GPU Implementations of Object Detection using HOG Features and Deformable Models

Manato Hirabayashi, Shinpei Kato, Masato Edahiro
Dept. of Information Engineering
Nagoya University

Kazuya Takeda
Dept. of Media Science
Nagoya University

Taiki Kawano and Seiichi Mita
Research Center for Smart Vehicles
Toyota Technological Institute

Visione-based object detection using camera sensors is an essential piece of perception for autonomous vehicles. Various combinations of features and models can be applied to increase the quality and the speed of object detection. A well-known approach uses histograms of oriented gradients (HOG) with deformable models to detect a car in an image [14]. A major challenge of this approach can be found in computational cost introducing a real-time constraint relevant to the real world. In this paper, we present an implementation technique using graphics processing units (GPUs) to accelerate computations of scoring similarity of the input image and the pre-defined models. Our implementation considers the entire program structure as well as the specific algorithm for practical use. We apply the presented technique to the real-world vehicle detection program and demonstrate that our implementation using commodity GPUs can achieve speedups of 3x to 5x in frame-rate over sequential and multithreaded implementations using traditional CPUs.

***Index Terms*—GPGPU; Computer Vision; Object Detection**

I. Introduction

Grand challenges of cyber-physical systems (CPS) include a high computational cost of understanding the physical world. Object detection is one of compute-intensive tasks for CPS. For example, an autonomous vehicle needs to detect and track other vehicles by itself. Current autonomous driving technologies [6], [13], [23] tend to rely on active sensors such as GPS, RADAR, and LIDAR [11], [21] together with very accurate pre-configured maps, but the use of passive camera sensors is becoming more practical due to recent advances in computer vision [2]–[4]: vision-based object detection can be applied for various ranges and orientations. In particular, histograms of oriented gradients (HOG) [2] features provide reliable high-level representations of an image underlying many state-of-the-art object detection algorithms [3], [5], [20], [22], [24]. However, a major concern of HOG-based object detection remains in computational cost.

Previous work on the implementation of HOG-based object detection are limited to either hardware implementations [7], [8], [12] or specific parts of HOG algorithms [1], [19]. There is even no quantitative investigation of what implementation issues could prevent HOG-based object detection from being deployed in real-world applications. Given recent innovations in commodity hardware technology such as multicores and graphics processing units (GPUs), it is worth exploring if the current state of the arts meets computational requirements of cutting-edge object detection implementations.

Contribution: This paper presents GPU implementations of HOG-based object detection in consideration of real-world applications using deformable part models [3]. While this is a popular vision-based object detection approach, what remains an open question is a genelized programming technique and a quantification of performance characteristics for practical use. We begin with an analysis of traditional CPU implementations to find fundamental performance bottlenecks of HOG-based object detection. This analysis reasons about our approach to GPU implementations where we parallelize compute-intensive blocks of the object detection program using the GPU step by step to minimize its makespan. The experimental results obstained from a real-world car detection program using a commodity GPU show that the GPU outperforms the CPU by 3x to 5x in frame-rate, while another 5x improvement would be needed at least to deploy in the real world.

Organization: The rest of this paper is organized as follows. Section II describes the assumption behind this paper. Section III presents an analysis of HOG-based object detection and our GPU implementation technique. Section IV evaluates the performance benefit of our technique over traditional CPU implementations. This paper concludes in Section V.

II. Assumption

We consider the system composed of a multicore CPU and commodity GPU. They communicate with each other via the PCIe bus. We use CUDA [17] for GPU programming, whose development environment can be downloaded from NVIDIA's website [18]. Input images are loaded from pre-captured JPEG files, since we focus on a high computational cost of image processing. Systemized coordinations of computations and I/O devices are outside the scope of this paper. The use of multiple GPUs is also not in consideration.

We follow the object detection method presented by Felzenszwalb *et. al.* [3], where objects are represented by HOG features [2] and the detectors is composed of a "root" filter plus a set of "parts" filters that allow visual appearance to be modeled at multiple scales. This is one of the most recognized approach to object detection. See [3] for the detail.

Object detection often requires a machine learning phase to construct the object models. We assume that this learning

phase has already been done a priori and the object models are stored in the system. Particularly we restrict our attention to vehicle detection in this paper, utilizing the vehicle models provided by prior work [14]. Although these models achieve a high detection rate, the computational cost of scoring similarity of an imput image and the models using HOG features is very expensive. Specifically they include 2 root filters and 12 part filters, each of which needs to be scored against 32 resized images. The scoring could be conducted for every squire of a few pixels independently. In consequence, there are approximately 10 million computational blocks for a single high-definition image, while the frame-rate needs to meet 10∼20 frames per second (FPS) for practical use. This data-parallel compute-intensive nature of HOG-based object detection motivates the use of GPUs in this paper.

The CPU implementation of HOG-based vehicle detection has already been developed in prior work [14]. It leverages the POSIX *pthread* to parallelize the scoring per filter on a multicore CPU. While we use this multicore implementation for a performance comparison as it is, we also serialize it to execute on a single core so that we can compare our GPU implementation to two variants of the CPU implementation.

III. GPU Implementation

This paper presents GPU implementations of the existing object detection program using a popular computer vision technique [14]. Our contribution is distinguished from prior GPU implementations work [1], [19] in that we analyze the performance characteristics of the object detection program to figure out what part of the program should be accelerated using the GPU and our implementations build on this analysis optimizing performance.

Note that this paper focuses on vehicle detection but the presented GPU implementations and our technical contribution can be applied for other object detection methods using HOG features and deformable models.

A. Basic Understanding

In GPU programming, the GPU code and the input data typically need to be copied from the host to the device memory before we launch a function, *a.k.a.*, a compute kernel, on the GPU. The output data also needs to be copied back from the device to the host memory so that the CPU can read them. Hence the GPU-accelerated computation comes at the expense of the offloading overhead. Another shortcoming of the GPU is its relatively low operating frequency as compared to the CPU due to the presence of a significant number of compute cores. These trade-offs must be addressed to benefit from GPU programming; it is a complex undertaking for programmers to ascertain appropriate computational blocks that can accelerate on the GPU. Nonetheless this massively parallel computing architecture is becoming a trend in the state of the art. Given that GPUs outperform traditional multithreaded and multicore CPUs in peak performance by an order of magnitude [9], it is worth exploring a more efficient way of GPU programming. This paper provides a guideline of how to use GPUs in an efficient way for vision-based object detection.

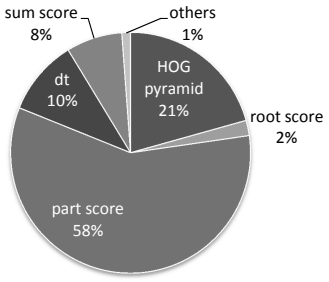

Fig. 1. The breakdown of computation times of the sequential implementation.

As aforementioned, we use CUDA [17] for GPU programming. A unit of code that is individually launched on the GPU is called a *kernel*. The kernel is composed of multiple *threads* that execute the code in parallel. A unit of threads that are co-scheduled by hardware is called a *block*, while a collection of blocks for the corresponding kernel is called a *grid*. The maximum number of threads that can be contained by an individual block is defined by the GPU architecture.

B. Program Analysis

As aforementioned, the usage of the GPU depends highly on the program structure. If the program does not contain data-parallel compute-intensive blocks, the GPU is not effective at all. Therefore it is important to analyze the program prior to coding and implementation. The following is a summary of the program sequence for HOG-based object detection using the deformable models. The detailed procedure and algorithm description are presented in [3], [14].

1) Load an input image.
2) Load the pre-defined object models.
3) Calculate HOG features for all resized variants of the input image, often referred to as a *HOG pyramid*.
4) Calculate similarity scores for every set of the root/part filters and the resized HOG images.
5) Detect an object region based on a summation of the similarity scores.
6) Recognize an object based on the detection result.
7) Output the recognition result.

In order to identify computationally time-consuming blocks of the object detection program, we conducted a preliminary measurement running the original sequential code [14] on a generic Intel Core i7 3930K CPU (3.2GHz). The measurement method is straightforward. We find high-level *for* loops (in case that the program is written in the C-like language) by scanning the program structure and make a timestamp for each loop. The result of measurement is shown in Fig. 1. Note that a label "others" represents the computation time excluding the high-level *for* loops. This breakdown of computation times provides us with a hint of how to approach GPU implementations. Specifically an obvious computational bottleneck

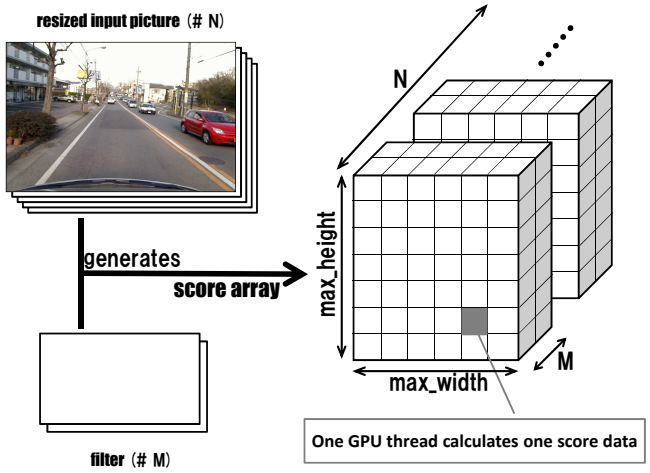

Fig. 2. The number of compute threads and their block shape.

appears in the calculation of similarity scores against the part filters, which corresponds to partly "Step 4)" in the above program sequence. This block dominates 58% of the total time. On the other hand, the calculation of HOG features corresponding "Step 3)" spends 21% of the total time, while the detection and recognition of the object, *i.e.*, "Step 5)" and "Step 6)", contribute to 10% and 8% of the total time respectively.

Our analysis conducted herein implies that even a simple time measurement highlighting only the program structure without awareness of the program context is helpful enough to understand computational bottlenecks of the program. In our case, it turned out that the most portion of the total time is spent in the *for* loops, which means that this program contains a very high degree of parallelism and could be accelerated on the GPU.

C. Implementation Approach

We parallelize all the high-level *for* loops of the program using the GPU. According to the measurement result in Fig. 1, the scoring of the root filters is a minor factor (2%) of the total time, but its program structure is almost identical to that of the part filters and they adhere to continuous blocks. Hence we include it to the GPU code. In the rest of this section, we focus on the implementation of the scoring of these filters, which is the most dominant part of the program, due to a space constraint.

Fig. 2 illustrates a conceptual structure and flow of our GPU implementation. We have 32 resized pictures to be able to detect different sizes of the object, *i.e.*, `N` is 32. We also have 2 root filters and 12 part filters to be able to detect different shapes or angles of the object, *i.e.*, `M` is 2 or 12. A minimum piece of the similarity score for a HOG representation and an object model can be calculated for each pixel of the filtered area independently. This area could be larger than a squire of 100 pixels (equivalent to `max_width` × `max_height`) for a 640 × 480 size of an image when using the trained data obtained from previous work [14]. Therefore the number of producible compute threads could be an order of millions in this setup. We shape these threads by blocks and grids for CUDA as shown in Fig. 2 where each block size is `max_width` × `max_height` × `M` and each grid size is `N`. This shape is considered for ease of programming and a different shape may further improve performance but such a fine-grained performance tuning is outside the scope of this paper.

D. GPU Programming

Once computational blocks of the program to be parallelized are determined, we can focus on the program structure rather than the context when implementing the GPU code. Listing 1 illustrates a loop structure of the procedure to score similarity of the input HOG image and the pre-defined object models. We find this structure containing fairly high parallelism, which means that the impact of GPU implementations is significant, and apply the implementation approach described in Fig. 2; the depth of the third (`C_height`) and the forth (`C_width`) loops is variable, and they could reach `max_height` and `max_width` respectively. Since these loops are independent with each other, we unroll all the loops and assign all the elements of the iteration to millions of individual threads on the GPU as shown in Fig. 2.

Listing 1. The program structure of similarity scoring

```
1  for(int level=0; level<RESIZED_INPUT_NUM; level++) {
2    for(int i=0; i<ROOTFILTER_NUM; i++) {
3      for(int j=0; j<C_height; j++) {
4        for(int k=0; k<C_width; k++) {
5          .....
6        }
7      }
8    }
9    for(int i=0; i<PARTFILTER_NUM; i++) {
10     for(int j=0; j<C_height; j++) {
11       for(int k=0; k<C_width; k++) {
12         .....
13       }
14     }
15   }
16 }
```

As aforementioned, GPU programming involes some trade-offs. It is not straightforward to address these trade-offs due to a complex architecture of the GPU. For example, parallel threads may conflict on some functional unit. Reducing the number of parallel threads mitigates this conflict but results in less parallelism. The best trade-off is obtained from optimized shapes of blocks and grids depending on the GPU architecture. We adopt comprehensive shapes of blocks and grids as presented in Fig. 2 because it simplifies programming while still providing much better performance than CPU implementations. An optimization of GPU programming is left open for future work.

Listing 2 and 3 illustrate the remainig parts of the program that we parallelize using the GPU. We also unroll all the loops of these blocks to accelerate computations on the GPU. Due

to a space constraint, we skip the details of implementation approaches.

Listing 2. The program structure of region detection

```
1  for(int level=0; level<RESIZED_INPUT_NUM; level++) {
2    for(int cmp; cmp<COMPONENT_NUM; cmp++) {
3      for(int kk=0; kk<numpart[cmp]; kk++) {
4        for(int x=0; x<dims[0]; x++) {
5          ......
6        }
7        for(int y=0; y<dims[1]; y++) {
8          ......
9        }
10       .....
11     }
12     sum_score(.....);
13   }
14 }
```

Listing 3. The program structure of HOG calculation

```
1  for(int ii=0; ii<interval; ii++) {
2    for(int x=0; x<vis_R[1]; x++) {
3      for(int y=0; y<vis_R[0]; y++) {
4        .....
5        *(Htemp+ ixp_b+iyp)+=vy1*vx1Xv; // need atomicity
6        ......
7      }
8    }
9  }
```

IV. Evaluation

We now demonstrate performance improvements brought by our GPU implementations using the existing vehicle detection program [14]. We further provide the details of performance comparisons among our GPU implementations and prior CPU implementations to discuss fundamental factors that allowed the GPU to outperform the CPU.

A. Experimental Setup

We prepare three variants of the vehicle detection program implemented using (i) a single core of the multicore CPU, (ii) multiple cores of the multicore CPU, (iii) and massively parallel compute cores of the GPU. The CPU implementations use the Intel Core i7 3930K series while we provide several varied GPUs for the GPU implementations: namely NVIDIA GeForce GTX 560 Ti, GTX 580, GTX 680, TITAN, and K20Xm. The same set of 10 images as previous work [14] is used as input data and their average computation time is considered as a major performance metrics. Note that this computation time includes all relevant pieces of image processing such as image loading and output rendering in addition to the primary object detection part.

B. Experimental Results

Fig. 3 shows the computation times of all variants of the vehicle detection program configured to use the single precision for floating operations. The dimensions of input images are 640×480 pixels. "sequential" uses a single CPU core while "multicore" uses multiple CPU cores with *pthread*. Other labels except for "best" represent our GPU implementations using corresponding GPUs. "best" is the best combination

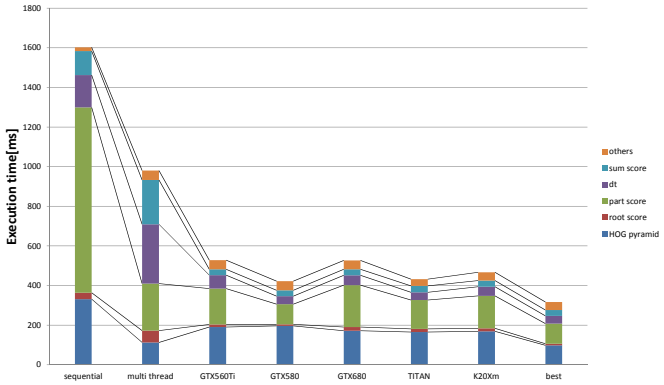

Fig. 3. computation times of the single precision floating point program.

of the GPU and CPU implementations. Most computational blocks benefit from GPUs; only the HOG calculation prefers the multicore implementation. This is attributed to the fact that the HOG calculation contains atomic operations that squeeze massively parallel threads of the GPU.

A comparison of different GPUs provides an interesting observation. The GPUs based on the state-of-the-art *Kepler* architecture [16] are inferior to those based on the previous *Fermi* architecture [15]. The Kepler GPUs employ a significant number of compute cores with the enhanced multithreading mechanism. However they operate at a lower frequency than the Fermi GPUs due to their complex architecture. Since the vehicle detection program is compute-intensive as depicted through Listing 1 to 3, the operating frequency is more dominating than the architectural benefit. This is a useful finding toward the future development of image processing with GPUs.

As a result, the best performance is obtained from such a setup that uses the multicore implementation for the HOG calculation while using the GeForce GTX 580 GPU for other computational blocks offloaded. It speeds up the execution of vehicle detection by more than 5x over the traditional single-core CPU implementation and 3x over the multicore CPU implementation respectively. A 3∼5x speed-up for the overall execution of a complex real-world application program is a significant contribution, whereas often an order-of-magnitude speed-up is reported for a particular part of the program or the algorithm.

Fig. 4 shows the computation times of all variants of the vehicle detection problem configuired to use the double precision for floating operations. Unlike the single-precision scenario, the Kepler GPUs outperform the Fermi GPUs. This explains that the double-precision performance of GPUs is improved as the generation of GPUs advances. Another notable finding is that the TITAN GPU is slightly faster than the K20Xm GPU for our vehicle detection program. Given that the TITAN GPU is a consumer price while the K20Xm is very expensive for supercomputing, we suggest that the vehicle detection program uses the TITAN GPU for a better cost performance.

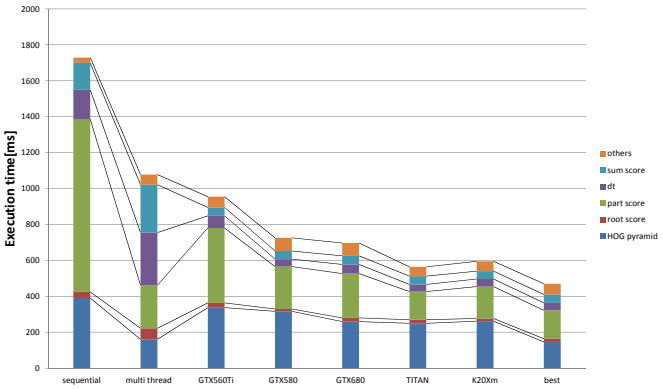

Fig. 4. computation times of the double precision floating point program.

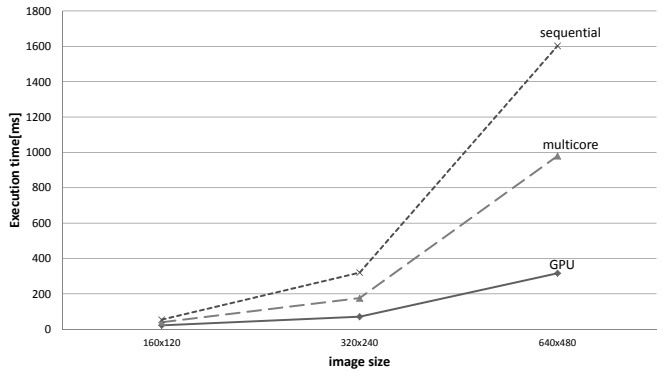

Fig. 5. Impact of the image size on computation times.

Fig. 5 shows the impact of the image size on computation times. We herein use the program configured to use the single precision for floating-point operations. The GPU implementation uses the GeForce GTX 580 GPU, which is the best performer in all the GPUs demonstrated in Fig. 3. The lessons learned from this experiment are that the execution time of the vehicle detection program is proportionally influenced by the input image size. This means that the benefit of our GPU implementations as compared to the traditional CPU implementations would hold for more high-resolution image processing.

V. Conclusion

In this paper, we have presented GPU implementations of HOG-based object detection and their performance evaluation. Unlike preceding work that highly stressed on performance improvements, our implementations are based on an analysis of performance bottlenecks posed due to an introduction of the deformable models in HOG-based object detection. This approach ensures that the GPU truly accelerates approapriate computational blocks. Our evaluation using a commodity GPU showed that our GPU implementation can speed up the existing HOG-based vehicle detection program tailored to the deformable models by 3x to 5x over traditional CPU

ADD FIGURE HERE

Fig. 6. Impact of the block and thread shapes on computation times.

ADD FIGURE HERE

Fig. 7. The breakdown of computation times of the GPU implementation.

implementations. Given that this performance improvement is obtained from the entire program runtime rather than particular algorithm parts of the program, our contribution is useful and significant for real-world applications of vision-based object detection.

To the best of our knowledge, this is the first piece of work that made a *tight* coordination of object detection and parallel computing – a core challenge of CPS. Specifically we showed that a measured and structured way of GPU programming is efficient for the object detection program and quantified the impact of GPUs in performance. Our conclusion is that GPUs are promising to meet the required performance of vision-based object detection in the real world.

In future work, we plan to complement this work with systemized coordinations of computations and I/O devices. Since real-world applications require camera sensors to obtain input images while GPUs are compute devices off the host computer, the data I/O latency could become a non-trivial bottleneck on the data bus. In this scenario, we need enhanced system support such as zero-copy approaches [10] to minimize the data latency raised between camera sensors and GPUs. We also plan to augment our GPU implementations using multiple GPUs in order to meet the real-time and real-fast requirement of real-world CPS applications.

References

[1] Y-P. Chen, S-Z. Li, and X-M. Lin. Fast HOG Feature Computation based on CUDA. In *Proc. of the IEEE International Conference on Computer Science and Automation Engineering*, pages 748–751, 2011.

[2] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. In *Proc. of the IEEE Conference on Compute Vision and Pattern Recognition*, pages 886–893, 2005.

[3] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan. Object Detection with Discriminatively Trained Part Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[4] P. Felzenszwalb and D. Huttenlocher. Pictorial Structures for Object Recognition. *International Journal of Computer Vision*, 61(1), 2005.

[5] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Proc. of the IEEE Conference on Compute Vision and Pattern Recognition*, pages 3354–3361, 2012.

[6] E. Guizzo11. How Google's Self-Driving Car Works. IEEE Spectrum, 2011.

[7] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware Architecture for HOG Feature Extraction. In *Proc. of the International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 1330–1333, 2009.

[8] F. Karakaya, H. Altun, and V. Cavuslu. Implementation of HOG algorithm for real time object recognition applications on FPGA based embedded system. In *Proc. of the IEEE Signal Processing and Communications Applications Conference*, pages 508–511, 2009.

[9] S. Kato. Implementing Open-Source CUDA Runtime. In *Proc. of the 54the Programming Symposium*, 2013.

[10] S. Kato, J. Aumiller, and S. Brandt. Zero-Copy I/O Processing for Low-Latency GPU Computing. In *Proc. of the IEEE/ACM International Conference on Cyber-Physical Systems*, 2013 (to appear).

[11] A. Kirchner and C. Ameling. Integrated Obstacle and Road Tracking using a Laser Scanner. In *Proc. of the IEEE Intelligent Vehicles Symposium*, pages 675–681, 2000.

[12] M. Komorkiewicz, M. Kluczewski, and M. Gorgon. Floating Point HOG Implementation for Real-Time Multiple Object Detection. In *Proc. of the International Conference on Field Programmable Logic and Applications*, pages 711–714, 2012.

[13] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J.Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun. Towards Fully Autonomous Driving: Systems and Algorithms. In *Proc. of the IEEE Intelligent Vehicles Symposium*, pages 163–168, 2011.

[14] H. Niknejad, T. Kawano, M. Simizu, and S. Mita. Vehicle detection using discriminatively trained part templates with variable size. In *Proc. of the IEEE Intelligent Vehicles Symposium*, pages 766–771, 2000.

[15] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[16] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[17] NVIDIA. CUDA Documents. http://docs.nvidia.com/cuda/, 2013.

[18] NVIDIA. CUDA TOOLKIT 5.0. http://developer.nvidia.com/cuda/cuda-downloads, 2013.

[19] V. Prisacariu and I. Reid. fastHOG – a Real-Time GPU Implementation of HOG. Technical Report 2310/09, Department of Engineering Science, University of Oxford, 2009.

[20] P. Rybski, D. Huber, D. Morris, and R. Hoffman. Visual Classification of Coarse Vehicle Orientation using Histogram of Oriented Gradients Features. In *Proc. of the IEEE Intelligent Vehicles Symposium*, pages 921–928, 2010.

[21] D. Streller, K. Furstenberg, and K. Dietmayer. Vehicle and Object Models for Robust Tracking in Traffic Scenes using Laser Range Images. In *Proc. of the IEEE International Conference on Intelligent Transportation Systems*, pages 118–123, 2002.

[22] F. Suard, A. Rakotomamonjy, A. Bensrhair, and A. Broggi. Pedestrian Detection using Infrared Images and Histograms of Oriented Gradients. In *Proc. of the IEEE Intelligent Vehicles Symposium*, pages 206–212, 2006.

[23] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, C. Baker, R. Bittner, T. Brown, M. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y-W. Seo, S. Singh, J. Snider, J. Struble, A. Stentz, M. Taylor, W. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[24] Q. Zhu, M-C. Yeh, K-T. Cheng, and S. Avidan. Histograms of Oriented Gradients for Human Detection. In *Proc. of the IEEE Conference on Compute Vision and Pattern Recognition*, pages 1491–1498, 2006.