J Ambient Intell Human Comput (2014) 5:635–643
DOI 10.1007/s12652-013-0188-5

ORIGINAL RESEARCH

Faster and more intelligent object detection by combining OpenCL and KR

Floris De Smedt · Lars Struyf · Sander Beckers · Joost Vennekens · Gorik De Samblanx · Toon Goedemé

Received: 27 December 2012 / Accepted: 10 June 2013 / Published online: 3 July 2013

Abstract In this paper we present a fast implementation of a robust object detector by using OpenCL. The use of fast object detection is of great use for a broad range of applications in multiple domains. OpenCL allows for scalability to more performant and different types of hardware, with minimal changes to the implementation. By using a GPU as execution device, we exploit the data parallelism opportunities of the algorithm. We also discuss the use of knowledge representation as a means to integrate expert knowledge into applications. This can be used both for faster processing by limiting the searching space, and for applications to work more autonomous by exploiting a higher level of intelligence.

Keywords OpenCL · Data parallelism · Part based object detection · Knowledge representation

1 Introduction

Object detection has endless possibilities in many application areas of computer vision. For example the detection of humans can be used in surveillance applications, detection of vulnerable road users (Van Beeck et al. 2011; Cho et al. 2012), blurring of persons in mobile mapping images for privacy issues, human–robot interaction, e-health applications such as the detection of falling elderly people (Willems et al. 2009), …

It is important that the detection of the object happens as fast as possible while at the same time as accurately as possible. Many applications expect real-time performance while having a small amount of false positives. Even faster as real-time detection speeds are beneficial since this allows more processing time to be consumed by post-processing steps.

Recently, a number of state-of-the-art object detection algorithms are described in literature that have a very high recognition performance (Felzenszwalb et al. 2008, 2010a; Leibe et al. 2004; Gall et al. 2011; Dollár et al. 2009a). The downside of these powerful algorithms is that they come with a high computational cost. The algorithm we chose to implement (Felzenszwalb et al. 2010b) is a very robust algorithm based on histograms of oriented gradients proposed by Dalal and Triggs (2005). To increase performance, we implement this algorithm in OpenCL, a novel open standard for heterogeneous computing. This allows us to execute the algorithm on dedicated hardware that exploits the opportunity of data parallelism.

In Sect. 2, we give an overview of object detection and explain how the algorithm we implement works. We will also explain why a faster implementation of such a robust algorithm would be beneficial. In Sect. 3, we will discuss in detail the implementation of the construction of the feature pyramid, which is the searching space for model evaluation. In this section we handle the advantages and disadvantages of these choices and how we can circumvent the obstacles. In Sect. 4, we will discuss the experiments we have done and the timing results. In Sect. 5 we describe the

F. De Smedt (✉) · L. Struyf · S. Beckers · J. Vennekens · G. De Samblanx · T. Goedemé
Campus De Nayer, Lessius Mechelen, Association KU Leuven, Leuven, Belgium
e-mail: floris.desmedt@esat.kuleuven.be

F. De Smedt · T. Goedemé
Department of Electrical Engineering, KU Leuven, Leuven, Belgium

S. Beckers · J. Vennekens · G. De Samblanx
Department of Computing Science, KU Leuven, Leuven, Belgium

combination of knowledge representation with the object detector. Knowledge representation allows the system to directly exploit the knowledge of human experts when interpreting a scene. The knowledge representation language we use is IDP, which is a model expansion system for FO. We will describe the value of this combination and discuss the levels of integration we can distinguish. Section 7 discusses the conclusions we can make based on this implementation.

2 Object detection

The complexity of object detection is related to the appearance variation (color, pose, size, aspect ratio, …) of the objects to detect. This is an obvious fact, since the complexity in appearance, which will increase with variation, is related to the required dimensions to obtain a correct separation plane to perform classification. The art of object detection is to select features able to distinguish true from false, and additional be fast to compute. The accuracy of a detector is of great importance, since this gives a score of how reliable the results are. For some applications it is necessary to detect all object, e.g. for the detection of vulnerable road users in traffic applications. This detector setting comes in most cases with a lot of false detections, which makes an alarm unreliable. An other case is when a false detection comes with a huge cost, e.g. when a detection would result in a complete machine stop in a machine room. In this case the detection process will also miss a lot of true detections resulting in applications that can not be trust as only safety measure. An improved accuracy is beneficial for each application using object detection. A precision-recall curve, as shown in Fig. 1, visualises the accuracy performed by different detectors running over all possible thresholds and so representing all possible detector settings. The precision is defined as the fraction of the detections that is a correct detection, while the recall is defined as the fraction of objects in the image that detected.

The algorithm we use is based on Histograms of Oriented Gradients (HOG) for human detection proposed by Dalal and Triggs (2005), who claim that the use of HOG outperforms other feature sets. Based on this detector, we can distinguish two approaches to improve robustness of the HOG detector: more features or a more complex model based on the same features. Dollár et al. (2009a) did this by extending the amount of features used with color based features. This, combined with the use of integral images, as used by Viola and Jones for face detection (Viola and Jones 2001), leads to a very robust detector. This detector is later improved in speed by using approximations of feature

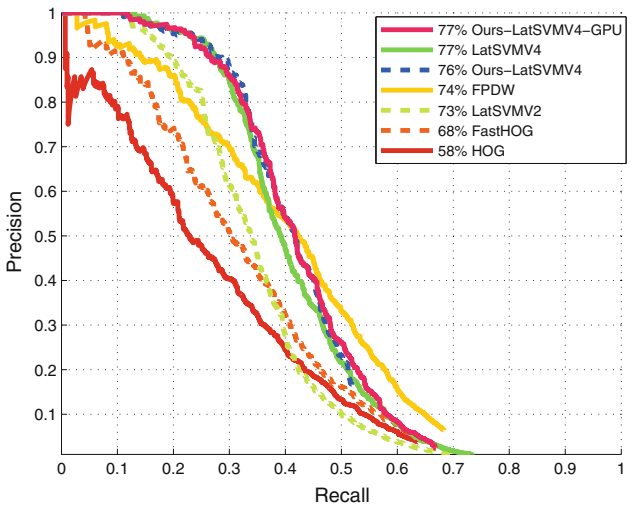

Fig. 1 Comparison of accuracy of different detectors based on the Caltech dataset (Dollár et al. 2009b)

layers to limit the required amount of computations to construct the feature pyramid (Dollár et al. 2010).

Another robustness approach is performed by Felzenszwalb et al.(2008) by using a more complex model which integrates the use of part models in a deformable configuration, representing for example the limbs of a person. Figure 1 presents the improvement in robustness of these two detectors over the default HOG implementation. We can observe the curves of the FPDW detector (which is equally accurate as the integral channel feature detector) and the part-based detector crossing each other at a certain point. This is probably due to the fact that the use of parts outperforms integral channel features when the object is large enough in the image (and hereby contains enough pixel information to distinguish the parts). Figure 1 shows our implementation compared with the state-of-the-art methods. We could also observe that the reimplementations we describe in this paper, "Ours-LatSVM4" and "Ours-LatSVM4-GPU", do not come at the cost of a decrease in accuracy.

Most fast implementations of the part-based object detector known in literature are using scene information such as ground-plane assumption (Cho et al. 2012) to limit the search space. This leads obviously to a reduction in processing time. In our implementation we obtain a speedup by using a more efficient implementation of the original algorithm, so our implementation is complementary to scene based strategies.

The improvement in accuracy over the HOG-detector comes at the cost of an increased calculation time. This is partly solved in (Felzenszwalb et al. 2010a) which proposes a cascaded implementation which uses partial hypothesis pruning. A similar approach was used by Viola and Jones (2001) where simple filters are used to prune

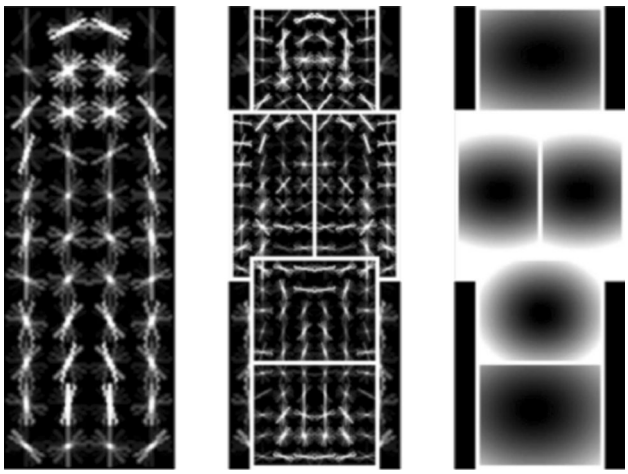

Fig. 2 Person model, from *left to right*: root model, parts model, probability of finding this part on this location on the root model

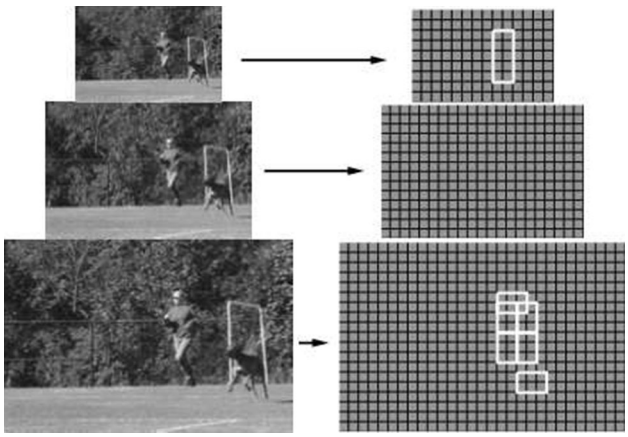

Fig. 3 Scale space pyramid and resulting feature pyramid. The root model is searched for at half the resolution as the parts

most of the search space, so more computation intensive filters are only used in detection areas with a higher probability of containing a detection. In this paper we show our work to speed up this implementation even more by using dedicated hardware (GPU) that exploits the opportunity of data parallelization.

The algorithm can be divided in two main parts:

1. The construction of the feature pyramid.
2. Model evaluation, the search for a pretrained model in the feature pyramid.

In this paper, we focus on the implementation of the first part, while in later work we will focus on speeding up the model evaluation of the detection algorithm. The calculated feature pyramid is independent of the model we are looking for. Our optimized implementation can thus be used in a detector for any arbitrary object class, as long as a pretrained model is available: pedestrians, bicycles, horses,

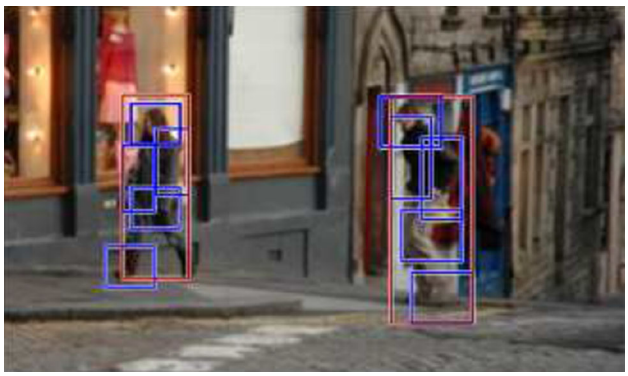

Fig. 4 Detection of pedestrians

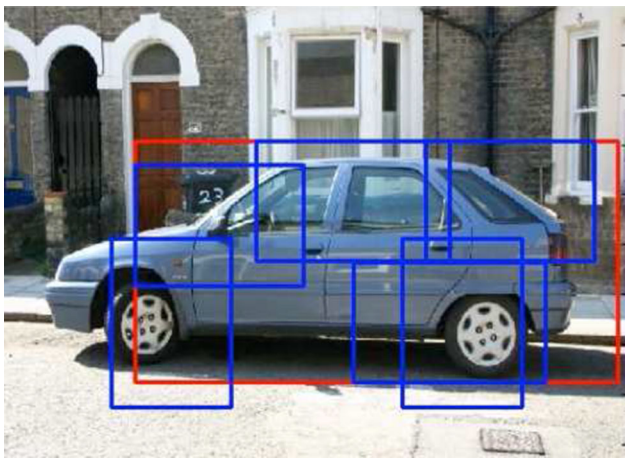

Fig. 5 Detection of a car

cars, …. In Fig. 2 the model for a person is shown. At the left we can see the root model for the object as a whole, in the middle the part models and at the right the probability of finding a part at that position on the root model. This deformation allows a flexibility in the pose of an object, which is not present in object detectors with only a single part model.

The construction of the feature pyramid can be subdivided in four stages:

1. Rescale the image to different resolutions of the same image, resulting in a scale-space pyramid. This allows to find the model on different sizes without the need to rescale the model.
2. Calculate the gradients of the pixels for each layer in the scale-space pyramid. Using the gradients creates an invariance for illumination changes.
3. Create histograms of the orientations of the gradients (HOG).
4. Use these histograms to calculate the features of each layer.

Once the feature pyramid is built, it can be used to search for a model on different scales. Each model exists of

a root model, which is used to find the object as a whole (comparable to the model used by Dalal and Triggs), and multiple part models. The part models, used to detect small parts whose position can vary with respect to the root model, are searched for at twice the resolution of the root model. The higher resolution offers more image information since more pixels of the same image area are present. This can be seen in Fig. 3. At the left the scale-space pyramid is shown, at the right we can observe the resulting feature pyramid and the layers the different parts are applied to. In Figs. 4 and 5 some detection results for the pedestrian model and the car model are shown.

3 Implementation

In this section we go deeper into our OpenCL specific implementation details of the feature pyramid. We will explain how the different parts exactly work, and point out the advantages and disadvantages of our implementation. Our implementation is based on a publicly available Matlab implementation released by Felzenszwalb et al. (2010c). We first reimplemented this algorithm in C++, which is easier to port to OpenCL since the kernels are written in a language based on C99.

3.1 OpenCL

Modern computation platforms typically include one or more CPUs, GPUs, DSPs, … All these hardware types are designed and optimized for a specific type of calculations. Optimized hardware leads to faster execution and/or less power consumption, so it is beneficial to use the most optimal hardware as much as possible. The problem is that each kind of hardware has its own instruction set, and so requires very specific programming. OpenCL, Open Computing Language (Khronos 2011), is a novel open standard for heterogeneous computing. It is a framework for writing programs that can use these platforms in a heterogenous way, in contrast to CUDA which was developed by Nvidia, specifically for its own GPU hardware. This allows to write an efficient and portable implementation of an algorithm which exploits the possibilities of each part of the algorithm on the most suitable device (multi-core CPU, GPU, cell-type architectures or other parallel processors). Since it is heterogenous, we do not have to know in advance which hardware will be used to execute the algorithm. The used platform can easily be changed by changing an initialisation variable of the program. Since different devices have different instruction sets, the compiling of the OpenCL kernels can even happen at runtime. In this article, we focus on the exploitation of data parallelism opportunities of an object detection algorithm. Since GPU hardware is optimized for data parallelism, this is the hardware we will use.

The code is written in the form of kernels. A kernel is a block of code, written in a language based on C99, that can be executed in parallel. For example, when each element of a matrix has to be multiplied by a certain value, the kernel may contain the code for one multiplication and this kernel will be executed for all elements of the matrix. The execution of the NDRange (all threads that have to execute the

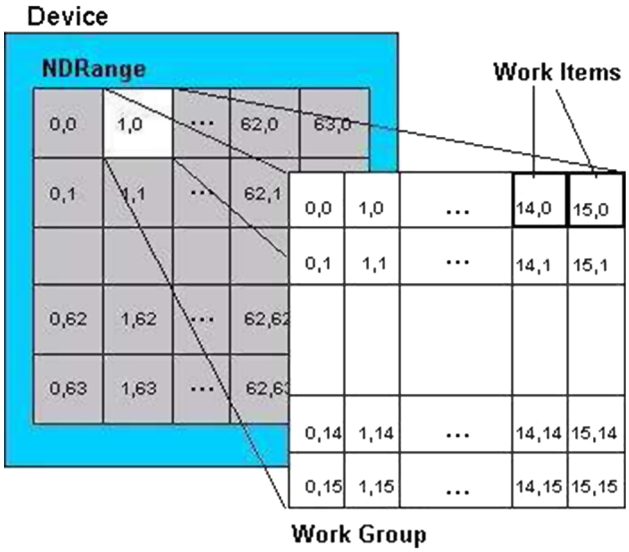

Fig. 6 The execution of the kernels is divided in workgroups, which can be subdivided in work items. Each work item executes an instance of the kernel

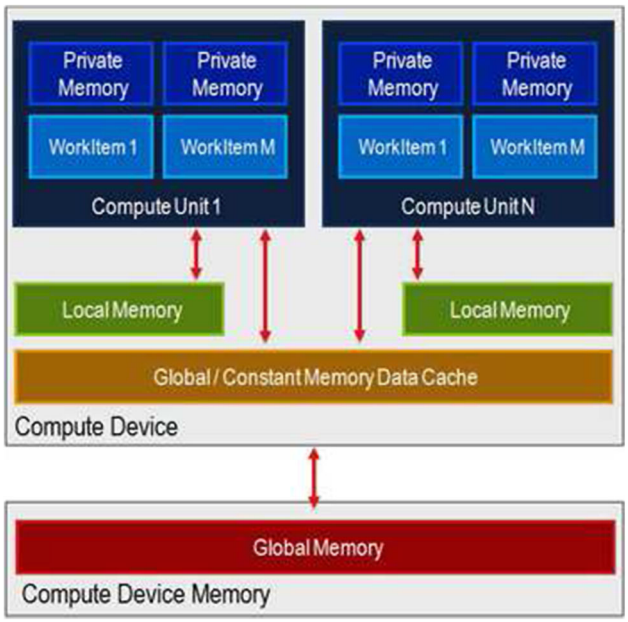

Fig. 7 Memory model of a GPU, from slow to fast: global memory, local memory, private memory

kernel code) is divided in workgroups. A workgroup is subdivided in work-items, which will execute the kernel code in parallel (Fig. 6).

To distinguish different execution threads, each thread has a unique global id, and within a workgroup each thread has a unique local id. Both are assigned for each dimension.

Figure 7 shows the memory model of a GPU device. The memory access times are going from the slowest at the bottom (starting with the memory of the host computer) to the fastest at the top (private memory). The global memory of the GPU (and CPU) is shared over all executing work items, the local memory is shared over the work items in the same workgroup and the private memory is only accessible by the running work item.

3.2 Rescale

The first step in the construction of the feature pyramid is the construction of a scale-space pyramid, which contains rescaled versions of the input image using linear interpolation. This allows to detect the model at different sizes, since for each layer the features will be computed, which will later be used for model evaluation.

We used two implementation approaches: a straightforward implementation of linear interpolation (like we did on CPU) and an implementation using texture memory. In both implementations, we launch one thread for each pixel of the destination image. This allows for maximum parallelization.

3.2.1 Implementing linear interpolation

In this implementation, we choose to launch one thread for each destination pixel. Since the rescaling consists of a vertical and horizontal rescaling, the kernel executed by each thread is split up in these two directions. At the beginning of the thread we calculate which pixels will be needed by the linear interpolation process. These pixels are then used to rescale vertically and the result is stored in private memory (registers of the GPU, memory with the best access time). In the next step, these vertically rescaled pixels are used in a horizontal rescaling step which results in the destination pixel. The calculated pixels are written to global memory as part of a layer in the scale-space pyramid.

The disadvantage is the non-linear access pattern of the needed pixels. This problem can be solved by using texture memory.

3.2.2 Using texture memory

Texture memory is a special kind of memory that can be used by GPU hardware. It is optimised for a more random access pattern and it is cached. Since GPUs are mostly used for processing image content, certain functions are very frequently used. To speed up these functions, a hardware optimised version is present on most GPUs. The use of texture memory allows the use of these functions. Since not all OpenCL-capable hardware supports the use of texture memory, it is not included in the OpenCL specifications, which makes the code only useable on a GPU platform.

3.3 Histogram

When the scale-space pyramid is built, we can create histograms for each layer based on the orientation of the gradients of the images. A pixel gradient is based on the horizontal and vertical derivative, which are obtained by subtracting two subsequent horizontal or vertical pixels. In the case of color images, only the strongest (largest) gradient of the color channels is used for the histogram. To determine the orientation of the gradient, the horizontal and vertical derivatives are multiplied with respectively the cosine and the sine of the bin orientation (the use of 18 bins results in 20° per orientation bin) and are then summed. The maximum response gives us the orientation of the gradient.

Each pixel votes in four neighbouring histograms, thereby avoiding abrupt changes as a pixel smoothly changes from one histogram to another. The influence of the votes is determined by trilinear interpolation. The construction of the well-known SIFT local feature descriptor (Lowe 2004), which is also based on HOG, uses a very similar approach. Due to this trilinear interpolation, the vote coefficients are not linear anymore, which avoid the opportunity of dividing the histograms into smaller parts to be calculated in parallel. Since each position inside the histogram block (the group of pixels voting for the same histograms) has a different coefficient, even the use of vectorisation is not possible. These restrictions make this part of the algorithm the most difficult to parallelize and even the bottleneck.

Each histogram contains the votes of a limited amount of pixels (4×4 or 8×8). When we would use a similar approach like in the rescaling part, and use one thread per voting pixel, we face the problem that multiple pixels need to have write access to the same memory addresses. We found out that the classic solution of using a semaphore to lock a memory location is very complex to implement on GPU, since the program counter of multiple threads is shared for performance reasons. Sharing of the program counter has the effect that the code for waiting on the release of the lock is shared with the thread that has the lock, so the lock is never released which results in a deadlock. An extra disadvantage of the semaphore approach is that it is against the philosophy of parallel

programming because we create a bottleneck by waiting for the release of the memory lock, which prevents gaining computation speed by parallel execution.

Our solution to this problem is to keep the four groups of histograms separately and launch one thread for each block of pixels which are voting in the same histogram (4×4 or 8×8). With this approach the kernels do not have to wait to write their result. When the four groups of histograms are filled in, they can be summed together, with respect to their disalignment, to get the final histograms of the image layer.

3.4 Features

The last step in the creation of the feature pyramid is the calculation of the features out of the HOGs. The feature pyramid has 32 layers, containing four types of features, shown in Fig. 8. Each kind of feature emphasizes a specific property that can be used to distinguish possible detections from negatives. For the calculation of the features, we chose the number of threads launched to be equal to the number of places in a feature layer. So each thread calculates 32 values.

With the techniques described above, we now have OpenCL implementations of the scale-space pyramid (Rescale), histogram and feature space computations, which are ready to be tested and compared.

4 Experimental timing results

In this section we will present the timing results from different experiments. We will begin with our reference implementation on CPU and go step by step to a total implementation of the feature pyramid in OpenCL.

4.1 Experiment specifications

All experiments are executed on the same platform, with a core i7 965 (3.2 GHz) CPU and a dedicated Nvidia GeForce GTX 295 GPU. This GPU has the possibility to be used as two parallel devices, but we only use one. We run our experiments under the linux operating system.

The experimental timing results we got are from 795 images with a resolution of 600×480 from the PETS2010 dataset (PETS 2010), which are processed three times each. For OpenCL profiling we used the visual profiler released by Nvidia, which runs seven times the different implementations over 30 images. To profile the C implementation we made use of callgrind.

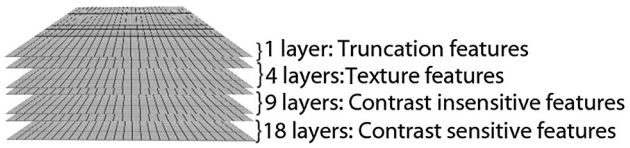

Fig. 8 The layers of the feature pyramid. Each layer emphasizes a specific type of feature

4.2 C implementation

The CPU implementation of the algorithm is used as a reference. Since OpenCL is an extension of the C programming language, using a C implementation as a starting point is of great use. In Table 1 the division of the calculation time for the CPU-implementation can be observed. The largest share is spend by the calculation of the histogram.

4.3 Rescale in OpenCL

As a first experiment, the rescaling of the images is executed on the GPU. The source image is transferred one time to the GPU and is used multiple times to be rescaled. The resulting scale-space pyramid needs to be transferred back to host memory for further processing. In Fig. 9 one can observe that the amount of time spent transferring information is large compared to the actual computation time on GPU, namely the rescaling of the images. Still we can observe a big profit in time compared to the CPU version, as can be seen in Fig. 9 which visualise the step-by-step conversion from a pure C-implementation to an OpenCL implementation. Each time an extra part of the algorithm is performed using OpenCL on GPU, while the remaining part is still executed on CPU.

4.4 Rescale and histogram in OpenCL

In this second experiment we execute two parts of the feature pyramid on GPU, namely the rescaling of the images and the HOGs from these images. After calculating the histograms based on the gradients of the images from the scale-space pyramid, these are transferred back to host memory for the calculation of the features, which is performed on CPU. In Fig. 9 we can observe that almost all computation time on CPU is consumed by the rescaling

Table 1 Distribution of calculation time on CPU

Function	Share of calculation time (%)
Transform	0.31
Rescale	20.05
Histogram	69.39
Energy	0.52
Feature calculation	9.73

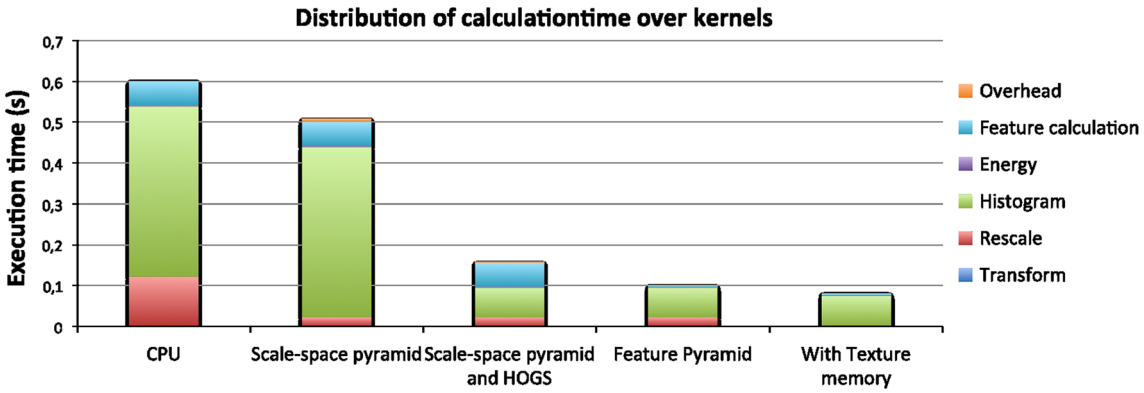

Fig. 9 Distribution of calculation time on GPU over kernels

and the calculation of the histogram. As mentioned earlier, the non-linear access pattern limits the speed of this implementation. Figure 9 shows that the implementation of these two functions result in the most time profit. This can be explained by the potential speed incrementation of Amdahl's law (S for sequential part, P for parallel part):

$$\text{Speedup} = \frac{1}{S + \frac{P}{\#cores}}.$$

We learned from Table 1 that these two functions are the most computational intensive on the CPU, so by parallelizing these functions we can gain the most overall speedup.

4.5 Total feature pyramid in OpenCL

In our final experiment we execute the total feature pyramid on GPU. The initial image is transferred to device memory and after the execution of all the kernels the total feature pyramid is transferred back to host memory. Although the share of memory transfer is large compared to other implementation (Fig. 9).

4.6 Total feature pyramid in OpenCL using texture memory

As mentioned before, using GPU as the execution platform allows the use of dedicated functions by using texture memory. In this implementation we use this texture memory to speed-up the rescaling step. As we can see in Fig. 9, the use of texture memory decreases the execution time of the rescale kernels to a minimum.

4.7 Comparison of results

In Fig. 9, a comparison of the experimental timing results is given. We can observe that the use of dedicated hardware results in a feature pyramid over six times as fast as the CPU implementation. We can also notice that the largest speed up is obtained in the parts that are most computationally intensive on CPU, namely the image rescaling and the calculation of the histograms. The speed we gain by implementing functions in OpenCL is almost directly proportional to the time needed on CPU. We obtained a frame rate of approximately 2 Hz for a complete pedestrian detection (on all scales) on the PETS dataset.

5 Combining of object detection with knowledge representation

5.1 The value of knowledge representation

For many applications, detecting objects is simply a first step towards achieving a broader understanding of a scene. One way in which this broader goal could be achieved is by adding a Knowledge representation (KR) layer to the system. KR languages allow human experts to formally write down their knowledge about a problem domain in a natural and concise way. General reasoning algorithms can then be applied to this knowledge, in order to achieve the desired behaviour of the system. Much attention in KR has gone towards the study of dynamic domains, which can be formalized in, e.g., situation or event calculus (Reiter 2001; Kowalski and Sergot 1986). Modern reasoning tools such as (Lifschitz 2002) are able to use such knowledge to efficiently calculate the effects of sequences of actions on the state of the world.

The object detector described above is ideally suited to combine with knowledge representation to interpret the scene because the information coming from the object detector is very trustworthy. The integration of computer vision and knowledge representation can be performed at different levels, as we discuss in Sect. 5.2

The language we use is IDP (Wittocx et al. 2008), which is a model expansion system for FO., which is an extension

of classical (first order) logic. The strength of IDP lies both in its rich input language and its efficiency, which makes it the perfect choice as a knowledge representation language for us.

5.2 Levels of integration

Knowledge representation can support a computer vision application on multiple levels. Each level comes with an increased level of integration complexity but also intelligence:

5.2.1 Pruning of false detections

Notwithstanding the robust algorithms we use for our object detection, false detections can still be present. By describing the scene, we can eliminate some of these false detection. For example a car can never be detected in the air, since cars can not fly. Pruning these false detections leads to better accuracy and speed.

5.2.2 Interpreting the scene for information retrieval

In this step, the description we use is more advanced. We use knowledge representation to describe the scene with the goal of retrieving information about the scene we are observing. An example of this is the detection of cars driving on unpermissible places like sidewalks, the lawn, … These will lead to other consequences than normal car behaviour. Of course the rules are made dependent on the object we are detecting (rules for cars differ from rules for pedestrians). The advantage of this integration level allows a high level of information retrieval which can be made as complex as necessary. The disadvantage of this approach is the dependency on the correct results of the object detection.

5.2.3 Cooperating with the object detector for a smarter and/or faster detection process

Until now, the application process we described can be seen as 2 phases. First object detection is performed, and its results are then used by a knowledge representation system for interpretation. However, it is also possible to integrate the two phases. The knowledge representation system creates a hypothesis about the scene based on detection results of the object detector. To substantiate this hypothesis, the system can employ the object detector, possibly with other parameters, for more information. As a simple example: when a car stops for a zebra crossing, the hypothesis of a pedestrian crossing the street is created. When there is no pedestrian detected around the zebra crossing, it is useful to employ an object detection on pedestrians with a lower threshold, since it is possible the pedestrian was not detected because it was partially occluded. The result of the detection will lead to a higher probability of the hypotheses matching the real situation, or an adjustment of the hypotheses. Although this approach is very complex to implement, a collaboration of computer vision and knowledge representation will result in far more intelligent and accurate applications as currently available in literature.

5.3 Example application

We are working on an application focussing on traffic monitoring. It observes if cars live up to the traffic regulations, which can be used to measure the safety of crossroad infrastructure. The safety of crossroad infrastructure is traditionally measured based on the number of accidents, since manually monitoring the crossroad would be far to time intensive. The knowledge representation system replaces the task of human interpretation. This enlarges the amount of information that can be used as a safety score.

5.3.1 Preliminary work

We use of knowledge representation for pruning of false detections, as mentioned in Sect. 5.2.1. For the second level of integration we divide the scene into multiple zones according to the roads, the sidewalk and bicycle path. In Fig. 10 we can see the resulting combination of the object detector, using the model for cars and bicycles, and the segmentation into zones. By defining rules using knowledge representation, we can detect a number of illegal activities based on the zone an object is found in and the action it performs. The biggest challenge of this to bring the detection accuracy as high as possible on real life data.

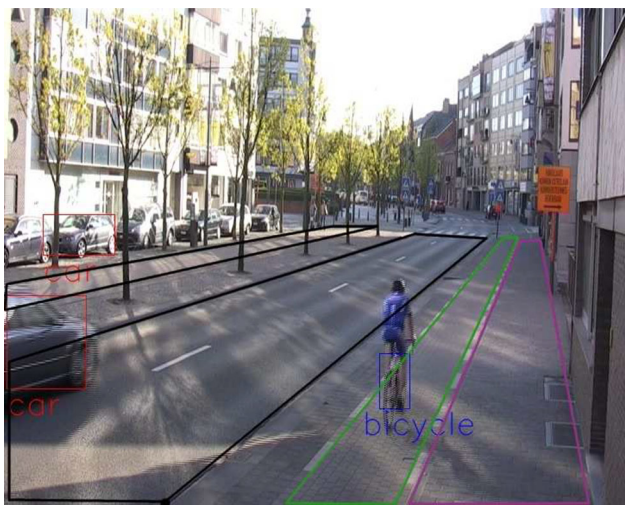

Fig. 10 The combination of the object detector and the zone-segmentation. We can distinguish the road in *black*, the bicycle path in *green* and the sidewalk in *pink* (color figure online)

6 Future work

In the future we will further reduce the detection time of the object detector. This will be done both by optimizing the model evaluation part of the algorithm and integrating scene knowledge into the application to minimize the search space.

We also plan to extend the traffic regulation system with priority rules on street crossings. The flexibility of IDP allows a fast adaption of this system to new scenes and a fast adaption of new rules.

7 Conclusion

In this paper, we presented our experiences with the implementation of an algorithm for object detection in OpenCL. We discussed the opportunities we exploited by parallelizing parts of the algorithm on GPU using OpenCL. We used a CPU implementation as a reference, and perceived a speedup from circa 0.60–0.08 s for the construction of the feature pyramid for images with a resolution of 600×480.

We also discussed the use of knowledge representation in combination with object detection. The use of knowledge representation allows us to integrate human knowledge in our applications. We defined three levels of integration, which comes with increased complexity and intelligence. We also presented the tip of the iceberg of possibilities of the combination of object detection and knowledge representation using an applications on traffic regulations.

Acknowledgments This work is supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT) via the Tetra project *S.O.S. OpenCL—Multicore cooking*.

References

Cho H, Rybski P, Bar-Hillel A, Zhang W (2012) Real-time pedestrian detection with deformable part models. In: IEEE intelligent vehicles symposium

Dalal N, Triggs B (2005) Histograms of oriented gradients for human detection. In: International conference on CVPR, vol 2, pp 886–893

Dollár P, Belongie S, Perona P (2010) The fastest pedestrian detector in the west. In: BMVC

Dollár P, Tu Z, Perona P, Belongie S (2009a) Integral channel features. In: BMVC

Dollár P, Wojek C, Schiele B, Perona P (2009b) Pedestrian detection: a benchmark. In: CVPR

Felzenszwalb P, Girschick R, McAllester D (2010a) Cascade object detection with deformable part models. In: Proceedings of the IEEE conference on CVPR

Felzenszwalb P, Girshick R, McAllester D (2010b) Discriminatively trained deformable part models, release 4. http://people.cs.uchicago.edu/~pff/latent-release4/

Felzenszwalb P, Girschick R, McAllester D, Ramanan D (2010c) Object detection with discriminatively trained part based models. IEEE Trans Pattern Anal Mach Intell 32(9):1627–1645. doi:10.1109/TPAMI.2009.167. http://dx.doi.org/10.1109/TPAMI.2009.167

Felzenszwalb P, McAllester D, Ramanan D (2008) A discriminatively trained, multiscale, deformable part model. In: Proceedings of the IEEE Conference on CVPR

Gall J, Yao A, Razavi N, Van Gool L, Lempitsky V (2011) Hough forests for object detection, tracking, and action recognition. IEEE Trans Pattern Anal Mach Intell 33(11):2188–2202. doi:10.1109/TPAMI.2011.70. http://dx.doi.org/10.1109/TPAMI.2011.70

Khronos G (2011) OpenCL—the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/

Kowalski R, Sergot M (1986) A logic-based calculus of events. New Gener Comput 4(1):67–95

Leibe B, Leonardis A, Schiele B (2004) Combined object categorization and segmentation with an implicit shape model. In: ECCV'04 workshop on statistical learning in computer vision

Lifschitz V (2002) Answer set programming and plan generation. Artif Intell 138:39–54. http://www.cs.utexas.edu/users/ai-lab/pub-view.php?PubID=924

Lowe DG (2004) Distinctive image features from scale-invariant keypoints. Int J Comput Vis 60(2):91–110. doi:10.1023/B:VISI.0000029664.99615.94. http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94

PETS (2010) Pets 2010 benchmark data. http://www.cvg.rdg.ac.uk/PETS2010/a.html

Reiter R (2001) Knowledge in action: logical foundations for describing and implementing dynamical systems. MIT Press, Cambridge

Van Beeck K, De Smedt F, Beckers S, Struyf L, Vennekens J, De Samblanx G, Goedemé T, Tuytelaars T (2011) Towards robust automatic detection of vulnerable road users: monocular pedestrian tracking from a moving vehicle. In: Proceedings of ATINER 7th annual international conference on computer science and information systems

Viola P, Jones M (2001) Rapid object detection using a boosted cascade of simple features. In: Proceedings of the IEEE conference on CVPR

Willems J, Debard G, Bonroy B, Vanrumste B, Goedeme T (2009) How to detect human fall in video? In: An overview. Positioning and context-awareness international conference, POCA

Wittocx W, Marien M, Denecker M (2008) The IDP system: a model expansion system for an extension of classical logic. In: Proceedings of the 2nd workshop on logic and search