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

BENCHMARKS FOR SMARTCAM DEVELOPMENT

[1] [2]*Wouter Caarls,* [2]*Pieter Jonker,* [3]*Henk Corporaal*

[1]`wcaarls@ph.tn.tudelft.nl`
[2]Delft University of Technology, Lorentzweg 1, Delft, The Netherlands
[3]Eindhoven University of Technology, Eindhoven, The Netherlands

ABSTRACT

The PROGRESS/STW *SmartCam* project aims to quantify the design of smart camera processors by providing a well-defined design trajectory. This trajectory includes specifying an architecture template, creating a parallelising compiler, and using simulation and analysis tools to find an optimal mapping of the intended application to some instance of the template. In order to design and tune the trajectory, however, we need applications to benchmark and test the tooling. This paper describes the selection and subsequent decomposition of these applications, and we conclude that *algorithmic kernels* can be used to test the architecture and analysis tools, while we need real-world applications to verify the compiler trajectory.

1. INTRODUCTION

Smart cameras are devices that integrate a vision sensor and image processing hardware in a single, small package. Current solutions use off-the-shelf digital signal processors, which are not optimally suited for the image processing domain. In the *SmartCam* project [1], we suggest the use of *SIMD* (single instruction, multiple data) processors for low-level operations such as filtering and colour segmentation, and *ILP* (instruction-level parallel) processors for intermediate- and high level algorithms such as object detection, feature extraction, and classification (see figure 2). SIMD processors have been used for image processing since the early 1970s [2], and more recent miniaturisation efforts have shown them to be useful in near-sensor processing tasks [3, 4]. ILP processors can extract the more irregular operational parallelism in high-level tasks, while task parallelism is exploited by having multiple processors.
Constraints such as processing power, power consumption and cost vary wildly between applications, and thus there is no single solution that fits all needs. We wish to quantify the selection of a suitable architecture by providing an integrated trajectory for the design of smart camera processors. This trajectory will take a C program annotated with parallelisation opportunities and various constraints like frame rate and cost as input, and suggest architecture possibilities by parallelising, simulating and analysing the application (see figure 3). We will base the tools for this project on our previous work in the areas of parameterised ILP design [5], image processing environments [6] and automatic parallelisation [7].

Before starting the integration, enhancement, and SmartCam-specific tuning of the tooling, we first need to research which range of applications we want to support, as this directly influences our architecture template in terms of available operational-, data-, and task-level parallelism and required connectivity. We will then find and extract the algorithmic kernels they use for our simulation and analysis tools. Finally, we need to acquire actual C implementations of these algorithms in order to verify our parallelisation and mapping environment.

2. APPLICATIONS

The application areas of smart cameras range from battery-run toys to high speed industrial applications. We will use the following definition to limit the scope of applications we are interested in: "*A Smart Camera is a small user programmable camera which, in its primary mode of operation, outputs not images but control decisions, symbolic data, and/or small (processed) regions of interest.*". We specifically include programmability, because in our view the continuing evolution of image processing algorithms precludes the sole use of hardwired *ASIC*s. As shown in figure 2, special-purpose function units can of course be used to speed up certain frequently used operations.
The type of output limits the class of applications to those in which not only low-level (image to image) processing is done, but intermediate and perhaps high-level as well. This still covers a host of applications, though; we will focus on the following representative set, which was selected from a larger set of applications with which we have experience [8].

PDAs with integrated cameras for purposes such as owner recognition, localisation, and distributed model building and path planning [9]. These need to be low-power and very small.

Security camera (networks) for tracking and identifying suspects. There could be inter-camera communication for handover purposes.

Industrial inspection tasks such as strip steel inspection might need multiple cameras to cover the product width, and feature extraction over camera boundaries, requiring high-speed interconnections for fast strip speeds (in excess of 60 km/h).

Intelligent transport systems (*ITS*) in cars that use lane detection and object/traffic sign detection for intelligent cruise control. Fast processing is needed for high speeds, and there might be fault tolerance requirements.

This set includes devices with different size-, power-, and processing requirements, as well as a good collection of image processing algorithms. This is important to be able to test the simulation and analysis tools with a wide variety of well-defined algorithmic kernels. As an additional benefit, we have access to actual implementations of these systems, which allows us to test the compilation and parallelisation tools in a real-world environment.

3. ALGORITHMS

We can decompose the chosen applications into a sequence of image processing algorithms. Our architecture template will need to be able to run those algorithms efficiently, such as to minimise power consumption and cost. We thus need to know the available parallelism and the connectivity requirements the algorithms pose; for this we will use the kernels of these algorithms as benchmarks and use the profiling information to steer our architecture template and the instantiation thereof. We need to use this domain-specific kernel set because more general benchmark suites such as EEMBC [10] and MediaBench [11] include lots of compression algorithms, which are very specific and generally not applicable to our domain, and do not include some other important algorithms, such as classifiers.

In the following subsections, we describe a number of algorithmic kernels that are used in the chosen application set. Since they do not interface with other systems except by input and output, it is possible to run them individually on prerecorded data and extract clean profiling information. We have split the kernels according to the prototypical image processing flow of *low-level*, *intermediate-level* and *high-level* tasks (see figure 1).

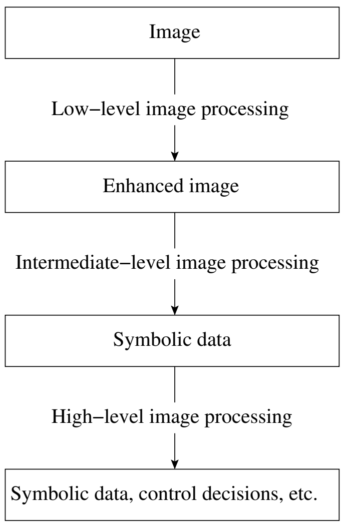
Image
Low−level image processing
Enhanced image
Intermediate−level image processing
Symbolic data
High−level image processing
Symbolic data, control decisions, etc.

Figure 1: Prototypical image processing flow.

3.1. Low-level algorithmic kernels

Low-level image processing algorithms take an image as input and output another image, which has a direct pixel correspondence with the input:

Colour space conversion is usually the first step for segmentation operations, which work better in HSI, YUV, or dedicated colourspaces instead of RGB. It is a point operation, which means that it can be done in parallel for all pixels without communication.

Colour segmentation is used for skin tone recognition for face detection in PDAs and security cameras, and object detection for ITS. This is also a point operation.

Generalised convolution is a superclass of operations like noise removal, edge detection, etc. These operations are used in all applications as preprocessing steps. This is a local neighbourhood operation, meaning that for each pixel information about its immediate neighbourhood is needed for processing.

Distance transform calculates for each pixel the distance to the nearest edge. This is sometimes used as a preprocessing step for template matching, such as in traffic sign detection for ITS. It can be performed in two steps by using a *recursive* neighbourhood operation [12]. This means that some neighbouring *output* pixels are required, limiting the amount of available parallelism.

Histogram equalisation is often used to preprocess images before feeding them to a pattern recognition system, such as in face detection for PDAs and security cameras. Histogramming is a global operation, requiring information from all pixels, but it can be parallelised because histogram levels are independent and the reduction is associative [13].

Stitching the images from multiple cameras together such as in steel inspection requires translation, scaling and rotation operations. Image borders need to be exchanged, and stitched to the image according to some (precalibrated) parameters. Depending on the frame rate, alignment and overlap of the cameras, high-speed interconnect could be necessary.

3.2. Intermediate level algorithmic kernels

Intermedate level image processing extracts objects, features or other information from images, and outputs this in symbolic data structures. Several are used in our application set (only a subset is given here):

Hough transform extracts shapes from an image by transforming it into a parameter space and finding the local maxima in this space [14]. It can be used for contour detection in PDA localisation and lane following in ITS. The transformation contains parallelism that might be difficult to extract, and needs to be analysed. The maximum search is a local neighbourhood operation.

Connected component analysis finds connected sets of pixels belonging to the same object, allowing the extraction of face regions in PDAs and security cameras, or asphalt regions in ITS. The way of presenting these objects to a high-level algorithm can make it difficult to parallelise, while the basic operation is a recursive neighbourhood operation much like in the distance transform.

Centre-of-gravity calculation on segmented or grey-valued images is used for feature extraction in industrial inspection. It is a global operation (within a certain region of interest (*ROI*)), but there is independence between X and Y components, and again associativity in the reduction like in histogramming.

Statistical operations on various low-level transformed images such as gradient magnitude and direction, are also used in feature extraction. There is quite a range of possible operations, but we will stick to average, median and variance, as they are abundant in our steel inspection application.

3.3. High level algorithmic kernels

High level image processing algorithms transform symbolic data structures into other symbolic data, decisions, or control signals. This ranges from simple post-processing to compute-intensive classifiers; we will only use algorithms with well-defined computation intensive kernels:

Coordinate transformations on lines and objects found in the image are needed to convert image coordinates to world coordinates in the localisation of PDAs, and object positioning in security cameras and ITS. This is usually done by multiplying homogeneous-coordinate matrices and vectors, involving numerous multiply-and-accumulate operations[1].

Neural network classifiers are used in face recognition for PDAs and security cameras, and fault detection in industrial inspection. Feeding a large input vector (5000 inputs is not unheard of in face recognition tasks [15]) through the network requires a vast amount of floating point multiply-and-accumulate operations and activation function evaluations. The computation can be done in parallel by rotating the data, but storage requirements for the weights might be an issue.

Support vector classifiers have been used in industrial inspection for the classification of defects. The classification phase is very similar to neural networks, basically consisting of computing and adding kernel functions between the new point and a number of *support vectors*, and is easy to parallelise. Using specialised kernel functions, however, may allow for fixed point solutions.

Path planning is used in PDAs for navigation. Many path planning and search algorithms, such as depth first, breadth first, uniform cost, A*, etc. can be captured in the same kernel and differ only in the order in which nodes are put in an "Open list". Parallelisation problems lie in updating this queue, but algorithms do exist [16].

4. IMPLEMENTATIONS

While using algorithmic kernels is very convenient for architecture testing and simulation, leading to nice task- and data parallelism exploitation, real-world applications are rarely this well implemented. It is thus imperative that we use actual application code to verify our parallelising compiler.
We propose the use of *annotations* in order to mold legacy applications into parallelisable programs, and a set of *library functions* for new and revised applications. Naturally, optimised library routines will allow for the extraction of more parallelism than annotations.

[1]Note that the low-level variant of this algorithm is included in the *Stitching* kernel.

4.1. Annotations

Automatic parallelisation is a very difficult process, requiring the temporal analysis of data dependencies between variables to be successful. Source-code annotations like in High Performance Fortran [17] have since long been used to help the parallelising compiler by explicitly stating independent variables and data distributions. More recently such annotations have been proposed for task parallelism as well [18, 19]. Using annotations instead of a parallel language such as DPCE [20] or CC++ [21] allows the use of any standard compiler for debugging or back-porting purposes, and is thus preferable.
To demonstrate why we need actual application code, we will use a simple example that finds the centre of gravity of a certain colour range in an image. The original code may look as follows:

```
(gx, gy) = Function SegmentGrav (image, bounds)
  gx = gy = n = 0;
  for (0 ≤ y < HEIGHT)
    for (0 ≤ x < WIDTH)
      if (InBounds(image[y][x], bounds))
        gx = gx + x; gy = gy + y; n = n + 1;
  gx = gx / n; gy = gy / n;
```

Original centre-of-gravity code for one image

By observing that **InBounds** is a pure function, and that the loop iterations are thus independent except for the counter variables *gx*, *gy*, and *n*, which are updated using a commutative and associative operator (and thus computable using a *reduction tree*), we can parallelise this loop by adding some HPF-like annotations:

```
(gx, gy) = Function SegmentGrav (image, bounds)
  #pragma DISTRIBUTE image(BLOCK,
                           BLOCK)
  gx = gy = n = 0;
  #pragma INDEPENDENT, NEW(x),
          REDUCTION(gx, gy, n)
  for (0 ≤ y < HEIGHT)
    #pragma INDEPENDENT
    for (0 ≤ x < WIDTH)
      if (InBounds(image[y][x], bounds))
        gx = gx + x; gy = gy + y; n = n + 1;
  gx = gx / n; gy = gy / n;
```

Data parallel code, based on High Performance Fortran. Note that the top loop is only independent if a different *x* is chosen for each iteration, hence the `NEW(x)`.

Note, however, that executing **InBounds** and updating the variables is not done concurrently, as splitting this dependency can only be done by pipelining the operations across a sequence of images, and this requires rewriting the code. It is expected that this situation will occur frequently in actual applications, which are often optimised for sequential processing. If not enough task parallelism is exploitable through just the addition of annotations, one should consider rewriting parts of the code using the library functions proposed in the next section.

4.2. Image processing library

For writing new applications, and for revising unstructured legacy applications, using optimised library routines is much more efficient than (re)writing your own algorithms. An added benefit is that the resulting code will have distinct functions with well-defined interfaces, allowing for easy exploitation of task parallelism.
Soviany [22] proposes the use of *algorithmic skeletons* for such library functions. These skeletons specify the structure of computation, while functions passed to the skeleton implement the computation itself. As an example, a monadic point operation with one input image and one output image has a structure in which each pixel is processed independently, in any order. A colour segmentation algorithm would then specify only the segmentation of one individual pixel, and pass this function to the skeleton. The mapper and scheduler can then use the properties of the structure of computation for the actual data parallel execution.
Using skeletons, and task parallelism annotations based on HPF 2.0 [19] for our segmentation and centre-of-gravity example could yield the following code:

```
Subroutine Segment(in, bounds, out)
  out = InBounds(in, bounds)

Subroutine Grav (in, x, y, {gx, gy, n})
  if (in) gx = gx + x; gy = gy + y; n = n + 1;

(gx, gy) = Function SegmentGrav (image, bounds)
  gx = gy = n = 0;
  #pragma TASK_REGION
    #pragma ON(SET_1)
      PointOp(image, segout, bounds, Segment)

    gravin = segout

    #pragma ON(SET_2) BEGIN
      PointReductionOp(gravin, {gx, gy, n}, Grav)
      gx = gx / n; gy = gy / n;
    #pragma END ON
  #pragma END TASK_REGION
```

Data- and task parallelised code using skeletons. The two skeletons can run independently, while the communication (*gravin* = *segout*) must be executed by all processors.

While it is not possible to execute the tasks in parallel in this specific example, with the given annotations it can be extended to be used in a pipelined environment: if *image* were an image source a **while** `(1)` within the `TASK_REGION` would do.

5. CONCLUSIONS

We have shown that representative benchmarks are necessary to define a suitable architectural template for image processing, to test architecture analysis tools, and to verify a parallelising compiler trajectory for our SmartCam environment. We have argued that while deriving *algorithmic kernels* from application domains simplifies the analysis for purposes of architecture definition and tooling, it is not sufficient for verifying the trajectory an application developer follows using our parallelising compiler. For this purpose, we will use real-world implementations of our chosen set of applications.

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

6. REFERENCES

[1] 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.

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

[3] 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.

[4] S. Kyo, “A 51.2GOPS programmable video recognition processor for vision based intelligent cruise control applications,” in *Proceedings of the 2002 IAPR Workshop on Machine Vision Applications*, K. Ikeuchi, Ed. International Association for Pattern Recognition, December 11–13 2002, pp. 632–635.

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

[6] C. Nicolescu and P.P. Jonker, “EASY PIPE - an ”EASY to use” Parallel Image Processing Environment based on algorithmic skeletons,” in *Proceedings of the Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (held in conjunction with IPDPS)*, 2001.

[7] I. Karkowsi and H. Corporaal, “FP-MAP - An approach to the functional pipelining of embedded programs,” in *Proceedings of the 4th International Conference on High Performance Computing*, V. Prasanna, Ed., December 1997.

[8] W. Caarls, “Testbench algorithms for SmartCam,” Tech. Rep., Delft University of Technology, 2003.

[9] R.L. Lagendijk and P.J. van Vliet, “CACTUS impulse research project,” Website, 2002, `http://www.cactus.tudelft.nl`.

[10] Embedded Microprocessor Benchmark Consortium, “Embedded microprocessor benchmark consortium website,” Website, `http://http://www.eembc.org/`.

[11] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems,” in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997, pp. 330–335.

[12] G. Borgefors, “Distance transforms in digital images,” *Computer Vision, Graphics and Image Processing*, vol. 34, pp. 344–371, 1986.

[13] 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.

[14] 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.

[15] J. Haddadnia, K. Faez, and P. Moallem, “Human face recognition with moment invariants based on shape information,” in *Proceedings of the International Conference on Information Systems, Analysis and Synthesis*, Orlando, Florida, USA, 2001, International Institute of Informatics and Systemics, vol. 20.

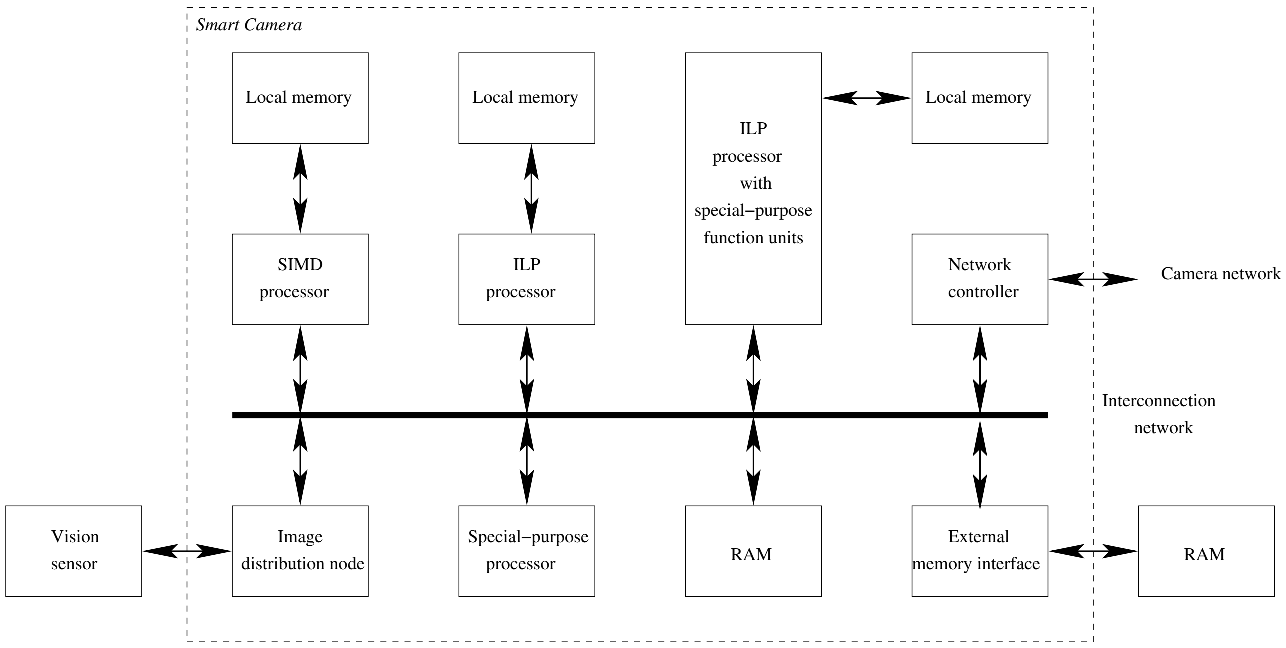

Figure 2: A possible architecture template for a smart camera device, containing SIMD, ILP, and special-purpose processors. All components, including the interconnection network, are subject to adjustment by the architecture exploration.

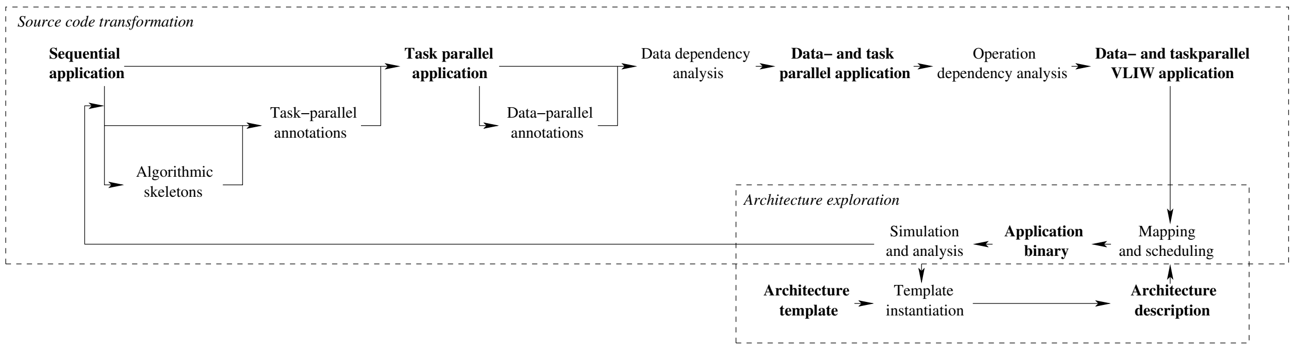

Figure 3: SmartCam design flow, with the integration of source code transformation and architecture exploration.

[16] G.S. Brodal, J.L. Träff, and C.D. Zaroliagis, "A parallel priority data structure with applications," in *Proceedings of the 11th International Parallel Processing Symposium*, 1997, pp. 689–693.

[17] High Performance Fortran Forum, *High Performance Fortran Language Specification*, 1993, Version 1.0.

[18] T. Gross, D. O'Hallaron, and J. Subhlok, "Task parallelism in a high performance fortran framework," *IEEE Parallel & Distributed Technology*, vol. 2, no. 2, pp. 16–26, 1994.

[19] High Performance Fortran Forum, *High Performance Fortran Language Specification*, 1997, Version 2.0.

[20] Numerical C Extensions Group of X3J11 DPCE Subcommittee, "Data parallel C extensions," Tech. Rep., ANSI, December 1994.

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

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