Algorithmic Skeletons for Stream Programming in Embedded Heterogeneous Parallel Image Processing Applications

W. Caarls[1], P.P. Jonker[1], and H. Corporaal[2]

[1]Delft University of Technology
Quantitative Imaging Group
Lorentzweg 1, Delft, The Netherlands
{w.caarls, p.p.jonker}@tudelft.nl

[2]Eindhoven University of Technology
Dept. of Electrical Engineering
Den Dolech 2, Eindhoven, The Netherlands
h.corporaal@tue.nl

Abstract

Algorithmic skeletons can be used to write architecture independent programs, shielding application developers from the details of a parallel implementation. In this paper, we present a C-like skeleton implementation language, PEPCI, that uses term rewriting and partial evaluation to specify skeletons for parallel C dialects. By using skeletons to control the iteration of kernel functions, we provide a stream programming language that is better tailored to the user as well as the underlying architecture. Skeleton merging allows us to reduce the overheads usually associated with breaking an application into small kernels.

We have implemented an example image processing application on a heterogeneous embedded prototype platform consisting of an SIMD and ILP processor, and show that a significant speedup can be achieved without requiring knowledge of data parallel processing.

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 [6] investigates how an application-specific processing device can be generated for the specific field of intelligent cameras, using design space exploration.

The architecture template on which the design space exploration is based contains data-parallel (SIMD) as well as instruction-parallel (ILP) processors, because image processing applications use regular, high-bandwidth as well as irregular operations. Since the design space exploration should be automatic, a single application program must run on all architectures within the template, and the image processing operations must be written in an architecture-independent way.

This same architecture independence shields the application programmer from the parallel implementation details of his operations, such as operation mapping, data distribution, border handling, etc.

This paper describes how we use algorithmic skeletons[5] and stream programming to achieve architecture independence, and how these skeletons are specified in a new language, PEPCI. The paper is structured as follows: first, some background and motivation on stream programming and algorithmic skeletons is discussed. Then, the SmartCam framework and PEPCI language are introduced. After that, we present the prototype architecture on which we have evaluated our system, and the results of the evaluation. Finally, we discuss some conclusions and future work.

2 Background

2.1 Stream programming

Streams are sequences of data elements (often pixels, in image processing). In *stream programming* these streams are transformed by *kernels*, functions that are applied to each element of a stream. Only kernels can access stream elements, and they cannot access global variables (in some systems, the kernels can have internal state memory[9], while in others they cannot[3]). As such, they can be executed efficiently on a wide range of architectures, in particular parallel distributed memory architectures where communication is overlapped with computation (*stream processors*).

Applications in which stream programming is natural are regular, and are presented with large (or al-

1-4244-0054-6/06/$20.00 ©2006 IEEE

most infinite) input streams, the elements of which can be processed mostly independently. Low-level image processing clearly fits into this category, and many intermediate-level and pattern recognition tasks can be expressed like this as well.

Of course, the stream processing abstraction comes with a drawback: algorithms that cannot be naturally mapped to the paradigm must often be awkwardly rewritten, and require multiple passes with different kernels. The underlying hardware may have the capability to efficiently execute it, but the abstraction does not allow us to transparently make use of it. We therefore propose to use a more general way of specifying kernels, using *algorithmic skeletons*, while still using the streaming paradigm to connect them.

2.2 Algorithmic skeletons

Algorithmic skeletons were introduced in [5] to separate the structure of a (parallel) computation from the computation itself, thereby freeing the programmer from the implementation details of that structure, such as how to map it to the available processors. By choosing a skeleton, the application programmer makes a statement about the parallelism in his computation, without specifying how to exploit it, and this freedom can be used to optimally map it to different architectures.

A stream kernel can be seen as a specific type of skeleton: one that calls the kernel for every element of the stream (program 1 shows an example of such a skeleton implemented as a C higher-order function). However, we can imagine different skeletons, specifying different types of parallelism. One might make the *neighbourhood* of a stream element available to the kernel, or might allow the kernel to add new elements to the input, allowing the implemention of recursive and stack operations. The key concept is that we do not fix this when designing our language, but rather allow (specialized) programmers to add their own skeletons.

Because in principle skeletons can be written to perform any kind of operation, they allow us to present all kinds of hardware to the user in an easy-to-use way. Of course, there is a balance between generality (allowing re-use for many different architectures and user kernels) and specificity (for very efficient implementations and very easy interfaces to the user kernels).

2.3 Design goals

We wish to use algorithmic skeletons as interfaces between a hardware platform and kernel functions. A specialized programmer (if not the application programmer himself) should be able to write new skeletons to exploit new hardware or support new operations. To promote adoption, the kernels are written in C, while the target hardware is often programmed in a parallel C dialect. The design goals for our skeleton language are therefore as follows:

1. The language should resemble C, because it deals with C input and C-dialect output. In addition, the skeleton programmer is most likely familiar with the language.
2. The language should be able to *translate* between C and C dialects, and *transform* the kernel from using the interface presented by the skeleton to the interface provided by the underlying runtime system.
3. The output of the skeleton should compile efficiently to the target hardware.

The processor types and number of processors in our platform are unknown at design time, and in order to avoid a static mapping of operations to processor types (or even specific processors) by the user, a single skeleton should be implemented for all processors that support it. A runtime system can then decide which implementation to use, based on benchmarking and load information.

We will first briefly discuss our framework and runtime system, because it specifies how skeletons are called and

Program 1 Sequential pixel stream skeleton implemented as a C higher-order function, and an example binarization kernel using it

```
PixelStream(int_stream_t *in, int_stream_t *out,
  int in_threshold,
  void (*kernel)(int *i, int *o, int threshold))
{
  int iElem, oElem;

  while (read(in, &iElem))
  {
    kernel(&iElem, &oElem, in_threshold);
    write(out, &oElem);
  }
}

void binarize(int *i, int *o, int threshold)
{
  *o = (*i > threshold);
}

/* ...  Stream declaration and
* initialization ...  */
Pixelstream(in, out, 128, binarize);
```

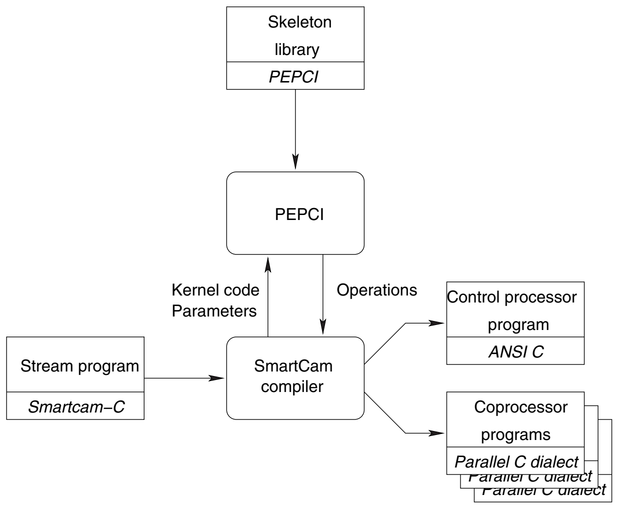

Figure 1: Compilation process of the SMARTCAM framework

what their input will be, both at compile-time and runtime.

3 The SmartCam framework

The SMARTCAM framework[4] specifies how kernels are to be defined and called, and how they are executed at runtime. Figure 1 indicates how a stream program is parsed by the SMARTCAM compiler, how the individual kernels are transformed into operations by the appropriate skeletons through the PEPCI tool (discussed in section 4), and how the compiler generates a control program, as well as a coprocessor program for every coprocessor in the system. In this framework, operations are required to operate on buffered streams of data. Buffer allocation and data distribution is handled by the runtime system, and therefore of no concern to the skeleton.

3.1 Defining kernels

In contrast to normal stream programming, SMARTCAM kernel definitions need to specify which skeleton is used by the kernel. Program 2 shows the kernel definition for a pixel operation, using the **PixelToPixelOp** skeleton.

The direction of the **stream** arguments is specified using **in** and **out**. The skeleton is polymorphic in the number of kernel arguments and their types, and supports non-stream input parameters.

Program 2 SMARTCAM kernel definition for a binarization

```
PixelToPixelOp()
binarize(
  in stream int *i,
  out stream int *o,
  int *threshold)
{
  *o = (*i > *threshold);
}
```

3.2 Defining streams and calling kernels

Streams are declared using the **STREAM** type, and are dynamically typed. Kernels are called as functions that operate on these streams, even though they typically operate on stream elements: the skeleton specified in the kernel definition determines the iteration.

Program 3 shows a complete SMARTCAM stream program.

Program 3 A SMARTCAM program, demonstrating stream definition and kernel calling

```
STREAM a;
scInit();
while (1)
{
  capture(&a); // Capture image from sensor
  binarize(&a, &a, scint(128));
  display(&a); // Display image on screen
}
```

As discussed in [4], kernels are called asynchronously in order to exploit task parallelism, and use single assignment semantics for their stream arguments (if a **STREAM** variable is passed as both input and output, the output points to a *different* stream). Mapping is done at runtime by a control processor, and the various coprocessors perform local scheduling.

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 buffer interactions and scheduling overhead. We would therefore like to compose and statically schedule series of kernels that are known to run together, and this merging is another task for our skeleton language.

4 The PEPCI language

Efficiently implementing algorithmic skeletons, especially on constrained architectures, is a daunting task.

Much research on the subject is done using higher-order functions in functional programming languages, but these are not a natural fit if we want our kernels to be written in C. Using C higher-order functions suffers from a lack of polymorphism, both in parameter types and the number of parameters. While very good C++ skeleton libraries are available (like [8]), C++ compilers for embedded parallel processors are rare. However, the main problem with these approaches (and skeleton languages such as SKIL[2]), is that they do not deal with the *transformation* of kernel code, only with the implementation of structure.

We want our skeletons to be able to arbitrarily change the kernel code to best fit the C dialect and hardware implementation, avoiding overheads such as parameter set-up. These transformations should not be done separately from the implementation of the computational structure, because the structure is what drives the transformations. We have therefore opted to embed term rewriting in a C-like language, PEPCI.

In the following sections, we will discuss a sequential implementation of the **PixelToPixelOp** skeleton used in program 2. The kernel definition will be parsed by the SmartCam compiler into program 4, fed through the skeleton (defined in programs 6 and 7), to yield the output in program 8.

Although this particular case does not require the generality of our skeleton language, it does make use of the most important concepts.

4.1 The code datatype

The input to a skeleton is the code of the kernel and a list of its parameters. They are passed using a new primitive datatype, **code**. To assign code to a **code** variable, we need to distinguish the code to be assigned from the normal program text. The backtick character (‘) is used for this, as shown in program 4.

Program 4 Code assignment and quotation, as used by the SmartCam compiler to pass the kernel code and first argument of the **binarize** kernel to the **PixelToPixelOp** skeleton

```
kernel = ‘*o = (*i > *threshold);‘;
args[0].id = ‘i‘;
args[0].type = STREAM;
args[0].direction = IN;
args[0].datatype = ‘int‘;
```

Because the kernel code will need to be executed as part of the skeleton, it is necessary to provide a way to evaluate the contents of a **code** variable. For this, we introduce the code dereference operator “@”. Furthermore, to aid in code construction, we provide an antiquotation operator “$”, which inserts the value of its argument into a quoted piece of code. Program 5 shows how these operators are used.

Program 5 Code evaluation and antiquotation

```
code assign(code lhs, code rhs)
{
  return ‘$@lhs = $@rhs;‘;
}

int a;
@assign(‘a‘, ‘42‘);
```

The **assign** function takes the left-hand side and right-hand side of an assignment, and constructs the code for it. It does this by antiquoting the sides into an assigment statement, and returning that. Note that dereferenced codes are inserted, because otherwise the statement would become

‘*a*‘ = ‘42‘;,

which is incorrect. The result of the function call is evaluated using the “@” operator as well, resulting in the assignment of 42 to the variable *a*.

4.2 Term rewriting

In order to manipulate **code** variables beyond assignment and evaluation, we employ term rewriting. Term rewriting works by matching a certain code pattern, and replacing it with another. Program 6 uses term rewriting to change the kernel code from using pointer dereferencing to access its inputs and outputs into using array indexing.

Program 6 Rewriting pointer dereferencing into array indexing

```
for(i=0; i < arguments; i++)
  if (args[i].type == STREAM)
    kernel = replace(args[i].id,
                     ‘&$@(args[i].id)[c]‘,
                     kernel);
```

replace is a term rewriting strategy that performs a topdown search through the abstract syntax tree representing the code in its third argument, and replaces all occurrences of its first argument with its second argument.

Again, “$” is the antiquotation operator that inserts its argument in the quoted code. In this case, its argument is the code dereference (“@”) of *args*[*i*].id (being

i or o). Using our binarization kernel as input to this operation, we will get the following output:

$$o[c] = (i[c] > *threshold);$$

All term rewriting in our system is achieved by embedding the Stratego[11] language. It is possible to define custom strategies in the same source code as the skeleton, but the most common ones are provided in a library. Unfortunately, a full description of Stratego is beyond the scope of this paper.

4.3 Partial evaluation

Apart from rewriting the kernel, a skeleton should also output the code that is to be evaluated at runtime. We have chosen to avoid a manual distinction between generating and generated code, and use *partial evaluation* to make this distinction automatically. All code in programs 5-6 can be evaluated at compile time, but consider program 7, which implements the main loop of our skeleton.

Program 7 The main loop of the **PixelToPixelOp** skeleton

```
while (...)
{
  for (i=0; i < arguments; i++)
    if (args[i].type == STREAM)
      if (args[i].direction == IN)
        read(args[i].stream, &@args[i].id, bytes);
      else
        allocate(args[i].stream, &@args[i].id,
                 bytes);
  for (c=0; c < bytes; c++)
    @kernel;
  for (i=0; i < arguments; i++)
    if (args[i].type == STREAM)
      release(args[i].stream, bytes);
}
```

The output should not include the looping over the arguments, both for efficiency reasons, and because the code may include constructs that are not understood by the target compiler. Partial evaluation separates the parts of the code that can be executed at compile time by propagating information about static and dynamic variables. Because PEPCI is implemented as an interpreter, this information is easily accessible through the symbol table that is used record the actual variable values.

The basis of the interpreter is a **reduce** function that not only returns the value of a computed subexpression, but also a residual syntax tree, the nodes of which can be *tentative*. As long as all values are known, the interpreter proceeds normally, with the exception that all declarations and assignments are tentatively entered in the residual syntax tree.

When a statement is encountered that references unknown values, the declarations and most recent assignments of all symbols used in it are unmarked, and will be printed when the residual tree is output. Unknown expressions in branching and looping constructs will cause the different branches to be evaluated separately; values changed in these branches are unknown for further evaluation, and the construct itself is entered into the residual syntax tree. To ensure termination, recursive functions called with unknown parameters are residualized.

Program 8 shows the result of program 7 applied to the rewritten binarization kernel. Only the program parts that depend on variables that are not known at compile time, or on functions that cannot be evaluated at compile time, are output.

Program 8 Result of partially evaluating program 7 applied to the binarization kernel

```
while (...)
{
  read(args[0].stream, &i, bytes);
  allocate(args[1].stream, &o, bytes);

  for (c=0; c < bytes; c++)
    o[c] = (i[c] > *threshold);

  release(args[0].stream, bytes);
  release(args[1].stream, bytes);
}
```

4.4 Skeleton merging

As discussed in section 3.2, the SmartCam framework uses runtime scheduling to interleave multiple kernels running on the same processor. This can be inefficient, both because of the context switching overhead, and because of the limited instruction parallelism available in a single kernel. Especially SIMD processors do not work well with runtime scheduling, as hundreds of computations happen each cycle, and control flow operations therefore incur a large overhead. Our solution is to compose and statically schedule series of kernels that are known to run together, as is the case when they occur in a single basic block of the stream program.

We perform kernel composition and static scheduling by recursively reducing a stream graph through skeleton merging. The algorithm proceeds as follows:

1. Extract a series of kernel calls from the stream program, satisfying the following conditions:
   - Intermediate streams are not used outside the sequence
   - All the skeletons used by the kernels employ the same *metaskeleton*
2. Merge the first two kernels in the sequence, by applying a *merge metaskeleton* that is specific to the metaskeleton.
3. Repeat 2 until the sequence is reduced to one kernel call.
4. Replace the series in the stream program by the reduced version.

Because skeletons are general programs, with arbitrary output, merging them is not straightforward. There needs to be some form of restriction on their output, and this is achieved by their use of a metaskeleton. A metaskeleton is a PEPCI function that has the code of a skeleton as its input, in a structured form (such as initialization, prologue, body, epilogue).

When no merging is to take place, the metaskeleton evaluates its arguments in the appropriate way. If two skeletons are to be merged, however, the arguments they both pass to the metaskeleton are passed to a merge metaskeleton instead, which performs the merging. The merged result again uses the original metaskeleton, so that the merged skeleton can be used in subsequent recursive merge operations, see figure 2.

In our library, the **PixelToPixelOp** skeleton uses the **pixelbypixel** metaskeleton, while the **NeighbourhoodToPixelOp** and **RecursiveNeighbourhoodToPixelOp** skeletons use the **linebyline** metaskeleton. Other skeletons, such as **StackOp** and **GlobalOp**, do not use a metaskeleton, and therefore cannot be merged.

Program 9 shows a simplified merge metaskeleton for pixel operations. In this piece of code, we assume that in the stream program, the second (output) argument of the first kernel is connected to the first (input) argument of the second kernel.

First, we construct a new prologue as a composition of the constituent prologues; the **declare** rewriting strategy declares a new intermediate variable of the appropriate datatype. Then, we replace the first kernel's output and second kernel's input with this new variable. Finally, we compose the kernels and again call the **pixelbypixel** metaskeleton. Assuming we're merging a logarithm and binarization, the newly constructed kernel is:

```
intermediate = log(*i);
*o = (intermediate > *threshold);
```

Program 9 A simplified **pixelbypixel-merge** metaskeleton for merging two pixel operations

```
pixelbypixel-merge(arg_t *args, int arguments,
                        instance_t *instance)
{
  code prologue, kernel;
  prologue = ‘{
                declare(
                  $instance[0].args[1].datatype,
                  ‘intermediate‘);
                @$instance[0].prologue;
                @$instance[1].prologue;
              }‘;
  instance[0].kernel =
    replace(instance[0].args[1].id,
      ‘&intermediate‘, instance[0].kernel);
  instance[1].kernel =
    replace(instance[1].args[0].id,
      ‘&intermediate‘, instance[1].kernel);
  kernel = ‘{
              @$instance[0].kernel;
              @$instance[1].kernel;
            }‘;
  pixelbypixel(args, arguments, prologue, kernel);
}
```

More elaborate merge metaskeletons, such as **linebyline-merge**, need to deal with the appropriate buffer delays, and of course must be fully polymorphic in the stream connections. They should also take care of variable nameclashes.

5 Prototype setup

In order to evaluate the programmability and efficiency of our system, we have implemented it on a mixed ILP-SIMD prototype architecture.

5.1 Architecture

Our prototype architecture is the Philips CFT Inca+ prototype. This is a minimal implementation of our architecture template, consisting of one XeTaL [1] SIMD processor and one TriMedia[10] VLIW processor. 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 [7], and is schematically summarized in figure 3.

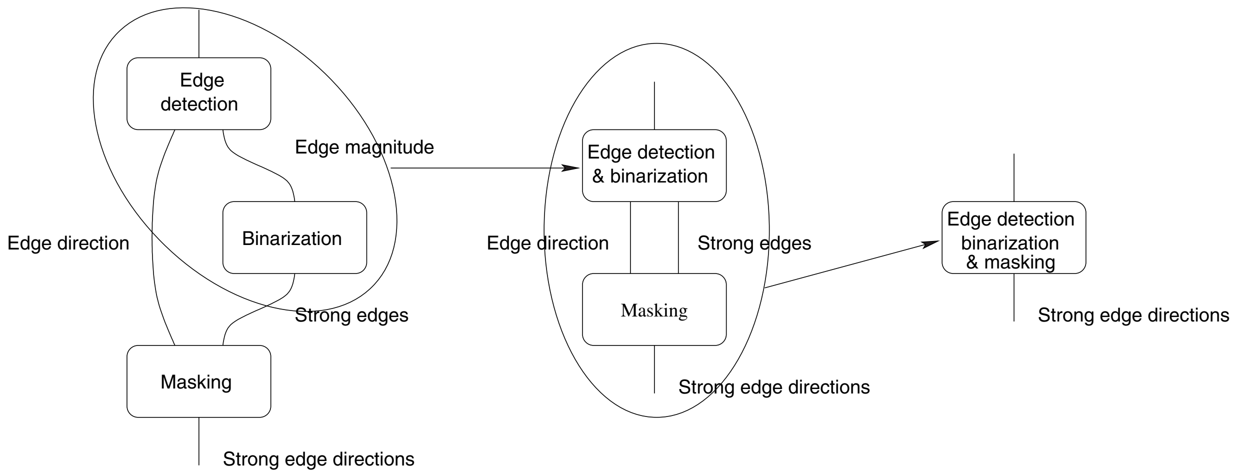

Figure 2: Recursive skeleton merging

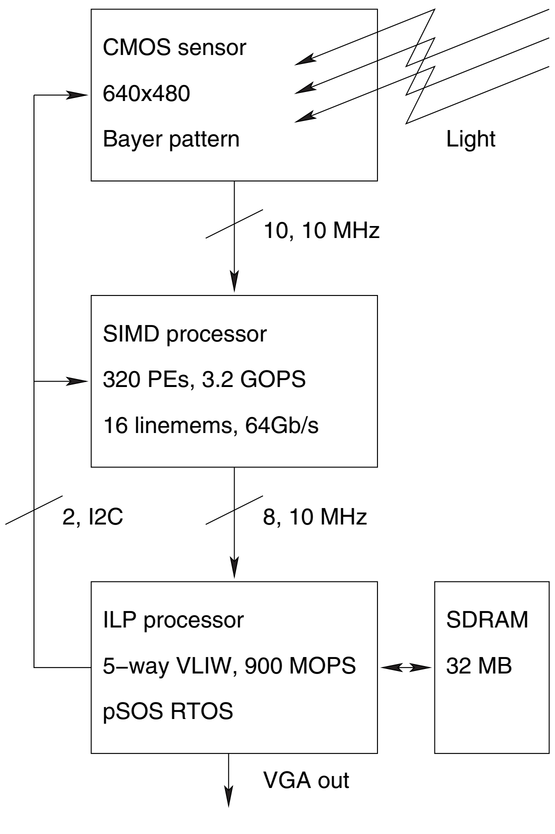

Figure 3: Inca+ prototype architecture

The XeTaL chip consists of 320 PEs and a control processor, running at pixel clock. It can therefore process 320 instructions per pixel, and has enough memory to store 16 image lines. The TriMedia is a 5-way VLIW processor running at 180MHz. At the same video speed that means around 100 operations per pixel. An external 32MB SDRAM provides enough storage for most applications at this resolution. The TriMedia runs the pSOS multithreaded real-time operating system. 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. 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 platform that is under development. In this new prototype, the XeTaL processor may also run faster than the pixel clock, achieving 7.7 10-bit GMACs per second.

5.2 Programming

The TriMedia processor can be programmed in ANSI C. However, for certain optimizations to take place, it is necessary to instruct the compiler with additional information in the form of pragmas and type specifiers. Our algorithmic skeletons use knowledge about their parallelism to supply this information.

The dataparallel C-dialect used by the XeTaL SIMD processor (called XTC) mainly adds a new datatype, **lmem**, that represents an entire line. Arithmetic operations on the datatype are carried out on the entire line, and **lmem** variables can be relatively indexed to access neighbours. The processor does not provide indirect ad-

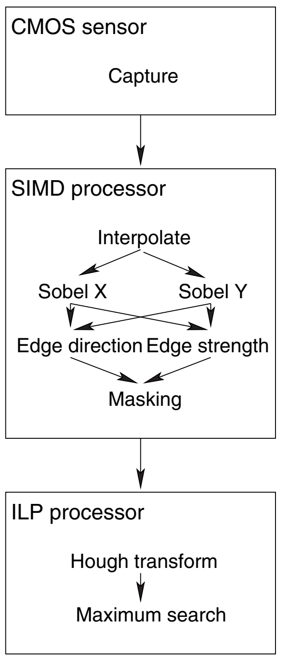

Figure 4: Mapping the ball detection application on the Inca+ prototype

dressing or per-PE branching, and the language reflects this, prohibiting array indexing and **if** statements on **lmem** variables.

However, because kernels *can* use these constructs, our XTC skeletons rewrite branches into guarded expressions, and static indirect neighbourhood addresses into separate variables. Loop unrolling is used to transform apparently-dynamic indices into static ones. Even then, not all kernel codes will compile to XTC, and these will have to be executed on the TriMedia.

5.3 Application

As a test application, we have implemented a ball detection algorithm based on the Hough transform. Our version uses the edge direction to draw a circle arc in the Hough accumulation space instead of the entire circle. The global maximum of the accumulation space is used as the ball position.

The edge detection (using the Sobel operators) and edge strength binarization can run on our SIMD processor, while the Hough transform itself cannot 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. See figure 4.

Setup	**Time(ms)**
TriMedia Baseline	133
TriMedia Optimized	100
TriMedia Kernelized (5-line bufs)	216
TriMedia Kernelized (25-line bufs)	160
TriMedia Merged	134
TriMedia + XeTaL Merged	54

Table 1: Performance evaluation of the ball finding algorithm

6 Results

We have evaluated a number of different setups. First, a handcoded C implementation on the TriMedia only, as a baseline. Second, an optimized version using TriMedia-specific pragmas and ISO C99 *restrict* specifiers. Next, we split the algorithm into basic kernels (interpolation, sobel x, sobel y, edge direction, edge magnitude, masking, Hough transform, global maximum) and ran this on the TriMedia using different buffer sizes. Finally, we enabled skeleton merging and ran the program on the TriMedia as well as the TriMedia and XeTaL combination. Table 1 summarizes the results.

It is evident that a fine-grained kernelization of the algorithm using small buffers between the kernels results in a severe slowdown (more than twice as slow as the optimized version). The overhead introduced by context switching approaches zero if larger buffers are used, but this is unrealistic, and we still retain a 60% total overhead. By merging the kernels, allowing the compiler to exploit instruction parallelism and data locality as well as avoiding context switching, we achieve the same speed as the baseline. Finally, by leveraging the SIMD processor, the program runs twice as fast as the optimized ILP version.

The reason the merged version does not achieve the speed of the optimized one is the inability to merge pixel and line metaskeletons, as well as some overhead in the merged operations themselves. In fact, the speedup depends very much on the input image and types of operations. In the case of the ball detection application on our prototype, the SIMD processor is only processing 1/2 of the time, as it is bandwidth-limited by the video speed input and output channels. 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.

7 Conclusion

We have presented a language and interpreter for implementing algorithmic skeletons for C-like target languages, and a framework to use them to write architecture independent streaming image processing applications for parallel heterogeneous embedded systems. By embedding a term rewriting language, skeletons can perform source-to-source translation in order to support data parallel targets from sequential kernel sources. Partial evaluation is used to avoid a manual distinction between generating and generated program.

Our results show that an example ball-detection application written using our framework can achieve the same performance on an ILP processor as an ANSI C implementation, while the architecture independence allows it to run unchanged on a mixed SIMD-ILP platform at twice the speed as an optimized ILP version. These results are achieved by merging the kernel functions in order to reduce buffer interaction and context switching overhead, and improve instruction parallelism and data locality. No parts of our example application required explicit parallel programming, or knowledge of the parallel implementation of the operations.

As future work, we would like to extend PEPCI to allow external library calls such as math functions. We would also like to relax some of the merging requirements, such as allowing the merging of **pixelbypixel** and **linebyline** skeletons by promoting the **pixelbypixel** skeleton. Finally, we want to expand our hardware platform in order to investigate larger and more dynamic applications.

Acknowledgements

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

References

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

[2] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004. Special Issue: Proceedings of the 2004 SIGGRAPH Conference.

[4] W. Caarls, P. Jonker, and H. Corporaal. Skeletons and asynchronous RPC for embedded data- and task parallel image processing. In K. Ikeuchi, editor, *Proceedings of the 9th IAPR Conference on Machine Vision Applications*. MVA Conference Committee, May 16-18 2005.

[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] P. Jonker and W. Caarls. Application driven design of embedded real-time image processors. In *Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems)*. Ghent University, September 2-5 2003.

[7] 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. In *Proceedings of ProRISC 2003*, pages 1–7. Technology Foundation STW, November 26-27 2003.

[8] H. Kuchen. A skeleton library. In B. Monien and R. Feldman, editors, *Proceedings of the 8th International Euro-Par Conference*, volume 2400 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[9] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 2001.

[10] G. Slavenburg. *TM1000 Databook*. TriMedia Division, Philips Semiconductors, 1997.

[11] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.