IS THE GAME WORTH THE CANDLE?
Evaluation of OpenCL for Object Detection Algorithm Optimization

Floris De Smedt[1], Lars Stuyf[1], Sander Beckers[1], Joost Vennekens[1,2], Gorik De Samblanx[1,2] and Toon Goedemé[1,3]

[1]*Campus De Nayer, Lessius Mechelen, Association K.U. Leuven, Leuven, Belgium*

[2]*Department of Computing science, K.U. Leuven, Leuven, Belgium*

[3]*Department of electrical engineering, K.U. Leuven, Leuven, Belgium*

Keywords: OpenCL, Object Detection, HOG.

Abstract: In this paper we present out experiences with the implementation of an object detector using OpenCL. With this implementation we fullfil the need for fast and robust object detection, necessary in many applications in multiple domains (surveillance, traffic, image retrieval, ...). The algorithm lends itself to be implemented in a parallel way. We exploit this opportunity by implementing it on a GPU. For this implementation, we have choosen to use the OpenCL programming language, since this allows for scalability to more performant and different types of hardware, with minimal changes to the implementation. We will discuss how the parallelization is done, and discuss the challenges we met. We will also discuss the experimental timing results we achieved and evaluate the ease-of-use of OpenCL.

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 in blind spot cameras (Van Beeck et al., 2011), blurring of persons for privacy issues, e-health applications (detecting falling elderly people), ... but also in image retrieval applications where large databases of images are used to search for a specific class of objects.

It is important that the detection of the object happens as fast as possible. Many applications expect real time performance while having a small amount of false positives. 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), (Felzenszwalb et al., 2010a), (Leibe et al., 2004), (Gall et al., 2011). 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 (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 exploit the opportunity of parallelization. We see this implementation task as a testcase for OpenCL. In this paper we present our experiences with this process and evaluate the ease-of-use of OpenCL and the optimization results.

In section 2, we will explain how the chosen algorithm works, and which steps are taken to go from a raw image to the detection of objects. In section 3, we will discuss in detail the implementation of the construction of the feature pyramid, which is used to find objects. In this section we also handle the advantages and disadvantages of these choices and how we can circumvent the obstacles. In section 4, we will discuss the experiments we have done and the resulting timing results. In section 5, we will share our experience with the use of OpenCL. We conclude with a brief conclusion in section 6, and present possible improvements in future work.

2 OBJECT DETECTION ALGORITHM

As mentioned in the introduction, we chose to implement an algorithm proposed by Felzenszwalb

De Smedt F., Struyf L., Beckers S., Vennekens J., De Samblanx G. and Goedemé T..
IS THE GAME WORTH THE CANDLE? - Evaluation of OpenCL for Object Detection Algorithm Optimization.
DOI: 10.5220/0003821002840291
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems* (PECCS-2012), pages 284-291
ISBN: 978-989-8565-00-6

(Felzenszwalb et al., 2010b) for object detection. This algorithm is based on Histograms of Oriented Gradients (HOG) for human detection proposed by Dalal and Triggs (Dalal and Triggs, 2005), who claim that the use of HOG outperforms other feature sets. To improve the detection rate, Felzenszwalb uses parts, for example the limbs of a person, in a deformable configuration to model an object. They applied this technique with succes for multiple objects, not only humans. The improvement of robustness came at the cost of an increased calculation time, so they proposed a cascaded implementation (Felzenszwalb et al., 2010a) which uses partial hypothesis pruning. A similar approach was used by Viola and Jones (Viola and Jones, 2001) where simple filters are used to prune most of the search space, so more computation intensive filters only work on possible detection areas. In this paper we show our work in progress to speed up this implementation even more by using dedicated hardware that exploits the opportunity of parallelization.

The algorithm can be divided in two main parts:

1. The construction of the feature pyramid.
2. The search for a model in this feature pyramid.

In this paper, we focus on the implementation of the first part. So, the results we will present are independent of the model we are looking for. Our optimized implementation can thus be used in a detector for any arbitrary object class, if a trained model of it is available: pedestrians, bicycles, horses, cars, ... In figure 1 the models for a person and the front view of a car are shown. At the left we can see the root model, in the middle the parts and at the right the probability of finding the part at that position on the root model.

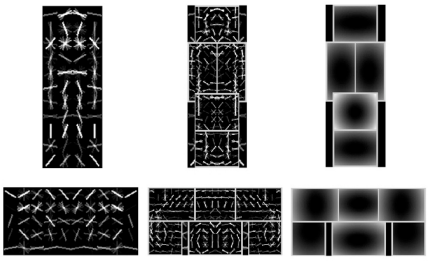

Figure 1: Person model (top) and car model (bottom). From left to right: Root model, parts model, probablity of finding this part on this spot on the root model.

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, which is very complex.
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, which are searching for small parts that can have a deformable placement respectively to the root model, are working on 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 figure 2. 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 figure 3 and figure 4 some detection results for the pedestrian model and the car model are shown.

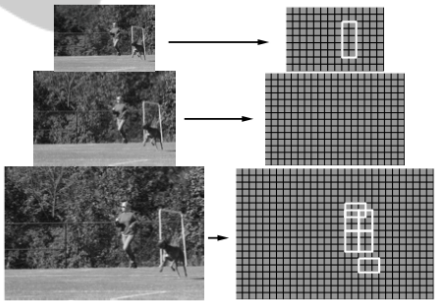

Figure 2: Scale space pyramid and resulting feature pyramid.

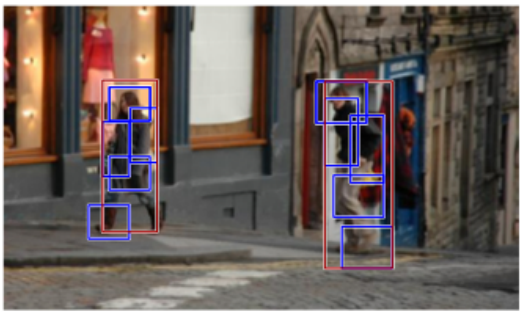

Figure 3: Detection of pedestrians.

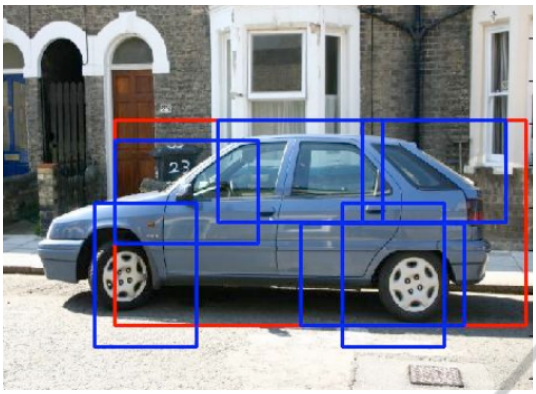

Figure 4: Detection of car using the side-view model.

3 IMPLEMENTATION

In this section we go deeper into the OpenCL specific implementation details of this feature pyramid. We will explain how the different parts exactly work, and point out the advantages and disadvantages for our implementation. Our implementation is based on a publicly available Matlab implementation released by Felzenszwalb (Felzenszwalb et al., 2010c). We reimplemented this algorithm first to C code, which is easier to port to OpenCL. The values resuling from the different steps of our own implementation are identical to these of the original implementation. This way we obtain identical detection results.

3.1 OpenCL

Modern platforms include one or more CPUs, GPUs, DSPs, ... All these hardware types are designed and optimized for a specific type of calculations. OpenCL, Open Computing Language (Group, 2011), is a novel open standard for heterogeneous computing. It is a framework for writing programs that can use these platforms in an heterogeous way, in contrast to CUDA, developed by Nvidia, for using GPU hardware. This allows us to write an efficient and portable implementation of an algorithm which exploits the possibilities for parallizing parts of the algorithm on the most suitable devices (multi-core CPU, GPU, cell-type architectures or other parallel processors). Since it is heterogenous, we don't have to know in advance which hardware will be used to execute the algorithm. The used platform can easely be changed by changing an initialisation variable of the program. Since different devices have different instruction sets, the compiling op the OpenCL kernels happens online (during the execution of the program).

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 value of a matrix has to be multiplied by a certain value, each kernel contains the code for one multiplication and this kernel will be executed for all elements of the matrix. The execution of the NDRange (all kernels that have to be executed) is subdivided in workgroups. A workgroup is subdivided in work-items, which will execute the kernels in parallel (figure 5).

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

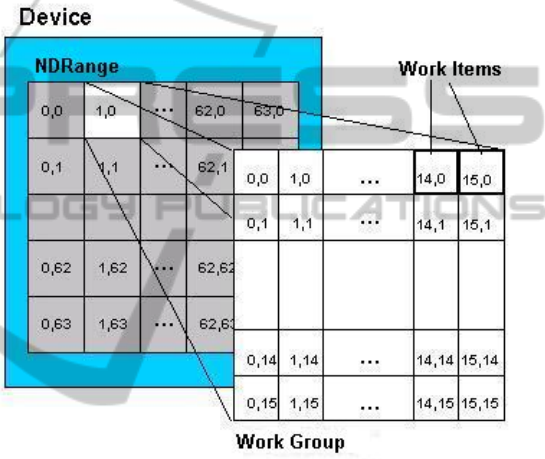

Figure 5: The execution of the kernels is divided in workgroups, which can be subdivided in work items. Each work item executes an instance of the kernel.

Figure 6 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 compute unit (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. Unlike most implementations of linear interpolation (e.g., those typically provided in hardware on GPUs), more than only the directly neighbouring pixels are used. For each power of two by which the source resolution is bigger than

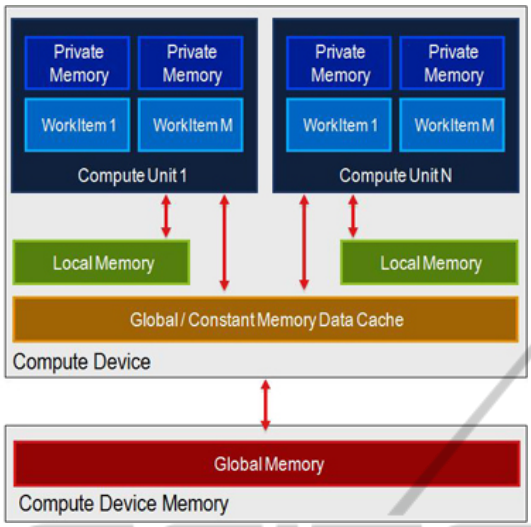

Figure 6: Memory model of a GPU, from slow to fast: Global memory, Local memory, Private memory.

the target resolution, an extra pixel is used in horizontal and vertical direction.

In our implementation, we choose to launch one thread (parallel running instance) for each destination pixel. Since the rescaling can be subdivided in 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 in vertical direction and the result is stored in a temporary pixel array 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 advantage of this method is that each pixel can be calculated independently, so we have a maximal parallel execution which can be exploited by an increase of available execution units. The disadvantage is the non-linear access pattern of the needed pixels, which makes the rescaling of images one of the most computationally intensive tasks in the creation of the feature pyramid (see section 4). This problem can probably be solved by using texture memory, since this is capable of handling more random access read- and write-patterns (see section 7).

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 these pixels. Subtracting each previous pixel from the subsequent pixel in horizontal and vertical direction yields the horizontal, respectively the vertical derivative of that pixel. The gradient is computed by making the squared sum of the derivatives. Only the strongest (largest) gradient of the three color channels is used to vote in 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 degrees 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. This trilinear interpolation makes it very difficult to split up the histograms in smaller parts, which would make parallelization easier.

Each histogram contains the votes of a limited amount of pixels (4x4 or 8x8). 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 acces to the same memory adresses. 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 time 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 (4x4 or 8x8). With this approach the kernels don't 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. Since the pixels we use in a single histogram are not aligned in memory, the same disadvantage of random read pattern arises. Therefore, also in this part of the algorithm the use of texture memory can offer additional speedup (section 7).

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 figure 7. 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. This equals the number of histograms in horizontal and vertical direction minus two.

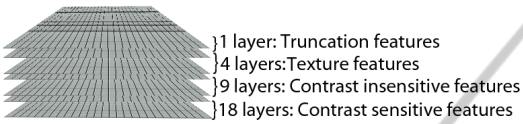

Figure 7: The layers of the feature pyramid.

The first type are the contrast sensitive features, which takes 18 layers, one orientation per layer, and uses the energy of the histograms combined with the original content of the histograms. The energy is calculated by summing the squared sum of opposite orientations of each block. For this purpose, a separate kernel is used, since the consecutive memory access is exploited this way. The energy is used as a sum over four neighbouring histogram places. Since the same groups are used multiple times, we calculate these in advance to save processing time.

To calculate the contrast-sensitive features, each orientation in the histograms is multiplied four times (four possible groups, see figure 8) with four values we calculated in advance. The feature value is half the sum of the resulting four values. Since the results of these multiplications, summed over all orientations, are used for the texture features, we store them in global memory. This results in four blocks of memory, each the size of the number of histograms.

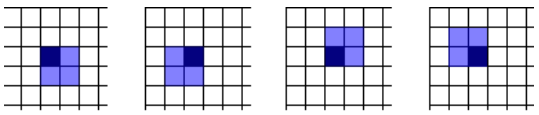

Figure 8: Four groups from energy histogram for a position from original histograms.

The next nine layers of the feature pyramid are filled with contrast insensitive features. These are calculated very similary to the contrast sensitive features, but instead of the original histogram values, the sum of opposite orientations is used.

The third type of features, texture features, is responsible for four layers in the feature pyramid. Here, the four blocks of memory we stored during the calculation of the contrast-sensitive features are used. Each created memory block results in one layer by multiplying the values with a fixed number. Also, here the consecutive memory access results in low calculation cost.

The last layer contains the truncation features, and exists of all zeros.

With the techniques described above, we have now 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 600x480 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.

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

4.2 C Implementation (Implementation A)

The CPU implementation of the algorithm is used as a reference. Since OpenCL is an extention of the C programming language, using a C implementation as

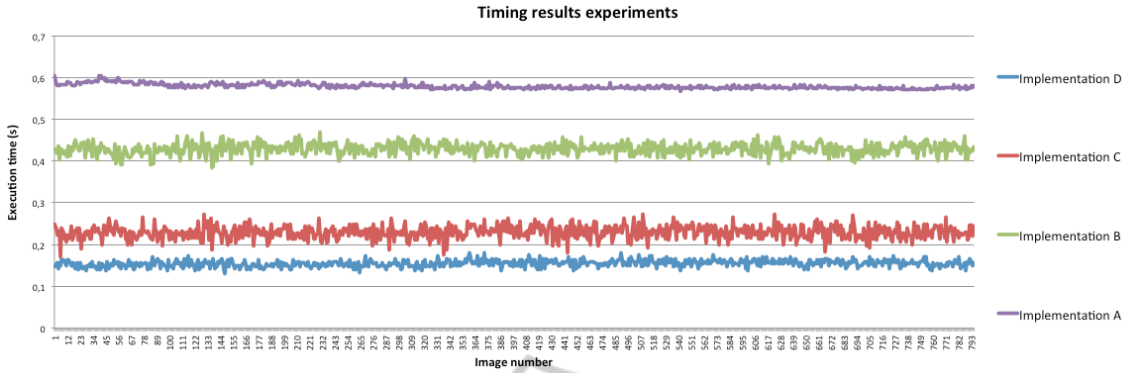

Figure 9: Comparison of the time needed to calculate the total feature pyramid of the four implementations.

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. The C-implementation is compiled with the gcc compiler using the -O3 option for optimization.

4.3 Rescale in OpenCL (Implementation B)

As a first experiment, the rescaling of the images is executed on the GPU. The source image is transfered 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 figure 10 you can observe the amount of time spent transferring information is large compared to the actual computation time. Still we can observe a big profit in time compared to the CPU version, which can be seen in figure 9.

4.4 Rescale and Histogram in OpenCL (Implementation C)

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 HOGs, these are transfered back to host memory for the calculation of the features. In figure 10 we can observe that almost all computation time is consumed by the rescaling and the calculation of the histogram. Like we mentioned in section 3.2 and section 3.3, these functions are limited by the memory access speed and non-sequential memory access pattern. Figure 9 learns 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):

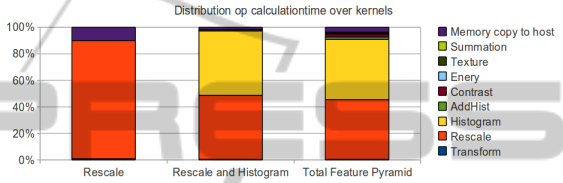

Figure 10: Distribution of calculation time on GPU over kernels.

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

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

Table 2: Distribution of calculation time on GPU for the three implementations.

Function	B (%)	C (%)	D (%)
Transform	0.823	0.431	0.576
Rescale	88.99	48.41	44.397
Histogram	/	48.28	45.55
AddHist	/	1.164	1.374
Contrast	/	/	2.48
Energy	/	/	0.443
Texture	/	/	0.443
Summation	/	/	0.266
Mem Copy	10.19	1.72	3.90

4.5 Total Feature Pyramid in OpenCL (Implementation D)

In our final experiment we execute the total feature pyramid on GPU. The initial image is tranferred to device memory and after the execution of all the kernels the total feature pyramid is transferred back to host memory. Altough the share of memory transfer is large compared to the previous implementation

(Implementation C, figure 10), the implementation is worth it. We still obtain a speed up, which can be seen in figure 9.

4.6 Comparison of Results

In figure 9, a comparison of the experimental timing results is given. We can observe that the use of dedicated hardware results in a feature pyramid four 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, namely the image rescaling and the calculation of the histogram. The speed we gain by implementing functions in OpenCL is almost directly proportional to the time needed on CPU.

5 OpenCL EVALUATION

In this section we will discuss our experiences with the use of OpenCL as a way to optimize an algorithm by running it on GPU. We will comment on the learning curve (5.1) and give tips and tricks for development (5.2) and debugging and profiling (5.3)

5.1 Learning Curve

Since it is a quite novel standard, the available literature is still growing. The specifications released by the khronos group (Group, 2011) is very valuable as a reference for function calls while developing. It does not only explain how to use the functions, but also gives possible errors and shows how to prevent them. Although it is a great help, it does not contain enough information to exploit the possibilities of OpenCL to produce the best implementation. It is necessary to know how OpenCL works, to fully exploit these oportinities. When we started using OpenCL, the learning was mostly based on examples and a trial and error-approach, which results in a longer learning curve. Now the available literature ((Tsuchiyama et al., 2009), (Benedict et al., 2011)) offers a more complete range of books which reduces the learning curve drasticaly and can also teach the reader a correct way of programming for a high performance gain.

5.2 Development

OpenCL focuses on heterogeneity. This comes at the cost of a lot of function calls to set up your execution enviroment (creating a platform, creating devices, creating a program, creating command queues, ...). This can be seen as a disadvantage, but once these functions are written, they can easely be reused in later projects without losing the flexibility it offers. This flexibility allows an easy change of execution device without modifying your kernel code.

To make optimal use of the possibilities of OpenCL, it is necessary to understand every detail of the algorithm to implement. Just copy-n-pasting existing source code to kernel code can give a speed up, but this will be small compared to an implementation which exploits the availibility of fast memory, the highest parallelization possible and sequencial memory access. The upcomming amount of (public) available libraries of optimized implementations of commonly used functions (matrix multiplication, image filtering, convolution, ...) can limit the developing time, since the developer only needs to focus on the rest of the algorithm.

5.3 Debugging and Profiling

We used a linux (Ubuntu) station, with a dedicated Nvidia GTX295 card, as a development platform. This approach is very useful, since the graphical shell of the operating system can occupy the second GPU device so that incorrect memory use on your main GPU will not result in a freeze of your graphical enviroment. Nvidia released a bundle of the newest driver, cuda and opencl drivers, and examples. These examples can be used to learn how to develop, but also as a template for your own projects.

The use of an OpenCL compatible CPU can be very useful during development, since this allows debugging, but since the architecture of CPUs and GPUs differ, some problems are very difficult to track. For example the implementation of a semaphore is quite easy on CPU, but the same code can result in a deadlock when executed on a GPU, since the program counter is shared for optimization. The difficulty of debugging is mostly due to the absence of available debugging enviroments for OpenCL on linux, which are already released for the Windows operating system.

Also for profiling the availability of profiling tools is quite sparse and mostly hardware manufacturer dependent. In this paper we used the visual profiler released by Nvidia, which can give detailed information about memory use and execution times of the kernel code. The possibility of getting timing information from within a kernel is not yet available, but could be quite useful to track down bottlenecks inside the kernels.

6 CONCLUSIONS

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 GPUs. We used a CPU implementation as a reference, and percieved a speedup from circa 0.60 seconds to 0.15 seconds for the construction of the feature pyramid for images with a resolution of 600x480. During this implementation we encountered different challenges. The most important one is the simultaneous write operation to the same memory location, which called for an approach that allows execution in a more parallel way.

We discussed also the ease-of-use of OpenCL. The flexibility comes at the cost of needing many function calls before the actual kernels can be executed, but these calls can be reused in other projects which makes the big advantage of heterogeneity and scalability outweighs the extra work. Since it is a novel standard, the available literature is limited, but still growing. Overall, we can certainly state thus that the optimization game is worth the OpenCL candle.

7 FUTURE WORK

In the future, we will extend our implementation with the use of texture memory, which allows a more random access pattern in memory. Recent preliminary experiments show we could reach an access speed of three to four times the current. We will also integrate the use of vectors, which allows an operation to be executed on multiple elements at once. This requires an additional padding of memory to be a multiple of the vector size. We will also implement the search for models in OpenCL. This part of the algorithm is mostly the execution of convolutions with the model. This overcomes the need to transfer the total feature pyramid to host memory, but only the coordinates of the detections.

ACKNOWLEDGEMENTS

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

Benedict, G. R., David, K., Perhaad, M., and Dana, S. (2011). *Heterogeneous Computing with OpenCL*. Morgan Kaupmann.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *International Conf. on CVPR*, volume 2, pages 886–893.

Felzenszwalb, P., Girschick, R., and McAllester, D. (2010a). Cascade object detection with deformable part models. In *Proc. of the IEEE Conf. on CVPR*.

Felzenszwalb, P., Girschick, R., McAllester, D., and Ramanan, D. (2010b). Object detection with discriminatively trained part based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9).

Felzenszwalb, P., McAllester, D., and Ramanan, D. (2008). A discriminatively trained, multiscale, deformable part model. In *Proc. of the IEEE Conf. on CVPR*.

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

Gall, J., Yao, A., Razavi, N., Van Gool, L., and Lempitsky, V. (2011). Hough forests for object detection, tracking, and action recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Group, K. (2011). Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

Leibe, B., Leonardis, A., and B.Schiele (2004). Combined object categorization and segmentation with an implicit shape model. In *ECCV'04 Workshop on Statistical Learning in Computer Vision*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*.

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

Tsuchiyama, R., Nakamura, T., Lizuka, T., Asahara, A., and Miki, S. (2009). *The OpenCL Programming book*. Fixstars.

Van Beeck, K., De Smedt, F., Beckers, S., Struyf, L., Vennekens, J., De Samblanx, G., Goedemé, T., and Tuytelaars, T. (2011). Towards robust automatic detection of vulnerable road users: Monocular pedestrian tracking from a moving vehicle,. In *Proc. of ATINER 7th Annual International Conf. on Computer Science and Information Systems*.

Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Proc. of the IEEE Conf. on CVPR*.