Multicore Processing for Object Recognition in Mobile Devices

Edmund Coersmeier[1], Sven Jaborek[1], Patrick Paul[1], Martin Bucker[1], Marc Hoffmann[1], Lukas Pustina[2], Simon Schwarzer[2], Felix Leder[2], Peter Martini[2]

[1] Nokia Research Center, D-44799 Bochum, Germany, firstname.lastname@nokia.com
[2] Institute of Computer Science IV, University of Bonn, D-53117 Bonn, Germany

Abstract. There exist several algorithm setups to realize object recognition systems. But actually it is a challenging task to implement these technologies for real-time applications in embedded, mobile devices. One reason for that is that the required processing power for real-time algorithms, which are required to offer a reliable system, is not available. One potential solution to this problem is the use of multi-processor platforms. Depending on the number of processors, such platforms generally offer significant more instructions per time than single processor systems. This paper investigates how to speedup a traffic sign recognition system for mobile devices. A four parallel processor ARM platform will be utilized to test the algorithm speedup in practice. Therefore, the paper gives a first basic insight to the mobile traffic sign recognition system by introducing the required algorithms and by analyzing in a first step how some of the specific algorithms behave on a multi-processor platform.

Keywords: Multi-processor platform, parallelization, image processing, object recognition

1 Introduction

Traffic signs are important instruments to keep the traffic flow up and running while volume of traffic continuously increases. Especially for car drivers it is important to remember always the actual speed limit. Therefore, it is a big help if cars are equipped with an automatic traffic sign recognition system, which always informs the driver about the actual speed limit. Besides the user interface, which plays a significant role for smooth and quick information access, it is important that the system is working in real-time. Real-time processing means in this particular case, that enough camera images per time are processed to guarantee that non of the traffic signs will be overlooked. This is only the case if the available processing power is strong enough to identify the relevant traffic signs at any speed of the car. Assuming that a system requires 10 images per second then there will be delivered at a speed of 50km/h one image every 1.38m, at a speed of 130km/h one image within every 3.6m. If the processor platform is too slow it cannot be guaranteed that every camera image can be taken into account. If a camera image is lost, then several meters distance are covered before the next image is taken into account. During that time important traffic signs can be passed by the car and object recognition system is not aware about the actual speed limit information.

Therefore, parallel processing in embedded solutions is a good approach to guarantee real-time processing. Especially for image processing applications there is potential to parallelize the algorithms. If one considers image pre-processing, pixel-based operations can be often done independently from results of other pixel operations. Different tasks like region of interest selection and final object classification, which do not work only on pixel base, have also significant potential for parallelization.

This paper introduces an algorithm setup for traffic sign object recognition and investigates selected algorithms from parallel processing perspective.

2 Algorithm Setup for Traffic Sign Object Recognition

The algorithm setup is shown in Figure 1. In the particular case 640x480 pixel images from the camera are delivered to the image pre-processing unit. Here Gauss filtering and edge detection are done.

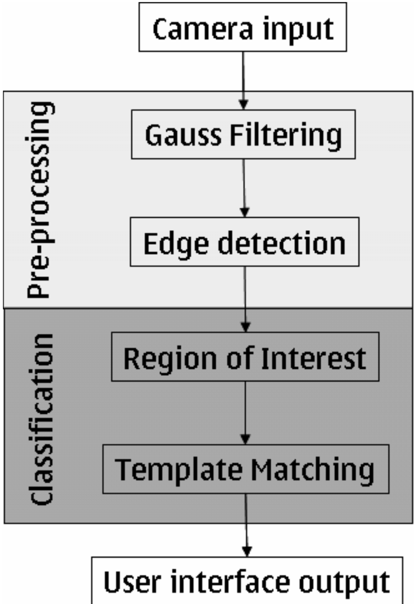

Figure 1 Algorithm setup for embedded traffic sign recognition

Object classification is done via region of interest selection and finally template matching. The identified object information can be displayed in various ways on the mobile device screen and informs the driver about the actual speed limit. Figure 2 shows an example how an original input image from camera view is looking like. There is only one target object from speed limit point of view – the 70 km/h speed limit traffic sign on the right side. The rest of the image is not important for the actual application. Thus the application needs to identify that only one relevant circle is included into this image. All the rest of the image needs to be disregarded.

Figure 2 Original image

The first step in algorithm processing from Figure 1 is the Gauss filtering, which smoothes the image from noise. Image smoothing is done on pixel base. The desired pixel is convolved with the pixels in the near neighborhood. Figure 3 shows the linear filtering process.

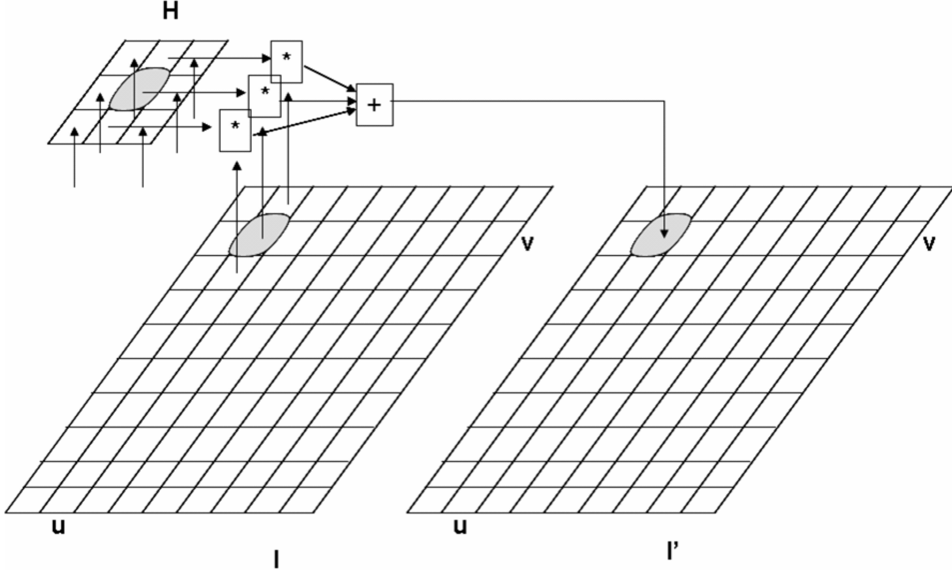

Figure 3 Linear filtering process, see also [1]

The size of the filter region is an important parameter, because it defines how many pixels from the neighborhood contribute to the pixel smoothing. Equation (1) shows the mathematical description.

$$I'(u,v) \leftarrow \sum_{(i,j)\in R} I(u+i, v+j) \cdot H(i,j) \quad (1)$$

The new pixel I' at position (u,v) is calculated from the convolution of the original pixels I at different positions $(u+i, v+j)$ and filter function H . R defines the filter region.

One problem, which occurs, is the filtering at the image boundaries. The filter function requires the availability of pixels from the specified neighborhood. If the target pixel, which should be filtered, is already located at the image boundaries, there are not enough pixels available for filling up the filter matrix H. From practical point of view this means that the size of the filtered image I' is smaller than the size of the original image I . If assuming a typical filter size of 3x3 pixels then one outer row and one outer column at each side of the image disappear for the filtered image I' .

Finally Figure 4 provides the comparison between the original and filtered image. Looking at the traffic sign at the right side or any other of the various objects it can be seen that the right image has less clearness than the left one. The edges have been smoothed and the transitions between the different objects are less precise.

Figure 4 Image before (left) and after (right) filtering

After image filtering has been finalized the next step in algorithm setup is the edge detection, which identifies all potential edges in the image. As a result the image is converted into an new format, which only contains edges, see Figure 5.

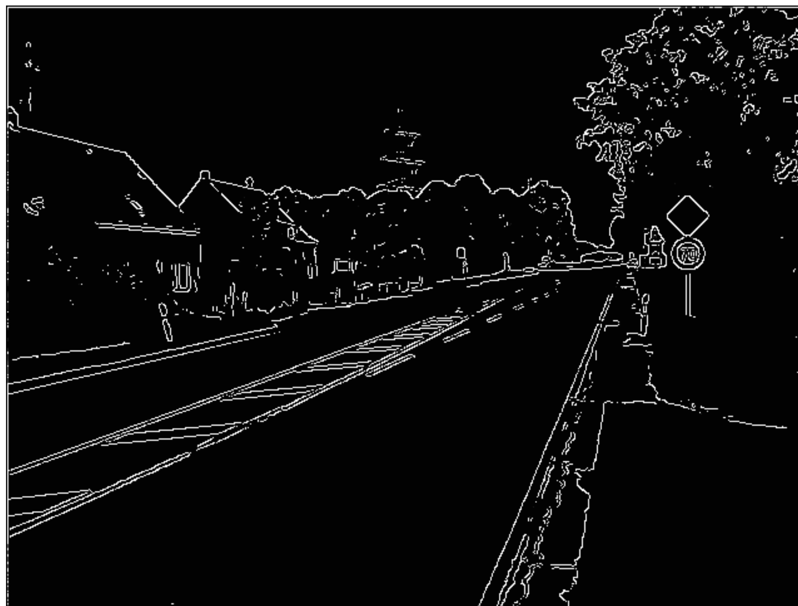

Figure 5 Edge detection

Region of interest selection is based in this approach on Hough transformation [1]. All available edges are investigated, whether they belong to the category *circle*. If not, the edge information is discarded. If yes, see Figure 6, the identified circle is classified as region of interest and delivered to the final algorithm stage, in the actual approach it is template matching. Template matching compares the region of interest with pre-defined objects. Actually, speed limit traffic signs are stored in various sizes for template matching. Template matching investigates whether a stored object provides significant matching with the region of interest. If yes, then a traffic sign has been recognized. If no, the next template is taken into account. If non of the templates match, then the region of interest does not contain the expected object.

Figure 6 Hough transformation

After an insight to the traffic sign recognition setup, the next section investigates general items with regard to parallelization

3 Finding Concurrency

This section presents theoretical considerations in the context of parallel processing. Under ideal conditions, it is possible to reduce the computation time of a program or an algorithm by the number of available processing units. Thus, if an one-processor system requires time T_1, a system with p processors would require only a fraction of T_1. Ideally, this time T_p required by the multi-processor system would be T_1/p.

Amdahl shows in [3] that the ideal conditions assumed in the above considerations are not possible in parallel computing. Every algorithm has one or more parts, which require sequential execution. Acquiring input data or combine data for output are examples for sequential parts. Sequential parts cannot only be found on the algorithm level, but also in blocks inside algorithms if they have strong interdependencies. These parts have to be computed sequentially and execute in time t_{seq}, which is independent from the number of processors. The serial fraction of the execution time of an algorithm is defined as $s := t_{seq}/T_1$ [4].

The execution time of an algorithm which consists of parts which can be processed in parallel and parts which have to be computed sequentially is given by the following equation: $T_1 = t_{seq} + t_{par}$. The execution time on a multi-processor system would be $T_p = t_{seq} + t_{par}/p$. The speedup of an algorithm being executed in parallel is presented in equation (2), see [4].

$$speedup = \frac{T_1}{T_p} = \frac{T_1}{t_{seq} + \frac{t_{par}}{p}} = \frac{1}{s\left(1 - \frac{1}{p}\right) + \frac{1}{p}} \tag{2}$$

This equation shows that the speedup depends on the serial fraction of an algorithm and the number available processors. Thus, for the speedup holds $speedup \geq 1$ because the serial fraction of an algorithm is lower or equal than one $s \leq 1$. Therefore, it is important to be able to determine the serial fraction of an algorithm to estimate the performance gain on a multi-processor system.

The presented considerations and equations do not regard the overhead, which is caused by distributed computing. However, this aspect is an important factor for the evaluation whether a parallel execution of an algorithm is faster than the execution on a single processor. The overhead of a distributed execution of an algorithm is originated by the need to create a thread for each part of an algorithm, which has to be executed on another processor. Creation, destruction, scheduling, and managing these threads consume additional time which is not considered in the above mentioned equations. Moreover, the different threads will have to synchronize with each other. At a certain point in the algorithm, the results have to be combined before they can be passed to the next calculation or algorithm. Therefore, threads have to be stopped and have to wait for other threads to be finished with their parts of the calculation. This results in an additional overhead, because some threads remain idle and waiting. The overhead caused by the introduction of threads will be referred to as *synchronization time* t_{sync} in the following. In [5] the serial fraction is proposed as metric to discover potential performance problems.

In the theoretical considerations about the speedup of parallel executions for an algorithm, the synchronization time has to be respected, too. It has to be noted, that the synchronization time increases with the number of involved processors. The overhead added by threads increases with the number of involved processors. Thus the total time overhead is $t_{sync} \cdot p$. The execution time of a distributed algorithm regarding this overhead is $T_p = (t_{sync} \cdot p) + t_{seq} + \frac{t_{par}}{p}$. These considerations lead to a refinement of equation (2) with respect to the parallel overhead. Equation (3) accounts for these considerations.

$$speedup = \frac{T_1}{(t_{sync} \cdot p) + t_{seq} + \frac{t_{par}}{p}} = \frac{1}{s\left(1 - \frac{1}{p}\right) + \frac{1}{p} + \frac{(t_{sync} \cdot p)}{T_1}} \tag{3}$$

Equation (3) shows that a parallel approach may slow down the total execution time of an algorithm if T_1 is small compared to the synchronization time, which is introduced for each processor. Thus, the speedup may become a speed-down.

Therefore, it is important to identify parts of an algorithm, which can be executed concurrently. The authors of [6] present design patterns for parallel application programs. Design patterns are descriptions of recurring problems and their solutions [7]. The pattern language in [6] consists of four design spaces. First, the *finding concurrency* space, which covers the phase of identifying the concurrency of a problem and defining tasks for parallel execution. Second, the *algorithm structure* space deals with the challenge of finding a structure for the algorithm to take advantage of the possible parallel computation. In the third design space, called *supporting structures*, abstract data types are introduced which may support the programmer in realizing the parallel execution. The forth design space covers low-level implementation issues and therefore, is named *implementation mechanisms*. The separation into the above mentioned four parts is a good approach to follow while porting an algorithm from a single-processor system to a multi-processor system.

4 Parallel Implementation

Before investigating the parallelization of the object recognition system, it is important to analyze the overall processor load when the different algorithms are active. Figure 7 provides the overall load division of the different algorithms in a single processor environment. The first operation starts with the conversion of RGB colors into a grey image. After that, Gauss filtering with 12% and edge detection based on Canny with 16% processor load takes place.

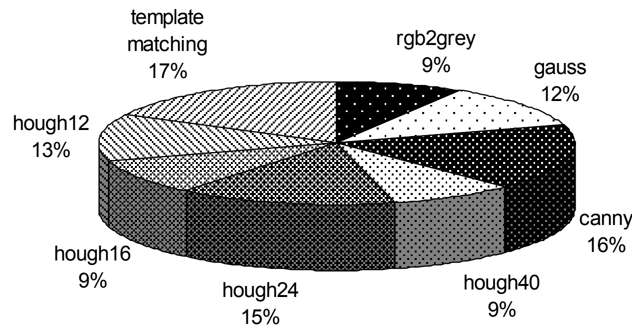

Figure 7 Division of the overall processor load in a single processor environment

For Hough transformation radii with different values (12, 16, 24, 40 pixels) have been applied to find a potential traffic sign with different sizes. The Hough transformation consumes about 46% of the overall processing time. Finally, template matching takes 17% processor time. In the following Gauss and Hough transformation are investigated in detail.

Taking the *finding concurrency* space into account the described object recognition system can be investigated for parallel implementation. The Gauss filtering process fulfills the qualifications for parallel execution. From equation (1) one can see, that each resulting pixel could be assigned to an own thread for parallel execution. This is possible because no serial dependencies internally Gauss filter require the algorithm to wait for provisional results. The only interdependency for such an extreme case is the memory management, because smoothing one pixel $I'(u,v)$ requires convolution with original neighboring pixels $I(u+i,v+j)$ from region R, not with already smoothed new neighboring pixels $I'(u+i,v+j)$. Guaranteeing the separation between these two memory spaces, full parallelization is possible. When going back to the *finding concurrency* space theory, one can define a task decomposition to execute the Gauss filtering in multithreading manner.

In the current case it makes no sense to provide e.g. 640x480 = 307200 single threads because so many processing units are not available from the test platform. In this paper an ARM four-processor platform has been used. The ARM processors are based on a 32-bit integer RISC architecture. The used ARM11 cores are clocked with 200 MHz and can use 32 kB of dedicated level-1-cache for data and instructions respectively. A shared second level cache with a size of 1 MB, which runs at core frequency, allows fast data exchange between the processors.

Thus there has been investigated how 1, 2 and 4 threads behave from the relative time consumption and speed up perspective. One image row has been assigned to one thread, thus, 1, 2, or 4 rows are computed in parallel, respectively. Figure 8 shows the normalized processing time.

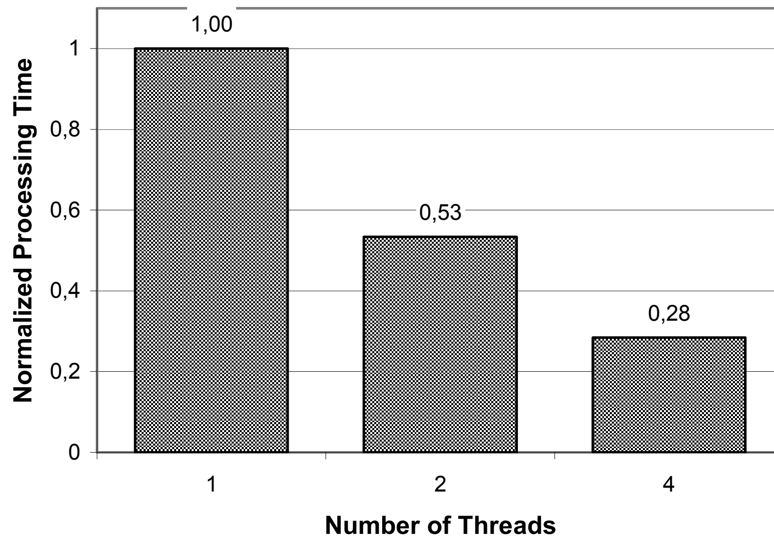

Figure 8 Gauss filter multithreading with 1,2 and 4 threads, 640 x 480 pixels

The single thread approach has been used for normalization and thus requires 1 single time unit. Two threads can speed up the overall image processing for Gauss filtering to 0.53 time units, which is practically 6% from ideal time consumption of 0.5 time units. Four parallel threads speed up to 0.28 time units or 12% from ideal time processing with 0.25 time units.

Very similar results can be achieved for Hough transformation from Figure 1 and Figure 6, respectively, In this particular case, circle detection is considered. Processing time measurement results from a 640 x 480 pixel image can be seen from Figure 9.

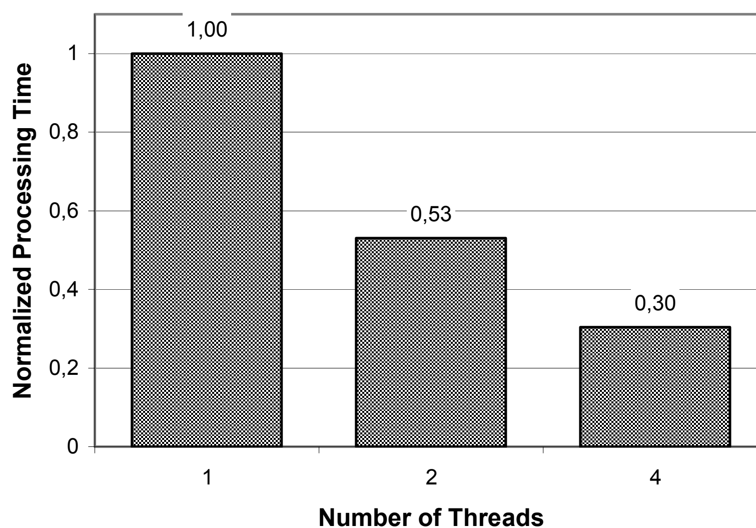

Figure 9 Hough circle multi-threading with 1,2 and 4 threads

The algorithm, which belongs in Figure 1 to the classification section of the overall object recognition approach, provides also an overhead of 6% when running with two threads and 20% when utilizing four parallel processors.

But for embedded systems it is not always possible that image sizes of 640 x 480 pixels can be processed in real time. This is because of the overall algorithm complexity for pre-processing and classification, see Figure 1, and available processor power in the mobile device. In this implementation the image size often needs to be chosen with lower resolution than 640 x 480 pixels. Figure 10 and Figure 11 show that image size of 128 x 96 pixels provide a cross-over point for processing time and speed up factor between one- and multi-processor approaches.

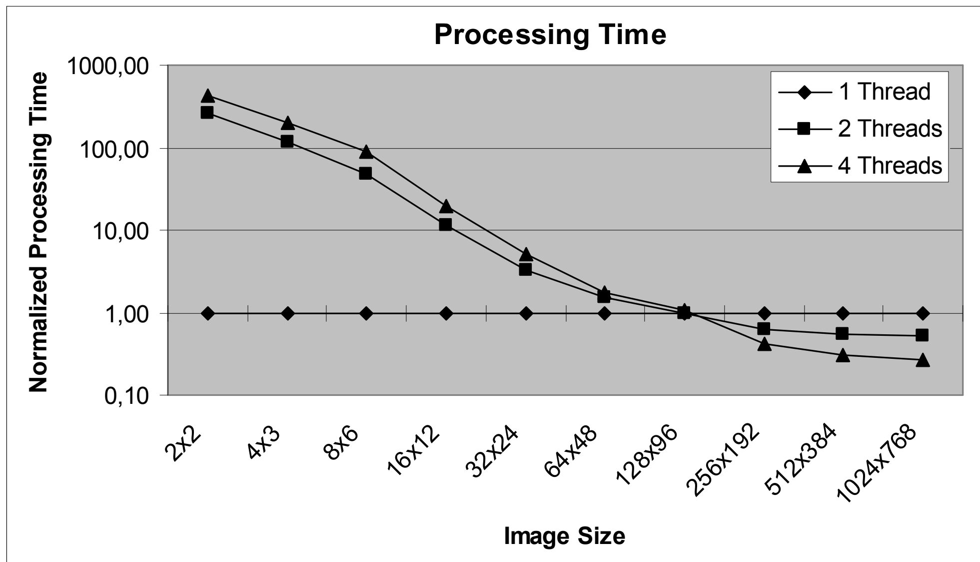

Figure 10 Normalized processing time for various images sizes

Images with smaller sizes than 128 x 96 pixels consume more processing time the smaller the image sizes are and the larger the number of threads is. If the image size is larger than 128 x 96 pixels, the multicore platform provides more performance than a single processor approach. Figure 11 shows the speed up factor measurements.

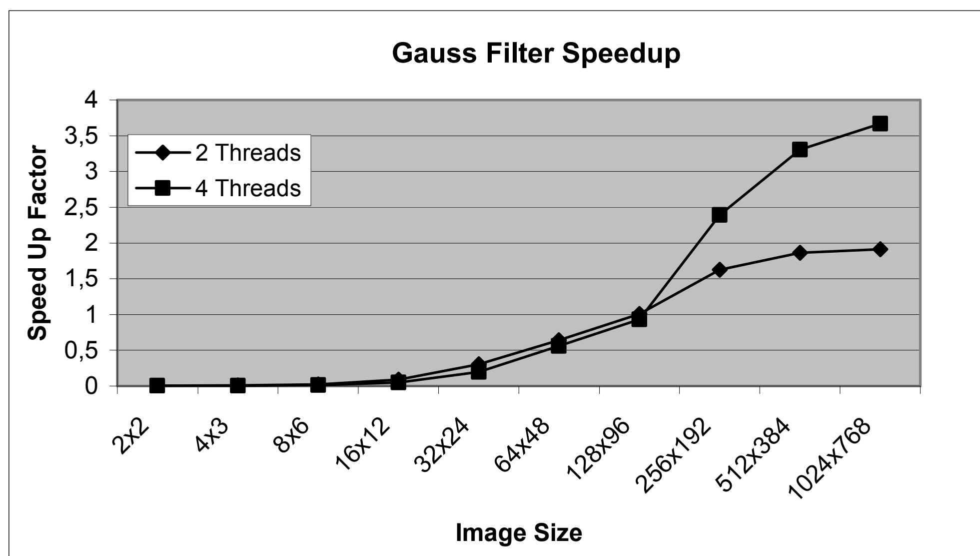

Figure 11 Gauss filter speed up factors for 2 and 4 threads

With very large image sizes like 1024 x 768 pixels the two threads approach speeds up to factor 1.91 or 4.5% less than ideal speed up factor 2, and the four threads approach speeds up to factor 3.67 or 8.25% less than ideal speed up factor 4. The two threads approach is more close to optimum with an image size of 1024 x 768 pixels than the four threads approach, which requires even larger image sizes to be as efficient as possible.

For the embedded system design approach from algorithm level as well as multi-processor architecture point of view, this means that there is most probably an iterative approach for optimization required. The potential parallelization of all algorithms, the real-time requirements, e.g. required number of images per second, as well as the number of processors and architecture of the multicore platform, needs to be adjusted to each other as long as the best setup from performance, cost and power consumption perspective has been reached.

5 Conclusion

This paper has introduced an object recognition system for speed traffic signs. The system is targeting to run on mobile devices and thus, the system needs to work in an embedded environment. The available processing power in embedded systems is limited, but the application needs to work in real-time. Therefore it is a good alternative to employ a multi-processor platform to provide enough processing power.

Because parallelization of a system is not a simple task, it needs to be identified, which parts of the system can be utilized for parallel execution. Therefore one can start to dig into the *finding concurrency* space, which provides on an abstract level an indication about the parallelization potential within an application. Two algorithms, Gauss filtering and Hough transformation, have been investigated for parallelization. Results show that parallelization is efficient and normalized time consumption is roughly proportional to the number of processors if the image size is large enough. For small images, the actual parallel processing setup can significantly slow down the processing speed.

Thus this paper is the starting point to investigate how to optimally choose system parameters, parallel platform architecture and finally algorithm architectures for object recognition in embedded systems.

References

1. Wilhelm Burger, Mark James Burge, Digitale Bildverarbeitung, 2. Aufl., ISSN1439-3107, Springer-Verlag Berlin Heidelberg, 2005
2. John L. Gustafson, Reevaluating Amdahl's law, Commun. ACM, volume = 31, number = 5, 1988, 0001-0782, p.532--533, http://doi.acm.org/10.1145/42411.42415, ACM Press, New York, NY, USA,
3. Gene M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, AFIPS spring joint computer conference, 1967, 0001-0782
4. Michael J. Flynn; Kevin W. Rudd, Parallel architectures, ACM Computing Surveys, volume 28, No. 1, p.67--70, 1996
5. B. Massingill, T. Mattson, B. Sanders, Patterns for parallel application programs, Proceedings of the Sixth Pattern Languages of Programs Workshop PLoP99, 1999
6. Erich Gamma, Richard Helm, Ralph E. Johnson, John M. Vlissides, Design Patterns: Abstraction and Reuse of Object-Oriented Design, ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, 1993, ISBN 3-540-57120-5, p.406--431, Springer-Verlag,, London, UK, National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov
7. Alan H. Karp, Horace P. Flatt, Measuring parallel processor performance, Commun. ACM, vol. 33, No. 5, 1990, ISSN 0001-0782, p.539--543, http://doi.acm.org/10.1145/78607.78614, ACM Press, New York, NY, USA