Edge Vectorization for Embedded Real-Time Systems using the CV-SDF Model

Dirk Stichling, Bernd Kleinjohann
University of Paderborn / C–LAB
Fürstenallee 11, D-33102 Paderborn, Germany
{tichel, bernd}@c-lab.de

Abstract In this paper an edge vectorization algorithm is presented. The target architecture of the algorithm are Embedded Real-Time Systems as can be found in vehicles, robots or toys. The most important aspect of this work is not the algorithm as such but the way how such an algorithm for Embedded Systems with restricted resources is realized. The edge vectorization is designed using the CV-SDF model, a computer vision extension to SDF (Synchronous Data Flow graphs) which are widely used in the domain of Real-Time Systems. Finally an implementation of the algorithm on a TriMedia TM1100 embedded processor is presented.

Keywords: real-time computer vision, edge vectorization, CV-SDF, embedded real-time systems, TriMedia 1100

1 Introduction

Computer Vision systems more and more tend to be used in consumer electronic products like Mobile Phones, Automotives, Toys, Personal Robots etc. All these products are Embedded Real-Time Systems with special restrictions like little memory, little computing power and low costs. Hard timing constraints have to be fulfilled by most applications like an obstacle detection system for vehicles or robots. Thus *real-time* means to meet timing deadlines therefore *Worst-Case Execution Time Analysis* (WCET) is an important manner.

Only little attention has been paid at combining the world of Embedded Systems and the world of Computer Vision. Computer Vision algorithms sometimes are called *real-time* but this often only means that they run with a frame-rate of 10 frames per second or higher on some kind of PC or workstation. But only little systematic work has been done on real *real-time* computer vision. The CV-SDF model is one way to specify real-time capable computer vision algorithms.

The structure of this paper is as follows: Section 3 gives an introduction to the CV-SDF model which is an attempt to combine the worlds of embedded systems and computer vision. Based on this model an edge vectorization algorithm is proposed in Section 4 to show how to realize a real-time computer vision algorithm. The edge vectorization algorithm is used in an Augmented Reality project called AR-PDA[1] where it is used to extract edges of household appliances like ovens and dishwashers. It is also used by our soccer playing robots to find the lines of the soccer field. Section 5 presents the results of our implementation. The paper closes with a conclusion in Section 6.

2 Related Work

In [3] a modular software architecture for real-time video processing is proposed. The cornerstone of the architecture is the *Flow Scheduling Framework (FSF)*. It specifies and implements a common generic data and processing model designed to support stream synchronization in a concurrent processing framework. The FSF is a middleware between the operating system and the application.

In [4] a design methodology of real-time vision based embedded systems is presented. Based on a real-time kernel *RTKER* realized on a TriMedia multimedia processor implementations of three computer vision algorithms are presented.

In [5] a Domain Specific Language (DSL) for computer vision systems called *FVision* is introduced. The language bases on *Haskell*, a general purpose, purely functional programming language. FVision uses the XVision C++ library for the computational operations and therefore is not really real-time capable in the sense of timing guarantees etc.

All systems described above use some kind of operating system as lowest layer. Implementations

[1]AR-PDA is funded by the German ministry for education and research (Bundesministerium fr Bildung und Forschung, BMBF).

of the CV-SDF algorithms do not necessarily need an underlying operating system and thus also support pure hardware implementations or mixed hardware/software architectures. In contrast to the other propositions the CV-SDF model directly addresses the problem of partitioning the image frames into smaller parts to minimize memory buffers and latency which is important for embedded systems due to their restricted resources.

3 CV-SDF

The edge vectorization algorithm proposed in this paper is based on the CV-SDF model first introduced in [6]. The CV-SDF model bases on Synchronous Data Flow Graphs (SDF) which are often used in real-time signal processing applications with streaming data like audio compression and decompression [2]. SDF graphs consist of nodes which contain the program code and edges between the nodes. The edges are FIFO buffers. Each edge is annotated with the number of tokens the start node produces within one execution of the node and the number of tokens the end node consumes. Due to the static nature of SDF graphs *schedules* for the execution order of the nodes can be calculated at compile time. One of the main features of the SDF model is the real-time capability. It is achieved due to the static scheduling and static buffer management.

But the SDF model lacks support for data access often used in image processing applications like kernel-based filter operations and tracking mechanisms. Therefore the CV-SDF model introduces *StructuredBuffers*. Instead of simply using a FIFO buffer with only `add` and `remove` operations StructuredBuffers also allow access to other parts of the buffers.

The CV-SDF model partitions image frames into *slices* of identical size. A slice may be one image line, a 8x8 pixel block or something similar. These slices are the tokens (the data instances) which are transferred over the edges of the graph.

The nodes of the CV-SDF graph (called *Modules*) contain the computer vision operations in form of one function for each module. With each execution of a module (called *activation*) a fixed number of slices is consumed and produced by the module.

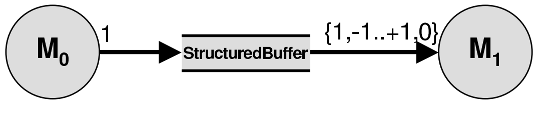

Figure 1: Example of a simple CV-SDF graph

Figure 1 shows a simple CV-SDF graph consisting of two modules M_0 and M_1. The output edge of a module is annotated with the number of slices the module produces with each activation. The input edge of a module is annotated with a triplet $\{consumed, slice\_interval, previous\_frames\}$. *consumed* is the number of slices a module consumes with each activation, *slice_interval* is the interval of neighboring slices the module needs access to and *previous_frames* is the number of previous image frames the module wants to access. The last two values of this triplet are new in contrast to the SDF model. They allow easy and effective implementations of low-level computer vision operations.

All modules comply to two principles, that were introduced to decrease the latency of the algorithm:

1. *Linear Processing*: The input images of nearly all devices are transmitted linearly line-by-line and from left-to-right similar to the analog video format. Therefore to minimize system latency all modules of a CV-SDF algorithm also have to work top-down and from left-to-right without waiting for a whole image.

2. *Incremental Concurrency*: All modules have to provide their output as soon as possible. This minimizes the size of temporary buffers and also minimizes the latency of the system.

The main advantages of the CV-SDF model are the minimization of the buffer usage, the minimization of latencies and the real-time capability.

4 The Algorithm

The CV-SDF graph of the proposed edge vectorization algorithm is shown in Figure 2.

All slice data-types are *line-based*. That means that each module consumes and produces slices which each correspond to one line of the input image. There are three different slice data type:

1. *GreyImageLine*: This data type stores the lightness of the pixels of exactly one line of the image.

2. *GradientLine*: This data type stores the gradients of the pixels as a result of a Sobel filter. It also stores the data for one image line.

3. *StraightEdgesLine*: This data type stores the numbers of the StraightEdgesBuffer-slots (see below) the pixels of exactly one line belongs to.

The CV-SDF model is designed for low-level computer vision algorithms and works best with pixel-based slice-data-types. The vectorized edges are not

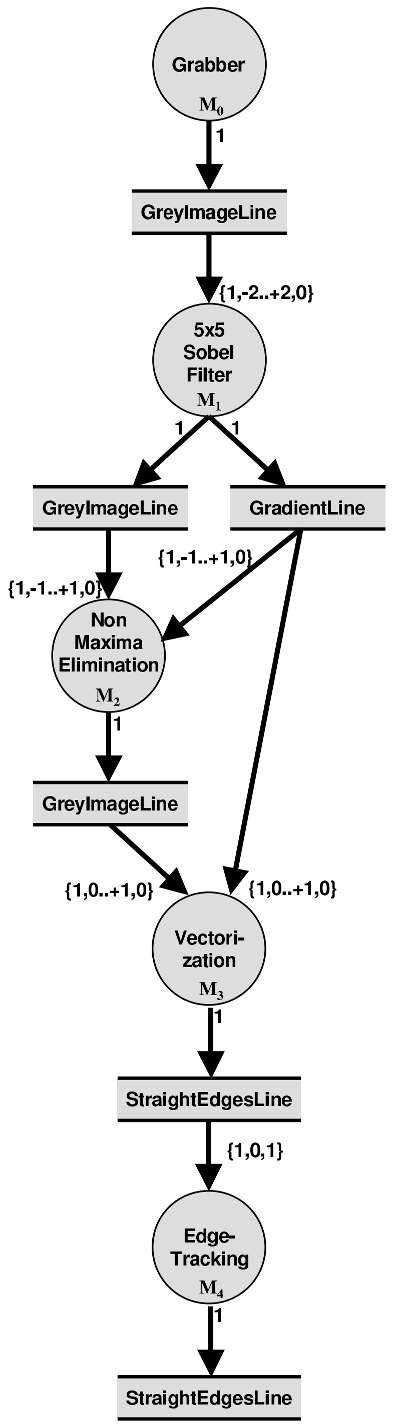

Figure 2: CV-SDF graph of the proposed algorithm

pixel-based therefore one buffer associated to each frame is introduced to store the edges. The buffer is called `StraightEdgesBuffer` and the instances are called `StraightEdge`. The size of the buffer is limited at compile time to avoid dynamic memory allocation.

There are two possible representations for `StraightEdge`:

1. Using *Start- and End-Points*: Only the start- and the end-pixel of a line is stored. If a new pixel has to be added to a StraightLine (during the execution of the algorithm) it has to be decided whether this new pixel replaces the start- or the end-point.

2. Using *Moments*: Similar to the representation of regions in [7] the lines are stored using the moments of first order and central moments of second order.

Which representation to use depends on the application. The first representation yields better results for the real end-points of the lines and the second representation yields more accurate results (sub-pixel accuracy) in respect of the whole line because statistical calculations are used.

As shown in Figure 2 the algorithm consists of the following modules: the *Grabber module* (M_0) is a module representing an image source, a *Sobel filter module* (M_1) to extract edges, a *Non-Maxima Elimination module* (M_2) to thin out the edges, the *Vectorization module* (M_3) to extract the lines from the pixel image and the *Edge Tracking module* (M_4) to connect the vectorized edges with the corresponding edges of the previous image frame.

The *Sobel filter module* (M_1) is a standard kernel operation with size 5x5. It can be easily implemented as CV-SDF module so that it complies to the principles *Linear Processing* and *Incremental Concurrency*. One processing step calculates the kernel values for the pixels of one image line. Therefore it needs the pixel values of the two previous and the two subsequent image lines. It consumes exactly one line and needs no access to previous frames. Thus the annotation of the input edge is $\{1, -2..+2, 0\}$.

The Sobel filter yields information about the strength and the gradient of the edge. The gradient is abstracted by one of 8 different values as needed by the Vectorization module.

The *Non-Maxima Elimination module* (M_2) is a standard computer vision operation similar to the Sobel filter. It uses the 8-neighborhood of the actual pixel. Therefore it can be implemented in a similar way as the Sobel filter. The Non-Maxima Elimination thins out the edges produced by the Sobel filter. This is needed to implement a line-following algorithm.

The *Vectorization module* (M_3) implements a line-following algorithm to vectorize the edges into

StraightEdges. The algorithm is based on ideas published in [1]. We adapted the algorithm to be compliant with the concepts *Linear Processing* and *Incremental Concurrency*.

The input to this module are lines with strength information (GreyImageLine) and the orientation information (GradientLine). The orientation for each pixel is abstracted by one of 8 directions as shown in Figure 3.

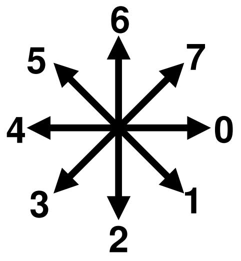

Figure 3: Quantization of gradients

The module processes the lines pixel by pixel from left to right. For the actually processed pixel the set of potential neighbors is determined. This set only depends on the orientation of the actual pixel. Because the algorithm works top-down and from left to right only the neighbor pixels which have not yet been processed have to be compared with the actual pixel. The actual pixel matches with one or more of its neighbors if the orientation of the actual pixel and the orientation of the neighbor pixel only differs one step (in reference to the quantization value 8).

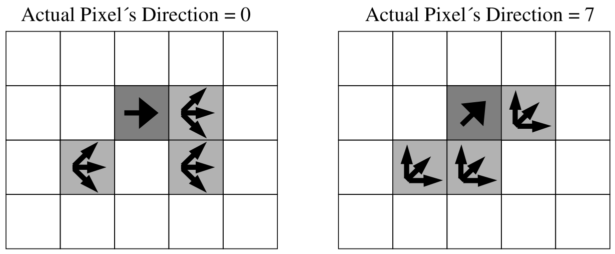

Figure 4: Example of pixel neighborhoods

Figure 4 shows two examples: the neighborhood of a pixel with orientation 0 and one with orientation 7. The potential neighbors of the pixel with orientation 0 may have orientation 0, 1 or 7. The position of potential neighbors are all positions of the 8-neighborhood which have not yet been processed and are not parallel to the actual pixel in respect of the orientation (e.g. the position below the actual pixel with orientation 0).

If two or more neighbors match the actual pixel one of the following actions is performed:

- If all pixels are not yet assigned to a StraightEdge a new StraightEdge is initialized with the matching pixels.
- If exactly one of the pixels has been assigned to a StraightEdge all non-assigned pixels are added to that StraightEdge.
- If two pixels are assigned to different StraightEdges these StraightEdges are merged and all non-assigned matching pixels are added to that merged StraightEdge.
- If two pixels are assigned to the same StraightEdge all additional non-assigned matching pixels are added to that merged StraightEdge.

Due to the method used for scanning the image it is not possible that more than two pixels are already assigned to a StraightEdge. The adding of pixels and the mergence of lines can be done in constant time. This allows the calculation of a WCET. If a pixel is already assigned to a StraightEdge the orientation of the complete StraightEdge is taken as orientation instead of the orientation of the single pixel. This is important to only extract straight lines and not any arbitrary lines. At the end all edges are vectorized without any further processing.

A StraightEdge is called *finished* if no pixel of the subsequent line has been added to the StraightEdge after the processing of the actual line. No more pixels will be added to finished edges.

The vectorization module only needs the actual and the subsequent image line and consumes and produces exactly one image line. Therefore the annotations of the input edges is $\{1, 0..+1, 0\}$.

For the start- and the end-pixel of finished StraightEdges the *EdgeTracking module* (M_4) searches inside the pixel's neighborhood of the previous frame for a corresponding edge. If a matching edge is found the StraightEdges are logically connected. A fixed pixel window size around the actual pixel is used to guarantee a maximum run-time so that a WCET analysis is feasible.

5 Implementation and Results

The main target architecture of the edge vectorization algorithm is the TriMedia TM1100 processor[2] which is a 100MHz DSP-like multimedia processor developed by Philips.

[2]http://www.trimedia.com

The TriMedia processor has hardware support for some aspects of computer vision like color space conversion, scaling and filtering. It has a 5-tap 1-dimensional hardware filter which we use to implement most parts of the 5x5 Sobel filter.

The image grabber is also part of the TriMedia processor. It grabs the input image and stores the data in the main memory without the need of the processor's core CPU. Therefore only the Modules M_2, M_3 and M_4 and some parts of M_1 are implemented using the core CPU.

	Memory	*Run-Time*
M_0: Grabber	5 lines = 880 Bytes	HW: 20ms SW: /
M_1: 5x5 Sobel	2x3 lines = 1056 Bytes	HW: 10ms SW: 4ms
M_2: Non-Max Elim.	2 lines = 352 Bytes	HW: / SW: 8ms
M_3: Vectorization	1 image = 50 kBytes	HW: / SW: 10ms
M_4: Tracking	1 image = 50 kBytes	HW: / SW: 12ms
Total:	101.8 kBytes	HW: 30ms SW: 34ms

Table 1: Memory consumption and run-times of the TriMedia implementation

Table 1 shows the memory consumption and the run-times of the modules of the TriMedia implementation. The image size was 176x144 pixels. The memory values correspond to the output-buffers of each module. The run-times are devided into hardware (HW) and software (SW) run-times.

The run-time of the analysis of one image is 34ms. This corresponds to a framerate of more than 25 frames per second.

These values show that pixel-based low-level computer vision algorithms can be realized on embedded real-time systems with the full frame-rate of 25 frames per second while a maximum run-time is guaranteed.

All modules of this edge vectorization implementation are real-time capable but we do not have any analytic results of the worst case execution times of the modules yet. This will be done as a next step.

Figure 5 shows an vectorization example. The left part is the original image and the right part shows the vectorized edges. The image shows a typical scene we are confronted with within our AR-PDA project.

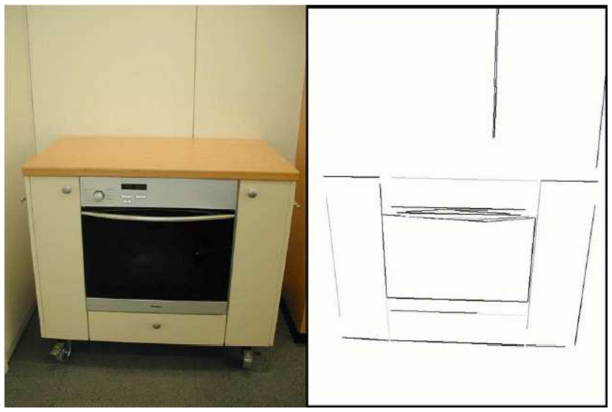

Figure 5: Edge vectorization: Input image and vectorized edges

6 Conclusion

In this paper we presented an edge vectorization algorithm based on the CV-SDF model. The algorithm as such is not the most interresting part but the way how to implement such algorithms on embedded real-time systems. First an introduction to the CV-SDF model was given and then the algorithm using 5 modules was presented. The results of the implementation show that it is possible to implement computer vision algorithms with very little memory footprints while guaranteeing deadlines using a worst case execution time analysis.

Based on the CV-SDF model we also realized a color segmentation algorithm and an obstacle detection algorithm. Both algorithms are used by our soccer playing robots which are taking part at the RoboCup middle-size league competition.

All these algorithms are used seperatly at the moment. In the future we will combine these three algorithms to built one single system.

References

[1] M. Aste, M. Boninsegna, and B. Caprile. A Fast Straight Line Extractor for VisionGuided Robot Navigation. Technical report, Istituto per la Ricerca Scientifica e Tecnologica, 1994.

[2] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, 21:151–166, 1999.

[3] Alexandre R.J. François and Gérard G. Medioni. A modular software architecture for real-time

video processing. In B. Schiele and G. Sagerer, editors, *Computer Vision Systems, Second International Workshop, ICVS 2001*, pages 35–49, Vancouver, Canada, 2001. Springer-Verlag.

[4] Vivek Haldar. Design of embedded systems for real-time vision. In *Indian Conference on Graphics, Vision and Image Processing (ICVGIP)*, 2000.

[5] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Proceedings of the 21st International Conference on Software Engineering*, pages 484–493. ACM Press, May 1999.

[6] Dirk Stichling and Bernd Kleinjohann. CV-SDF - a synchronous data flow model for real-time computer vision applications. In *IWSSIP 2002: 9th International Workshop on Systems, Signals and Image Processing*, Manchester, UK, November 2002.

[7] Dirk Stichling and Bernd Kleinjohann. Low latency color segmentation on embedded real-time systems. In *DIPES 2002: IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems*, Montreal, Canada, August 2002.