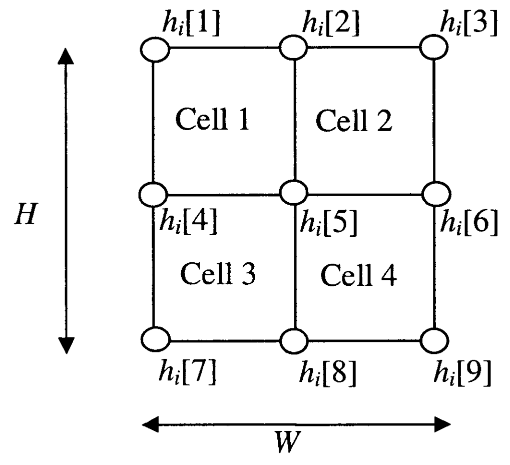


Figure 5-3: Computing the block histogram for a block with size $W{\times}H$. Using the i^{th} integral histogram, it is possible to compute the i^{th} elements of the cell histograms using 4 image accesses. For instance, i^{th} element of cell 1's orientation histogram is computed as $h_i[5] + h_i[1] - h_i[2] - h_i[4]$. Overall, we need 9 accesses for the i^{th} bin, and 81 for all 9 bins.

5.4 Training the Cascade with AdaBoost

Paralleling [2], we use 36 dimensional histogram blocks as base learners in constructing the cascade classifier. These learners are linear SVMs trained on the positive and negative training examples. Each stage of the cascade is a strong classifier formulated as an ensemble of these base learners,

$$s_i(\underline{x}) = \begin{cases} 1, & \text{if } \sum_{t=1}^{n_i} \alpha_{it} f_{it}(\underline{x}) \geq T_i \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

where s_i (.) is the strong classifier (ensemble) at stage i, f_{it} (.) is the t^{th} base learner of this stage with voting weight α_{it}, n_i is the number of the base learners in this stage, and T_i is the detection threshold. Each base learner has the form

$$f(\underline{x}) = \begin{cases} 1, & \text{if } \underline{x}^T \underline{\theta} - \theta_0 \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

and the parameters $\underline{\theta}$ and θ_0 are learned with our modified version of the SVM package SVMLight [10].

Since there are more than 5000 possible features in our pool to choose from, we randomly sample 125 blocks at each round of the AdaBoost algorithm and train linear SVMs. As noted in [2], choosing the best feature from about 59 random samples will guarantee nearly as good performance as if we used all the features. By settling for 125, we substantially decrease the training time meanwhile keeping feature quality reasonably high.

For all stages, we use the 2416 positive images of size 64×128 from the INRIA database [11]. For the first stage, we randomly sample 2416 negative windows of size 64×128 from the 1654 full-size ($\geq$ 320×240) negative images from INRIA. Then, each next stage in the cascade uses the false positives obtained by running the current cascade classifier over these full-size negative images as the negative training set. We randomly subsample these false positives since they exceed 2416. Because each new stage is forced to classify examples that the current cascade fails to classify, stages tend to contain progressively more features, hence they become more complex.

Below, we detail the cascade training steps with the AdaBoost algorithm.

Algorithm: Training the cascade with AdaBoost

User selects values for f_{max}, the maximum acceptable false positive rate per stage, d_{min}, the minimum acceptable detection rate per stage and F_{target}, target overall false positive rate.

Pos: set of positive samples (INRIA training positives)

Neg: set of negative samples (sampled from INRIA training full-size negatives)

```
initialization: $i = 0, D_i = 1.0, F_i = 1.0$
while $F_i > F_{target}$
        $i = i + 1, f_i = 1.0$
        while $f_i > f_{max}$
        • Train 125 randomly sampled linear SVMs using Pos and Neg
        • Add the best SVM into the ensemble with the appropriate vote determined by
          AdaBoost
        • Update weights of the examples in AdaBoost manner
        • Evaluate Pos and Neg with the current ensemble
        • Decrease the threshold $T_i$ until $d_{min}$ holds
        • Compute $f_i$ under this threshold
        $F_{i+1} = F_i \times f_i$
        $D_{i+1} = D_i \times d_{min}$
        Empty set Neg
        if $F_i > F_{target}$
        • Evaluate the current cascaded detector on the set of full-size negatives and add any
          false positives into Neg, subsample if necessary.
```

At each stage of the cascade, we keep adding base learners until the predefined quality requirements are met. In our case, we require the minimum detection rate of each stage to be 99%, and the maximum false positive rate to be 0.65. We trained 23 stages to reach about $0.65^{23} \approx 5 \cdot 10^{-5}$ FPPW on the training set, which corresponds to about 8 false positives in a 1280×960 image with dense scanning. The training took several days running on a PC with 2.5GHz CPU and 3GB memory.

5.5 CPU Implementation Considerations

For the training part of the algorithm, we integrated the SVM training package SVMLight [10] into our code, and modified it so that it can admit binary inputs, rather than reading from text files. Since no ready-for-use software function is available for AdaBoost training with SVM features, we provided the code for the algorithm.

The implementation for the cascaded detector consists of several parts (Figure 5-4). After acquiring the image and converting it to grayscale, gradient magnitude and orientations are computed for each pixel. Because the arctangent function is costly to evaluate, we use a look-up table to efficiently calculate the orientation bins. Next, we form the integral histogram images for each of the 9 bins, and generate the 36-D block histograms by accessing the histogram images according to Figure 5-3. After L2-norm normalization, we take the inner product of the block histogram with a linear SVM describing the current base learner. We evaluate all the features until rejection (negative window), or completion (positive window). After downsampling the image, we repeat this process until all scales are accounted for.

Scanning the classifier across all positions and scales in the image returns multiple detections for the same object at similar scales and positions. Hence, neighboring detections need to be fused together (non-maximum suppression). As in Section 3.3, we follow [13] and achieve this using a mean shift algorithm in 3D position/scale space. Since this time the detections are binary without the discriminant information, we take the detection weights as $t(w) = 1$, as opposed to the formulation in Eq. 3.3. For a 1280×960 image with 8 pixel horizontal and vertical window strides and a scale ratio of 1.05, the whole algorithm takes 5.4 seconds on the CPU.

Our implementation utilizes OpenCV libraries [12] for image acquisition, gradient computation and forming the integral histogram images.

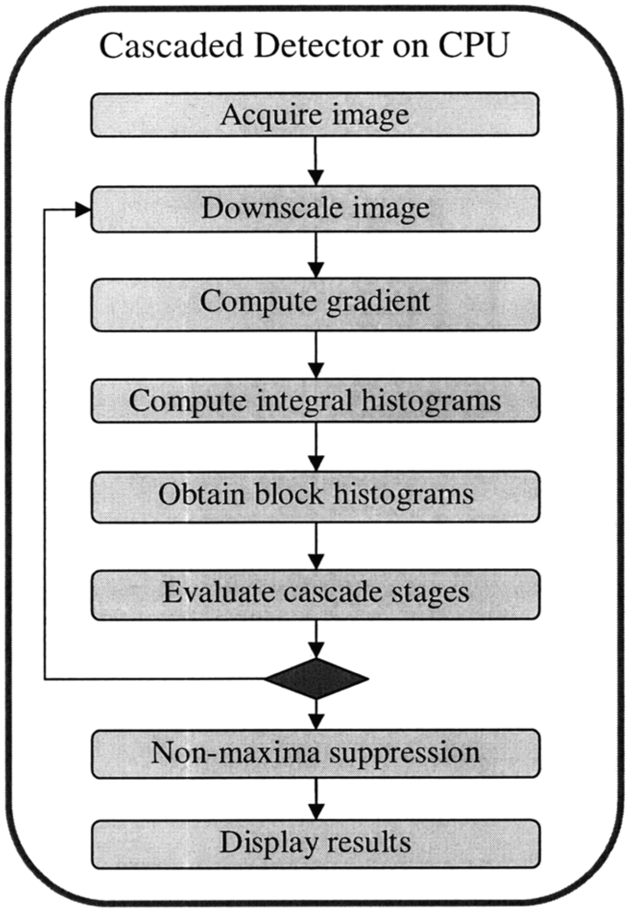


Figure 5-4: Steps of object localization using the cascaded HoG detector on the CPU.

5.6 Experiments and Evaluation

The cascaded classifier in our experiments consists of 23 stages and it reaches about $5 \cdot 10^{-5}$ FPPW false positive, and $0.99^{23} \approx 0.8$ detection rates on the training set. However, due to the fact that we generate many hypotheses for each object by searching densely in space and scale, the detection rate is about 4% higher in the test set. Figure 5-5 provides details about our cascade. To assess the depicted rejection rates at given stage numbers, we scanned a test set of negative images that contains over 1 million detection windows with the cascaded classifier. We note that the method achieves to reject more than 90% of the detection windows at the end of 4 stages, which contain only 16 features in total. More complex stages are needed for only the hardest windows, and this early rejection strategy is what gives the method a significant speed up over the Dalal-Triggs algorithm. Since each stage of the detector is trained on the false positives of the current cascaded detector, it takes more involved ensembles with larger number of base learners to attain the same false positive rates as the number of stages increases (Figure 5-5a).

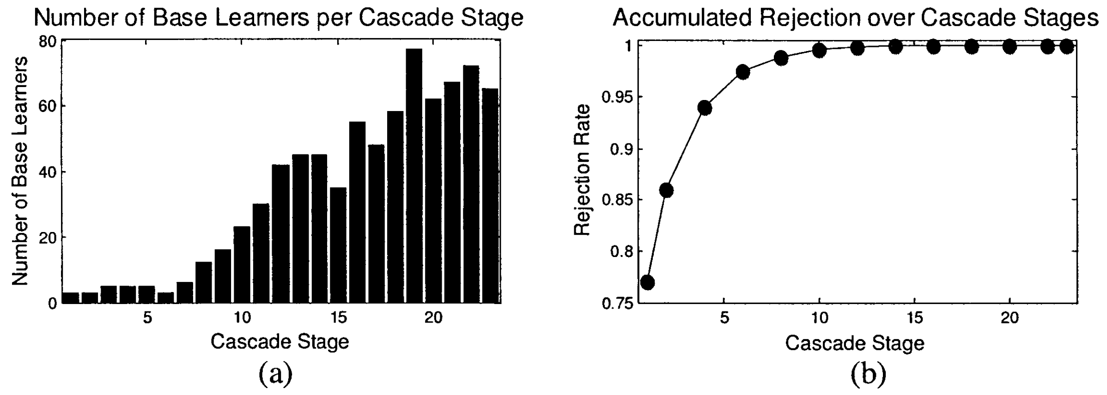


Figure 5-5: Cascaded classifier that uses variable-size HoG blocks as features in detail. The cascade consists of 23 stages where the base learners are linear SVMs with 36-D features of block histograms, chosen out of a feature pool of 5029 blocks with the AdaBoost algorithm. (a) The number of base learners at each stage. (b) The rejection rate as a cumulative sum over the cascade stages. We note that 4 stages are enough to reject more than 90% of the detection windows, providing significant speed up to the algorithm.

In Table 5-1 we compare three techniques running on the CPU: Dalal and Triggs, the cascaded detector implementation of Zhu *et al.* in [2] with L2-normalization, and our approach. We note that the detector of Zhu *et al.* has 30 stages and attains about 10^{-5} FPPW. However, our detector has 23 stages, and we would expect it run slower if we had trained an equal number of stages as [2].

On the average, 6.7 block evaluations are needed to classify a detection window in our method. Compared to the 105 block evaluations made in the Dalal-Triggs approach, we require 15.7 times less block evaluations, as evidenced by the 14.7 speed up our implementation achieves. Nevertheless, Zhu *et al.*'s method is about two times faster than ours, evaluating 4.6 blocks on the average. We present the miss-rate/FPPW curves of our cascade, Zhu *et al.*'s and Dalal-Triggs' approaches in Figure 5-6, and note that our results are comparable with the other two, especially when FPPW goes up. We think that the difference between the two cascades arises from using different sets of parameters f_{max}, d_{min} in training.

CPU detectors	Sparse scan (800 windows / image)	Dense scan (12800 windows / image)
Dalal & Triggs [1]	500 ms	7 sec
Zhu *et al.*[2]	30 ms	250 ms
Our cascade	82 ms	475 ms

Table 5-1: Time required to evaluate a 240×320 image with the three different methods. Sparse scan corresponds to using 8x8 spatial stride and 1.2 downsampling ratio. Dense scan generates more hypotheses by using 4×4 spatial stride with 1.05 scaling ratio.

To inspect the most informative blocks selected by the AdaBoost algorithm, we visualize the blocks in cascade stages 1, 3 and 5 in Figure 5-7. These blocks are the ones with the lowest weighted error at that round out of 125 randomly sampled blocks from the feature pool. Since the pool contains more than 5000 blocks, the sample size is about 2.5% of the total, hence the selected blocks may not be the best ones globally. We observe that the depicted blocks are located in certain positions such as torso, legs, and head, so the AdaBoost algorithm manages to

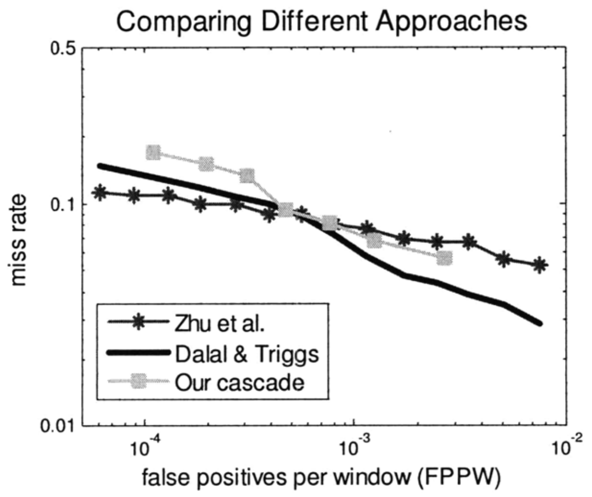


Figure 5-6: Comparing Zhu *et al.*, Dalal & Triggs and our cascade. Our implementation is comparable with the other two, especially when FPPW goes up.

select the histogram blocks that have semantic meanings in the human body. We also observed that the features in the early stages generally have sizes much larger than the 16×16 blocks used in the Dalal-Triggs approach. This fact gives us the power to rapidly summarize the contents within windows and reject them if they do not contain a person.

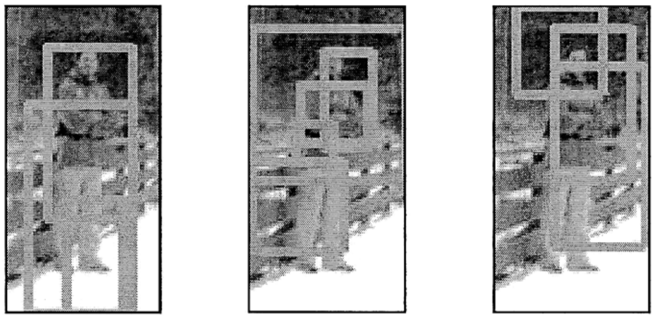


Figure 5-7: Visualizing the selected blocks by the AdaBoost algorithm. (a) Blocks in the first stage, (b) blocks in stage 3, and (c) blocks in stage 5 of the cascade. The blocks in (a)-(c) are the blocks with lowest weighted error out of 125 features sampled randomly from a feature pool of 5029 blocks.

We finish with some examples of detections obtained on the INRIA test set with our cascaded HoG detector.

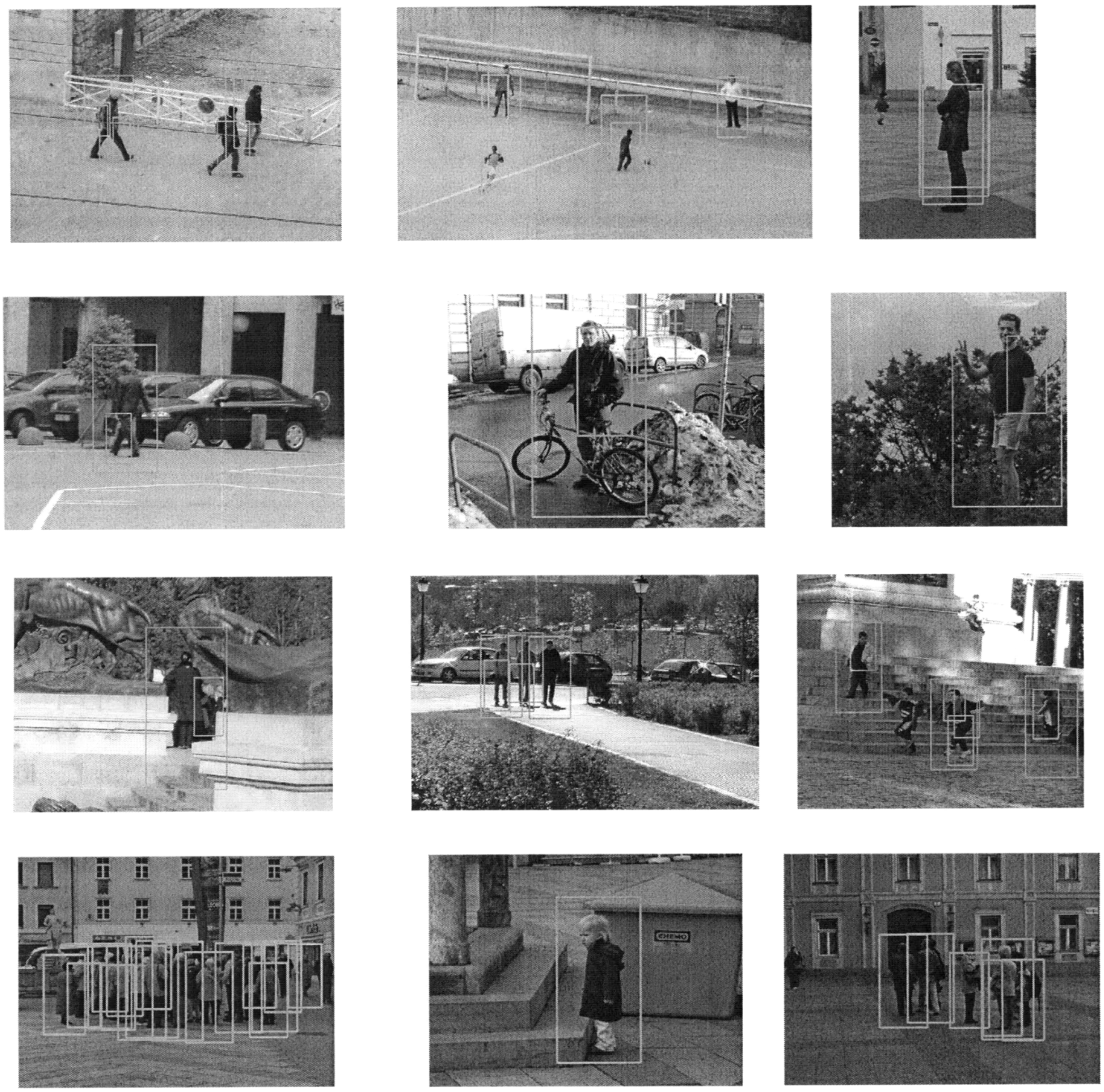


Figure 5-8: Detection results obtained with the 23 stage cascade detector that reaches about 10^{-4} FPPW on the test set and $5 \cdot 10^{-5}$ FPPW on the training set. The window strides are 8×8 and the downsampling ratio is 1.2.

Chapter 6

Efficient Integral Image Computation on the GPU

The integral image is a data structure that has been significantly useful in object detectors that employ features depending on sums of rectangular pixel regions. As we have discussed in Chapter 5, it also plays a central role in the cascade detector with HoG features, this time in the context of integral histograms. Here, we investigate an efficient parallel implementation of this important image representation, so that this implementation can be employed as a subroutine to speed up computer vision algorithms that rely on features with pixel sums.

6.1 Background on the Integral Image

The use of integral images for rapid feature evaluation became popular with the seminal face detection algorithm proposed by Viola and Jones [4]. The features employed in the detector are reminiscent of Haar basis functions and form an overcomplete set for image representation. Obtaining the proposed features involves computing sums of pixel values over rectangular regions. Since these sums can be calculated by using only 4 array references with the integral image, evaluating this set of Haar-like features is very cheap, once the integral image is computed.

An alternative motivation for the integral image arises from the signal processing literature. In the "boxlets" work of Simard *et al.* [38], authors point out that in the case of linear operators (e.g. the inner product $f \cdot h$), any invertible linear operation can be applied to either f or h if the inverse operation is applied to the other operand. From this point of view, the integral image can be expressed as a dot product, $i \cdot r$, where i is the input image and r is the box car function that takes the value 1 inside the rectangle of interest and 0 outside. This summation can be written as

$$i \cdot r = \left(\iint i\right) \cdot r''$$

where the double integral of the image, obtained by summation first along the rows and then along the columns, is in fact the integral image and the second derivative of the boxcar function gives rise to four delta functions at the corners of the image. This is exactly the same idea as using 4 array references to compute the integral image.

This integral image formulation has allowed the Viola-Jones face detector to run in real-time, and influenced the development of several other computer vision algorithms. Among these, [2, 39] apply the integral image to histograms, thus extending its usage from Haar-like wavelets to more complex features such as the Histograms of Oriented Gradients [1] descriptors.

Even though the systems that incorporate the integral image approach as an intermediate component have been reported [2, 8, 40] to have training times in the order of days, they experience significant performance benefits. It is possible to build on this boost in speed by realizing such methods on general purpose GPUs, and obtain real-time performances [40].

A sequential implementation for integral image computation would require $2 \cdot w \cdot h$ operations for an image of size $w \times h$. As the size gets larger, this cost represents a significant overhead for the overall algorithm. Messom and Barczak [41] adopt a parallel processing approach to reduce this overhead. Their realization is based on the Brook stream processing language and demonstrates that employing the GPGPU paradigm results in significant performance benefits.

6.2 The Sequential Implementation

For an image of size $w \times h$, we form the integral image on the CPU using the algorithm given below.

```
**Algorithm:** Sequential integral image formulation
$I$ : input image with size $w \times h$
$I_{int}$ : integral image with size $w \times h$
Array elements are accessed in row major order.

for $x = 0$ to $w-1$ do
    $I_{int}[x] \leftarrow 0$
for $y = 1$ to $h-1$ do
    $I_{int}[y \cdot w] \leftarrow 0$
    $s \leftarrow 0$
for $x = 0$ to $w-1$ do
    $s \leftarrow s + I[x + (y-1) \cdot w]$
    $I_{int}[x + y \cdot w + 1] \leftarrow s + I_{int}[x + (y-1) \cdot w + 1]$
```

We note that the output of this algorithm is an exclusive integral image, which is padded on the first row and the column by zeros and has the same size as the input. For instance, the image

$$I = \begin{bmatrix} 2 & 1 & 3 & 1 \\ 3 & 2 & 1 & 1 \\ 4 & 1 & 3 & 1 \end{bmatrix} \text{ produces } I_{\text{int}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 6 \\ 0 & 5 & 8 & 12 \end{bmatrix}$$

6.3 Parallel Algorithms for Integral Image Computation

We start by explaining the parallel prefix sum (scan) algorithm [42] which constitutes the foundation of our method. Next, we relate how this algorithm can be used as a building block by applying it first on the rows of the image, then taking the transpose, and again applying parallel scan on the rows of the transposed array to obtain the integral image.

6.3.1 Parallel Fix Sum (Scan)

The *all-prefix-sums* operation takes a binary associative operator $\oplus$, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

If we let the operator $\oplus$ be summation, we obtain the *inclusive scan* operation. If we shift the resulting array to the right by one element and insert the identity in the beginning, we end up with the *exclusive scan* operation, which returns

$$[0, a_0, (a_0 + a_1), \ldots, (a_0 + a_1 + \ldots + a_{n-2})]$$

In the rest of this work, we will be focusing on the exclusive version of the operation, and simply refer to it as *scan*.

For an input array with size n, the scan algorithm has computational complexity of $O(n)$, and it consists of two phases: the *reduce phase* (or the *up-sweep phase*) and the *down-sweep phase*. We can visualize the reduce phase as building a binary tree (Figure 6-1), at each level reducing the number of nodes by half, and making one addition per node. Since the operations are performed in place using shared memory, the tree we build is not an actual data structure, but helps explaining the algorithm.

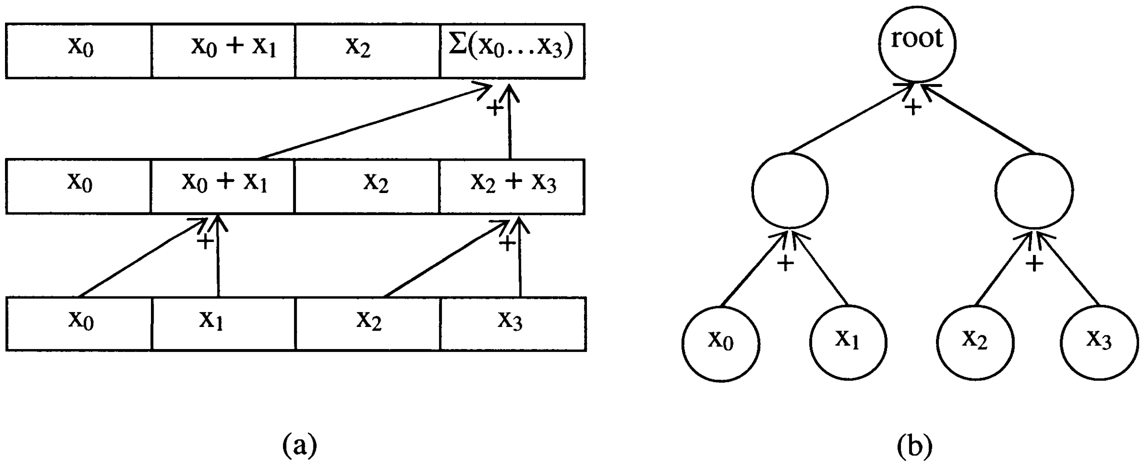


Figure 6-1: (a) The reduce phase applied on an array of four elements. (b) Binary tree view of the algorithm. Scanning is performed from the leaves to the root, where the root contains the sum of all four elements.

In the down-sweep phase, we traverse the tree from the root to the leaves, and use the partial sums we computed in the reduce phase to obtain the scanned array. We note that the last element is set to zero in the beginning and it propagates to reach the beginning of the array, thus resulting in an exclusive computation (Figure 6-2).

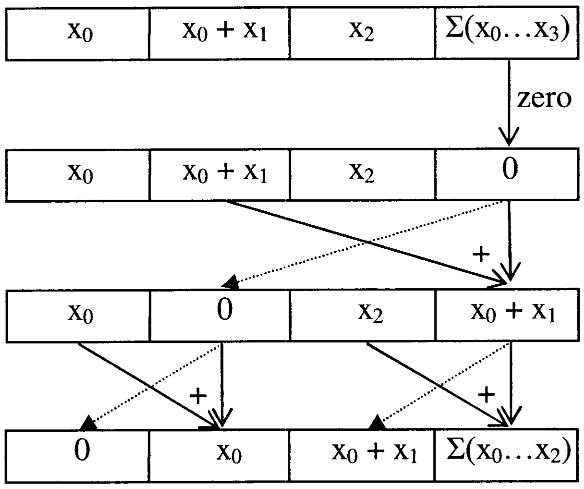


Figure 6-2: The down-sweep phase. At each level of the tree, there are as many swapping operations as summations.

The overall cost of these phases is $2(n-1)$ summations and $(n-1)$ swaps, which is in $O(n)$ time, same as the sequential algorithm. Following [42], we provide the CUDA kernel that implements the scan algorithm below.

Even though this kernel is work efficient, it suffers from bank conflicts in the shared memory. In our implementation, we try to avoid these conflicts by adding a variable amount of padding to each shared memory index we use, again as suggested in [42]. The amount we add is equal to the value of the index divided by the number of memory banks, which is equal to 16 for our graphics card.

As it is, this kernel is unable to scan arrays with sizes larger than 1024, since the maximum number of threads per block is 512 and a single thread loads and processes two data elements. Influenced by [10], we solve this problem by employing several thread blocks and making them responsible for a certain part of the input. If we let the input array contain n elements and if each block processes b of the entries, we need to launch n/b thread blocks and $b/2$ threads in each

CUDA Code: Scan kernel for the GPU

```
__global__ void scan(float *input, float *output, int n)
{
  extern __shared__ float temp[];
  int tdx = threadIdx.x; int offset = 1;

  temp[2*tdx]   = input[2*tdx];
  temp[2*tdx+1] = input[2*tdx+1];

  for(int d = n>>1; d > 0; d >>= 1)
  {
    __syncthreads();
   if(tdx < d)
    {
      int ai = offset*(2*tdx+1)-1;
      int bi = offset*(2*tdx+2)-1;
      temp[bi] += temp[ai];
    }
    offset *= 2;
  }

  if(tdx == 0) temp[n - 1] = 0;

  for(int d = 1; d < n; d *= 2)
  {
    offset >>= 1; __syncthreads();
    if(tdx < d)
    {
      int ai = offset*(2*tdx+1)-1;
      int bi = offset*(2*tdx+2)-1;

      float t  = temp[ai];
      temp[ai] = temp[bi];
      temp[bi] += t;
    }
  }
  __syncthreads();
  output[2*tdx]   = temp[2*tdx];
  output[2*tdx+1] = temp[2*tdx+1];
}
```

block. With the usual scan algorithm, each thread block scans its part of the array, but before zeroing the last element that contains the sum of all the elements in that segment, we register it to an auxiliary array I_{sum}. We then scan this array in place and add $I_{sum}[i]$ to all elements of the segment that $(i+1)^{st}$ thread block is responsible for. Figure 6-3 tries to further illustrate this. To handle inputs with a size that is not a power of two, we pad the last segment of the array before scanning.

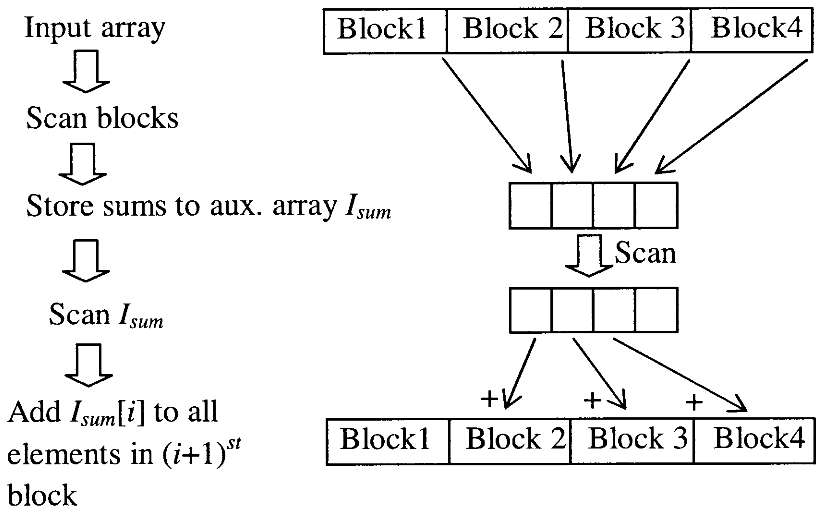


Figure 6-3: Scanning arrays of arbitrary size.

6.3.2 Scanning the Image Rows

We treat each row of the image as an independent array and scan the rows in parallel. In our implementation, each row is divided into segments of 512 pixels, and each segment is processed by a thread block consisting of 256 threads. Hence, we launch a scan kernel using a grid with dimensions $n_{seg} \times h$, where n_{seg} is the number of segments in each row, and h is the height of the image.

6.3.3 Computing the Transpose

After scanning the rows of the image, we take the transpose of the resultant array, so that we can use the same scanning kernel twice in order to compute the integral image. Taking the transpose is the cheapest routine in our method, because we utilize the shared memory to provide coalescence, and apply padding to the shared memory in order to avoid bank conflicts, as suggested in [43]. We present the transpose kernel next, where we take `BLOCK_DIM` as 16.

CUDA Code: Transpose kernel for the GPU

```
__global__ void transpose(float *input, float *output, int width, int height)
{

 __shared__ float temp[BLOCK_DIM][BLOCK_DIM+1];
 int xIndex = blockIdx.x*BLOCK_DIM + threadIdx.x;
 int yIndex = blockIdx.y*BLOCK_DIM + threadIdx.y;

 if((xIndex < width) && (yIndex < height))
 {
       int id_in = yIndex * width + xIndex;
   temp[threadIdx.y][threadIdx.x] = input[id_in];
 }

 __syncthreads();

 xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
 yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

 if((xIndex < height) && (yIndex < width))
 {
      int id_out = yIndex * height + xIndex;
      output[id_out] = temp[threadIdx.x][threadIdx.y];
 }

}
```

After transposing, we scan the rows of the transposed array to obtain the integral image. We launch a scan kernel with grid dimensions $\tilde{n}_{seg} \times w$, where $\tilde{n}_{seg}$ is the number of thread blocks, and w is the width of the image. We note that the resulting integral image is in transposed form, but this poses no difficulties since the pixel at position (x, y) can be accessed by the index $(y+x \cdot h)$.

6.4 Experiments and Evaluation

Here, we report integral image building times as a function of the input image size for single and double precision floating point arithmetic, and for vector type data. We compare our GPU processing times with our sequential implementation, as well as the GPU implementation in [41]. In all of our results, we exclude the time spent for data transfer and report only the GPU computation times, which are obtained on an NVIDIA GeForce GTX 295 graphics card. We use a PC with 2.5 GHz CPU and 3GB memory.

6.4.1 Single Precision Floating Point Computation

A multiprocessor consists of eight single precision thread processors, two special function units, on-chip shared memory, an instruction unit, and a single double precision unit. Therefore, GPUs are optimized for single precision computations, and there is an order of magnitude difference in the theoretical performance bandwidth between single and double precision operations. Figure 6-4 compares the results obtained with the sequential algorithm running on the CPU and the two single precision GPU implementations. For a 4 megapixel input, our system works about 3 times faster than the proposed method in [41], which is implemented with the Brook language and runs on a ATI graphics card.

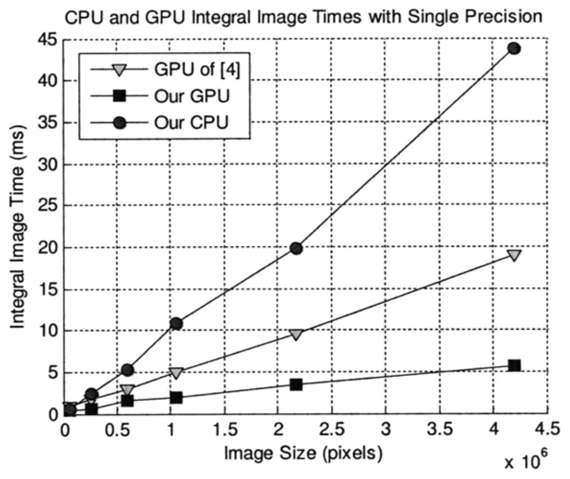


Figure 6-4: Performance comparison of single precision integral image computation on the CPU and the GPU. Results for [41] are replicated from their work.

6.4.2 Single Precision Vector Processing

In addition to standard data types, CUDA also provides packed data structures to ease access to multi-dimensional inputs. The vector type formed by a bundle of four floating point numbers is called `float4`. Since the size of this structure is 16 bytes, it satisfies two important properties that increase the maximum memory bandwidth. First, the GPU is capable of reading 16-byte words from global memory into registers in a single instruction. Second, global memory bandwidth is used most efficiently when the memory accesses of the threads in a half-warp can be coalesced into a single memory transaction of 32, 64, or 128 bytes. In the case of `float4` data, this results in only two 128 byte transactions per half-warp, given that the threads access the words in sequence. Therefore, it is possible to process four times more data with a smaller impact on the memory bandwidth. This point is illustrated in Figure 6-5.

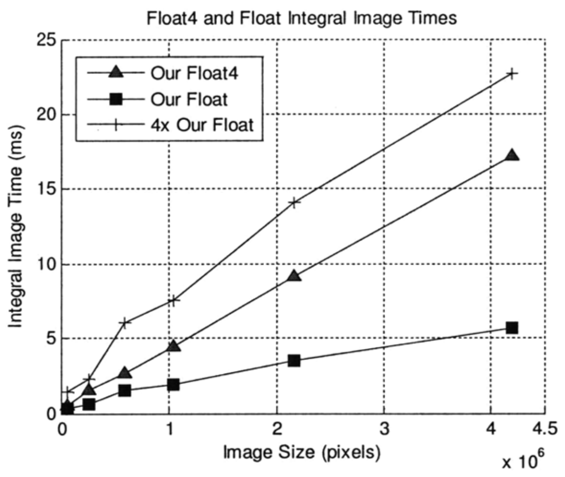


Figure 6-5: Comparing the GPU processing times of our `float4` implementation with four times the processing time of our `float` integral image. We are able to process four times more data using `float4` vector type, with a smaller impact on the memory bandwidth.

6.4.3 Double Precision Floating Point Computation

As GPUs are optimized for single precision arithmetic, double precision implementation results in a lower performance as depicted in Figure 6-6. For large image sizes, this performance degradation may be traded-off for higher accuracy computation. We note that our results are about 4 times faster than the implementation by [41] for a 2048×2048 size image.

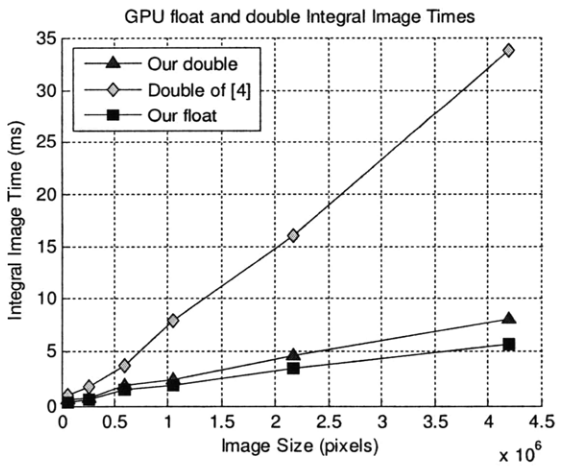


Figure 6-6: Performance comparison of double and single precision GPU implementations. Results for [41] are replicated from their work.

We finalize our discussion by noting that even though using double precision arithmetic reduces the GPU performance, it is still 9 times faster than the double precision CPU implementation, for a 4 megapixel input (Figure 6-7). As the input size gets smaller, we see that the performance difference is reduced. This is mainly because the CPU implementation makes use of its large cache and it is not possible to utilize all GPU processors at small image sizes.

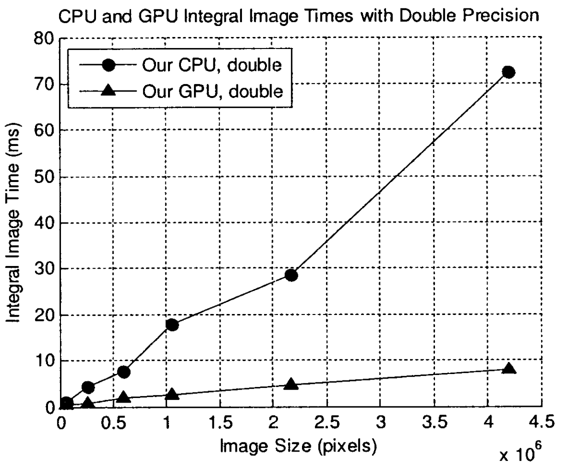


Figure 6-7: Integral image computation times with double precision on the CPU and the GPU. We report a speed up by a factor of 9 with the GPU implementation.

6.4.4 Kernel Occupancy and Performance

Maximum *occupancy* per kernel is a way of measuring CUDA code performance by quantifying how efficiently a multiprocessor is being used. Occupancy is defined as the ratio of the active warps to the maximum number of warps supported on a multiprocessor and determined by the shared memory and register usage and the thread block configuration of a kernel.

Table 6-1 presents occupancies as well as the processing times related with each kernel in our method. We note that all the kernels involved in integral image computation works with full occupancy.

Kernel	Occupancy	Mem. Throughput	Shared Mem.	Registers	Threads/Blk
Scan array	100 %	17 GB/s	2224	11	256
Increment block	100 %	50 GB/s	48	5	256
Transpose	100 %	49 GB/s	1120	8	16x16

Table 6-1: Shared memory and register usage, as well as the thread block configuration affects the kernel occupancies. The kernels in our implementation work at full occupancy, which is an indicator of good performance. The overall memory throughput reflects how fast the kernels access data from the global memory.

Chapter 7
Fast Human Detection with Cascaded Ensembles on the GPU

We investigate the feasibility of realizing the cascaded HoG detector introduced in Chapter 5 on a data parallel basis. Since the integral image idea is a key point in accelerating this detector, we make use of our parallel algorithms detailed in Chapter 6 for integral histogram computation. We report a speed up by a factor of 13 relative to our CPU implementation by carrying the detector over to our graphics processor.

7.1 GPU Implementation of the Cascaded Detector

Figure 7-1 shows the steps of our implementation, which starts with transferring the image from the CPU to the GPU's global memory. At each scale integral histograms for the discretized gradient orientations are computed, and these are then evaluated by the ensemble classifiers for object localization. When all scales are accounted for, the part of the GPU's global memory that contains the detection results are copied to CPU's main memory. Visualization of the detected objects is presented in the form of bounding boxes, and (optionally) mode estimation can be applied to fuse the neighboring positives. In what follows, we detail the steps of our detector.

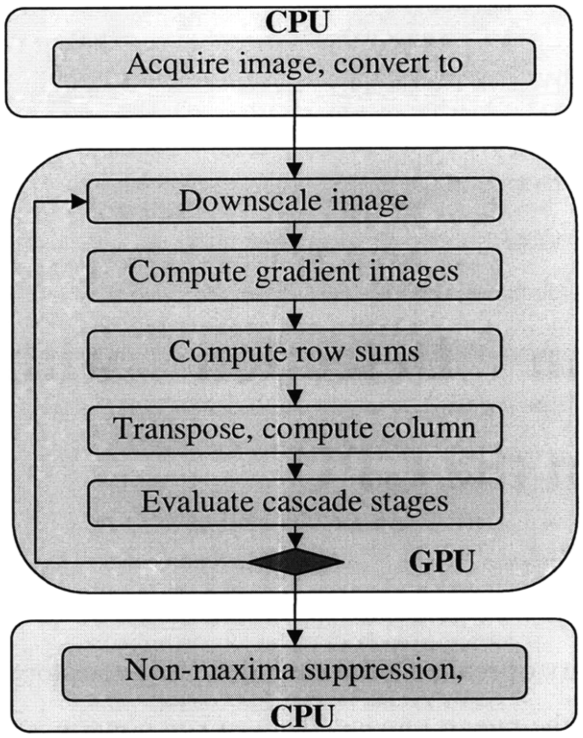


Figure 7-1: Steps of localization on the GPU

7.1.1 Image Acquisition and Preprocessing

The input is loaded to CPU memory and converted to grayscale with OpenCV routines. Next, it is copied to a CUDA array residing in the GPU's global memory and bound a 2 dimensional texture. By setting the `ReadMode` attribute of the texture appropriately, it is possible to get 32 bit floating point image values scaled to [0, 1] from the integer valued image pixels directly.

7.1.2 Downscaling and Gradient Computation

We evaluate these steps inside a single kernel. Each thread in this kernel corresponds to a single pixel, and they are grouped in 8×8 thread blocks for optimum efficiency. For downscaling, we take advantage of the texturing unit to efficiently subsample the target image by bilinear interpolation using the `tex2D` function. At each pixel, horizontal and vertical gradients are computed using centered convolution kernels [-1, 0, 1]; which we implement by simply taking the difference of the neighboring pixels around the pixel of interest. In this step, we also compute the gradient magnitude and the histogram bin that it corresponds to. To register the magnitudes, we use two `float4` arrays $I_{1\text{-}4}$ and $I_{5\text{-}8}$, and one `float` array I_9. Hence, at a given pixel, we store its magnitude in the appropriate field of $I_{1\text{-}4}$ if its orientation is between 0° and 80°, in $I_{5\text{-}8}$ if the orientation is between 80° and 160°, and in I_9 if it is larger than 160°.

7.1.3 Computing the Integral Histograms

This step is largely based on our discussion in Chapter 6. To compute the integral histograms for bins 1-4 and 5-8, we simply compute the integral images for the `float4` vector type using the inputs $I_{1\text{-}4}$ and $I_{5\text{-}8}$. For the ninth bin, we compute the single precision floating point integral image for the input I_9.

7.1.4 Evaluating the Cascade Stages

This stage contains the random memory access operations that do not fit well in the CUDA memory model. In order to evaluate a single HoG feature, we need to access 9 different positions within 9 integral histogram images $I_{1\text{-}4}$, $I_{5\text{-}8}$, and I_9 (Figure 5-3). Since the relative positions of the accessed points are determined by boosting, they are not continuous; hence memory coalescence becomes a problem while reading data from the global memory. There are two possible ways to overcome this problem, we can either employ shared memory or use textures. Let us explain why either method is not viable in our case.

Using shared memory for feature evaluation: We let each thread block be responsible for a detection window. Shared memory allowance of each thread block is 16kB, which corresponds to 4096 floating point numbers. Since our detection windows have size 64×128, we need $2^{13} \cdot 9 \cdot 4 = 288$kB of space to hold each window in the shared memory for efficient random access. This is clearly not possible.

Using texture memory: Cache working set for texture memory is 6 to 8kB for each multiprocessor. Even if we assume that a multiprocessor executes one block at a time, the texture cache is far smaller than our needs. Also, each time we subsample the image we need to rebind the global memory that holds the integral histograms to the texture memory, but the programming model does not support writes to textures bound to CUDA arrays. Hence we directly access the global memory for feature evaluation.

We launch kernels sequentially for each ensemble. The number of features in the early stages is much lower than it is for the late stages. To make better use of the CUDA memory model, we exploit this property by evaluating early and late stages with different kernels:

Feature evaluation, early stages: Each thread block works on a single detection window, and consists of $3 \times 3 \times n_i$ threads, where n_i is the number of base learners in the stage. Thus, a *group* of 9 threads is responsible for a feature, and all features are processed in parallel within a window. Each thread in a group accesses a single address in the integral histograms, reading all 9 bin values and recording them to shared memory. When the 81 required elements for a base learner are written to shared memory, threads go on to form the 36 dimensional histogram descriptor. In order not to use any additional shared memory, we make the necessary computations to form the descriptors in place. We store the linear SVM classifiers in a 1 dimensional texture, and compute the dot product between the descriptor and the classifier to get the vote of each base learner. After all evaluations within the block are completed, we compare the sum of the votes against the stage threshold T_i, and reject the window if it falls below it. In this kernel formulation, a thread block requires $9 \times 9 \times 4 \times n_i$ bytes, which becomes larger than 8kB when $n_i > 25$. Hence, for stages with more than 25 learners, we utilize the following kernel:

Feature evaluation, late stages: When the number of base learners is so high that it is not possible to launch more than one thread block due to shared memory pressure, we resort to a different kernel formulation. Now we employ two dimensional blocks with size $3 \times n_i$, where a group of 3 threads are responsible for a single base learner. Each of these 3 threads operate on one of $I_{1\text{-}4}$, $I_{5\text{-}8}$, or I_9, compute all 4 cell histograms using the corresponding integral histogram image, and write it to the shared memory. We note that by sacrificing some parallelism, we are able to accommodate more than 50 base learners in parallel within a single thread block, before reaching a shared memory requirement of 8kB. This is because we consume only $36 \times 4 \times n_i$ bytes of shared memory per block now. Normalization and taking the dot product with the linear SVM features is again carried out in this kernel, and each detection window is evaluated to either rejection or completion. We note that by utilizing two different types of kernels for feature evaluation, we observed an improvement of 15ms for a 1280×960 image, with 1.05 subsampling ratio.

7.1.5 Mode Estimation and Displaying the Results

After processing all scales, we copy the array that contains the binary detection results belonging to each window to CPU's main memory. We can optionally apply non-maxima suppression before visualizing the detection results.

7.2 Experiments and Evaluation

The cascaded classifier used in our GPU experiments is the same detector with 23 stages and reaches about $5 \cdot 10^{-5}$ FPPW false positive, and $0.99^{23} \approx 0.8$ detection rates on the training set. Due to single precision computations of integral histograms, detection results show slight differences between the CPU and the GPU realizations. Since the average difference between integral histogram bins computed on the two architectures is less than 10^{-3} per pixel, we believe the parallel implementation represents the sequential one faithfully.

Table 7-1 presents a performance comparison between two GPU implementations, Wojek *et al.*'s realization of the Dalal-Triggs method [3] and our GPU approach for the cascaded detector. We note that our implementation is slower by 10% when the subsampling factor is 1.05, but has about the same speed when it is 1.2. This difference should be caused from our CPU dependent steps, whose number increase as the number of scales increases. We also observe a 13x speed up when our method runs on the GPU.

Detectors	Scaling: 1.05	Scaling: 1.1	Scaling: 1.2
Wojek *et al.*[3]	385 ms	216 ms	133 ms
Our GPU	422 ms	228 ms	131 ms
Our CPU	5470 ms	2963 ms	1710 ms

Table 7-1: Processing times for a 1280x960 image. Presented results are for three different downscaling factors using 8x8 spatial strides. We note that our results exclude mode estimation and [3] uses a different GPU card than ours.

The HoG detector in the experiments of [34] reaches a speed of 353 ms per 1280x960 image which is further faster than our cascade, however they do not report results for downscaling ratios other than 1.05. We believe that our detector has comparable speed performance as these two suggested implementations, demonstrating real-time performance.

We conclude our discussion about the parallel implementation of the cascade detector with an inspection of occupancies and processing times for each part in our method:

Processing step	Occupancy	Memory throughput	Processing time
Data transfers	–	–	18 ms
Gradient kernel	50%	56 GB/s	6 ms
Integral hist.	100%	17 – 50 GB/s	50 ms
Early cascade st.	25 – 50%	18 GB/s	16 ms
Late cascade st.	19 – 31%	7 GB/s	41 ms

Table 7-2: Average performance results for a 1280×960 image with 1.2 subsampling ratio. Values for kernels used in integral histograms are reported in Chapter 5. Occupancies of classification kernels depend on the number of features at a given stage.

Chapter 8
Conclusion

In this thesis, we investigated the Histograms of Oriented Gradient (HoG) algorithm for person detection, as well as two different variations of this method and presented a study on the effect of these modifications on the detection speed and accuracy.

By formulating the detection algorithm in terms of a cascade of strong classifiers, we demonstrated that it is possible obtain a significant speed up. This is achieved by rapidly rejecting trivial detection windows and spending more time and resources on the more complex ones. The features employed in the ensembles are automatically chosen by the AdaBoost algorithm, so that they complement each other to satisfy rejection and detection rates at each stage of the attentional cascade. We reported a similar ROC curve and more than order of magnitude acceleration relative to the original Dalal & Triggs method.

The integral image is a powerful alternative image representation that has opened the path to real time performance for cascade-of-rejectors type detectors. By providing parallel algorithms to efficiently compute this data structure, we observed a speed up of about an order of magnitude compared to the sequential image. We also presented a study on the effect of using double precision floating point arithmetic and vector type data on the computation performance.

We also investigated the feasibility of a data parallel realization of the cascade detector with HoG features on the GPU. Even though the evaluation of these features requires random memory accesses which do not fit well in the GPU memory model, we obtained a significant performance boost compared to our sequential implementation. This is thanks to the efficient integral image algorithms we introduced and using two different kernels to better suit the complexity of the cascade stages. For a 1280×960 image, we observed a 13× speed up relative our CPU realization, and noted comparable detection times with the proposed GPU implementations of the Dalal-Triggs algorithm.

Bibliography

[1] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005

[2] Q. Zhu, S. Avidan, M. Yeh, and K. Cheng. Fast Human Detection using a Cascade of Histograms of Oriented Gradients. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006

[3] C. Wojek, G. Dorkó, A. Schulz, and B Schiele. Sliding-Windows for Rapid Object Class Localization: A Parallel Technique. *DAGM-Symposium* 2008: 71-81

[4] P. Viola and M. Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2001

[5] M. Enzweiler and D. M. Gavrila. Monocular Pedestrian Detection: Survey and Experiments. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31(12) pp. 2179-2195

[6] C. Papageorgiou and T. Poggio. A Trainable System for Object Detection. *International Journal of Computer Vision (IJCV)*, 38(1):15-33, 2000

[7] D. M. Gavrila and V. Philomin. Real-Time Object Detection for Smart Vehicles. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 1999

[8] P. Viola, M. Jones and D. Snow. Detecting Pedestrians Using Patterns of Motion and Appearance. *International Conference on Computer Vision (ICCV)*, 2003

[9] L. Zhang and R. Nevatia. Efficient Scan-Window Based Object Detection Using GPGPU. *Proc. IEEE CVPR Workshops (CVPRW'08)*, pp. 1–7, Jun 2008.

[10] T. Joachims. Making Large-Scale SVM Learning Practical. *Advances in Kernel Methods - Support Vector Learning, B. Schölkopf and C. Burges and A. Smola (ed.).* MIT-Press, 1999.

[11] INRIA Object Detection and Localization Toolkit
http://pascal.inrialpes.fr/soft/olt

[12] OpenCV, Open Computer Vision Library
http://opencv.willowgarage.com/wiki/

[13] Navneet Dalal. Finding People in Images and Videos. *PhD Thesis. Institut National Polytechnique de Grenoble / INRIA Rhône-Alpes, Grenoble, July 2006*

[14] O. Tuzel, F. Porikli, and P. Meer. Human Detection via Classification on Riemannian Manifolds. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007

[15] NVIDIA: NVIDIA CUDA SDK code samples: Scan

[16] B. Bilgic, B.K.P. Horn, I. Masaki. Efficient Integral Image Computation on the GPU. *Submitted to IEEE Intelligent Vehicles Symposium, 2010*

[17] N. Dalal, B. Triggs, and C. Schmid. Human Detection using Oriented Histograms of Flow and Appearance. *In Proc. of the European Conference on Computer Vision (ECCV)*, pages 428–441, 2006.

[18] M. Enzweiler, P. Kanter, and D. M. Gavrila. Monocular Pedestrian Recognition using Motion Parallax. *In Proc. of the IEEE Intelligent Vehicles Symposium,* pages 792–797, 2008.

[19] D. M. Gavrila and S. Munder. Multi-Cue Pedestrian Detection and Tracking from a Moving Vehicle. *International Journal of Computer Vision*, 73(1):41–59, 2007.

[20] C. Stauffer and W.E.L. Grimson. Adaptive Background Mixture Models for Real-Time Tracking. *Proc. Computer Vision and Pattern Recognition 1999 (CVPR 1999)*, June 1999.

[21] B. K. P. Horn and B. G. Schunck. Determining Optical Flow. *Artif. Intell.,* vol. 17, pp. 185–203, 1981.

[22] MIT 6.869 Advances in Computer Vision lecture notes

[23] K.Mikolajczyk, C.Schmid. Indexing Based on Scale Invariant Interest Points. *ICCV 2001*

[24] D.Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *IJCV 2004*

[25] K. Mikolajczyk, C. Schmid, and A. Zisserman. Human Detection Based on a Probabilistic Assembly of Robust Part Detectors. *In Proc. of the European Conference on Computer Vision (ECCV)*, pages 69–81, 2004.

[26] A. Mohan, C. Papageorgiou, and T. Poggio. Example-Based Object Detection in Images by Components. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 23(4):349–361, 2001

[27] D. M. Gavrila. A Bayesian, Exemplar-Based Approach to Hierarchical Shape Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(8):1408–1421, 2007.

[28] M. Enzweiler and D. M. Gavrila. A Mixed Generative-Discriminative Framework for Pedestrian Classification. *In Proc. of the International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.

[29] R. Fergus, P. Perona, and A. Zisserman. Object Class Recognition by Unsupervised Scale-Invariant Learning. *In Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2003

[30] V. Vapnik. The Nature of Statistical Learning Theory. Springer-Verlag, 1995.

[31] R. E. Schapire. The Boosting Approach to Machine Learning, an Overview. *In MSRI Workshop on Nonlinear Estimation and Classification*, 2002.

[32] C. Wöhler and J. Anlauf. An Adaptable Time-Delay Neural-Network Algorithm for Image Sequence Analysis. *IEEE Transactions on Neural Networks*, 10(6):1531–1536, 1999.

[33] D. Comaniciu. An Algorithm for Data-Driven Bandwidth Selection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(2):281–288, 2003

[34] Victor Prisacariu and Ian Reid. FastHOG - a Real-Time GPU Implementation of HOG. Technical Report 2310/09, Department of Engineering Science, Oxford University, 2009.

[35] NIVIDA CUDA *Compute Unified Device Architecture, Programming Guide 2.3*

[36] B. Bilgic, B.K.P. Horn, I. Masaki. Fast Human Detection with Cascaded Ensembles on the GPU. *Submitted to IEEE Intelligent Vehicles Symposium, 2010*

[37] Yoav Freund and Robert E. Schapire. A Decision-Theoretic Generalization of On-line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119-139, 1997

[38] P.Y. Simard, L. Bottou, P. Haffner, and Y.L. Cun. Boxlets: a Fast Convolution Algorithm for Signal Processing and Neural Networks. In M. Kearns, S. Solla, and D. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11, pages 571–577, 1999

[39] F. Porikli. Integral histogram: A Fast Way to Extract Histograms in Cartesian Spaces. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005

[40] B. Bilgic, B.K.P. Horn, I. Masaki. Fast Human Detection with Cascaded Ensembles on the GPU. *Submitted to IEEE Intelligent Vehicles Symposium, 2010*

[41] C.H. Messom, A.L. Barczak. High Precision GPU based Integral Images for Moment Invariant Image Processing Systems. *Electronics New Zealand Conference (ENZCON'08), 2008*

[42] M. Harris. Parallel Prefix Sum (Scan) with CUDA. *NVIDIA CUDA SDK code samples*

[43] NVIDIA: NVIDIA CUDA SDK code samples, Transpose