GPU Implementations of Object Detection using HOG Features and Deformable Models

Manato Hirabayashi, Shinpei Kato, Masato Edahiro, and Kazuya Takeda
School of Information Science
Nagoya University

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

***Abstract*—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 [15]. 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], [14], [24] tend to rely on active sensors such as GPS, RADAR, and LIDAR [12], [22] 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], [21], [23], [25]. 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 [8], [9], [13] or specific parts of HOG algorithms [1], [20]. 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 can meet 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 that we offload only the detected compute-intensive blocks of the object detection program to the GPU. The experimental results obstained from a real-world vehicle detection program show that commodity GPUs outperform high-performance multi-core CPUs by 3x to 5x in frame-rate, though at least another 5x improvement would be needed 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 [18] for GPU programming, whose development environment can be downloaded from NVIDIA's website [19]. 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

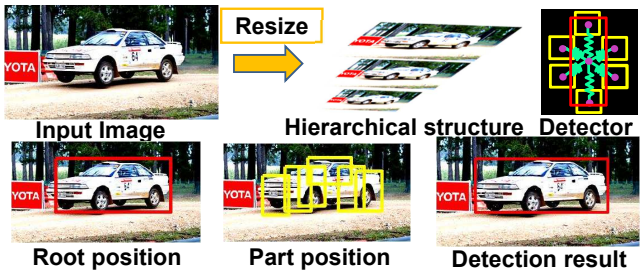

Fig. 1. Vehicle detection flow with deformable models.

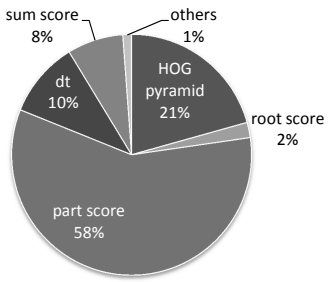

Fig. 2. The breakdown of execution times of the sequential implementation.

provided by prior work [15]. A brief concept of this approach is illustrated in Fig. 1. 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 [15]. 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 [15]. Our contribution is distinguished from prior GPU implementations work [1], [20] 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 [10], 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.

As aforementioned, we use CUDA [18] 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], [15].

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 [15] on a

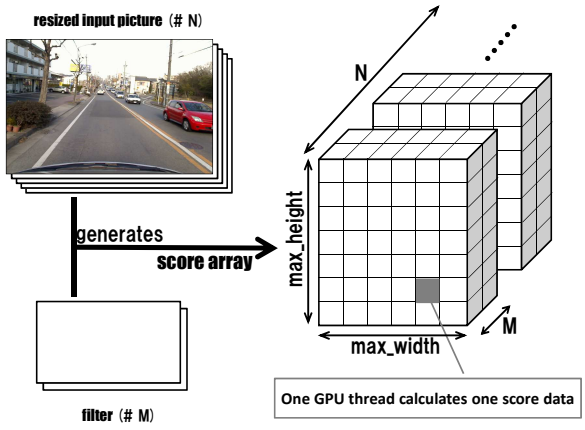

Fig. 3. The number of compute threads and their block shape.

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. 2. Note that a label "others" represents the execution time excluding the high-level *for* loops. This breakdown of execution times provides us with a hint of how to approach GPU implementations. Specifically an obvious computational bottleneck 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. 2, 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. 3 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` $\times$ `max_height`) for a 640×480 size of an image when using the trained data obtained from previous work [15]. Therefore the number of producible compute threads could be an order of millions in this setup. A conceptual structure of these threads is shown in Fig. 3. We further need to reshape this structure by blocks and grids for CUDA programming. Specifically we divide each "`max_width` $\times$ `max_height` $\times$ `M`" cube into multiple CUDA blocks, each of which is composed of $x \times y$ threads. They all fit in a single CUDA grid and there are `N` grids in total. 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. 3; 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. 3.

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. 3 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

This section demonstrates performance improvements brought by our GPU implementations for the existing vehicle detection program [15]. We also discuss the details of performance comparisons among our GPU implementations and traditional CPU implementations identifying the fundamental factors that allow GPUs to outperform CPUs.

A. Experimental Setup

We prepare three variants of the vehicle detection program implemented using (i) a single CPU core, (ii) multiple CPU cores, (iii) and massively parallel GPU compute cores. The CPU implementations use the Intel Core i7 3930K (@3.2GHz) and the Xeon E5-2643 (@3.3GHz) series while we provide several different GPUs for the GPU implementations: namely NVIDIA GeForce GTX 560 Ti, GTX 580, GTX 680, GTX TITAN, and Tesla K20Xm. The same set of 10 images as previous work [15] is used as input data and their average execution time is considered as major performance metrics. Note that this execution time includes all relevant pieces of image processing such as image loading and output rendering in addition to the primary object detection block.

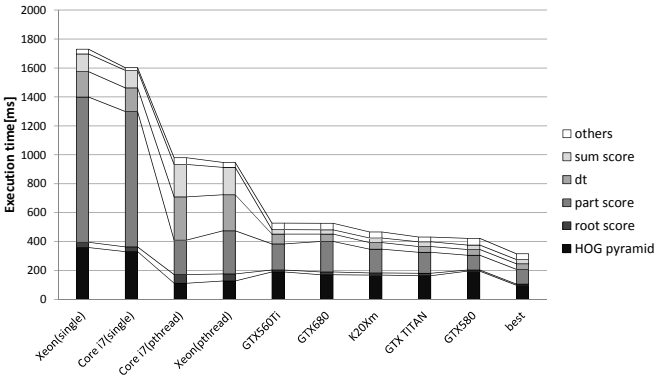

Fig. 4. Execution times of the single-precision floating-point program.

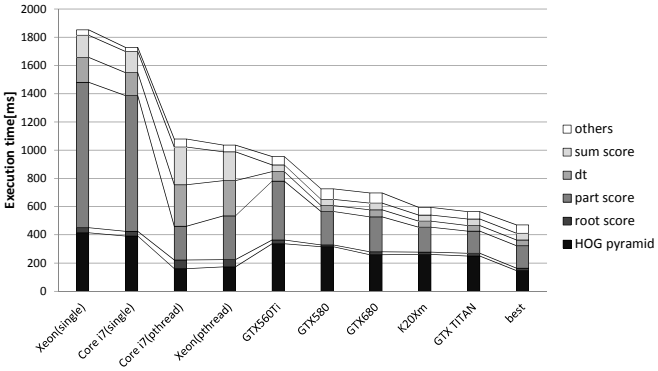

Fig. 5. Execution times of the double-precision floating-point program.

B. Experimental Results

Fig. 4 shows the execution times of all variants of the vehicle detection program configured to use single-precision floating-point operations. The dimensions of input images are 640×480 pixels. "XXX(single)" uses a single CPU core for the corresponding CPU series while "XXX(multicore)" uses multiple CPU cores with *pthread*. Other labels except for "best" represent our GPU implementations using the corresponding GPUs; "best" describes the best combination of the GPU and CPU implementations. For the GPU implementations, we shape each CUDA block by 8×8 threads. It is notable to see that most computational blocks benefit from GPUs but only the HOG calculation prefers the multicore implementation. This is attributed to the fact that the HOG calculation contains atomic operations as illustrated in Listing 3 that could squeeze massively parallel threads of the GPU.

Comparisons among the GPUs as well as those among the CPUs provide an intriguing observation. The GPUs based on the state-of-the-art *Kepler* architecture [17] are inferior to those based on the old *Fermi* architecture [16]. Albeit a significant number of compute cores with the enhanced multithreading mechanism, the Kepler GPUs operate at lower frequency than the Fermi GPUs due to their complex archi-

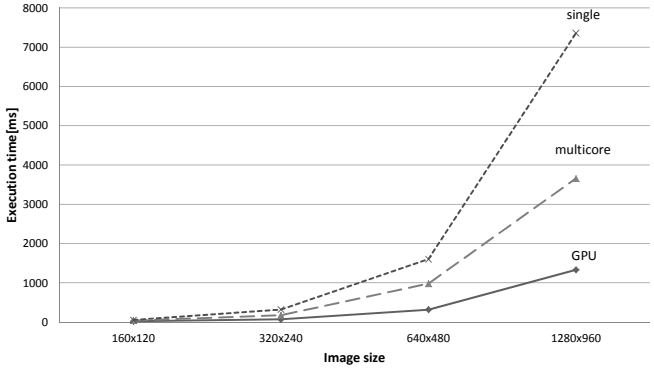

Fig. 6. Impact of the image size on execution times.

tecture. Since the vehicle detection program employs a lot of compute-intensive blocks 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 GPU-based image processing. It is also remarkable that the Core i7 CPU is slightly faster than the Xeon CPU. Since the experiment is limited to a single process, we suspect that a desktop-oriented design of the Core i7 CPU is preferred to a server-oriented design of the Xeon CPU.

As a whole, the best performance is obtained from such a setup that uses the multicore implementation for the HOG calculation while offloading other computational blocks to the GeForce GTX 580 GPU. It results in a speed-up of more than 3x to 5x for the execution of vehicle detection over the traditional single-core CPU implementation and the multicore CPU implementation respectively. This scale of speed-up for the overall execution of a complex real-world application is a significant contribution, whereas an order-of-magnitude speed-up has been reported for particular blocks of the program and the algorithm in previous work [1], [20]. Our results truly demonstrate the current performance status of state-of-the-art GPUs for practical vehicle detection.

Fig. 5 shows the execution times of all variants of the vehicle detection problem configuired to use double-precision floating-point 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. This is due to a slightly higher operating frequency of the TITAN GPU. Since the TITAN GPU is a consumer market price while the K20Xm is a very expensive supercomputing device, we suggest that the vehicle detection program uses the TITAN GPU for a better cost performance.

Henceforth we restrict our attention to the single-precision floating-point version of the vehicle detection program. Note that similar performance characteristics were also observed in the double-precision version through our experiments but they are omitted herein due to a space constraint.

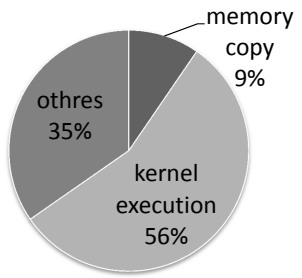

Fig. 7. The breakdown of execution times of the GPU implementation.

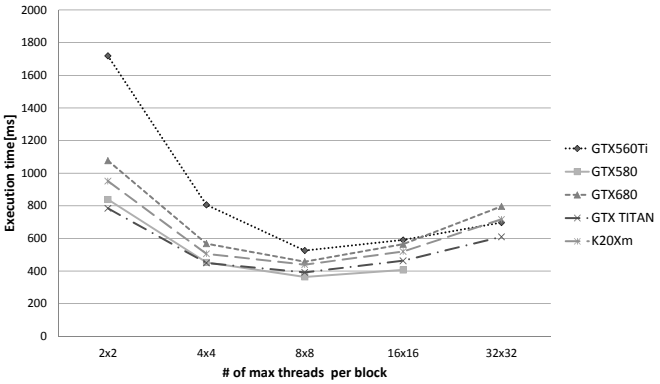

Fig. 8. Impact of the block shape on execution times.

Fig. 6 shows the impact of the image size on execution times. The GPU implementation uses the GeForce GTX 580 GPU, which is the best performer in all the GPUs demonstrated in Fig. 4. The lessons learned from this experiment are that the execution time of the vehicle detection program is proportionally influenced by the input image size. Therefore the benefit of our GPU implementations as compared to the traditional CPU implementations would hold for more high-resolution image processing.

Fig. 7 shows the breakdown of execution times of the GPU implementation that achieves the best performance for the vehicle detection program. The memory copy overhead is often claimed to be a bottleneck in GPU programming [7], but our analysis explains that it is not the case for the exhibited workload. This means that further advances of GPU technology will lead to faster implementations of the vehicle detection program, which encourages future work to use state-of-the-art GPUs.

Fig. 8 shows the impact of the block shape of GPU code on the execution times of the single-precision floating-point vehicle detection program. Particularly the number of threads in each block is varied to see how the performance is affected. Note that the Fermi GPUs cannot support 32×32 threads per block due to the hardware limitation. From what we observed in our experiment, a configuration of 8×8 threads exhibits the best performance for all the GPUs. This is somewhat an intu-

itive expectation because each warp of the GPU can contain up to 32 threads and a set of two warps is executed every two cycles according to the NVIDIA GPU architecture. Having less threads per block looses parallelism while introducing more threads could cause resource confliction within a block. Therefore a more in-depth investigation is required to truly optimize performance.

V. Conclusion

In this paper, we have presented GPU implementations of HOG-based object detection and their detailed performance evaluation. Unlike preceding work that highly stressed on performance improvements, our implementations are based on an analysis of performance bottlenecks posed by an introduction of the deformable models in HOG-based object detection. This approach ensures that the GPU truly accelerates appropriate computational blocks. Our experimental results using commodity GPUs showed that our GPU implementations can speed up the existing HOG-based vehicle detection program tailored to the deformable models by 3x to 5x over the traditional CPU implementations. Given that this performance improvement is obtained from the entire program execution rather than a particular algorithm within the program, we believe that 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, while performance optimizations remain open problems.

In future work, we plan to complement this work with systematized coordination 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 [11] to minimize the data latency raised between camera sensors and GPUs. We also plan to optimize and 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] T. Jablin, P. Prabhu, J. Jablin, N. Johnson, S. Beard, and D. August. Automatic CPU-GPU communication management and optimization. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, 2011.

[8] 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.

[9] 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.

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

[11] 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).

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. http://www.nvidia.com/, 2009.

[17] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110. http://www.nvidia.com/, 2012.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.