Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems), Ghent, Belgium, September 2-5, 2003

APPLICATION DRIVEN DESIGN OF EMBEDDED REAL-TIME IMAGE PROCESSORS

[1] [2]*Pieter Jonker,* [2]*Wouter Caarls*

[1]pieter@ph.tn.tudelft.nl
[2]Delft University of Technology, Lorentzweg 1, Delft, The Netherlands

ABSTRACT

Real-time image processing systems become more and more embedded in systems for industrial inspection, autonomous robots, photo-copying, traffic control, automotive control, surveillance, security, and the like. Starting in the 80's many systems - mainly for low-level image processing - have been developed. The architectures range from framegrabbers with attached Digital Signal Processors (*DSP*s), to systolic pipelines, square and linear single-instruction multiple-data (*SIMD*) systems, pyramids, PC-clusters, and smart cameras. Many of those systems lack a suitable software support, are based on a special programming language, are stand alone and cannot be tightly coupled to the rest of the processors of the embedded system. As a consequence, most often the embedded system cannot be programmed in one uniform way.
In this paper we will shortly review the archetypes of image processing architectures and their support, after which we will elaborate on a hard and software design framework for embedded image processors. In this framework we are able to schedule the inherent data and task parallelism in an application in such a way, that a balance is found for both data and task parallel parts of the application software. This schedule is optimal for a certain architecture description. For the selection of the best architecture in combination with the best schedule, one can cycle through design space exploration and scheduling.

1. INTRODUCTION

We aim to investigate the design of application-specific programmable smart cameras, with integrated sensor, SIMD-, and ILP processors. In such a camera, a lot of vision processing is done on-board, allowing the camera to actuate control systems, raise alarms, or output symbolic information. Single instruction multiple data (*SIMD*) processors are especially well suited for the pixel and neighbourhood operations common in low-level image processing, while a network of instruction-level parallel (*ILP*) processors can handle the more coarse-grained and irregular algorithms found in intermediate- and high-level tasks.

We are most interested in quantifying the design flow of such systems via the use of simulation and analysis in a design space exploration (*DSE*) environment, and in the development of an intuitive programming model. In this paper, we will first introduce the programming model, which is based on instantiating *algorithmic skeletons* in order to bring parallelism into a sequential code image. Then, we will show how this is integrated in the overall DSE framework, and how this allows a developer to generate the most appropriate architecture for his application.

Section 2 will review previous work in the field of fast image processing, providing a reference for our architecture template and programming model, described in sections 3 and 4. We will continue by introducing our DSE framework in section 5, and section 6 gives an example of the entire design flow. Finally, we will have some concluding remarks.

2. PREVIOUS WORK

The large amounts of data used in image processing, and the speed needed to process this information in a reasonable amount of time, has led the image processing community to look into special computer architectures since the early 1970s [1]. Subsequent miniaturization efforts have brought us to the point where it is possible to integrate an entire vision processing system in a single security-camera sized device.

2.1. Stand alone systems

Recognizing the data parallelism inherent in low-level vision operations such as point and local neighbourhood operations, image processing systems have been designed massively parallel from the start. This has taken many forms, such as (systolic) pipelines [2], SIMD processor arrays [3], and mesh connected multiprocessors[4]. At the start of the nineties, it became theoretically clear that only the Linear SIMD Processor would survive [5]. Pipelines and systolic wave-front arrays are difficult to program and cannot cope with dynamic control flow: the pipeline must be flushed at many program branches. Consequently, these systems can only fruitfully be used in, for example, the processing of streaming video, using a single dedicated algorithm. Square SIMD systems just became too large and costly, when every pixel had its own processor. A common solution to overcome this, was reducing the word-length of the Processing Elements (*PE*s) to 4 or 1 bit. Still,

Figure 1: UCL CLIP4 SIMD image processing system

in many systems the array size was smaller than the image size, which made scanning of the array over the image necessary. For square arrays this involves a considerable overhead, in contrast with linear arrays. It can be proven [5] that with the same amount of chip area, it is more beneficial to make a linear array with a larger word-length than a square array with a small wordlength. Both the square and the linear SIMD array have the benefit that they are very flexibly programmable, and provided with indirect addressing of the image memory, also algorithms that address the pixels in an irregular way can be supported [6]. Still there usually is a gap both in hardware and in software between the SIMD system and its host, a PC or workstation.

With ever increasing workstation processing speeds, the advent of cheap Beowulf-type commodity clusters [7], and the increase in their communications bandwidth, the other architectures have faded into the background. The sole survivor is the linear SIMD architecture, that can be found as accelerator board, architecture component in smart cameras and even in a rudimentary form in MMX/SSE/Altivec/... instruction sets in general purpose processors.

The clusters are usually programmed using the MPI message passing library [8]. Data-parallel language extensions such as CC++ [9] or the compiler-directive based OpenMP [10] are also used, but the amount of effort required keeps image processing researchers away, except when the added speed is absolutely necessary. Recent efforts have created specific image processing libraries which generate data-parallel [11] or even mixed data/task parallel [12] programs from sequential code images, which go a long way towards providing researchers with the benefits of parallel processing without the hassle.

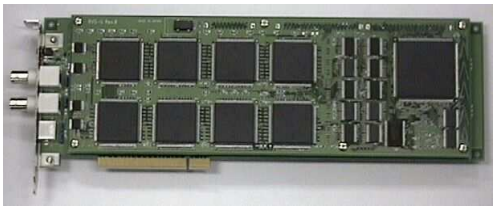

Figure 2: NEC IMAP-Vision SIMD vision accelerator

2.2. Vision accelerator boards

Because of the increased power and area efficiency, SIMD arrays, and in particular linear processor arrays (*LPAs*), are still frequently used in embedded applications. Vision accelerator boards are employed in real-time control systems where there is enough room to have a workstation. They contain LPAs (IMAP-Vision [13]), DSPs (FUGA [14]), or GP processors (Genesis*Plus* [15]).

The IMAP-Vision uses a data-parallel C extension called 1DC [16] to program the LPA, while the FUGA and Genesis*Plus* are programmable in standard C++. All boards provide optimized library routines for common image processing operations. In addition, the Genesis*Plus* uses the library routines to interface with a separate neighbourhood processor as well.
The use of an explicitly data parallel language makes the IMAP-Vision slightly more difficult to program, but also potentially faster. It occupies a place between assembly language, which is always fastest but not realistically used by image processing researchers, and a library-only based approach, which may shield the programmer too much to make any optimizations. It seems that a library-based system in which the user can descend to a (parallel) programming level, if necessary, is the best approach.

2.3. Smart cameras

For the even more embedded market, with a need to be very small and power efficient, cameras that integrate sensing and processing are emerging. Again, DSP (Vision Components [17], iMVS [18]) and GP (Legend [19],

Figure 3: Philips CFT Inca311 Intelligent Camera

mvBlueLYNX [20]) solutions are often used, but single-chip LPAs (Xetal [21], added to Inca311 [22]) and even integrated sensor/LPA chips (MAPP2500 [23]) exist as well.

Again, all systems are programmable using an image acquisition and processing library, but the single chip LPAs, because of the simplicity of their processing elements, cannot easily be programmed in C. Xetal tries to remedy this by providing a C-like macro language, while the MAPP2500 avoids the problem altogether by only providing a few algorithms specific to the expected application domain (range imaging). Both solutions are unsatisfying.
Two of the smart cameras, Inca311 and Legend, are also programmable using graphical programming languages. Both are targeted at industrial inspection, and allow novices in the field of image processing to graphically connect algorithms like sub-pixel edge detection, angle measurements and template matching. Efforts have been made to put such a user interface above a library-based approach, providing another level of abstraction in a single framework.

3. ARCHITECTURE TEMPLATE

In our SMARTCAM [24] DSE environment, an application designer will be able to generate an optimal smart camera hardware configuration for his specific domain, based on his application code and various constraints such as size, cost and power consumption. However, for this approach to be feasible it is necessary to restrict the search space by imposing an architecture template. Based on the previous work described in the previous section, our architecture template will consist of a sensor, LPA(s), instruction-level parallel (*ILP*) processor(s), memories, and communications peripherals (see figure 4). These will be parameterizable with regard to resolution, number of PEs and PE functionality, data width, the amount and type of functional units, etc.
The choice of an LPA is simple, because it is perfectly suited for the data parallelism inherent to low-level image processing operations. ILP processors, such as very long instruction word (*VLIW*) and *superscalar* processors, can execute multiple independent instructions per cycle, exploiting a finer-grained level of parallelism than LPAs. This is necessary because higher-level vision processing tasks are too irregular to execute on LPAs. Finally, using a network of processors allows us to take advantage of the independence between different image processing tasks, or between different stages in a pipeline.

4. PROGRAMMING MODEL

Of course, managing such a diverse set of parameterizable processors without putting too great a strain on the programmer requires a unified programming model. The programming languages for the systems described in section 2 fall in five categories: (parallel) assembly, specialized parallel languages, data-parallel extensions to a sequential language, (generalized) libraries, and graphical programming environments. We think that assembly is too time consuming, and specialized parallel languages require too much effort to learn to gain widespread use in real applications. This leaves us with three viable options; however, as described in [11], any deviation from a standard sequential programming model creates a barrier for adoption, and thus we would like to limit that as much as possible.
Thus, our programming model will consist of a C/C++ image processing library, possibly with a graphical programming environment on top. If the user wants to add library routines, either to accomodate new algorithms or to speed up existing ones, he can do so by using some data-parallel extensions in the form of compiler directives or *pragmas*. Note, though, that because of the possibly limited capabilities of some of the processors in the architectural template, he may have to provide several versions.

4.1. Algorithmic skeletons

Because extensions to the library should be as infrequent as possible, we will base it around the concept of algorithmic skeletons [25], also called template- or pattern-based parallel programming. This means that the library provides higher-order functions which only implement a certain structure and communication, while the user provides the code for the actual computation (see figure 5). A useful survey of different skeletons and implementations is contained in [26]. In the field of low-level image processing, examples are generalized skeletons for point operations, neighbourhood operations, and global reductions.

By using the algorithmic skeletons, the user is completely shielded from the parallel implementation of his algorithm,

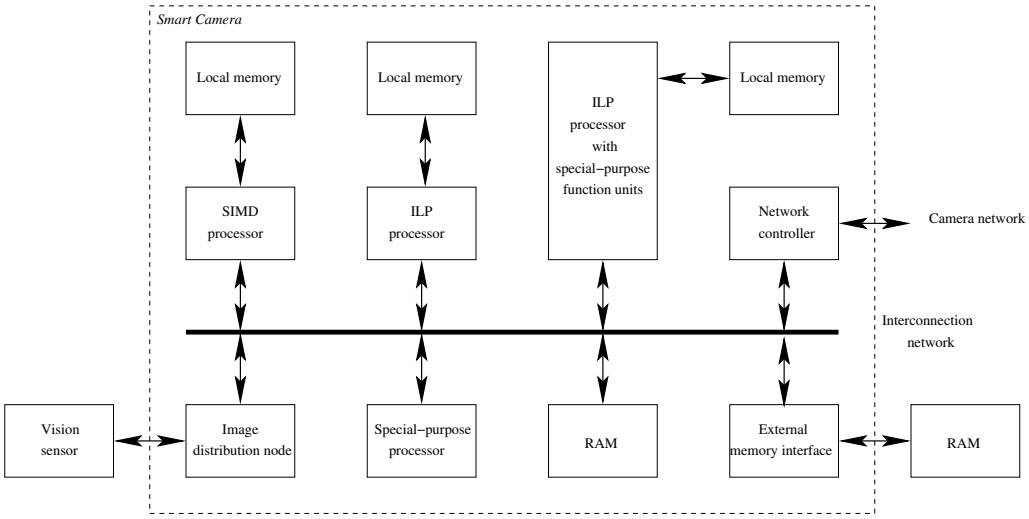

Figure 4: A possible architecture template for a smart camera device, containing SIMD, ILP, and special-purpose processors. All components, including the interconnection network, are subject to adjustment by the architecture exploration.

providing only the sequential code to process a single datum. The advantage, apart from providing the developer with a sequential interface and avoiding changes to the library, is that this abstraction allows the program to be executed on different processor architectures without changes to the user code: once a skeleton implementation has been provided for the architecture, it is possible to run any instantiation of it[1]. Skeletons which are not implemented on a certain architecture are simply never scheduled to those processors.

While this abstraction over communication as well as architecture is very convenient, algorithms which cannot be captured in one of the provided skeletons are executed sequentially. That can be avoided by providing the user with low-level communication primitives, but this might introduce problems with scheduling and maintainability. Therefore, it is better to allow the programmer to create his own skeletons, possibly based on already existing ones. In the ideal case this would be done in an architecture-independent manner, but for some architectures that is either impossible or inefficient. Thus, there are four levels of abstraction in our programming model:

1. *No abstraction* for the specification of skeletons for restricted architectures or fixed-function blocks.
2. *Abstraction over architecture* for the specification of skeletons for general architectures, such as those capable of executing C.
3. *Abstraction over communication* for the user program that makes use of the skeletons.
4. *Abstraction over syntax* for a graphical programming environment.

[1]Severely limited architectures – like single-ALU processing elements in SIMD systems – may have additional requirements on the skeleton instantiation functions, such as the absence of indirect addressing.

5. DSE FRAMEWORK

Writing the application is only the first step in our framework (see figure 6). The compilation trajectory takes the source code, instantiates the skeletons, extracts a macro dataflow graph, schedules the different skeleton instantiations to the available processors in the architecture template instantiation, and finally compiles the scheduled code for the different processor types. The design space exploration environment finds the most suitable processor architecture by structurally simulating and analyzing the application on different processor architectures (intra-processor optimization loop) and different combinations of processors (inter-processor optimization loop). Finally, the developer can also access the results, and use them to tune his program (source code transformation loop).

5.1. Compilation

The different skeleton instantiations in an image processing application are not fully dependent. Some can be run concurrently, while others can be pipelined. The compiler starts by extracting a macro dataflow graph (*MDG*) from the application in order to analyze the dependencies. It then makes a compromise between data-parallel (within the skeletons) and task-parallel (between the skeletons) execution [27]. It uses a cost model and profiling information to determine the weight of each task.

A higher-order function is a function which takes another function as input. We can use this to abstract over the structure of a certain computation. Consider the following code:

```
for (y=0; y<HEIGHT; y++)
  for (x=0; x<WIDTH; x++)
    out[y][x] = (in[y][x]>128);
```

Using a higher-order function, **PixelToPixelOp**, we can separate the structure from the computation. **PixelToPixelOp** will implement the loops, calling **binarize** every iteration.

```
int binarize(value)
  return (value>128);
void PixelToPixelOp(int in[HEIGHT][WIDTH], int out[HEIGHT][WIDTH], int (*op)(int))
  for (y=0; y<HEIGHT; y++)
    for (x=0; x<WIDTH; x++)
      out[y][x] = op(in[y][x]);

PixelToPixelOp(in, out, binarize);
```

Figure 5: Abstracting over structure using higher-order functions

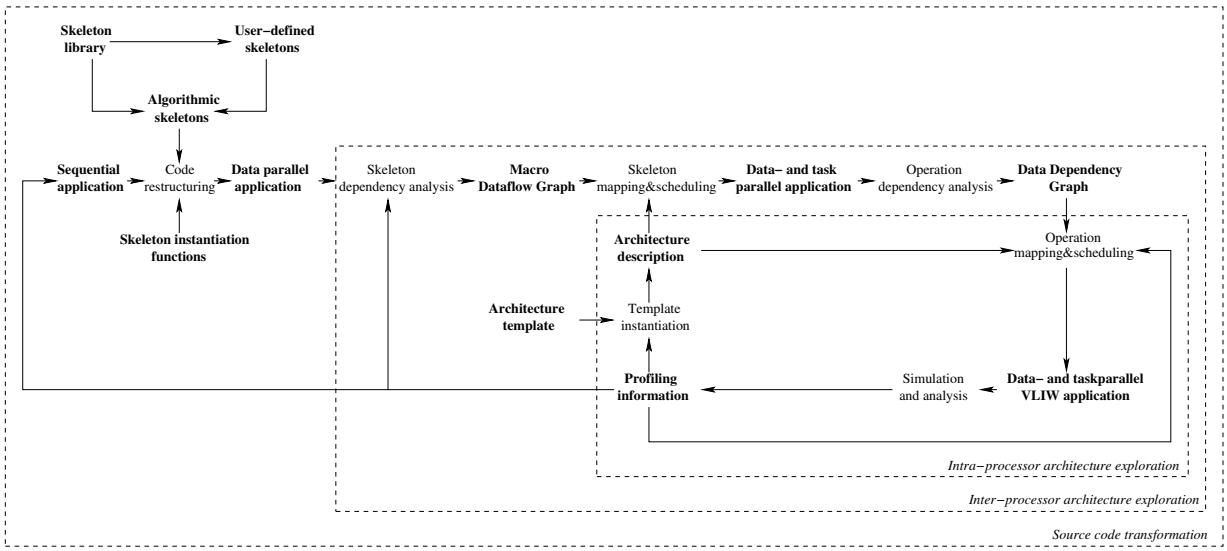

Figure 6: SmartCam design flow. Note the use of algorithmic skeletons to create data-parallel applications from a sequential code image, and the different types of design space exploration to find a suitable architecture.

5.2. Intra-processor optimization

Our architecture template specifies the type of processors that can be used, but not their exact composition, such as the number and types of functional units. The intra-processor optimization step heuristically iterates over the possibilities. Each iteration the application is simulated, and profiling information is used to steer the exploration of the design space [28]. The result is a set of architectures that are optimal with regard to speed, area and power consumption (the pareto-optimal set). The user can then make the trade-off himself.

5.3. Inter-processor optimization

Because the architecture template allows the use of more than one processor, an inter-processor optimization step is needed to find the best mix, and the interconnection between them. This follows much the same strategy as the intra-processor optimization, but a rescheduling of the macro dataflow graph is also necessary.

5.4. Source code transformation

Based on feedback from the profiling done in the optimization steps, the user can decide to rewrite parts of his application. For example, when rewriting a legacy application, he can start by replacing the easiest loops by skeleton instantiations in order to make them execute in parallel. If no architecture can be found that meets his requirements, he can replace more difficult constructions. In this way, it is possible to construct a parallel application with the least amount of effort.

6. DESIGN FLOW EXAMPLE

We will present a simulated design flow example. Suppose that we want to find lines in an image using the Hough transform [29] using as little power as possible, meaning that we want a low clockspeed because that allows us to lower the supply voltage.

The application first convolves the image with an edge detector, then binarizes on the edge strength, and finally transforms the edges to the (ρ, ϕ) space, where each point corresponds to a possible line. The sequential code might look like this:

```
while (1)
 getimage(in);
 for (y=1; y < HEIGHT-1; y++)
  for (x=1; x < WIDTH-1; x++)
   /* Sobel X */
   val=abs(-in[y-1][x-1]-2*in[y][x-1]-in[y+1][x-1]
    +in[y-1][x+1]+2*in[y][x+1]+in[y+1][x+1]);
   /* Sobel Y */
   val+=abs(in[y-1][x-1]+2*in[y-1][x]+in[y-1][x+1]
    -in[y+1][x-1]-2*in[y+1][x]-in[y+1][x+1]);
   trans = {0};
   /* Binarization */
   if (val > 128)
    /* Hough transform */
    for (phi=0; phi < PHI_RES; phi++)
     trans[phi][abs(x*cos(M_PI*phi/PHI_RES)
      +y*sin(M_PI*phi/PHI_RES))]++;
```

Simulating this code on an 8-bus TTA [30] ILP processor (with a lookup-table based Hough transform) results in 5.2 MCycles for a 320x240 image with 32 angles, or 156 MHz for video speed at 30fps (discounting readout and display). The intra-processor optimization step will find that there is no floating-point math, and instantiate the processor accordingly. If this does not meet the power requirements, the most logical step is to use a neighbourhood skeleton for the edge detector, and a pixel skeleton for the binarization:

```
int sobel(int **nbh)
 val = abs(-nbh[-1][-1]-2*nbh[0][-1]-nbh[+1][-1]
  +nbh[-1][+1]+2*nbh[0][+1]+nbh[+1][+1]);
 val += abs(nbh[-1][-1]+2*nbh[-1][0]+nbh[-1][+1]
  -nbh[+1][-1]-2*nbh[+1][0]-nbh[+1][+1]);
 return val;
int binarize(int val) return (val > 128);
while (1)
 getimage(in);
 NeighbourhoodToPixelOp(in, sob, 3, 3, sobel);
 PixelToPixelOp(sob, edge, binarize);
 trans = {0};
 for (y=1; y < HEIGHT-1; y++)
  for (x=1; x < WIDTH-1; x++)
   if (edge[y][x])
    for (phi=0; phi < PHI_RES; phi++)
     trans[phi][abs(x*cos(M_PI*phi/PHI_RES)
      +y*sin(M_PI*phi/PHI_RES))]++;
```

Within our template, the most energy-efficient way of computing neighbourhood and pixel operations is an LPA, and so the intra-processor optimization step instantiates one with 320 processors. By scheduling the application in a pipelined manner, this reduces the critical path to 4.2

MCycles/image, at the expense of a lot of area. Most of the computation is done in the transform, however, and parallelizing it is necessary:

```
int sobel(int **nbh)
  val = abs(-nbh[-1][-1]-2*nbh[0][-1]-nbh[+1][-1]
    +nbh[-1][+1]+2*nbh[0][+1]+nbh[+1][+1]);
  val += abs(nbh[-1][-1]+2*nbh[-1][0]+nbh[-1][+1]
    -nbh[+1][-1]-2*nbh[+1][0]-nbh[+1][+1]);
  return val;
int binarize(int val) return (val > 128);
int hough(int x, int y, int val, int **trans)
  if (val)
    for (phi=0; phi < PHI_RES; phi++)
      trans[phi][abs(x*cos(M_PI*phi/PHI_RES)
        +y*sin(M_PI*phi/PHI_RES))]++;
int add(int val1, int val2) return (val1+val2);
while (1)
  getimage(in);
  NeighbourhoodToPixelOp(in, sob, 3, 3, sobel);
  PixelToPixelOp(sob, edge, binarize);
  AnisoPixelToGlobalReductionOp(edge, trans,
    hough, add);
```

We assume a skeleton **AnisoPixelToGlobalReductionOp** which in the limit constructs a transform for each pixel, and adds them using a *reduction tree*. Because of the large amounts of communication time (the size of the transform times the depth of the tree) this does not scale well, but at 4 ILP processors it reduces the critical path to 1.2 MCycles/image, which may well suit our requirements.

7. CONCLUSION

Based on previous work, we have derived an architecture template and programming model for image processing in smart cameras. The architecture contains LPA and ILP processors, while the programming model is based on algorithmic skeletons. We have presented our DSE framework, which finds an optimal instantiation of the template for a particular application. An example has shown the iterative process in which the user transforms his source code to allow parallelization, and the optimizer finds the best quantity and configuration of processors.

This work is supported by the Dutch government in their PROGRESS research program.

8. REFERENCES

[1] J. Kittler and M.J.B. Duff, Eds., *Image Processing System Architectures*, Pattern Recognition & Image Processing Series. Research Studies Press, 1985.

[2] R.M. Lougheed and D.L. McCubbrey, "The Cytocomputer: A practical pipelined image processor," in *Proceedings of the 7th Annual Symposium on Computer Architecture*, 1980, pp. 271–277.

[3] M.J.B. Duff, *Special Architectures for Pattern Processing*, chapter 4 ("CLIP4"), pp. 65–86, CRC Press, 1982.

[4] S.P. Chambers, *TIPS: A Transputer Based Real-Time Vision System*, Ph.D. thesis, University of Liverpool, 1990.

[5] P.P. Jonker, "Why linear arrays are better image processors," in *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Los Alamitos, CA*. October 1994, vol. III, pp. 334–338, IEEE Computer Society Press.

[6] J.G.E. Olk, *Distributed Bucket Processing - A paradigm for parallel image processing*, Ph.D. thesis, Delft University of Technology, ASCI, September 2001, ASCI Dissertation Series 68.

[7] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "BEOWULF: A parallel workstation for scientific computation," in *Proceedings of the 24th International Conference on Parallel Processing*, 1995, pp. I:11–14.

[8] Message Passing Interface Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, 1994.

[9] P. Carlin, M. Chandy, and C. Kesselman, "The compositional C++ language definition," Tech. Rep., California Institute of Technology, 1993.

[10] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface*, 2002, Version 2.0.

[11] F.J. Seinstra, *User Transparent Parallel Image Processing*, Ph.D. thesis, University of Amsterdam, 2003, ISBN 90-5776-102-5.

[12] C. Soviany, *Embedding Data and Task Parallelism in Image Processing Applications*, Ph.D. thesis, Delft University of Technology, 2003, ISBN 90-9016878-8.

[13] Y.Fujita, N.Yamashita, and S.Okazaki, "IMAP-Vision: An SIMD processor with high-speed on-chip memory and large capacity external memory," in *Proceedings of the 1996 IAPR Workshop on Machine Vision Applications*, M. Takagi, Ed. International Association for Pattern Recognition, 1996.

[14] Philips CFT Industrial Vision, "Vision processor FUGA," website, http://www.cft.philips.com/industrialvision/products/fuga.pdf.

[15] Matrox Imaging, "Matrox GenesisPlus," website, October 2000, http://www.matrox.com/imaging/support/old_products/genesisplus/b_genesi%splus.pdf.

[16] S. Kyo and K. Sato, "Efficient implementation of image processing algorithms in linear processor arrays using the data parallel language 1DC," in *Proceedings of the 1996 IAPR Workshop on Machine Vision Applications*, M. Takagi, Ed. International Association for Pattern Recognition, 1996, pp. 160–165.

[17] Vision Components GmbH, "VC series smart cameras," website, http://www.vision-components.de/products.html.

[18] Fastcom Technology SA, "iMVS series intelligent machine vision systems," website, http://www.fastcom-technology.com/pages/products/process/userguides/iMV%S_Overview_eng.pdf.

[19] DVT Sensors, "Legend series SmartImage sensors," website, http://www.dvtsensors.com/products/LegendManager.php.

[20] Matrix Vision GmbH, "mvBlueLYNX," website, http://www.matrix-vision.com/pdf/mvbluelynx_e.pdf.

[21] A.A. Abbo, R.P. Kleihorst, L.Sevat, P. Wielage, R. van Veen, M.J.R. op de Beeck, and A. van der Avoird, "A low-power parallel processor IC for digital video cameras," in *Proceedings of the 27th European Solid-State Circuits Conference, Villach, Austria*. Carinthia Tech Institute, September 18–20 2001.

[22] Philips CFT, "Inca311 intelligent camera," website, http://www.cft.philips.com/industrialvision/products/inca.pdf.

[23] IVP, "MAPP2500 smart vision sensor," website, http://www.ivp.se/documentation/technology/TheSmartVisionSensor010518.p%df.

[24] W. Caarls, P.P. Jonker, and H. Corporaal, "SmartCam: Devices for embedded intelligent cameras," in *Proceedings of the 3rd PROGRESS workshop on Embedded Systems, Utrecht, The Netherlands*, Mariël Schweizer, Ed. Technology Foundation STW, October 24 2002.

[25] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989, ISBN 0-273-08807-6.

[26] D.K.G. Campbell, "Towards the classification of algorithmic skeletons," Tech. Rep. YCS-276, Department of Computer Science, University of York, 1996.

[27] A. Radulescu, C. Nicolescu, A. van Gemund, and P.P. Jonker, "Cpr: Mixed task and data parallel scheduling for distributed systems," in *Proceedings of the 15th International Parallel and Distributed Symposium*, 2001.

[28] H. Corporaal and J. Hoogerbrugge, "Cosynthesis with the move framework," in *Proceedings of the Multiconference on Computational Engineering in Systems Applications*, P. Borne, G. Dauphin-Tanguy, C. Sueur, and S. El Khattabi, Eds., Lille, France, July 1996, IMACS/IEEE.

[29] P.V.C. Hough, "Machine analysis of bubble chamber pictures," in *International Conference on High Energy Accelerators and Instrumentation*, L. Kowarski, Ed. CERN, September 14–19 1959, pp. 554–556.

[30] H. Corporaal, *Microprocessor architectures: From VLIW to TTA*, John Wiley and Son Ltd, 1998, ISBN 0-471-97157-X.