Efficient Scan-Window Based Object Detection using GPGPU

Li Zhang and Ramakant Nevatia
University of Southern California
Institute of Robotics and Intelligent Systems
{li.zhang|nevatia}@usc.edu

Abstract

We describe an efficient design for scan-window based object detectors using a general purpose graphics hardware computing (GPGPU) framework. While the design is particularly applied to built a pedestrian detector that uses histogram of oriented gradient (HOG) features and the support vector machine (SVM) classifiers, the methodology we use is generic and can be applied to other objects, using different features and classifiers. The GPGPU paradigm is utilized for feature extraction and classification, so that the scan windows can be processed in parallel. We further propose to precompute and cache all the histograms in advance, instead of using integral images, which greatly lowers the computation cost. A multi-scale reduce strategy is employed to save expensive CPU-GPU data transfers. Experimental results show that our implementation achieves a more-than-ten-times speed up with no loss on detection rates.

1. Introduction

Detection of objects of a specific class is an important task for many computer vision applications. The accuracy of such methods has improved greatly in recent years but the speed of processing remains a issue in their wide spread use. It is natural to consider utilizing powerful graphics hardware for speed up, which have become available for very affordable costs. However, because of the specialized architecture of those hardware, how to apply them to a general-purpose computational application, *i.e*. objection detection, is a highly non-trivial task.

Our objective in this paper, is to develop a generic GPGPU design that can be used for implementation of many object detection methods that share a similar *scan-window* structure, namely scanning of the image with windows of various sizes, computing features in each window, applying a classifier and combining the results; Figure 1 shows an example. We focus on implementing such a design for pedestrian detection. However, it can be applied to any object detection method that uses the scan-window paradigm. The algorithm is based on Dalal *et al*.'s work[4], since it is often considered as baseline with stable good results and simplicity. Their method uses HOG (Histogram of gradients) features and an SVM (Support Vector Machine) classifier. Many new types of features [4, 14, 13, 11, 12] and classifiers [4, 14, 16] have been tried under the scan-window framework to achieve a better detection performance. Our approach can also be translated to these improved features and classifiers.

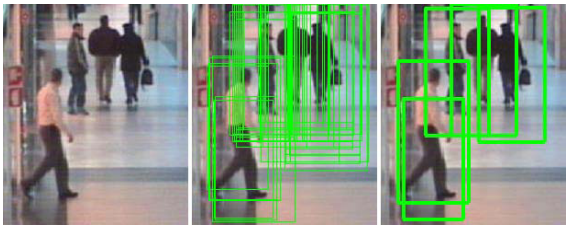

Figure 1. Pedestrian Detection: left) input image; mid) detection responses, *i.e*. the scan windows that have been classified as human; right) merged responses.

In the past few years, graphics process units (GPU) have evolved to become one of the most powerful computational hardware for a given price. Improvements in programmability also make them potentially useful for more general computation-centric tasks. The GPGPU framework is a result of the exploration of such potentiality. In GPGPU, computation is mapped into a graphics rendering process. Because rendering is executed highly in parallel by multiple shader units inside GPU, large improvement in speed can be achieved. The key point of applying GPGPU to a specific domain is how to setup a mapping between the computation and graphics rendering.

Our implementation is based on the observation that the computation for all scan windows in object detection follows an identical procedure: feature extraction and classification, which fits into GPGPU's stream programming model. To reduce the redundant computation, we extract the features separately from the scan windows, so that they

978-1-4244-2340-8/08/$25.00 ©2008 IEEE

can be shared among classification of different scan windows, which is more efficient than the previous integral image approaches. The graphics rendering is also used to build the scale-pyramid of the input image and combine detection results from different scales, so that only one upload and download is needed for CPU-GPU data transfer. Our detector achieves a more-than-ten-times speed up compared to the original detector with no loss in detection rates.

The rest of the paper is organized as follow. We first discuss related works in section 2, and review the GPGPU programming model briefly in section 3. The original pedestrian detector under sequential computing model is described in section 4. Our GPGPU-based detector is introduced in section 5. Evaluations are provided in section 6. The conclusion comes in section 7.

2. Related Work

The use of graphics hardware for general-purpose computation has been an area of active research for several years. Recently, rapid improvement in performance and programmability of graphics chips has made GPGPU an intriguing choice. [10] provides a detailed survey of GPGPU framework and a variety of applications. An online community about GPGPU can be found at `http://GPGPU.org`.

In computer vision, use of GPGPU has also been explored in many areas such as image filtering, tracking and geometry transformations. NVIDIA's "GPU Gems 2" book released in 2005 has a chapter dedicated to computer vision[5], which includes correcting radial distortions, Canny edge detector, tracking hands and computing image panoramas on GPU. Yang *et al*. [15] implemented real-time stereo on GPU. Ohmer *et al*. implemented a face recognition system using Kernel PCA and SVM [9]. Labatut *et al*. implemented level set-based multi-view stereo [8]. The well-known SIFT algorithm was implemented in 2007 by Heymann *et al*. [7]. There are also libraries providing reusable implementations of many basic vision algorithms such as MiniGPU[3] and OpenVIDIA[6]. More general GPGPU-architecture based programming language, *i.e*. CUDA[1] is also becoming popular. However, to the best of our knowledge, the use of GPGPU in high-level vision tasks such as object detection and recognition is rare among previous research works.

3. GPGPU Programming Model

The programming model of GPGPU has been described in many earlier surveys such as [10]. The entire GPU pipeline includes four stages: geometry, rasterization, fragment and composition. Like many general-purpose computation tasks, we only make use of the fragment and composition stages. Here we review this model briefly and highlight the key points. The concept of GPGPU program model

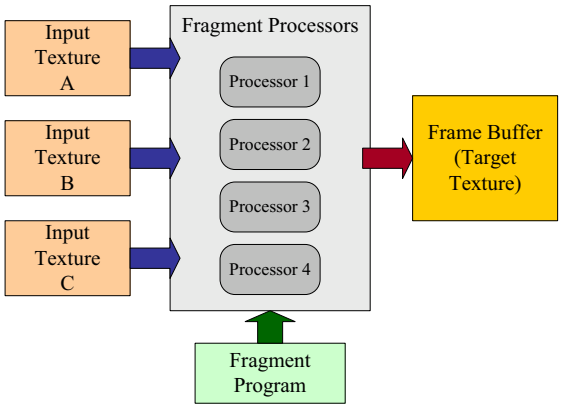

Figure 2. GPGPU Programming Model

(the fragment and composition stages) can be described as follows (Figure.2):

1. Input data are converted into 2D-textures and stored into GPU memory. A target 2D texture is attached to a frame buffer inside GPU to store the output of computation.

2. A graphics rendering of the frame buffer is triggered. Through the rendering procedure, a fragment program is executed for every target pixel in the frame buffer. Each execution of the program is allowed to take random access from all the input 2D textures but has to write the result into the corresponding target pixel.

3. The rendering procedure can be performed iteratively among several textures, with each of them taking turns as the target. This is often referred as *ping-ping* or *multi-pass*. After the final pass, the target texture needs to be read back into CPU memory.

Rendering is highly efficient because there are multiple fragment shader processors insides GPU which provide essential parallelism; additionally, the GPU operations themselves are vectorized and fast for float-point arithmetic.

However, there are two limitations for GPGPU programming model. First, the flow control, *i.e*. branching in fragment programs is limited. Currently, the GPU fragment processors only support Single Instruction Multiple Data(SIMD) processing, which means that if evaluation of the branch condition is identical for all processors, only the taken side of the branch will be evaluated; otherwise, both sides of the branch will be evaluated and the results of the taken sides are stored. It does not affect the correctness of programs, but may reduce the performance with redundant computation. Second, the data transfer between GPU and CPU has a large overhead. To make GPGPU profitable for the system, the data transfer should be minimized.

4. CPU-based HOG-SVM Detector

The HOG-SVM based pedestrian detector was originally proposed by Dalal *et al*. [4]. We review the typical CPU-

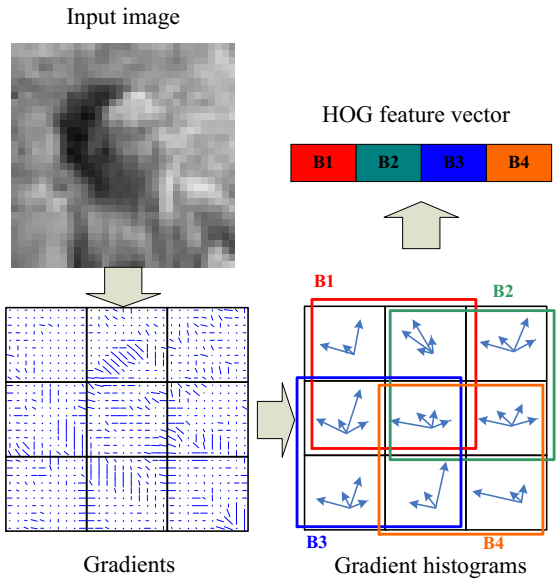

Figure 3. Extract HOG features

based implementation of the feature extraction and classification steps and analyze their running times. In this paper, we assume the size of the scan window is $w \times h$ where $w = 64$ and $h = 128$ as has been used in [4].

4.1. Histogram of Oriented Gradients

Histogram of Oriented Gradients(HOG) feature[4] is a grey-level image feature formed by a set of normalized gradient histograms as follow: the image pixels in a scan window are grouped into $r \times r$ rectangle cells. Within a cell, every pixel votes for its gradient orientation weighted by its gradient magnitude. All the votes are accumulated as a histogram. A fine quantization in gradient orientations is suggested for good performance. In our experiments, the orientations between $[0, \pi)$ are evenly divided into 8 histogram bins, The orientations between $[\pi, 2\pi)$ are mirrored, because the contrast of object and background is unreliable. The neighboring $c \times c$ cells are grouped into a block, in which $8c^2$ histogram bin values are combined as a vector B with $|B|$ normalized to one. Finally, all the blocks are concatenated into a high-dimensional HOG feature vector (Figure 3).

We calculate the computational cost in terms of number of memory-read accesses. For calculation of one HOG feature vector, assuming that the gradients are precomputed, it is equal to:

$$\frac{w}{r}\frac{h}{r}r^2 + 2L = wh + 2L \tag{1}$$

where

$$L = 8c^2(\frac{w}{r} - c + 1)(\frac{h}{r} - c + 1) \approx \frac{8c^2}{r^2}wh \tag{2}$$

is the length of the feature vector.

A common . way to improve the efficiency is by employing integral images for every gradient orientation and normalization factor [16]. Because the sum of any rectangle area can be calculated by 4 vertex accesses in the integral images, the total computation of the HOG feature is

$$4 \cdot 8 \cdot \frac{w}{r}\frac{h}{r} + 4\frac{L}{8c^2} + L = \frac{32}{r^2}wh + (1 + \frac{1}{2c^2})L \tag{3}$$

Comparing (1) and (3), it can be shown that the ratio of computational costs between raw HOG and using integral images is

$$\frac{r^2 + 16c^2}{36 + 8c^2}$$

In our implementation where we choose $r = 8$ and $c = 2$, the ratio is in favor of integral images. There's an overhead for precomputing the nine integral images. For a input image with size $W \times H$, it is equal to $16WH$ (WH for every orientation and $8WH$ for the normalization factor).

4.2. SVM Classifier

Support Vector Machines are a well-known statistical learning method; objective in SVM learning is to find a decision plane that maximizes the intra-class margin. Since the training procedure of SVMs does not affect the efficiency of the detector, we will only describe the usage of the trained classifiers.

A linear SVM can be simply represented as a dot-product between the feature vector and a weight vector, *i.e.*

$$f(x) = w \cdot x$$

The feature vector is classified as human if the output $f(x)$ is larger than a threshold b.

The SVM classification can be performed in a high dimensional kernel space by the "kernel trick". The kernel SVM usually outperforms linear one, as has been shown by Dalal *et al*. [4]. However, the storage and computation costs are much higher for non-linear kernel functions. We only use linear SVM. The computational cost in term of memory accesses is $2L$.

Summarizing the feature extraction and classification steps, the total computational cost of detection on one scale of the input image is equal to

$$\begin{aligned} & \frac{WH}{d^2}\left(\frac{32}{r^2}wh + (1 + \frac{1}{2c^2})L + 2L\right) + 16WH \\ \approx\ & WH\left(wh \cdot \frac{36 + 24c^2}{r^2d^2} + 16\right) \end{aligned} \tag{4}$$

where d is the interval between neighboring scan windows. The computation of dot-products for every scan window is almost inevitable. However, we will later show that the computation of HOG features can be further reduced by aligning the scan window grids with the histogram cells.

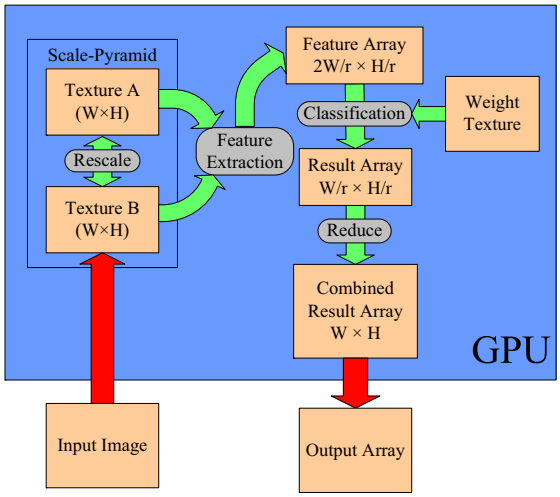

Figure 4. GPGPU-based Pedestrian Detector

5. GPGPU-based HOG-SVM Detector

In the CPU-based detector, the feature extraction and classification steps are simply repeated for every scan window. Therefore, a straightforward idea for its GPGPU implementation would be to combine the two steps in one fragment program, rendering to a $\frac{W}{d} \times \frac{H}{d}$ sized frame buffer, where each pixel represents the classification result of one scan window. However, this is unlikely to work. First, the computation of integral images is hard to fit into the GPGPU paradigm, because it's based on an order-dependent dynamic programming between successive image rows. Moreover, unlike a data copy operation within the main memory, a CPU-GPU data transfer is expensive. Therefore, the feature extraction module as well as the data transfer need to be carefully redesigned.

We divide the GPGPU-based detector into four modules: scaling, feature extraction, classification and reduction. The scaling module downloads the input image into GPU and resizes it to different scales (*i.e.* a scale-pyramid). The reduce module collects the detection results from all those scales. On each scale, the feature extraction module renders the feature array from the image, with one gradient histogram generated per fragment; then the classification module renders the result array from the feature array with one scan window per fragment. Our design, instead of going back to the raw method of HOG feature extraction, precomputes the features independently and reuses them among scan windows, which turns out to take even less computational cost. The entire detection procedure is illustrated in Figure 4.

5.1. Download and Build Scale-Pyramid

To reduce data transfer from main memory to GPU, the ideal choice is just to transfer the input image, and generate all other images of different scales inside the GPU; this is what we do in our implementation. We use the fast transfer technique from http://gpgpu.org to download the grey-level image onto a 2D 4-channel float-point texture on GPU. Then a typical "ping-pong" strategy is used to generate a scale-pyramid of images from two textures: keep one texture as input and the other as target, resize the image, and then swap. Resizing is done by linear interpolation.

5.2. Feature Extraction

Because the scan windows are applied over the entire image with a small stride, d, in between two placements, two neighboring scan windows have a large overlap. The histogram cells computed by the two scan window within the overlapping region produce almost the same information. To reduces this duplicated computation, our detector precomputes histogram cells independent of the scan windows, and shares them among different windows later in the classification module. In our implementation, we set the stride to be equal to the size of histogram cell, *i.e.* $d = r = 8$, so that the boundaries of histogram cells are always aligned with those of scan windows. The effect of this setting to the detection performance is small since the size of the cell approximates to the size of the smallest descriptive parts of the objects; the scale-pyramid also reduces the affect of the grid alignments.

Feature extraction is performed by a rendering from a $W \times H$ texture, *i.e.* one image in the scale-pyramid, to a $\frac{2W}{r} \times \frac{H}{r}$ texture as the feature array. The rendering procedure is described in Fragment Program 1.

- Let p_0 be the target coordinate, I be the input texture
- Decompose p_0 as $(2x + k, y)$ where $k \in 0, 1$
- Let histogram vector $H = (0, 0, 0, 0)$
- For pixel $p \in [rx, rx + r - 1] \times [ry, ry + r - 1]$
  - compute the gradient orientation $v \in \{0 \ldots 7\}$ and magnitude u at $I(p)$
  - if $4k \leq v < 4k + 4$
    let $H_{v-4k} = H_{v-4k} + u$
    (implemented by 4 "if else" statements)
- return H

Fragment Program 1:Compute the gradient histogram cell.

In Fragment Program 1, all of the textures use the 4-channel float-point type. Therefore a 8-bin histogram needs to be stored in two successive pixels, each of which stores 4 bins. Whether a pixel should accumulate the upper-four bins or lower-four bins is determined by whether its first coordinate is odd or even. In implementation, we actually move the calculation of gradient orientations and magni-

tudes into a separate fragment program to further remove redundant computation of gradient data.

GPUs usually have limited support for dynamic array indices and branching, which are often required methods for creating histograms. Therefore, how to build histograms efficiently is often of interest to GPU developers. We tried three different implementations, including simple"if else" enumeration, the "cos trick" from OpenVIDIA[6] and our "step trick". The "step trick" for a n-bin histogram is to create an indicator function by calculating

$$1 - \text{step}((x - [0, 1, \ldots, n])^2, 0.5)$$

, where $\text{step}(x, a)$ returns 1 if $x \geq a$ and 0 otherwise. After experiments, we find that there's no noticeable difference among the three implementations. We finally choose "if else" because of its readability.

The normalization factors are not precomputed in this feature extraction module but calculated on-the-fly in the classification module. This is because our experiments show that the cost of extra computation for normalization factors, *i.e.* 8 vector-multiplication and additions for a 2×2 block, is actually less than the cost of precomputing and caching the values.

5.3. Classification

For classification a rendering from the $\frac{2W}{r} \times \frac{H}{r}$ feature array (texture) to the $\frac{W}{r} \times \frac{H}{r}$ texture as the result array is executed. For each target pixel, a dot-product between the histogram values and the weight vector is computed, with normalization of the histograms applied on-the-fly. An extra texture storing the SVM weight vector is also used as a input of the fragment program. This extra texture needs to be downloaded onto GPU only once during the initialization of the detector. The rendering procedure is described as Fragment Program 2.

- Let p_0 be the target coordinate, I be the input texture, w be the weight texture
- Let $f = (0, 0, 0, 0)$
- For $p \in [0, w/r - c] \times [0, h/r - c]$
  - Let $p' = p * (2, 1)$, $s = (0, 0, 0, 0)$,$n = (0, 0, 0, 0)$
  - For $q \in [0, 2c - 1] \times [0, c - 1]$

    $s = s + I(p_0 + p' + q) * w(cp' + q)$

    $n = n + (I(p_0 + p' + q))^2$
  - $f = f + s/\sqrt{n_0 + n_1 + n_2 + n_3}$
- return $f_0 + f_1 + f_2 + f_3 - b$

Fragment Program 2:Classification.

Because we no longer need to compute the histograms for every scan window, the computational cost is reduced greatly. The number of combined data access operations for the feature extraction and classification steps is equal to

$$\frac{W}{r}\frac{H}{r}r^2 + \frac{WH}{d^2} \cdot 2L = WH(wh \cdot \frac{16c^2}{r^2d^2} + 1) \quad (5)$$

The ratio to (4) is

$$\frac{16c^2wh + r^2d^2}{(36 + 24c^2)wh + 16r^2d^2}$$

which is in favor of our approach under any choice of parameters (For $r = d = 8$ and $c = 2$, the ratio is 0.46). It means that even without the help of GPU, our implementation is twice as fast as the HOG-SVM detector using integral images.

5.4. Reduce and Read Back

For every image in the scale-pyramid, the feature extraction and classification module are executed with one result array (texture) as output. If the result array is read back to main memory every time it's available, the data transfer will become expensive and also block the rendering pipeline. Therefore, we choose to combine multiple result arrays into one texture inside the GPU, and only read back the combined texture after the entire scale-pyramid has been processed.

To do this, we simply copy the result array into different regions of the combined texture. The size of combined texture is $W \times H$, while the size of the result array is $\frac{W}{r} \times \frac{H}{r}$. Therefore, at most 64 level of scale-pyramid can be reduced for our choice of $r = 8$, which is usually sufficient. The readback again is by the fast data transfer technique.

6. Evaluations

We evaluate both the detection performance and speed performance of our detector.

To evaluate the detection performance, we use the INRIA person dataset, we use the same training and test test partition as in [4]. The parameters are $r = d = 8$, $c = 2$, and the scale ratio equals to 1.2. The error rate and false alarm rate are shown in Figure 5. The curve of Dalal *et al.*'s is copied from their paper. The result shows that our detector has comparable performance with the original HOG-SVM detector. The comparison of detection performance is carried out without merging multiple detection windows.

The test images in INRIA dataset are of several different sizes, which makes it not suitable for a speed test of our GPGPU-based detector, as the size of the GPU textures needs to be fixed during runtime. Instead, we evaluate the execution times on the CAVIAR dataset[2], where images

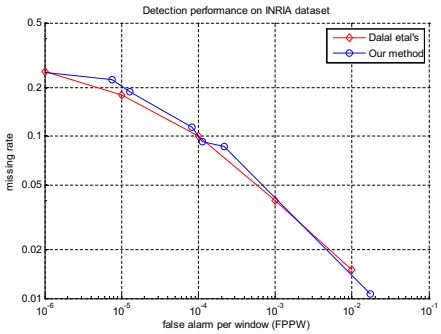

Figure 5. Performance comparison of detectors on INRIA dataset

are sampled from video sequences, which are all of the same size (384×288). We randomly selected 200 images from this dataset for testing. The detectors are still trained with INRIA person data. The graphics card we use is an NVIDIA Quadro FX 3500.

In this evaluation, we tested three different implementations: the CPU-based detector with integral images (CPU-II), the CPU-based detector with histograms precomputing and caching (CPU-HC), and our GPGPU-based detector (GPGPU). The overall execution times as well as those of different modules are listed in Table 1. Since the GPU rendering is non-block operations, we use glFinish() functions to force pipelines to finish rendering when measuring the execution times of individual modules in GPGPU-based detector. That is why the sum of time over all the modules of GPGPU is larger than its overall time.

The result shows that our GPGPU-based method significantly outperforms the other two methods on both scale ratios. Even with scale ratio equal to 1.05 (equivalent to 5637 window scans per image), our detector can still process at a speed of 73 ms per image (*i.e.* 13.6 frames per second), which is more-than-ten-times faster compared to the CPU-II detector whose speed is 1456 ms per image. The major speed boost-up comes from the feature extraction part, which has a 20x improvement compared to CPU-II. The feature precomputation and caching is more efficient than integral images even on the CPU-based version, which is consistent with our analysis in Section 5. Some detection results on CAVIAR dataset are shown in Figure 6. To merge overlapping detection results we use a greedy algorithm that iteratively combines two windows that are closest to each other, of which the running time can be neglected when the total number of detection windows is small. This merging procedure is done on CPU.

7. Conclusion

We described a fast pedestrian detector based on GPGPU. A more-than-ten-times speed up is achieved compared to the original HOG-SVM detector. Our implementation uses the HOG feature and linear SVM classifier. It is not difficult to generalize to more complicated feature such as shaplet[11], or classification models such as AdaBoost and cascade-structured classifiers[16], which can further improve the performance. To implement classifier with more complicated structure such as a cascade, data gather and scatter procedure need to be refined in order to handle branching. To the best of our knowledge, our work has revealed a new application of GPGPU in computer vision. Other specific object detection tasks that use similar CPU-based scan-window approaches, such as face detection and car detection, are also likely to benefit from GPGPU.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide*. 2

[2] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1. 5

[3] P. Babenko and M. Shah. MinGPU: A minimum GPU library for computer vision. *Technical report*. 2

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *CVPR*, 2005. 1, 2, 3, 5

[5] J. Fung. *GPU Gems 2*, chapter Computer Vision on the GPU. Addison Wesley, 2005. 2

[6] J. Fung and S. Mann. Openvidia: parallel GPU computer vision. *ACM MULTIMEDIA*, 2005. 2, 5

[7] S. Heymann, K. Muller, A. Smolic, B. Frohlich, and T. Wiegand. SIFT implementation and optimization for general-purpose GPU. *ICCG*, 2007. 2

[8] P. Labatut, R. Keriven, and J.-P. Pons. Fast level set multi-view stereo on graphics hardware. *3DPVT*, 2006. 2

[9] J. Ohmer, F. Maire, and B. R. Implementation of kernel methods on the GPU. *DICTA*, 2005. 2

[10] J. D. Owens, D. Luebke, N. Govindraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics, STAR report*, 2005. 2

[11] P. Sabzmeydani and G. Mori. Detecting pedestrians by learning shapelet features. *CVPR*, 2007. 1, 6

[12] O. Tuzel, F. Porikli, and P. Meer. Human detection via classification on riemannian manifolds. *CVPR*, 2007. 1

[13] P. Viola, M. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. *ICCV*, 2003. 1

[14] B. Wu and R. Nevatia. Detection and tracking of multiple, partially occluded humans by bayesian combination of edgelet based part detectors. *IJCV*, 2007. 1

[15] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. *CVPR*, 2003. 2

[16] Q. Zhu, S. Avidan, M. Yeh, and K. Cheng. Fast human detection using a cascade of histograms of oriented gradients. *CVPR*, 2006. 1, 3, 6

Figure 6. Detection results on CAVIAR dataset

scale ratio	methods	overall	scale	feature	class.	reduce	data transfer
1.2	CPU-II	432	3	242	187	N/A	N/A
	CPU-HC	186	2	96	88	N/A	N/A
	GPGPU	16	2	5	9	1	4
1.05	CPU-II	1456	10	844	602	N/A	N/A
	CPU-HC	624	10	344	270	N/A	N/A
	GPGPU	73	4	26	45	5	4

Table 1. Execution time per image(milliseconds)