HOG-SVM car detection on an embedded GPU

Nawaz, S.

Award date:
2015

HOG-SVM Car Detection on an Embedded GPU

Shah Nawaz
SPS/VCA Research Group, Department of Electrical Engineering
Technical University of Eindhoven
Eindhoven, Netherlands
<s.nawaz.1@student.tue.nl>

***Abstract*—The focus of this paper is on a vision-based car detection pipeline for a Cooperative Cruise Control system, based on Histograms of Oriented Gradients and Support Vector Machines. The pipeline is implemented on an automotive-grade embedded computation platform which features a Tegra K1 System on Chip, enclosing also a Graphics Processing Unit. The system is evaluated using the KITTI Vision Benchmark Suite, a standard in computer vision research. The results show that the pipeline reaches accuracies up to 65.5%, which is acceptable given the limited training dataset size of 1,174 samples and that application context has not yet been exploited.**

Keywords— Car Detection, HOG, Linear SVM, GPU.

I. INTRODUCTION

A major automotive innovation in the coming decades will come from intelligent communication, sensing and perception technologies that enable cooperative automated driving. The potential benefit of cooperative automated driving is that all vehicles jointly optimize their actions in order to improve traffic efficiency and safety. The transition from current-day driving to cooperative automated driving, has already started with Advanced Driver Assistance Systems (ADAS), such as Adaptive Cruise Control (ACC) and lane keeping assist. Existing autonomous driving technologies [1] [2] [3] tend to depend on active sensors such as LIDAR [4] [5] along with accurate pre-configured maps. However, the use of passive camera sensors is becoming more practical due to recent advances in machine learning-based computer vision algorithms [6], [7]. Vision-based object detection algorithms can be applied for various object ranges, orientations, and lighting conditions. The drawback of these algorithms is that they are computationally intensive, which makes real-time deployment in cooperative automated driving challenging.

Recent innovations in commodity hardware such as Graphics Processing Units (GPUs) have advanced into an extremely powerful computational resource that can meet the computational requirements of state-of-the-art vision-based object detection applications. The GPU provides a streaming-based, data-parallel arithmetic architecture, which performs repetitive computations on an array of data. This Single Instruction Multiple Data (SIMD) capability of GPUs is suitable for computer vision tasks, which traditionally have repeated calculations operating on an entire image. In this work, we exploit the capabilities of embedded GPUs to realize a real-time vision-based car detection pipeline.

Vision-based object detection algorithms consists typically of an image *feature detector*, which provides a digital description of local image content, and a *pattern recognizer* that classifies this local image content, e.g. as a car or background. Fig. 1 shows an example detection obtained with the developed vision-based car detection pipeline.

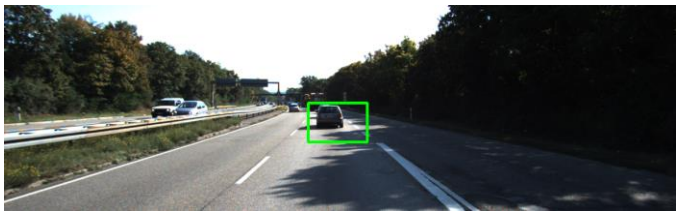

Fig. 1: *Example detection of the developed pipeline.*

There are different feature detectors, e.g. Haar-like [8], Local Binary Pattern (LBP) [14], and Histogram of Oriented Gradients [7] (HOG) which can be used to develop a car detection application. The HOG feature, proposed by Dalal and Triggs, has gained wide recognition as a successful feature-based method for object detection, especially for real-time applications. The HOG feature provides reliable high-level representations of image regions underlying many state-of-the-art object detection algorithms [9] [10] [11].

For our implementation, we have chosen to use the HOG features with linear Support Vector Machines (SVMs) as pattern classifier. A key benefit of using linear SVM, is that it is extremely efficient and therefore well-suited for real-time applications. The main focus of this work is to deploy the car detection pipeline, using existing open source libraries, on an automotive-grade embedded computation platform featuring a Tegra K1 SoC, which houses a GPU and ARM quad-cores. This work is a part of ongoing research to develop a dense multi-class detector.

Section II describes the methodology used to extract the HOG features for positive and negative data samples that are fed to linear SVM to obtain a trained model for the car detection pipeline. Section III presents an implementation of the car detection pipeline on the Tegra TK1 SoC. Section IV introduces the dataset that is used for training, testing, and evaluation of the car detection pipeline. Section V presents an overall evaluation and comparison using the publicly available KITTI Vision Benchmark Suite. It also presents a comprehensive analysis of the results. Section V concludes this paper.

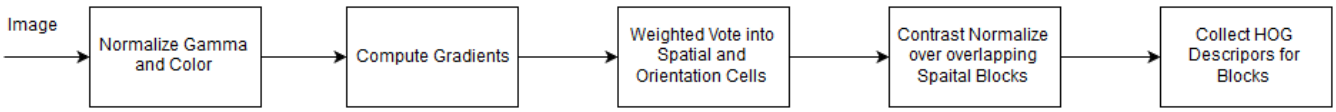

Fig. 2: *Feature detector chain.*

II. METHODOLOGY

This section provides an overview of the feature detector (HOG), the pattern recognizer (SVM), and the training pipeline used for our implementation.

A. Feature detector

The basic concept behind the HOG descriptor is that local object appearance and shape within an image, can be characterized by the distribution of intensity gradient magnitudes and directions. The HOG divides an image into small spatial connected regions, called cells, and the local 1-D histogram of gradient directions of each pixel within each cell is accumulated, according to the gradient magnitudes. These histograms are then concatenated to get the descriptors. The histograms of each cell, can be contrast-normalized for better invariance to illumination, shadowing, etc., by calculating a measure of the intensity across a larger region of the image, called blocks, and then using this value to normalize all cells within the block [7]. These steps are summarized in Fig. 2. Dalal and Triggs proposed overlapping blocks in which each cell contributes several components to the final descriptor vector, each normalized with respect to a different block. The normalized descriptor blocks are referred to as HOG descriptors.

For our car detection pipeline, we use the following HOG parameters:

- Window size is 96 × 64 pixels (Width × Height)
- Block size is 16 × 16 pixels
- Block stride is 8 × 8 pixels
- Cell size is 8 × 8 pixels
- Number of gradient bins per cell is 9

In order to be able to detect objects of different sizes, a multi-scale approach is used. For this, we rescale the original image with 12 different scale factors. For an original resolution of 1224×370 pixels, the scale factors, the multiple resolutions, and the effective window sizes, are provided in Table I. It can be observed that with our multi-scale approach we can detect objects of sizes in between 96×64 to 370×227 pixels.

B. Pattern Recognizer

A linear SVM trained on HOG features can be considered a de facto standard across many visual perception tasks [15]. The SVM algorithm fits a hyper plane in between positive and negative data samples (e.g. cars and non-cars). Fig. 3, illustrates this with + indicating positive data points and – indicating negative data points. This hyperplane is optimal in the sense that it is the maximum margin separating hyperplane between positive and negative data samples.

TABLE I. LEVEL, SCALE, RESOLUTION AND WINDOW SIZE INFOMATION

Level	Scale Stride	Resolution (WxH)	Window Size (WxH)
1	1.00	1224×370	96×64
2	1.11	1103×333	107×71
3	1.23	993×300	118×79
4	1.36	895×271	131×88
5	1.51	806×244	145×98
6	1.68	726×220	161×109
7	1.87	654×198	179×121
8	2.07	590×178	199×134
9	2.30	531×161	221×149
10	2.55	478×145	246×165
11	2.83	431×130	273×184
12	3.15	388×117	333×204
13	3.49	350×106	370×227

The hyperplane extruded with the margin, is often called the slab. The support vectors are the points which are on the boundary of the slab; these data points are closest to the separating hyperplane. Once the optimal maximum margin separating hyperplane is estimated on data samples, it can be used to classify new data samples. Each new data sample is classified on basis of on which side of the hyperplane it is. To tune the detector, one typically uses a distance threshold that offsets the hyperplane in a perpendicular direction (its normal vector).

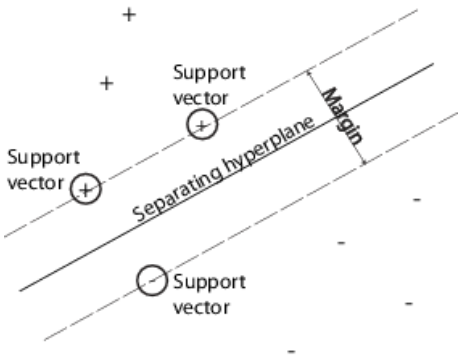

Fig. 3: *Support vector machine.*

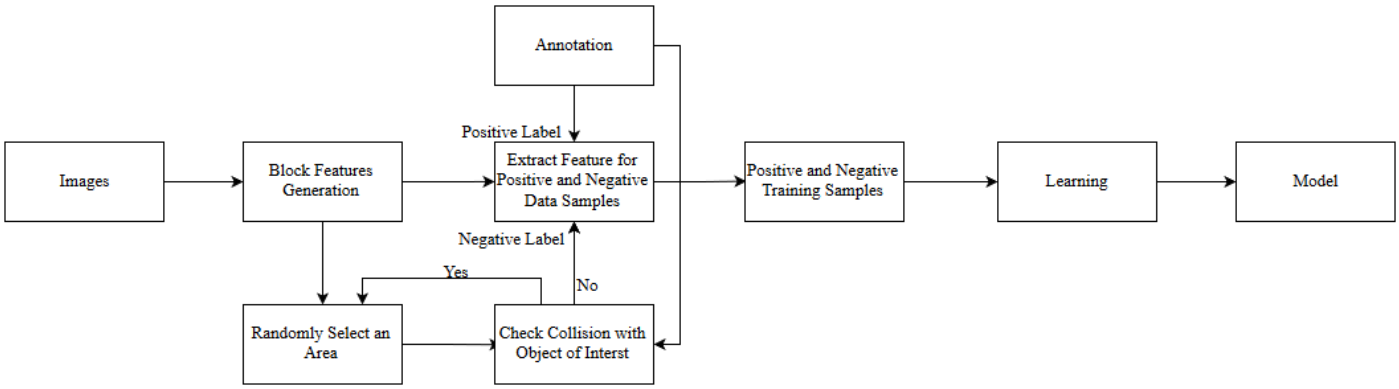

Fig. 4: *Positive and negative feature extraction chain*

C. Training Pipeline

To estimate the SVM hyperplane, positive and negative data samples have to be extracted from image data. This process is depicted in Fig. 4 and is explained in more detail below.

In order to extract positive samples, we have to use training labels. A training label minimally contains the coordinates of a rectangular region in a training image, which contains the object of interest. From these coordinates, the correct multi-scale level needs to be selected and the HOG descriptor for the particular rectangular image region needs to be extracted. To select the correct multi-scale level, we select that level for which effective window size (see last column of Table I) fully contains the rectangular region of a training label. Once we have the correct level, we can extract the HOG feature corresponding to the particular rectangular image region.

Similarly, in order to extract the HOG features that belong to negative samples, a random region is selected over the image. This selected area is then checked for collision with the object of interest using the label information of the image. In case there is a collision with the object of interest, a new random region is selected. This process is repeated until a region is found which does not collide with the object of interest. Once the randomly selected region that does not collide with the object of interest is chosen, the same block feature extraction method mentioned above for positive feature can be used in order to extract the blocks that correspond to the randomly selected region.

III. IMPLEMENTATION

The implementation of our car detection pipeline is based on the Open Source Computer Vision (OpenCV) library, and targets an embedded Linux development platform that features a Tegra K1 SoC, enclosing a GPU. The main reason to choose this embedded platform is that it provides the same architecture and advanced features as a modern desktop GPU, while still using the low power draw of a mobile chip. Furthermore, the Tegra K1 SoC will be used for many next-generation automotive systems. The Tegra TK1 development board, used to implement our car detection pipeline, is shown in the Fig. 5.

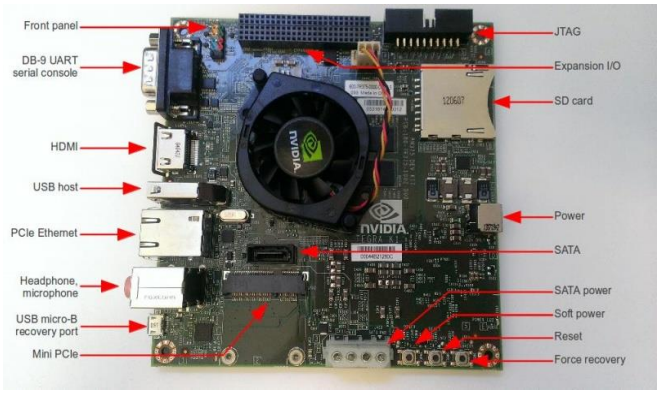

Fig. 5: *Tegra TK1 board (13×13 cm).* [12]

The Tegra K1 SoC has a Kepler GPU, which is clocked at 900 MHz and has 192 Compute Unified Device Architecture (CUDA) cores. Furthermore, it has 4 ARM cores that support SIMD operations (Neon).

The GPU can be programmed with the CUDA programming language, which is also used by OpenCV. The current version of OpenCV has a preconfigured HOG-SVM people detector. This we used as the basis for our car detection pipeline, see Fig. 6. We added new functionality to OpenCV that allows exporting HOG blocks from the GPU. This can be used to train general detectors other than people detectors. We also added functionality that allows importing pre-trained SVM models to the GPU. With this added functionality, we have a full-feature general training and detecting pipeline. In this work, it is used to detect cars but it can also be used for any other object.

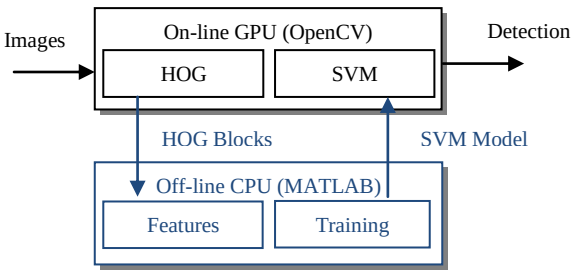

Fig. 6: *Detector training pipeline. (Everything in black is available in standard OpenCV, everything in blue is functionality that we added)*

IV. Dataset

The dataset used for training the car detection pipeline and for evaluating it, originates from the KITTI Vision Benchmark Suite [11]. This benchmark consists of 7,481 training images and 7,518 test images. Only for the training images, the ground truth labels are made publically available, giving a total of 80,256 labeled objects. Out of all label objects, which are cars, vans, pedestrians, etc., there are 1,174 labels related to rear-view of cars. In this work, we limit our car detection pipeline to detect rear-view of cars only. To train and evaluate our pipeline, we split the 1,174 rear-view car labels into two equal size sets, one set is used for training and the other set is used for evaluation. In the remainder of this section, we provide the details on how we use the KITTI Vision Benchmark Suite to obtain our positive and negative data samples.

For each label, there are 15 variables that are provided in Table II.

TABLE II. LABEL INFORMATION

Values	Name	Description
1	Type	Describes the type of object: 'Car', 'Van', 'Truck', 'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram', 'Misc' or 'DontCare'
1	Truncated	Float from 0 (non-truncated) to 1 (truncated), where truncated refers to the object leaving image boundaries
1	Occluded	Integer (0,1,2,3) indicating occlusion state: 0 = fully visible 1 = partly occluded 2 = largely occluded 3 = unknown
1	Alpha	Observation angle of object, ranging [-pi..pi]
4	Bbox	2D bounding box of object in the image (0-based index): contains left, top, right, bottom pixel coordinates
3	Dimensions	3D object dimensions: height, width, length (in meters)
3	Location	3D object location x, y, z in camera coordinates (in meters)
1	Rotation_y	Rotation ry around Y-axis in camera coordinates [-pi..pi]
1	Score	Only for results: Float, indicating confidence in detection, needed for p/r curves, higher is better

To obtain positive data samples, we use the Type variable, to select all Car labels. Next, we use the Truncated and Occluded variables, to only select Car labels which are fully visible and are truncated less than 15%. We use the Alpha variable, to select the rear view Car labels. Finally, we use 2-D bounding Bbox variable, to select Car labels whose height is larger than 40 pixels. This specific selection puts our evaluation in the Easy class, according to the KITTI guidelines that are provided in Table III.

TABLE III. DIFFICULT LEVEL DESCRIPTION

Sr. No.	Level	Description
1	Easy	These were separated for Easy Training, which is done by using the observation angle information of the type 'Car' in the labels of the images. Minimum bounding box height: 40 Pixel Maximum occlusion level: Fully visible Maximum truncation: 15 %
2	Moderate	Minimum bounding box height: 25 Pixel, Maximum occlusion level: Partly occluded, Maximum truncation: 30 %
3	Hard	Minimum bounding box height: 25 Pixel Maximum occlusion level: Difficult to see Maximum truncation: 50 %

To obtained negative data samples, we randomly generate rectangular images regions. For each region we checked if it overlaps with all Car labels (i.e. all observation angles, all truncation values, all occluded values, etc. are considered). If it overlaps, the negative data sample is rejected.

V. RESULTS AND EVALUATION

The dataset used to train, test, and evaluate the car detection pipeline originate from the KITTI Vision Benchmark Suite. There are 1,174 images in the dataset and half of the images (587) are used to train the linear SVM and the other half are used for evaluation, as mentioned in Section IV.

Prior to illustrating the performance of the classifier, some terminology related to the classifier performance is introduced. True Positive (TP) is the number of positive samples that are classified correctly, True Negative (TN) is the number of negative samples that are correctly classified, False Positive (FP) is the number of negative samples misclassified as positive samples, and False Negative (FN) is the number of positive samples wrongly classified as negative samples [13]. From these performance indicators, we construct our evaluation metric, i.e. Precision and Recall. These metrics are defined as:

$$Precision = TP / (TP + FP), \quad (3)$$
$$Recall = TP / (TP + FN).$$

The parameter Precision conceptually expresses the percentage of detections referring to true objects that we are looking for (cars). Recall indicates the percentage of the total detected cars during evaluation.

The third performance metric that we used is Accuracy. It is defined as

$$Accuracy = TP + TN / (TP + FP + FN + TN) \quad (4)$$

and it is providing the percentage of correctly classified objects in the evaluation dataset. In this case, objects are both the objects of interest (cars) and objects that are not of interest (e.g. houses). It requires that objects that are not of interests (e.g. houses) are also labelled in the evaluation dataset.

The precision and recall metrics are used to generate the Precision-Recall (PR) curves. They are obtained by sweeping the distance threshold parameter of the linear SVM between -3 to 2 with steps of 0.1. We have performed the following three experiments.

1. In the first experiment, we take 587 positive training samples and 1,761 negative training samples (a ratio 1:3) and generate the PR curve using 500 positive test samples. With this experiment we give insight in the Precision Recall trade-off of the car detection pipeline.
2. In the second experiment, we vary the ratio of positive and negative training samples between 1:1 and 1:3. This shows the influence of the ratio on the number of false positives, and hence on Precision.
3. In the third experiment, we have used the ratio of 1:3 but employ different numbers of positive and negative training samples. This experiment is performed to observe when the performance saturates at an increased number of training samples.

A. Results

First we show some qualitative results in Fig. 7. These results indicate that the car detection pipeline is working as it successfully detects the object of interest (car) in the testing images. Especially the detection of the object of interest (car) in the shadow in the middle image indicates the robustness of the detection. Besides this, the detector can also handle multiple detections simultaneously, as is shown in by the bottom subfigure.

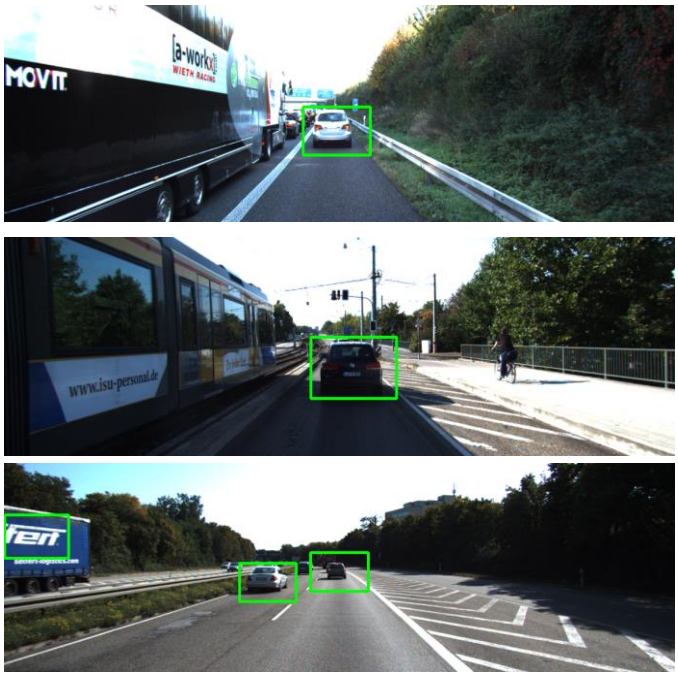

Fig. 7: *Car detection pipeline results.*

The numerical detection results of our first experiments are shown as PR curve in Fig. 8. The details of the experiment are provided by Table IV. The PR values are not as high as that of state-of-the-art vision-based object detection pipelines. However, they are acceptable, when considering that only 587 positive training samples are used and that application context information is absent.

TABLE IV. EXPERIMENT 1

Sr. No.	Positive Samples	Negative Samples	Pos/Neg. Ratio
SVM_587_1761	587	1,761	1:3

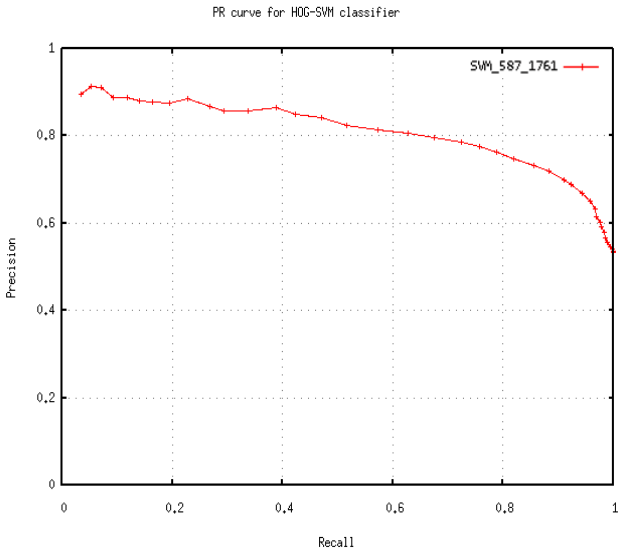

Fig. 8: *Precision Recall curve.*

The results of the second experiment are provided in Fig. 9, where the numerical details are provided in Table V. It can be observed that increasing the number of negative samples improves the Precision (as it reduces the number of false positives). The applied benefit is that one can use a higher value for the SVM distance threshold, thereby achieving a higher level of the Recall at the same level of Precision (one can detect more cars without increasing the number of wrong detections).

TABLE V. EXPERIMENT 2

Sr. No.	Positive Samples	Negative Samples	Pos/Neg. Ratio	Average Accuracy (%)
SVM_587_587	587	587	1:1	58.36
SVM_587_1761	587	1,761	1:3	65.51

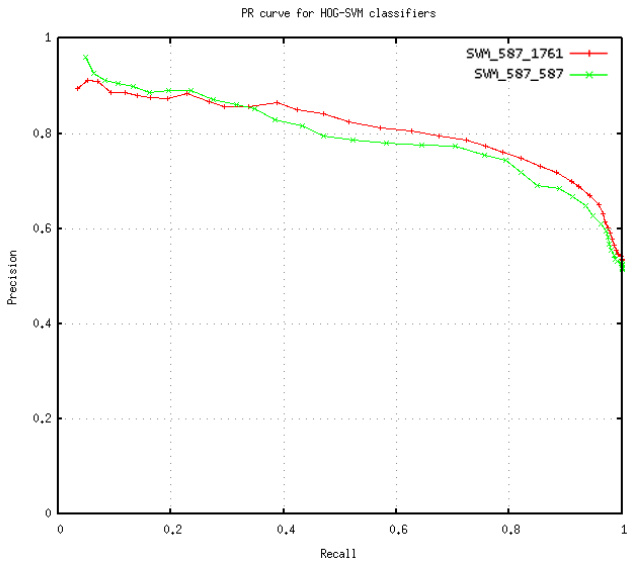

Fig. 9: *PR curves for two different pos/neg. ratios.*

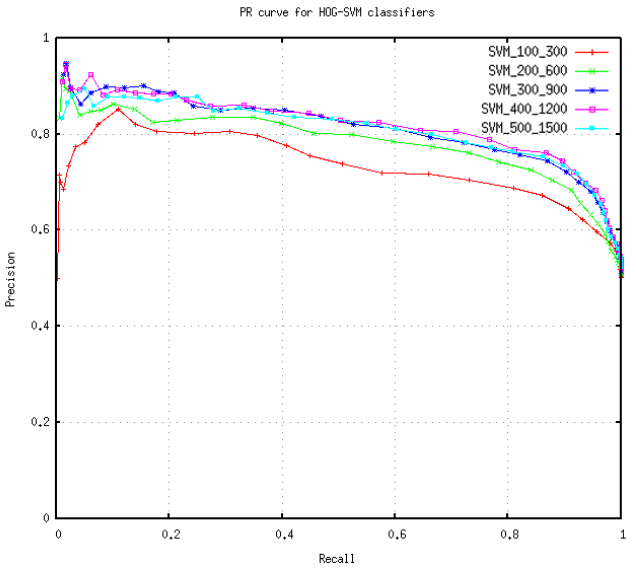

Fig. 10: *PR Curves with different number of training samples.*

The results of the third experiment are provided by Fig. 10 and the numerical details in Table VI. It can be observed that increasing the number of training samples improves the performance. Especially, the average accuracies reported in the Table VI and the last row in Table V show that the performance is not yet saturated. The improvement of using 587 positive training samples (last row Table V) over using 500 positive training samples (last row Table VI), is 1.28 % for 87 extra positive training samples. Unfortunately, we cannot increase the number of positive training samples further, as we also need test samples for our evaluation. Nevertheless, in a real application, using more training samples is recommended. After considering the experiments in more detail, we found that a number of positive data samples in the dataset are redundant, which decreases the overall quality of the positive data samples in terms of variation.

TABLE VI. EXPERIMENT 1

Sr. No.	Positive Samples	Negative Samples	Pos/Neg. Ratio	Average Accuracy (%)
SVM_100_300	100	300	1:3	57.95
SVM_200_600	200	600	1:3	60.66
SVM_300_900	300	900	1:3	63.07
SVM_400_1200	400	1,200	1:3	63.87
SVM_500_1500	500	1,500	1:3	64.23

Let us also report on the computational performance of the algorithm mapped on the proposed embedded computation platform. We present the number of frames that the pipeline can process per second, in Table VII. The throughput numbers are obtained with our GPU-based implementation and an CPU-based implementation. It can be observed that for higher resolution (1224×370 pixels), the GPU-based implementation executes a factor of 4 faster than the CPU-based implementation. The installation details of the Firewire1394 driver are provided as an Appendix at the end of this paper.

TABLE VII. FRAMES EXECUTED PER SECOND FOR GPU AND CPU

Resolution	GPU (FPS)	CPU (FPS)
1224×370	4.310	0.998
816×246	6.436	2.351
544×164	11.740	5.324

CONCLUSION

We have presented an implementation of a vision-based car detection pipeline, based on HOG and linear SVM. The pipeline is developed for a GPU of an automotive-grade embedded SoC, i.e. a Tegra K1. The results show that the pipeline reaches accuracies up to 65.51% and executes at 4 frames per second at a resolution of 1224×370 pixels. This yields a 4 times faster execution compared to a single-core ARM implementation. The accuracy can be improved by using more (non-redundant) training samples and exploiting application context information. The context helps in rejecting false positives, which in turn allows the pipeline to become more sensitive and detect more true positives. The presented pipeline can be utilized in future research to implement a dense multi-class detector.

ACKNOWLEDGMENT

This work gets help from Dr. Gijs Dubbelman, and the Video Coding and Architectures (VCA) Group in the Department of Electrical Engineering, Technical University of Eindhoven, Netherlands, with the group leader Prof. Dr. Peter de With.

REFERENCES

[1] E. Guizzo11, "How Google's self-driving car works". *IEEE Spectrum*, 2011.

[2] J. Levinson, J. Askeland, J. Becker, J. Dolson, et.al., "Towards fully autonomous driving: systems and algorithms". In *Procedding of the IEEE Intelligent Vehicles Symposium*, pages 163–168, 2011.

[3] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, et.al., "Autonomous driving in urban environments: boss and the urban challenge". *Journal of Field Robotics*, 25(8):425–466, 2008.

[4] A. Kirchner and C. Ameling, "Integrated obstacle and road tracking using a laser scanner". In *Proceeding of the IEEE Intelligent Vehicles Symposium*, pages 675–681, 2000.

[5] D. Streller, K. Furstenberg, and K. Dietmayer, "Vehicle and object models for robust tracking in traffic scenes using laser range images". *In Proceeding of the IEEE International Conference on Intelligent Transportation Systems*, pages 118–123, 2002.

[6] Viola, P., & Jones, M. J. (2004), "Robust real-time face detection". *International journal of computer vision*, 57(2), 137-154.

[7] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection". In *Proceeding of the IEEE Conference on Compute Vision and Pattern Recognition*, pages 886–893, 2005.

[8] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features". In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. CVPR 2001, vol. 1, 2001, pp. 511–518.

[9] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part based models". In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[10] P. Rybski, D. Huber, D. Morris, and R. Hoffman, "Visual classification of coarse vehicle orientation using histogram of oriented gradients features". In *Proceeding of the IEEE Intelligent Vehicles Symposium*, pages 921–928, 2010.

[11] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the KITTI vision benchmark suite". In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361.

[12] eLinux.org, "Jetson TK1", 2015. [Online]. Available: http://elinux.org/Jetson_TK1

[13] G. de Haan, "*Object detection," in Video processing for multimedia system*", Eindhoven, Netheralands, 2010 .

[14] S. Liao, X. Zhu, Z. Lei, L. Zhang, and S. Li, "*Learning multi-scale block local binary patterns for face recognition*," In *Advances in biometrics*, 2007, pp. 828–837.

[15] H. Bristow and S. Lucey, "*Why do linear SVMs trained on HOG features perform so well?*". In *arXiv preprint arXiv:*1406.2419, (2014).

APPENDIX

This section provides installation pipeline details that are needed in order to run the car detection pipeline, which includes a kernel for Firewire1394 driver support. Step 1 should be performed in order to check if the running kernel on Tegra Tk1 board provides Firewire1394 driver support. If the running kernel does not have the Firewire1394 driver support, then installation from Step 2 onwards should be followed.

Step 1: Check Firewire 1394 driver support

Use the following Linux command on Nvidia Tegra board to check if the running kernel provides Firewire 1394 driver support.

```
lsmod | grep –E -i "(1394|firewire)"
```

If there is no output, the running kernel does not have Firewire 1394 driver support. In that case follow step 2 onwards.

Step 2: Install L4T 21.3

Install Linux for Tegra 21.3 version 3.0 available on Nvidia Developer Zone.

Step 3: Extract L4T 21.3

The downloaded L4T 21.3 is extracted using the following Linux commands.

```
tar -xvf Tegra124_Linux_R21.3.0_armhf.tbz2
cd Linux_for_Tegra/rootfs
sudo tar xpf ../../Tegra_Linux_SampleRootFilesystem_R21.3.0_armhf.tbz2
```

Step 4: Apply binaries

The extracted file is run using the following Linux command.

```
sudo ./apply_binaries.sh
```

Step 5: Flash Jetson TK1Board

Tegra board should be flashed using the following Linux command.

```
sudo ./flash.sh jetsontk1 mmcblk0p1
```

Step 6: Login to Jetson TK1 board and download Grinch Kernel

After flashing the Jetson TK1 board, login to the board and download the Grinch kernel.

Step 7: Check MD5 sums

MD5 sums are checked in order to verify the MD5 hashes

```
md5sum zImage
a4a4ea10f2fe74fbb6b10eb2a3ad5409 zImage

md5sum jetsontk1grinch21.3.4modules.tar.bz2
3f84d425a13930af681cc463ad4cf3e6
jetsontk1grinch21.3.4modules.tar.bz2

md5sum jetsontk1grinch21.3.4firmware.tar.bz2
f80d37ca6ae31d03e86707ce0943eb7f
jetsontk1grinch21.3.4firmware.tar.bz2
```

Step 8: Update kernel

The kernel is updated using the following Linux commands.

```
sudo tar -C /lib/modules -vxjf
jetsontk1grinch21.3.4modules.tar.bz2
sudo tar -C /lib -vxjf jetsontk1grinch21.3.4firmwaretar.bz2
sudo cp zImage /boot/zImage
```

Step 9: Reboot the JetsonTK1

Reboot the Jetson TK1 board in order to make changes effective.