IEICE TRANS. INF. & SYST., VOL.E89–D, NO.7 JULY 2006

PAPER Special Section on Machine Vision Applications

Skeletons and Asynchronous RPC for Embedded Data and Task Parallel Image Processing*

Wouter CAARLS[†a)], ***Student Member***, **Pieter JONKER**[†b)], ***and*** **Henk CORPORAAL**[††c)], ***Nonmembers***

SUMMARY Developing embedded parallel image processing applications is usually a very hardware-dependent process, often using the single instruction multiple data (*SIMD*) paradigm, and requiring deep knowledge of the processors used. Furthermore, the application is tailored to a specific hardware platform, and if the chosen hardware does not meet the requirements, it must be rewritten for a new platform. We have proposed the use of *design space exploration* [9] to find the most suitable hardware platform for a certain application. This requires a hardware-independent program, and we use *algorithmic skeletons* [5] to achieve this, while exploiting the data parallelism inherent to low-level image processing. However, since different operations run best on different kinds of processors, we need to exploit task parallelism as well. This paper describes how we exploit task parallelism using an asynchronous remote procedure call (*RPC*) system, optimized for low-memory and sparsely connected systems such as smart cameras. It uses a *futures* [16]-like model to present a normal imperative C-interface to the user in which the skeleton calls are implicitly parallelized and pipelined. Simulation provides the task dependency graph and performance numbers for the mapping, which can be done at run time to facilitate data dependent branching. The result is an easy to program, platform independent framework which shields the user from the parallel implementation and mapping of his application, while efficiently utilizing on-chip memory and interconnect bandwidth.

key words: *design space exploration, heterogeneous architectures, constrained architectures, algorithmic skeletons, remote procedure call, futures, run-time scheduling*

1. Introduction

As processors are becoming faster, smaller, cheaper, and more efficient, new opportunities arise to integrate them into a wide range of devices. However, since there are so many different applications, there is no single processing device that meets all the requirements of every application. The SmartCam project [9] investigates how an application-specific processing device can be generated for the specific field of intelligent cameras, using design space exploration.

The processing done on an intelligent camera has very specific characteristics. On the one hand, low-level image processing operations such as interpolation, segmentation and edge enhancement are local, regular, and require vast amounts of bandwidth. On the other hand, high-level operations like classification, path planning, and control may be irregular while typically consuming less bandwidth [2]. The architecture template, on which the design space exploration is based, therefore contains data-parallel (SIMD) as well as instruction-parallel (ILP) processors.

One of the main goals of the project is keeping the system easy to program. This means that one single program should map to a wide range of configurations, of a wide range of processors. It also means that the application developer shouldn't have to learn a parallel programming language. The solution presented below is based on using algorithmic skeletons to exploit data parallelism within each operation, while a form of asynchronous RPC allows the operations to run concurrently.

The structure of this paper is as follows: Sect. 2 reviews some related work. Section 3 presents our prototype architecture, while Sects. 4 and 5 describe our programming environment and some optimizations. Section 6 details our implementation, and Sect. 7 presents some results from our prototype. Finally, Sect. 8 draws conclusions and points to future work.

2. Related Work

Even restricting ourselves to systems that fit inside a camera, there exist many different image processing architectures:

- **DSPs**, (VLIW) processors optimized for signal processing, like the Texas Instruments TMS320C6x series and the Philips TriMedia.
- **Vector architectures**, scalars or superscalars with an SIMD coprocessing unit, like the Berkeley VIRAM [11] project, and to a lesser degree Intel's MMX/SSE and Motorola's AltiVec.
- **SIMD arrays**, among many others, NEC IMAP [19] and Philips XeTaL [1].
- **FPGAs**, which can implement operations in hardware.

Recently, SIMD arrays of VLIW processors have also been developed, like NEC's IMAP-CE [12], and the Stanford Imagine [17] architecture. Many systems also contain a RISC processor for control and OS tasks.

Of these architectures, pure SIMD arrays are the most suited for low-level image processing, because they can most efficiently exploit the largest degree of regular data parallelism. Superscalars and VLIWs work best with irregular problems, allowing different instructions to run within

Manuscript received November 1, 2005.
Manuscript revised January 23, 2006.

[†]The authors are with the Quantitative Imaging Group, Delft University of Technology, The Netherlands.
[††]The author is with the Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands.

*This work is supported by the Dutch government in their PROGRESS research program under project EES.5411.

a) E-mail: w.caarls@tudelft.nl
b) E-mail: p.p.jonker@tudelft.nl
c) E-mail: h.corporaal@tue.nl

DOI: 10.1093/ietisy/e89–d.7.2036

one cycle. Vector architectures make a compromise to perform well on both domains, but in a multiprocessor system it makes sense to choose domain-specific architectures for best efficiency. Finally, FPGAs can be used for tasks that do not map well to any of the other architectures.

Programming environments for image and signal processing applications are also widely ranged. Tightly coupled systems usually have parallel extensions to a sequential language, like Celoxica's Handel-C [4] for FPGA programming, or 1DC [13] for the IMAP cards. More loosely coupled systems usually work with the concept of a *task* or *kernel*, and differ in how these tasks are programmed and composed.

Process networks such as YAPI [6] allow much freedom in specifying the tasks, but they are statically connected. StreamC/KernelC [14], developed for Imagine, reduces the allowed syntax within a kernel, but makes the interconnections dynamic by using *streams*. Their current implementation doesn't support task parallelism, however. EASY-PIPE [15] does, but requires a batch of tasks to be explicitly compiled and dispatched by the user. Their main contribution is the use of algorithmic skeletons to make programming the tasks easier. Finally, Seinstra [18] allows no user specification of the tasks, instead relying on an existing image processing library. It is also limited to data parallelism, but these restrictions allow it to be more transparent to the user, presenting a purely sequential model.

Futures were introduced in the MultiLisp [7] language for shared-memory multiprocessors. Requesting a future spawns a thread to calculate the value, while immediately returning to the caller, which only blocks when it tries to access it. Once the calculation is complete, the future is overwritten by the calculated value. Batched futures [16] apply this concept to RPC, but with the intent to reduce the amount of RPC calls by sending them in batches that may reference each other's results.

Our system uses RPC to support dynamic task parallel stream programs, while the kernels are programmed using algorithmic skeletons. Our futures can reference results from operations that are executed at *different* RPC servers. Using these mechanisms, much parallelism can be exploited even when it is not specifically kept in mind, but for full efficiency some care will be needed.

3. Architecture

Our prototype architecture is the Philips CFT Inca+ prototype (see Fig. 1). This is a minimal implementation of our architecture template, consisting of one XETAL [1] SIMD processor and one TriMedia VLIW processor; they can run concurrently to exploit task parallelism. There is one video speed channel from the sensor to the XETAL and one video speed channel from the XETAL to the TriMedia. The TriMedia can program the XETAL via the I2C bus. The architecture is described in more detail in [10], and is schematically summarized in Fig. 2.

The XETAL chip consists of 320 PEs and a control processor, running at pixel clock. At VGA resolution with a pixel clock of 16 MHz and 30 fps, it can process more than 500 instructions per pixel, and has enough memory to store 16 image lines. The TriMedia is a 5-way VLIW processor running at 180 MHz. At the same video speed that means around 100 operations per pixel, but the pixel accesses may be irregular. An external 32 MB SDRAM provides enough storage for most applications at this resolution. This architecture is suited for image processing because it takes advantage of the fact that image processing applications progress from low-level, high-bandwidth operations to high-level, low-bandwidth operations. Due to the exploited parallelism, its power efficiency is vastly improved over normal microprocessors.

One drawback is that, because there is no channel from TriMedia to XETAL, the TriMedia cannot be used as a temporary frame store. This will be remedied in a new prototype

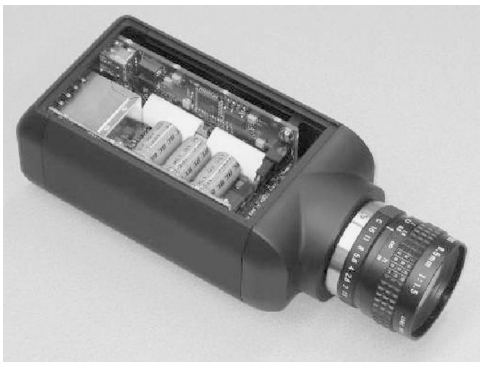

Fig. 1 Inca+ prototype smart camera.

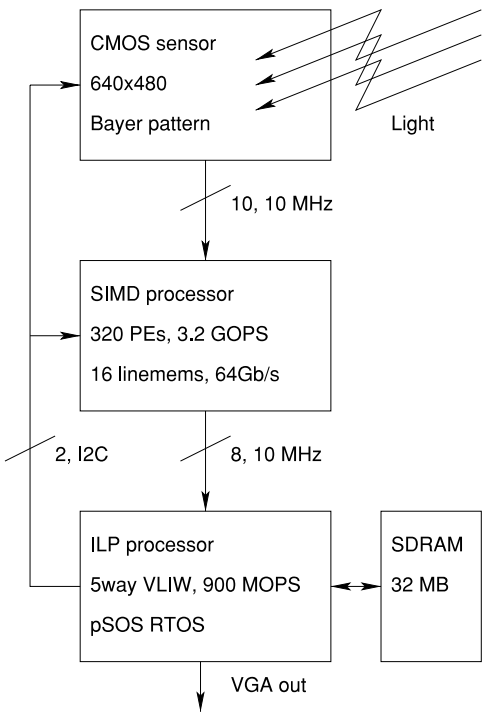

Fig. 2 Inca+ prototype architecture.

platform that is under development. It contains an FPGA for routing and intermediate-level processing, and the XETAL may also be decoupled from the pixel clock, achieving 7.7 10-bit GMACs per second.

4. Programming

Our programming environment is based on C, to provide an easy migration path. In principle, it is possible (although slow) to write a plain C program and run it on our system. In order to exploit concurrency, though, it is necessary to divide the program into a sequence of image processing operations, and to string these together using function calls. Parts of the program which cannot easily be converted can be left alone unless the speedup is absolutely necessary.

The main program, which calls the operations and includes the unconverted code, is run on a *control processor*, while the image processing operations themselves are run on the *coprocessors* that are available in the system (the control processor may also act as a coprocessor). Only this main program can make use of global variables; because of the distributed nature of the coprocessor memory, all data to and from the image processing operations must be passed using parameters.

4.1 Within-Operation Parallelism

The main source of parallelism in image processing is the locality of pixel-based operations. These low-level operations reference only a small neighborhood, and as such can be computed mostly in parallel. Another example is object-based parallelism, where a certain number of possible objects or regions-of-interest must be processed. Both cases refer to *data parallelism*, where the same operation is executed on different data (all pixels in one case, object pixels or objects in the other).

Data parallel image processing operations map particularly well on linear SIMD arrays (LPAs, [8]). However, since we don't want the application developer to write a parallel program, we need another way to allow him to specify the amount of parallelism present in his operations. For this purpose, we use *algorithmic skeletons*. These are *templates* of a certain computational flow that do not specify the actual operation, and can be thought of as higher-order functions, repeatedly calling a *kernel function* for every computation. Take the very simple binarization in program 1:

Program 1 Binarization

```
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 the **binarize** kernel every iteration. See program 2.

Program 2 Binarization implemented using a higher-order function

```
int binarize(int value)
  return (value>128);

void PixelToPixelOp(int (*op)(int),
int in[HEIGHT][WIDTH], int out[HEIGHT][WIDTH])
  for (y=0; y<HEIGHT; y++)
    for (x=0; x<WIDTH; x++)
      out[y][x] = op(in[y][x]);

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

Note that implementing **PixelToPixelOp** column-wise instead of row-wise – by reversing the loops – does not change the result, because there is no way for **op** to reference earlier results (side effects are not allowed). It can be said that by specifying the inputs and outputs of the kernel function, the skeleton characterizes the available parallelism. So, by choosing a skeleton, the programmer makes a statement about the parallelism in his operation, while not specifying how this should be exploited. This freedom will allow us to optimally map the operation to different architectures.

Another benefit is that the image processing library normally shipped with DSPs and other image processors is replaced by a skeleton and kernel library, which is more general and thus less in need of constant updates, since the skeletons it provides and uses for its own kernels can be used by the application developer as well. Of course, highly specialized operations can still be provided in a fixed library, provided they meet the requirements of the framework discussed in Sect. 5.2.

Not all operations can be data-parallelized as easily as pixel operations; more irregular operations place increasing demands on the autonomy and interconnection of the processing elements. For example, for efficient implementation, local neighborhood operations are straightforward, recursive neighborhood operations require indirect addressing, run-length encoding requires non-local communication, and edge following is mostly sequential. However, even in the sequential case the skeleton approach can still be used, if only to facilitate programming instead of parallelization. If specialized hardware then becomes available, it is easier to make use of it.

4.2 Between-Operation Parallelism

An image processing application consists of a number of operations described above, surrounded by control flow constructs. In order to provide an easy migration path, these operations can be called as higher-order functions, although the kernel functions are inlined at compile-time to ensure efficiency. Furthermore, because our hardware platform is heterogeneous, it is important that multiple of these operations are run concurrently, as not all processors can be working on the same computation. We are therefore using asynchronous RPC calls as a method to exploit this task-level parallelism.

In RPC, the *client* program calls *stubs* which signal a *server* to perform the actual computation. In our case, the application is the client program running on the control processor, while the skeleton instantiations are run on the co-processors. This alone does not imply parallelism, because the stub waits for the results of the server before returning. In asynchronous RPC, therefore, the stub returns immediately, and the client has to *block* on a certain operation before accessing the result. This allows the client program to run concurrent to the server program, as well as multiple server programs to run in parallel, as shown in program 3.

Program 3 Running client and server code in parallel using asynchronous RPC

```
IMAGE in, out1, out2;
Read(in);
_block(Read);
PixelToPixelOp(op1, in, out1);
PixelToPixelOp(op2, in, out2);
/* ...Concurrent client code ...                    */
_block(op1);
_block(op2);
```

However, this still has the disadvantage of requiring the client program to wait on the completion of *in* before proceeding, even though it never uses the results except to pass them on to other RPC calls. To address this problem, MultiLisp [7] introduced the concept of *futures*, placeholder objects which are only blocked upon when the value is needed for a computation. Since simple assignment is not a computation, passing the value to a function doesn't require blocking; once the called function needs the information, it will block itself until the data is available, without blocking the client program. Program 4 demonstrates this.

Program 4 Using futures to avoid blocking when passing variables between RPC calls

```
IMAGE in, out1, out2;
SCALAR out;
Read(in);
PixelToPixelOp(op1, in, inter1);
PixelToPixelOp(op2, in, inter2);
PixelReductionOp(op3, inter1, inter2, out);
/* ...Concurrent client code ...                    */
_block(out);
/* ...Dependent client code ...                     */
```

In this piece of code, **PixelReductionOp** still cannot run in parallel with both **PixelToPixelOp**s, though, because it has to wait for their output (the *inter1* and *inter2* frames) to become available, even though the client program can continue. Indeed, the RAW dependency makes it impossible to run them concurrently on the same image, but it *is* possible to run them concurrently on *different* images. We therefore introduce the concept of a *composite operation*, which behaves the same as a normal operation, except that it consists of more than one sub-operation. As such, it may wait for data independently of the calling program; see program 5.

Program 5 Using composite operations to avoid blocking the entire program when only a part has to wait

```
ProcessImage(IMAGE in)
  IMAGE inter1, inter2;
  SCALAR out;
  PixelToPixelOp(op1, in, inter1);
  PixelToPixelOp(op2, in, inter2);
  PixelReductionOp(op3, inter1, inter2, out);
  /* ...Concurrent client code ...                  */
  _block(out);
  /* ...Dependent client code ...                   */

IMAGE in;
while(1)
  Read(in);
  CompositeOp(ProcessImage, in)
  /* ...More client code ...                        */
```

In this instance, that allows us to run different stages of different images in parallel, because the next **Read** (and **ProcessImage**, once **Read** finishes) can start right after the previous one finished, instead of having to wait for the processing. Thus, in a fully loaded pipeline, **Read** is working on one image, the two **PixelToPixelOp**s on another, and **PixelReductionOp** on a third.

Note that the functions are called separately for each image frame. As such, function parameters and their sequencing can dynamically change each frame, possibly based on the result of earlier calls.

5. Optimizations

While our futures-like implementation is much less elaborate than MultiLisp's (requiring, for example, explicit blocks on results, although these could be inserted by the compiler), it does tackle two other problems: data distribution and memory usage. Both originate from our architecture template, which features distributed-memory processors with a relatively low amount of on-chip memory.

Furthermore, although the skeletons are called as higher-order functions in order to provide an easy migration path, we avoid the function call overhead by using *source-to-source transformations*. Using transformations also allows us to translate between different target processor languages, and to provide an efficient way to specify skeletons that are polymorphic in the number and types of their arguments. It also allows us to efficiently *merge* different operations.

5.1 Data Distribution

The data generated by most image processing operations is not accessed by the client program, but only by other operations. This data should therefore not be transported to the client (control) processor. In order to achieve this, we make a distinction between images and other variables.

Images are never sent to the control processor unless the user explicitly asks for them, and as such no memory is allocated and no bandwidth is wasted. Rather, they are

transported between coprocessors directly, thus avoiding the scatter-gather bottleneck present in some earlier work [15]. All other variables (thresholds, reduction results, etc.) are gathered to the control processor and distributed as necessary. These can be used without an explicit request.

The knowledge about which data to send where simply comes from the inputs and outputs to the skeleton operations: image variables are used to specify the connections between operations instead of actually holding images, and are also known as *streams*. Coprocessors are instructed to send the output of an operation to all peers that use it as an input.

5.2 Memory Usage

Our concern about memory usage stems from the fact that especially SIMD linear processor arrays for low-level image processing may not have enough memory to hold an entire frame, let alone multiple frames if independent tasks are mapped to it. These processors are usually programmed in a pipelined way, where each line of an image is successively led through a number of operations. We would like our system to conserve memory in the same way, and have therefore specified all our operations to read from and write to *FIFO buffers*.

The distribution mechanism allocates these buffers, and sets up transports as described above. The operations themselves read the needed information from the buffer, process it, and write the results to another buffer. A method is provided to signal that no more data will be forthcoming. This conserves memory, because even a series of buffers is generally much smaller than a frame. Simultaneously, it hides the origin of the data, making the operations independent of the producers of their input and the consumers of their output.

As an added advantage, this buffer scheme results in more fine grained task parallelism. Instead of working on two *different* images, operations can work on different *parts* of the *same* image, and this benefits the latency of the output. Of course, this only works if the operations support piece-by-piece processing; a frame memory is still required if they do not. Fortunately, many low-level operations support this.

5.3 Skeleton Merging

It is advantageous, both for reuse and the freedom of mapping, to split a program into as many kernels as possible. However, this results in a lot of scheduling overhead, especially on SIMD processors where hundreds of instructions are executed each cycle. On ILP processors it significantly reduces the amount of exploitable instruction-level parallelism.

We avoid this by merging operations that use compatible skeletons. Our compiler finds sequences of compatible skeleton calls in the application program, and uses information carried by the skeletons to merge them. Apart from specifying how to order the instructions, this information also contains the appropriate buffer sizes and delays, so that the synchronization of the operations is preserved. Currently, we can merge pixel operations on the one hand, and neighborhood and recursive neighborhood operations on the other.

After merging, the sequence of skeleton calls is replaced by a single call to the merged operation. Of course, this should only happen when the operations are always run together, in the same order and on the same processor. The ordering is preserved by only merging skeleton calls in the same *basic block* of the program, while the mapping information can be gathered from simulation.

6. Implementation

The skeleton instantiation and merging is discussed elsewhere [3], and in this paper we will discuss only our RPC system. This system has been implemented on both a network of workstations (*NOW*) and the Inca+ prototype. The traces in this section were generated on the NOW, while the performance figures in Sect. 7 are gathered from the prototype implementation.

The library consists of the following components: a *front-end* that enqueues operations, a *mapper* that maps operations to processors, and a *dispatcher* that dispatches operations, variables, and sets up buffers and transports. If administrative data is not destroyed, a *trace generator* can write a trace once the program finishes. This trace can be used to benchmark individual operations to assist the mapping.

6.1 Enqueueing

Each call to an RPC stub enqueues that operation in a list. All arguments are passed *by reference*, and are tracked by their address. As such, an operation that uses a variable that hasn't been produced yet can be marked as a consumer of that variable, such that it will be sent as soon as it is available. Note that this means our concept of futures is limited to their use in stub calls, and using an output variable outside of one requires an explicit **_block** (although these may be inserted by a compiler).

As a consequence of the use of buffers, it is necessary to know how many consumers a stream will eventually have, because the data in the (cyclic) buffer may not be overwritten unless it has been read by all consumers. We have chosen to require a *finalization* once all consumers have been specified. This makes it easy to conditionally add readers.

Composite operations are a special case. These are implemented using *threads*, but we want them to behave as much as function calls as possible. This means that their arguments should be passed *by value*, and thus we need to make explicit copies of their input arguments. If such an input is a future, a new *instance* is created, which allows the composite operation to independently **_block** on it. The same goes for stream inputs and outputs, which must be independently **_finalize**d (although there is an implicit final-

ization at the end of the operation). Composite operations may be arbitrarily nested.

We use a *single assignment* semantic, such that each buffer only has one writer. This means that if an output variable is reused, it points to a *different* buffer.

6.2 Mapping

Mapping means selecting the best processor for an operation to run on, optimizing throughput and latency. A *static* mapping can be generated if the course of the program is known. This is the case in parts of the program between data-dependent branches, called *basic blocks*. A problem is that we cannot finish one basic block before starting the next, as this requires a frame memory for every image that is passed between the two, and would effectively reduce task parallelism to zero at each branch.

Fortunately, the granularity of our operations is quite large (operating on images instead of single values), so we can spend a bit of time determining a mapping *dynamically* at run time. Currently, we employ a greedy strategy that will map an operation to the processor that most quickly produces its outputs, according to a simple network flow model. This model assumes that a processor's cycles or a connection's bandwidth are equally distributed over all operations or streams that are mapped to it. If an operation or stream can't use its share, the excess is distributed over the other contenders, etc.

6.3 Dispatching

After an operation has been mapped to a processor, it can be dispatched. During skeleton instantiation, each operation is assigned an identifier, and this is first sent to the coprocessor. Next, stream arguments are checked; if the stream doesn't have a buffer on the processor, a new buffer is created, and a transport is set up between the source buffer and the newly created one. Then, the buffer id is sent. Finally, non-stream input arguments are marshalled and sent as well.

The operation starts as soon as all its arguments have been received. It will run until it blocks because of reading an empty buffer, or writing a full buffer. The coprocessor then selects a new task that isn't blocked, and so on. In this way, the buffer sizes define the task switching granularity.

Even the SIMD processors run a (very light-weight) task scheduler. However, because XeTaL runs synchronous to the pixel clock, the operations cannot block, and always process one line of the image.

Once the operation has finished, it signals the control processor, and transmits its scalar results, if any. If these results are needed by another operation, they are sent back by the control processor. Any futures referencing the results are resolved by copying the data to their addresses, and threads blocking on them are unblocked.

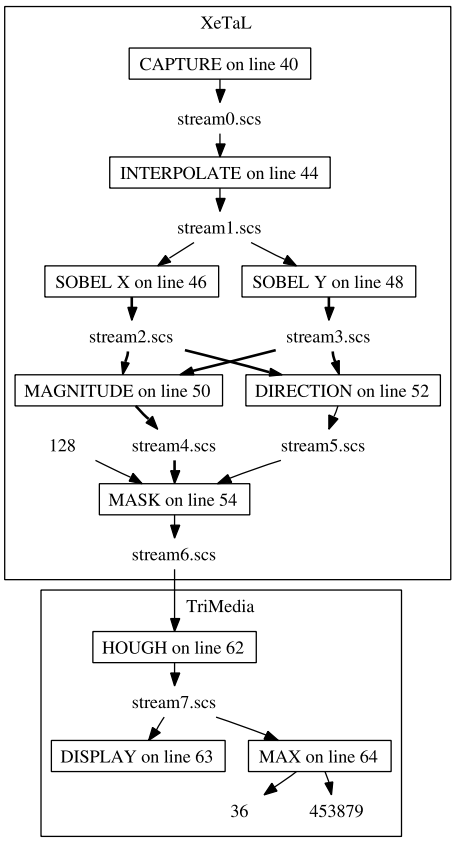

Fig. 3 Trace of a ball detection algorithm.

6.4 Trace Generation and Benchmarking

The mapping process needs information about the performance of each operation on each processor. This requires benchmarking all operations independently. By keeping a trace and storing all streams during a simulation run, we can generate the required information for such an independent benchmark: if we preload the input streams and preallocate output streams, we can get an estimated performance figure in the absence of network bandwidths and competing operations.

Figure 3 shows such a trace, with operations in boxes. Buffers are unboxed, and show the name of the file which stores the content; the line numbers refer to the place of the operation in the client program.

7. Results

We have implemented the ball detection algorithm of Fig. 3 on the architecture described in Sect. 3. In this algorithm, the Bayer pattern sensor output is first interpolated; then the Sobel X and Sobel Y edge detection filters are run and combined to get the edge direction and magnitude. The edge direction is masked when the magnitude is below a certain threshold, and around the remaining edge pixels a circle segment is drawn in a Hough accumulation space. The maxi-

Table 1 Timing results of the ball detection algorithm.

Trial	Processing time
Single operation (TriMedia)	100 ms
Split operations (TriMedia)	160 ms
Merged operations (TriMedia)	134 ms
Parallel (XeTaL+TriMedia)	54 ms

mum of the accumulation space is taken, and used as the position of the ball.

The Hough transform cannot run on the XeTaL, because it requires a frame memory and dynamic indirect addressing. As there is no channel back to the SIMD processor in our prototype architecture, this means the maximum search of the Hough transform has to be performed on the TriMedia as well.

Four situations were compared: one in which the entire algorithm was implemented in a single operation on the TriMedia, as a baseline for how a sequential application would be written. Next, the operation was split into tasks as shown in the trace, and all tasks were mapped to the TriMedia; this shows the overhead caused by the task switching and buffer interaction. Then, we applied skeleton merging to bring down the overhead. Finally, all low-level operations were mapped to the XeTaL, while the Hough transform, maximum search and display were mapped to TriMedia; this resembles the situation as it would run on our system.

Because XeTaL only has 16 line memories, the buffers between the filters were 1 line. On the TriMedia, they were 25 lines, to reduce context switching. An allocate-and-release scheme was used on the TriMedia, so that no extra state memory was needed in the filters, and no unnecessary copies were made. See Table 1.

As can be seen, a fine-grained kernelization of the algorithm results in a significant overhead. By merging the kernels, allowing the compiler to exploit instruction parallelism and data locality as well as avoiding context switching, we reduce the overhead by half. Finally, by mapping all the low-level operations to the SIMD processor, the program runs almost twice as fast as the ILP-only version.

Note that in the last case the filtering and transform are done task parallel, and the processing time is bounded by the slowest operation, which is the transform (if not run in parallel, the pixel speed links would add an extra frame-time). In fact, the speedup depends very much on the input image and types of operations. In the case of the ball detection algorithm on our prototype, the SIMD processor is only processing at half speed. Amdahl's law applies, and in order to increase the throughput of applications with large sequential parts, we need a faster ILP processor, or to overlap the computation of multiple frames on different processors.

The effectiveness of skeleton merging on the SIMD speed cannot be gathered from this application, as its speed is constrained by the (synchronous) pixel clock. However, a cycle-accurate simulator has verified that the dynamic instruction count is reduced by a factor of 1.35, and this will be larger for asynchronous systems.

8. Conclusions and Future Work

We have presented a system in which an application developer can construct a parallel image processing application with minimal effort. Data parallelism is captured by specifying the way to process a single pixel or object, with the system handling distribution, border exchange, etc. Task parallelism of these data parallel operations is achieved through an RPC system, preserving the semantics of normal function calls as much as possible. Results from an actual prototype architecture have shown that the system works, and can achieve a significant speedup by using an SIMD processor for low-level vision processing.

Currently, streams are dynamically typed, and this can result in runtime errors. We wish to investigate static typing of streams, and type promotion rules. Secondly, we wish to include memory considerations in the mapping decision, so that operation combinations requiring too much memory will not be mapped to memory-constrained processors. We are also developing a new hardware platform and SIMD processor that will allow us to investigate more complex and demanding applications, such as real-time stereo vision.

References

[1] 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," Proc. 27th European Solid-State Circuits Conference, pp.137–140, Carinthia Tech Institute, Villach, Austria, Sept. 2001.

[2] W. Caarls and P.P. Jonker, "Benchmarks for smartcam development," Proc. ACIVS 2003, Ghent University, pp.81–86, Sept. 2003.

[3] W. Caarls, P.P. Jonker, and H. Corporaal, "Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications," Proc. 20th IEEE International Parallel and Distributed Processing Symposium, April 2006 (published).

[4] Celoxica Limited, Handel-C Language Reference Manual, 2003.

[5] 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.

[6] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers, "YAPI: Application modeling for signal processing systems," Proc. 37th Design Automation Conference (DAC2000), pp.402–405, June 2000.

[7] R.H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," ACM Trans. Programming Languages and Systems, vol.7, no.4, pp.501–538, Oct. 1985.

[8] P.P. Jonker, "Why linear arrays are better image processors," Proc. 12th IAPR International Conference on Pattern Recognition, vol.III, pp.334–338, IEEE Computer Society Press, Los Alamitos, CA, Oct. 1994.

[9] P.P. Jonker and W. Caarls, "Application driven design of embedded real-time image processors," Proc. Acivs 2003 (Advanced Concepts for Intelligent Vision Systems), pp.1–8, Ghent University, Sept. 2003.

[10] R. Kleihorst, H. Broers, A. Abbo, H. Embrahimmalek, H. Fatemi, H. Corporaal, and P. Jonker, "An SIMD-VLIW smart camera architecture for real-time face recognition," Proc. ProRISC 2003, pp.1–7, Technology Foundation STW, Nov. 2003.

[11] C. Kozyrakis, Scalable Vector Media Processors for Embedded Systems, PhD Thesis, University of California at Berkeley, May 2002.

[12] S. Kyo, T. Koga, S. Okazaki, and I. Kuroda, "A 51.2 gops scalable video recognition processor for intelligent cruise control base on a linear array of 128 four-way vliw processing elements," IEEE J. Solid-State Circuits, vol.38, no.11, pp.1992–2000, Nov. 2003.
[13] S. Kyo, S. Okazaki, and I. Kuroda, "An extended c language and compiler for efficient implementation of image filters on media extended micro-processors," Proc. ACIVS 2003, pp.234–241, Ghent University, Sept. 2003.
[14] P. Mattson, A Programming System for the Imagine Media Processor, PhD Thesis, Dept. of Electrical Engineering, Stanford University, 2001.
[15] C. Nicolescu and P.P. Jonker, "EASY PIPE—an "EASY to use" Parallel Image Processing Environment based on algorithmic skeletons," Proc. 15th International Parallel and Distributed Processing Symposium, pp.1151–1157, April 2001.
[16] P. Bogle and B. Liskov, "Reducing cross domain call overhead using batched futures," Proc. Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications, pp.341–354, ACM Press, 1994.
[17] S. Rixner, Stream Processor Architecture, PhD Thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2000.
[18] F.J. Seinstra and D. Koelma, "Lazy parallelization: A finite state machine based optimization approach for data parallel image processing applications," Proc. 17th International Parallel and Dsitribuied Processing Symposium, April 2003.
[19] Y. Fujita, N. Yamashita, and S. Okazaki, "IMAP-Vision: An SIMD processor with high-speed on-chip memory and large capacity external memory," Proc. 1996 MVA Workshop, ed. M. Takagi, IAPR, pp.170–173, 1996.

Wouter Caarls gained an M.Sc. in Artificial Intelligence from the University of Amsterdam, and is now a PhD student at the Quantitative Imaging Group, Department of Imaging Science and Technology, Faculty of Applied Sciences, Delft University of Technology.

Pieter Jonker gained an M.Sc. in Electrical Engineering from the University of Twente, and a Ph.D. in Physics from the Delft University of Technology. He is now an associate professor at the Quantitative Imaging Group, Department of Imaging Science and Technology, Faculty of Applied Sciences, Delft University of Technology.

Henk Corporaal gained an M.Sc. in Physics from the University of Groningen, and a Ph.D. in Electrical Engineering from the Delft University of Technology. He is now a full professor in Embedded System Architectures at the Faculty of Electrical Engineering, Eindhoven University of Technology.