2010 IEEE Intelligent Vehicles Symposium
University of California, San Diego, CA, USA
June 21-24, 2010

Fast Human Detection with Cascaded Ensembles on the GPU

Berkin Bilgic, Berthold K.P. Horn, and Ichiro Masaki

Abstract— **We investigate a fast pedestrian 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 humans 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, which is similar to the recent GPU implementation of the original HoG algorithm in [3].**

I. Introduction

Detecting humans 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 to represent a detection window. When the block histograms within the window are concatenated, the resulting feature vector is powerful enough to classify humans 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 with a sparse scanning methodology.

To speed up the method, Zhu *et al.* [2] combined the cascade-of-rejectors approach [4] that has been the benchmark method in face recognition with the HoG features. This 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. Since the Dalal-Triggs algorithm employs small, fixed-size histogram blocks defined in a dense grid, it is not possible to capture the "big picture" to make fast rejections with such blocks in the early stages of the cascade. To find out which combinations of blocks can be used together for summarizing a detection window in the early stages and for providing detail in the later stages, Zhu *et al.* proposed to choose the most informative blocks out of a large feature pool by using AdaBoost. The selected block features have the most suitable size, location and aspect ratio to complement the other features within each stage.

By this formulation, the method was reported to yield 4 to 30 FPS performance on a 320×240 image, depending on the scanning density. Although not reported, 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.

In our work, we aim to show that by using efficient parallel algorithms with a technology called general-purpose computation on graphics processing units (GPGPU), we can speed up the cascade detector by a factor of 13 relative to our CPU implementation to achieve a near real-time performance of 8 FPS for a 1280×960 image.

In the rest of this paper, we give a brief background information human detection and GPGPU, describe the cascade-of-rejectors algorithm, and give details about our CPU and GPU implementations. We finish with experimental results.

I. Background

There is an extensive literature on human detection, and [5] gives an elegant survey on this topic. Papageorgiou *et al.* [6] presents a detector that combines Haar wavelet features with a polynomial SVM classifier. Gavrila and Philomin [7] use the chamfer distance of edge images, and match them with a learned exemplar set. Viola *et al.* [8] use AdaBoost to train a chain of rejection rules that employ Haar-like wavelets and spatial-temporal differences. Dalal and Triggs [1] use a single window approach with a dense HoG descriptor and a linear SVM for classification. Tuzel *et al.* [14] demonstrate superior results over the Dalal-Triggs algorithm by using covariance matrices as object descriptors.

Due its simplicity and high descriptive power, several authors worked on the Dalal-Triggs algorithm to make it feasible for real time detection. Among these, Wojek *et al.* [3] and Zhang *et al.* [9] suggest using the GPGPU technology for implementation. In [3], a speed up by a factor of 34 over the original CPU implementation by [1] is

Final manuscript received May 15, 2010.

B. Bilgic is with the Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA 02139, USA (e-mail: berkin@mit.edu).

B.K.P. Horn is with the Department of Electrical Engineering and Computer Science and CSAIL, MIT, Cambridge, MA 02139, USA (e-mail: bkph@csail.mit.edu).

I. Masaki is with the Department of Electrical Engineering and Computer Science and MTL, MIT, Cambridge, MA 02139, USA (e-mail: IMasaki@aol.com).

reported, and this corresponds to a processing time of 385ms per 1280×960 image. Similarly, the work in [9] achieves a more than 10 times speed up over the original CPU code of [1]. Zhu *et al.* [2] suggest formulating the HoG algorithm as a rejection cascade and demonstrate 30× speed up over the original algorithm for 320×240 images, both working on the CPU.

II. The Dalal-Triggs Algorithm

The method starts by applying square root gamma correction to the input RGB image. Then gradients are computed for all three color channels using centered kernels [-1, 0, 1] in horizontal and vertical directions. For each pixel, the gradient magnitude is taken from the color channel with the largest gradient norm. Next, gradient orientations are computed and discretized into 9 orientation bins, lying between 0° and 180°. A weighted vote for an edge orientation histogram is calculated for a given pixel, and the votes are accumulated into orientation bins over local spatial regions called *cells*. To reduce the aliasing, the gradient votes are trilinearly interpolated between neighboring bin centers in orientation and space. Cells are taken to be of size 8×8 pixels, and grids of 2×2 cells are grouped in *blocks*. Since each cell is represented by 9 histogram bins, a histogram block then has a 36 dimensional feature vector. To increase robustness, each block is normalized with *L2-Hys* norm; L2-norm followed by clipping the maximum element in the vector to 0.2. Within a detection window of size 64×128 a total of 105 blocks can be defined, when overlapping by 8 pixels in both dimensions is allowed. This way, each window is represented by a 105×36 = 3780 dimensional feature vector, which is used to train a linear SVM classifier. The detection window scans over the whole image with vertical and horizontal strides of 8 pixels, then the image is downscaled with a ratio of 1.05. This process continues until all scales are accounted for. Finally, a mode estimator based on the mean shift algorithm fuses multiple detections in space and scale to return bounding boxes corresponding to detections. Given a 1280×960 image, all these correspond to evaluating about 150000 detection windows, which takes many seconds to run on a CPU.

III. The Cascade-of-Rejectors Algorithm

The success of the Dalal-Triggs 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, as noted in [2], 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 1). 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 our algorithm our feature pool contains 5029 HoG blocks, as opposed to the 105 fixed size blocks that define a window in the Dalal-Triggs algorithm. We also note that the first stages of the cascade employ large blocks that attempt to capture the "big picture" and the later stages include smaller ones to provide more detail.

We also note that this rejection cascade was first introduced in Viola & Jones' seminal face detection work [4], where they suggested using Haar-like wavelets as features. However, this methodology was reported to be unable to perform equally well in human detection, especially in a cluttered, complex dataset [5, 2].

Another important contribution in [4] was the introduction of the integral image for rapid evaluation of features. Paralleling [2], we also use the integral image idea to compute the orientation histogram of the image.

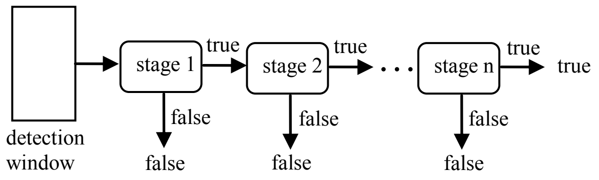


Figure 1: Cascade-of-rejectors. Detection window is passed to the stage 1 which decides true or false. A false determination stops further computation, and the window is classified to contain non-pedestrian. A true determination triggers the computation at the following stage. Only if the window passes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

A. 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, introduced in [4] and detailed in Figure 2, enables us to compute the sum of the elements within a rectangular region by using 4 image access operations. As in [2], we discretize each pixel's gradient orientation into 9 bins, then compute and store an integral image for each histogram bin. The HoG for any rectangular region then can be computed by 9 × 4 = 36 image access operations, 4 for each of the 9 bins.

This formulation 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.

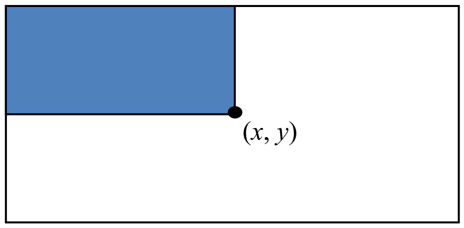


Figure 2: The value of the integral image I_{int} at point (x, y) is the sum of all the pixels above and to the left in the input image I;

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

B. Histogram Cells and Blocks

For a 64×128 detection window, 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. Each of these blocks is further divided into a grid of 2×2 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 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.

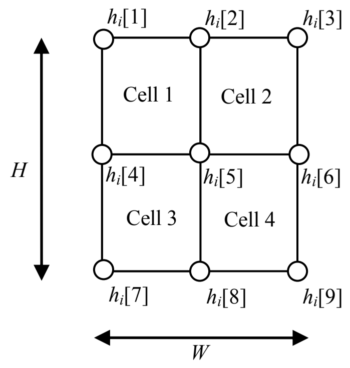


Figure 3: Computing the block histogram for a block with size $W \times H$. Using the i^{th} integral histogram, it is possible to compute the i^{th} elements of the cell histograms using 4 image accesses. For instance, i^{th} element of cell 1's orientation histogram is computed as $h_i[5] + h_i[1]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

C. 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,

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

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

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

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 in the INRIA database [11]. For the first stage, we randomly sample 2416 negative windows of size 64×128 from the 1654 full-size (≥ 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.

```
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.

D. CPU Implementation of the Cascaded Detector

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 4). After acquiring the image and converting it to grayscale, gradient magnitude and orientation are computed for each pixel. Since 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 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). Following [13], we achieve this using a mean shift algorithm in 3D position/scale space. 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.

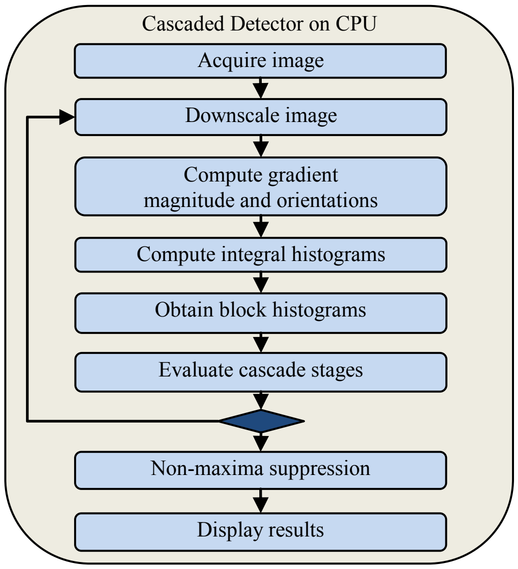


Figure 4: Steps of localization using the cascaded HoG detector on the CPU

IV. GPGPU AND NVIDIA CUDA

The term GPGPU refers to using graphics processing units to accelerate non-graphics problems. The many-core architecture of new generation GPUs enables them to execute thousands of threads in parallel and make more than 1 teraflops floating point operations per second. This computational power provides an excellent platform for computer vision algorithms, especially the ones that can be classified as “embarrassingly parallel”.

In our parallel implementation, we make use of NVIDIA’s CUDA programming model, which is a software platform for massively parallel high-performance computing on the GPUs. According to this model, parallel portions of an application are executed on the GPU as *kernels*, which are functions in written in the C language. CUDA allows these kernels to be executed multiple times by multiple *threads* simultaneously. A typical application would use thousands of threads to achieve efficiency.

For scalability, multiple threads are grouped in *thread blocks* and multiple blocks reside in a user specified *grid*. Up to 512 threads can be grouped in a block, and threads within the same block can cooperate via the block's on-chip *shared memory* (16kB) and synchronize their execution to coordinate memory access. Each block runs on the same multiprocessor, while a multiprocessor can execute several blocks at a time.

Threads may access several different memory types during their execution, albeit with different latencies. Apart from the shared memory that is visible to all threads within a block, each thread has a private local memory and registers. Additionally, all threads have access to three types of off-chip memory: The *global memory* (1792MB) has high latency and is not cached. The *constant memory* (64kB) is cached and typically fast if all threads are accessing the same address. The *texture memory* is also cached and optimized for 2D locality, and it allows virtually free hardware interpolation. However, constant and texture memory are read-only memory types.

When a thread needs to access the same address in the high latency global memory multiple times, copying data to the shared memory and accessing it from there would be preferable. This is because it takes about 400 to 600 clock cycles to issue a memory instruction for the global memory, which has about 150 times more latency than the shared memory. Each memory type in CUDA has different access patterns and maximum sizes, hence developers need to decide where the data are stored for the best performance.

In our experiments, we use a GeForce GTX 295 video card. It contains 2×30 multiprocessors, each one containing 8 thread processors. Hence, it is capable of running 480 threads simultaneously.

A. GPU Implementation of the Cascaded Detector

Figure 5 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 carried to fuse the neighboring positives. In what follows, we detail the steps of our detector.

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.

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_{1-4} and I_{5-8}, and one `float` array I_9. Hence, at a given pixel, we store its magnitude in the appropriate field of I_{1-4} if its orientation is between 0° and 80°, in I_{5-8} if the orientation is between 80° and 160°, and in I_9 if it is larger than 160°.

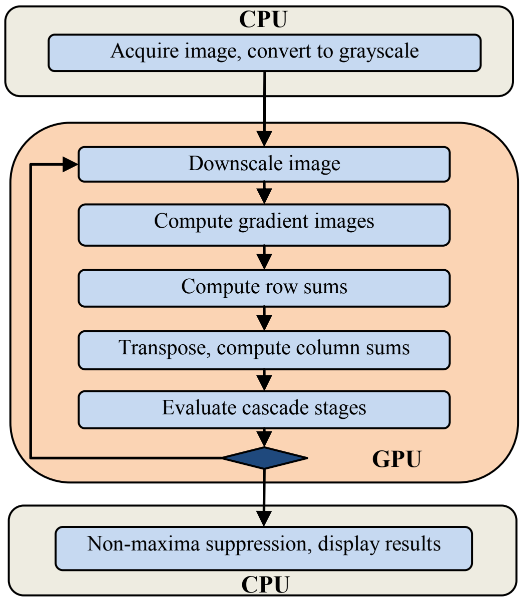
Computing the integral histograms: Our implementation is inspired by the Parallel Prefix Sum (Scan) example [15]. The *all-prefix-sums* operation takes a binary associative operator $\oplus$, and an array of n elements

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

and returns

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

In our case the operator $\oplus$ is summation, and it generates a new array where each element j is the sum of all elements up to j. We apply this operation on each row of the input image independently, then compute transpose of the image


CPU
Acquire image, convert to grayscale
Downscale image
Compute gradient images
Compute row sums
Transpose, compute column sums
Evaluate cascade stages
GPU
Non-maxima suppression, display results
CPU
Figure 5: Steps of localization on the GPU

efficiently based on the guidelines in [15]. If the rows of the transposed image are again scanned with this operator, the resultant array gives us the integral image. We perform these steps on I_{1-4}, I_{5-8}, and I_9 in order to compute the integral histograms. For a detailed discussion of this step, please refer to [16].

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_{1-4}, I_{5-8}, and I_9 (Figure 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 $2^{13} \cdot 9 \cdot 4 =$ 288kB 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:

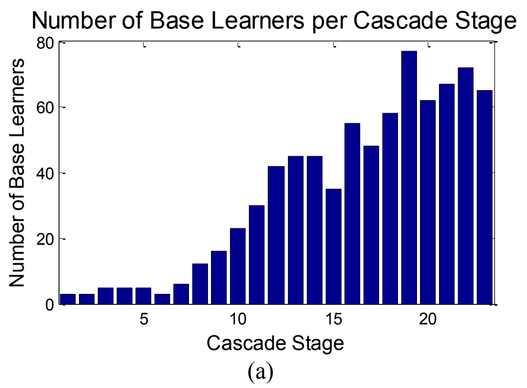
Feature evaluation, late stages: When the number of base learners is so high that it is not possible to launch more than one thread block due to shared memory pressure, we resort to a different kernel formulation. Now we employ two dimensional blocks with size $3 \times n_i$, where a group of 3 threads are responsible for a single base learner. Each of these 3 threads operate on one of I_{1-4}, I_{5-8}, 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.

V. Experiments

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 7 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 7a).

In Table 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



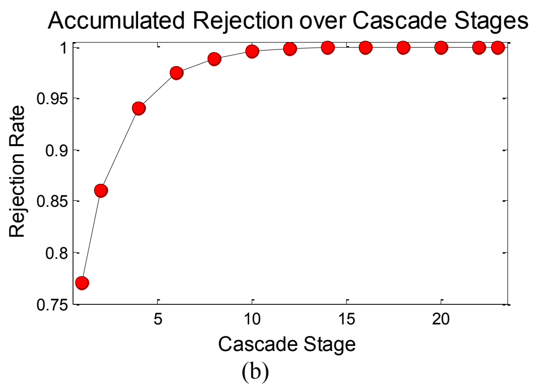


Figure 7: Cascaded classifier that uses variable-size HoG blocks as features in detail. The cascade consists of 23 stages where the base learners are linear SVMs with 36-D features of block histograms, chosen out of a feature pool of 5029 blocks with the AdaBoost algorithm. (a) The number of base learners 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.

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 8, 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	500 ms	7 sec
Zhu *et al.*	30 ms	250 ms
Our approach	82 ms	475 ms

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

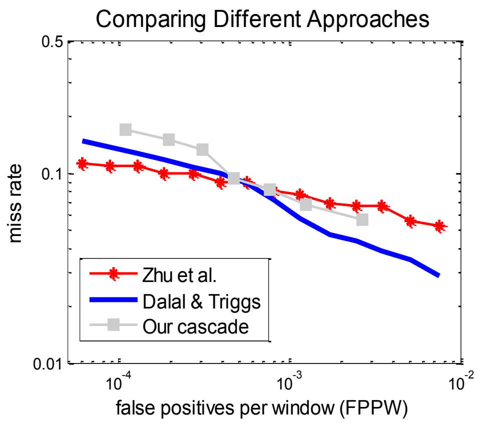


Figure 8: Comparing Zhu *et al.*, Dalal & Triggs and our cascade. Our implementation is comparable with the other two, especially when FPPW goes up.

Table 2 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 reaches a similar 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 13× speed up when our method runs on the GPU. In Table 3, we inspect occupancies and processing times for each part in our algorithm.

To inspect the most informative blocks selected by the AdaBoost algorithm, we visualize the blocks in cascade stages 1, 3 and 5 in Figure 9. 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 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.

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

Table 2: Processing times for a 1280×960 image. Presented results are for three different downscaling factors using 8×8 spatial strides. We note that our results exclude mode estimation and [3] uses a different GPU card than ours.

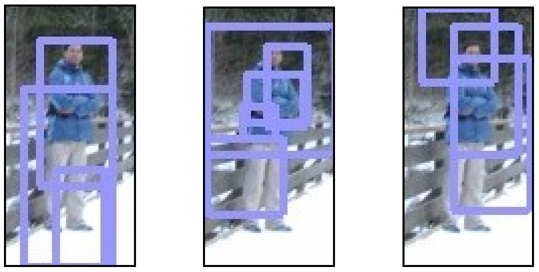


Figure 9: Visualizing the selected blocks by the AdaBoost algorithm. (a) Blocks in the first stage, (b) blocks in stage 3, and (c) blocks in stage 5 of the cascade. The blocks in (a)-(c) are the blocks with lowest weighted error out of 125 features sampled randomly from a feature pool of 5029 blocks.

Processing step	Occupancy	Memory throughput	Processing time
Data transfers	–	–	18 ms
Gradient kernel	50%	56 GB/s	6 ms
Integral hist.[16]	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 3: Average performance results for a 1280×960 image with 1.2 subsampling ratio. Values for kernels used in integral histograms are reported in [16]. Occupancies of classification kernels depend on the number of features at a given stage.

VI. CONCLUSIONS

We present a fast human detection framework that achieves about $5 \cdot 10^{-5}$ FPPW on the training set and runs at 8 frames per second for a 1.2 megapixel image. By combining the cascade-of-rejectors approach with the Histogram of Oriented Gradients (HoG) features and selecting the mutually most informative histogram blocks with the AdaBoost algorithm, we are able to demonstrate near real time detection performance. Our results are comparable in terms of speed and accuracy with recent data parallel person detectors.

REFERENCES

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

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