Invited Paper:
Adapting Algorithms for Hardware Implementation

Donald G Bailey
School of Engineering and Advanced Technology, Massey University
Palmerston North, New Zealand
D.G.Bailey@massey.ac.nz

Abstract

Embedded vision often requires balancing the computation and power requirements of an application. Hardware implementation of the vision algorithm using an FPGA enables parallelism to be exploited, allowing clock speeds to be significantly reduced. However, simply porting software algorithms usually gives disappointing performance. Software algorithms are usually optimised for serial implementation. An efficient FPGA implementation requires transforming the algorithm to make better use of parallelism. Several transformations are illustrated using connected components analysis.

1. Introduction

Embedded vision is the incorporation of vision within a product or appliance to enhance its usefulness of functionality in some way. However, there are a number of significant challenges in developing such systems:

- There is a large volume of data to process. Computer vision is computationally demanding. An input image contains a large number of pixels, and any non-trivial application requires many computations to be performed on each pixel.
- The data must be processed quickly. In many applications, but especially machine vision and robot vision, the processing speed is critical. Real time operation is essential for tasks where the data extracted from the vision system is used to control an activity. Delays complicate the design of a control system and can directly lead to instability.
- Power is limited. Many applications require the system to be operated off battery power. This can require running the system with a low clock speed, impacting on performance.
- Physical space and weight is limited. Often, devices incorporating embedded vision must be portable.

The final solution is often a compromise between processing speed on one hand, and the power required to operate successfully on the other.

Field programmable gate arrays (FPGAs) are increasingly seen as a practical platform for the implementation of embedded vision applications. They have a number of distinct advantages over conventional computing platforms. They are able to meet each of the challenges posed by embedded systems:

- As a hardware implementation, it is inherently parallel, and is able to exploit the parallelism implicit within low and intermediate level image processing operations. This can enable the large volumes of data to be processed efficiently
- By operating in a pipelined manner, the processing throughput can be increased and the latency reduced.
- Exploiting parallelism enables many algorithms to run at the pixel clock rate, which can be up to two orders of magnitude slower than that required for high end serial processors. This can give significant power savings.
- An FPGA can interface directly with CMOS sensors and many other peripherals. Consequently, a complete system can often be built with only two or three chips.

In spite of these clear advantages, using FPGAs to implement vision systems has one distinct disadvantage over software based implementations. FPGAs are difficult to program well. This difficulty has five main sources.

Firstly, hardware is parallel, and requires a hardware mindset. The added dimension of concurrency results in an explosion of complexity. Parallel programming is difficult, and even more so at the relatively low level of gates.

Secondly, the computational architecture of a CPU is well defined, and is given. When designing for an FPGA, the developer has a clean slate as far as the computational architecture is concerned. Both the architecture and the algorithm must be developed in tandem.

Thirdly, computer vision is traditionally seen as a software activity. Many of the algorithms have been developed and optimized for serial execution, and do not directly port well to a hardware implementation. The performance of complex algorithms on an FPGA can be quite sensitive to the quality of the implementation [1].

Fourthly, many of the languages used to program FPGAs are at quite a low level. Traditional hardware

description languages (VHDL, Verilog) are not well suited to programming complex vision algorithms. Using them is a little like programming in assembly language; it is necessary for the programmer to take care of every little detail. This has led to a proliferation of C based hardware description languages (for example Handel-C [2] formerly from Celoxica, now available from Mentor Graphics; Impulse C [3] from Impulse Accelerated Technologies; Catapault C [4] from Mentor Graphics; and many others). While these hide the low level details to varying degrees, there are still limitations in using a software based language to describe hardware [5].

Fifthly, the complexity of mapping a design onto the resources of an FPGA is a combinatorial problem. It can literally take hours to compile, map, place and route a complex design. This makes iterative development techniques less practical. Experimental development techniques commonly used for image processing applications do not fit well with FPGA development cycle times. This can be overcome to some extent by developing the high-level application algorithm in software first.

To effectively use FPGAs to accelerate image processing algorithms, it is necessary to overcome these limitations. Simply porting a software algorithm generally gives disappointing performance. The underlying software algorithm is inherently serial, and can limit the scope for exploiting parallelism. While modern compilers can use dataflow analysis to identify opportunities for parallelism, the algorithm basically remains the same. They cannot redesign the algorithm to make better use of hardware resources. To achieve the best results, it is necessary to transform the algorithm to exploit parallelism and make best use of the available hardware resources. In this paper, several key transformation principles will be discussed within the context of a connected components analysis task.

2. Connected components analysis

Connected components analysis is an important part of many image analysis applications. The basic structure of such an algorithm is shown in Figure 1.

In an embedded vision system, the image would be captured directly from a camera. Preprocessing typically includes one or more filtering stages to reduce noise and enhance edges, followed by thresholding to make the image binary. Connected components labelling assigns a unique label to each region within the image enabling distinct objects to be distinguished. Key features are then extracted from each region which enables the corresponding objects to be classified, and appropriate control signals generated (for example in a machine vision application).

A software implementation of the algorithm is analyzed

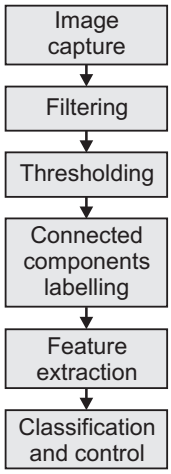

Figure 1: Basic structure of a connected components algorithm.

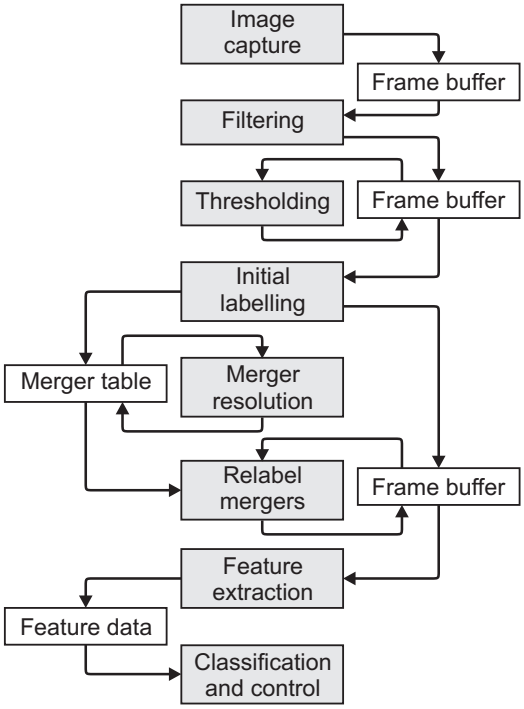

Figure 2: Expansion of the algorithm to show the high level data flow typical of a software implementation.

in more detail in Figure 2. Of particular importance is the data flow through the algorithm.

In a software based system, the image is typically captured into a memory based frame buffer. Any filtering will typically store the filtered output into a separate array in memory to prevent overwriting pixels in the input image required for filtering subsequent lines. Thresholding, being a point operation is often performed in place on the image.

The connected components labelling algorithm has been expanded out into its three separate phases or stages. The

classic two-pass connected components labelling algorithm [6] is used here. The first pass performs an initial labelling of the image using a raster scan to propagate labels down through the image. The four neighbours of an object pixel which have already been processed (assuming 8-connectivity) are examined, and if only one label is present, it is propagated to the current pixel. If none are labelled, a new label is generated and assigned. At the bottom of a 'U' shaped object, each of the branches will have a different label. Since these two labels refer to the same object, one must be relabelled. Rather than perform the relabelling immediately, since there are potentially many labels that must be relabelled in an image, relabelling is deferred to a second pass where they may all be changed at once. This requires recording the fact that such pairs of labels are equivalent, in some form of merger table. The merger table (or equivalence table) is often implemented by storing the smaller label in a table indexed by the larger label. At the end of the first pass, all label equivalences are resolved by assigning a new, final, label to each set of equivalent labels. This requires following chains of equivalent labels through the table to find the smallest label used within each region. The result is a lookup table where each label points to the smallest label in the equivalence set. The second pass uses this lookup table to relabel the image consistently. This second pass can be performed in place.

Feature extraction then processes the image for each label to calculate the features of the corresponding connected component. These features are then passed to the classification and control stage.

3. Transformation principles

Two observations may be made regarding this algorithm. The first is that it is largely sequential, with each image processing operation being applied to the image in turn. Within each operation, each pixel is operated on in turn. The algorithm is also memory bound. Each operation reads its input pixels from memory, processes them, and writes the results back to memory. The processing time will therefore be largely limited by the number and speed of memory accesses.

To reduce the clock speed (and power) it is necessary to transform the algorithm to exploit parallelism and reduce the necessary memory bandwidth.

3.1. Exploit appropriate forms of parallelism

Parallelism can be found in image processing algorithms at a number of levels.

At the highest level, an image processing algorithm consists of a sequence of image processing operations. This is the form directly illustrated in Figure 1. Each operation is independent in the sense that the processing performed by one operation is independent of the processing performed by each of the other operations. The only dependency is the flow of data from one operation to the next through the processing chain. This temporal parallelism may be exploited by pipelining, using a separate processor for each operation within the algorithm. Such pipelines work like assembly lines. At a given stage, once the input is available, the operation may be performed on the image with the results passed on to the next stage in the chain. It may then begin processing the next input, without having to wait until the processing for the previous image is complete. The independence of the operations allows each processor to work independently of the others, subject to the flow of data between them.

At the next level, many image processing operations perform the same operation for each pixel within the image. In software, this is represented by the outermost loop iterating through the pixels in the image. This therefore corresponds to spatial parallelism, which may be exploited by partitioning the image among several independent processors. The main factor in determining the best partitioning is to minimise the communication between processors, because any such communication is an overhead and reduces the effective speedup that can be achieved. Minimising communication usually corresponds to minimising the data required from other partitions. Such partitioning is therefore most effective when only local processing is required. This is one of the drivers for investigating separable algorithms, because separating row and column processing provides a partitioning that has inherently low communication overheads.

At the lowest level is logical or functional parallelism. This is where a single function block is reused many times within an operation. For example, a linear window filter implemented in software would consist of an inner loop which iterates through the pixels within a window, multiplying each pixel by the corresponding filter coefficient, and adding the result to an accumulator. The content of the inner loop – the multiply and accumulate – is repeated many times, but can readily be parallelised by building several multiply and accumulate blocks. Since most of the processing time is spent in these inner loops, parallelizing these can give the largest processing speedup.

3.2. Use stream processing where possible

The original serial algorithm is based on random access of pixels from memory. Many low and intermediate level operations are implemented with the outer loops performing a raster scan through the image. Serializing the image data, using a raster scan, converts spatial parallelism into temporal parallelism.

In a stream processing system, each stage of the processing pipeline operates one pixel at a time, rather

than a whole image at a time. In hardware, each operation works in parallel. Therefore, rather than write the results to memory, the pixels output from one operation are passed directly as input to the next operation. This eliminates the need to buffer the intermediate images in memory, significantly reducing the memory bandwidth required. Stream processing can also significantly reduce the latency because each operation in a pipeline can begin processing as soon as pixels start arriving, without having to wait until the complete image is available. Stream processing does require the downstream operation to consume data at the same rate as pixels are produced by the upstream operation. This may be relaxed if necessary by using a FIFO buffer between operations to smooth the data flows.

When interfacing to either a camera or display, the image data is normally passed sequentially, so using pipelined stream processing for operations close to image capture or display is an obvious optimization. Progressive scanning is assumed here; interlaced scanning complicates matters by requiring a frame buffer for de-interlacing.

Within the connected components analysis algorithm, many of the initial preprocessing operations can be transformed to use stream processing. Local filtering is usually implemented with the outer loops performing a raster scan through the image. The limitation is that for each output pixel, multiple input pixels are required. This requires that the computational architecture be designed with an appropriate caching scheme so that each input pixel is only input once. This will be described in more detail in the next section; assume for now that this is possible and the filter can be streamed.

Thresholding, as a point operation, also fits well with stream processing. However, in many applications the threshold level is not necessarily fixed, but determined from the image. Adaptive thresholding uses a local filter to determine the threshold level, so can be treated just like another filter. Dynamic thresholding requires capturing statistics from the complete image, for example in a histogram. This necessitates buffering the image while the statistics are being gathered and the threshold level calculated. If the statistics do not change significantly from one frame to the next, then data gathered from the previous frame may be used to threshold the current frame [7].

Each of the two labelling passes uses a raster scan, enabling stream processing to be used for these. The initial labelling pass propagates output labels assigned to the previous row; these can be cached locally. The image must be buffered between passes while mergers are resolved.

The resulting algorithm architecture is shown in Figure 3. The initial frame buffers are replaced by small local caches. No operations can be performed in place with stream processing, because the data in the input buffer will be replaced by the next frame. This requires a new frame buffer to hold the relabelled image. Similarly, the merger table needs to be split into two, with a separate lookup table for the relabelling.

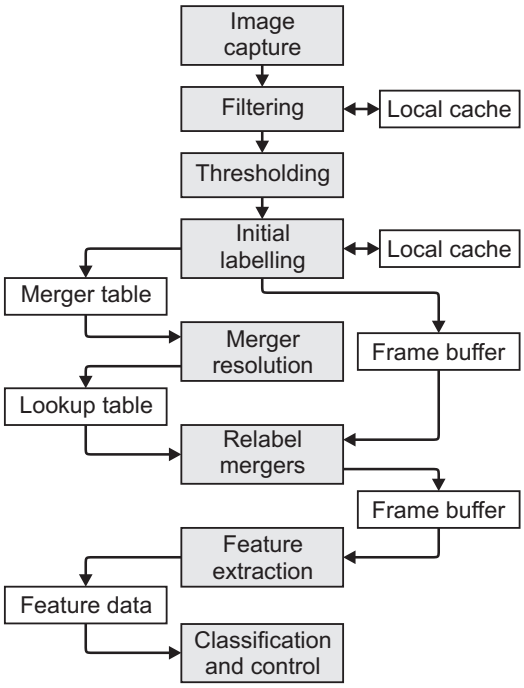

Figure 3: After applying stream processing to the original algorithm

3.3. Reduce memory access through local caches

Many image processing operations require multiple input pixels for each output pixel. Caching provides temporary storage for data that is potentially used multiple times, significantly reducing memory bandwidth [8].

Consider a 3×3 window filter – each output sample is a function of the nine pixel values within the window. Without caching, each pixel must be read nine times as the window is scanned through the image. Pixels adjacent horizontally are required for successive window positions, so may simply be buffered and delayed in registers. This reduces the number of reads to three pixels every window position. A row buffer caches the previous rows to avoid having to read the pixel values in again (see Figure 4). A 3×3 filter spans three rows: the current row and two previous rows; a new pixel is read in on the current row, so two row buffers are required to cache the pixel values of the previous two rows. Such a caching arrangement can enable one output pixel to be produced each clock cycle.

The regular access pattern simplifies the design of caches. Row buffers are typically implemented using either FIFOs or circular memory buffers made from dual-port RAM resources on the FPGA. This principle can be readily extended to larger window sizes or to other operations that require data from the previous rows of

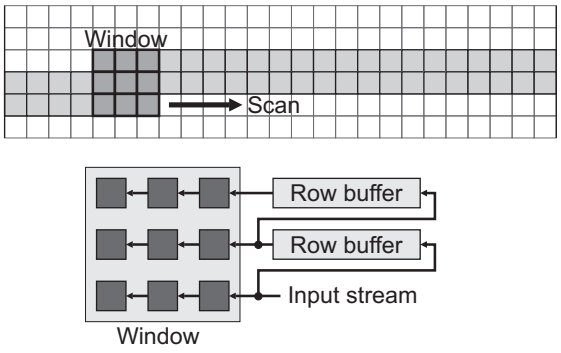

Figure 4: Row buffering caches previous rows so they only need to be loaded once.

either the input or the output.

Note that with filtering, another issue is managing image borders (where the window is not completely within the image). In software it is easy to add a few conditional tests to manage these edge effects, although the corresponding code can often be longer than the rest of the design. While the same is true in hardware, the additional logic can add considerably to the resource requirements of algorithm. By carefully designing the caching [9], the image borders can be easily extended with minimal resources without significantly affecting the rest of the design.

3.4. Strip mining and multiplexing

One of the bottlenecks within the connected components data flow is the multiple iterations required by the feature extraction step. Typically the algorithm iterates through the labelled image extracting data for each label in turn. However, since each iteration is processing a separate label, this step is readily parallelizable. Strip mining partially unrolls the outer loop, creating multiple copies of the loop body. Computation is accelerated by partitioning the input data over different processors.

On the surface, strip mining gives little relief from the memory bandwidth, since each processor requires access to the input data. One technique that can be used to overcome this is to allocate each processor local memory on the FPGA [10]. The availability of many smaller independent memory blocks on the FPGA overcomes the memory bandwidth problem. The input data is broadcast to the local memories in parallel, from which each processor can access its required data in parallel.

In the connected components analysis algorithm, if the features can be extracted from a streamed image, then the image can be streamed to all of the processors in parallel directly from the relabelling step. Many (but not all) useful object features can be accumulated using a raster scan through the image. These include object count, area and other moments, the perimeter (with a little extra processing to detect edge pixels), bounding box, average intensity or colour (if the original input image is buffered in parallel with the other processing). Features derived from these can also be computed: compactness, centre of gravity, orientation of the best fit ellipse, etc.

However, strip mining only partially solves the problem because potentially a large number of components may be present within the image, requiring a large number of processors. There are two solutions to this problem.

First observe that the number of components that may be being processed at any one time will be less than half of the width of the image. This places an upper bound on the number of parallel processors actually needed. Unfortunately, this is still a large number, and requires considerable hardware resources.

Second, observe that for feature extraction from connected components, each pixel can have only one label. Therefore, only one of the feature extraction processors will be active for any pixel. This implies that if stream processing is used, a single feature extraction processor may be shared for all of the iterations (strip rolling). Separate data registers must be maintained for each component with the particular registers used for processing each pixel multiplexed by the component label. Thus for each pixel, the label is used to select the data associated with that object. This data is then updated according to the pixel added, and saved again in the corresponding register. A single feature extraction processor can then measure the features of an arbitrary number of objects in a single pass through the image.

Such register multiplexing can be expensive in terms of hardware, especially if there is a large number of labels (hence registers) processed in parallel. Rather than use registers, a memory based data table can be used, with the addressing logic performing the multiplexing for free.

3.5. Rearrange the algorithm and substitute operations to simplify the processing

In some algorithms, it is possible to rearrange the order of the operations to simplify the processing complexity, or even eliminate some steps. A classic example of this is greyscale morphological filtering followed by thresholding. This sequence of operations is equivalent to thresholding first, followed by binary morphological filtering, and since the hardware requirements for binary processing are much simpler, this can result in significant savings. In this example, morphological noise filtering may be moved to after the threshold.

In the connected components labelling scheme, if the labels themselves are not required, then the feature data can be extracted directly during the initial labelling pass [11]. In the original algorithm, relabelling was deferred as it potentially involved updating many pixels. However, if the essential feature data is extracted during the first pass,

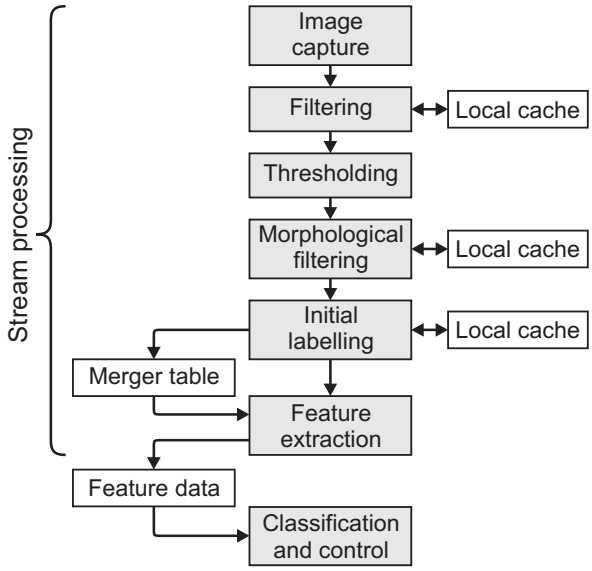

Figure 5: After extracting features during the first pass.

merging simply requires the data for the merged regions to be combined. The second relabelling pass is no longer necessary, and can be eliminated along with its associated input and output frame buffers.

As a result of this transformation (see Figure 5), the whole processing through to feature extraction can operate on streamed data, and the need for frame buffers is completely eliminated.

Other optimizations that may be used at this stage are to simplify operations by substituting simpler but computationally less expensive operations. A good example of this is replacing the L_2 norm in the Sobel filter with the computationally simpler L_1 or L_∞ norm [12], or the YCbCr colour transformation with the simpler reversible colour transform [13] or related transformation using power of two coefficients [14]. Note that substituting an operation may change the results of the algorithm. The resulting algorithm will be functionally similar but not necessarily equivalent. In many applications this does not matter and can be adjusted for elsewhere in the algorithm.

Appropriate use of separability, decomposition, and common sub-expression reuse can all result in reduced hardware.

3.6. Reduce data volume through coding

In many applications, the processing time is determined by the number of pixels processed. This is especially so with multi-pass or iterative algorithms. While not necessarily important with software processing, when using hardware, especially with a reduced clock speed, the processing time can be a critical constraint.

In such cases, one transformation that is sometimes possible is to compress the data, and operate on the compressed data. In this connected components analysis example, the processing can be reduced by run-length encoding the binary image before processing [15, 16]. While this complicates the label propagation logic, it does result in fewer mergers, and in the second pass, the labelled image can be processed whole runs at a time, rather than pixel by pixel.

3.7. Transform the complete algorithm, not just individual operations

The example here illustrates the importance of not just considering each operation in isolation. Considerable gains can be made by considering the interactions between the operations. This is especially the case where the operations can all be made to use stream based processing.

3.8. Select data and memory structures based on H/W rather than S/W requirements

In software, almost all data structures are memory based, and are mapped to a single monolithic memory. There is a wider range of hardware structures available on an FPGA. For example, memory is not monolithic, but consists of many smaller, independent blocks. Most of these are dual-port, enabling read-modify-write processing (with pipelining if necessary to satisfy setup and hold requirements).

Memory structures such as lists and trees can be bandwidth hogs, so less bandwidth intensive hardware alternatives should be considered, especially where bandwidth is the limiting constraint.

Several examples of these structures and optimizations are illustrated in Figure 6 for the computational architecture for labelling and feature extraction steps from Figure 5. It operates as follows [17]: The neighbourhood context provides the labels of the previously processed pixels adjacent to the current pixel. The row buffer caches assigned labels for access on the next row, and is effectively a one row delay. The label selection block selects the label for the current pixel based on he labels of its neighbours, following the same approach as the classical algorithm for propagating labels.

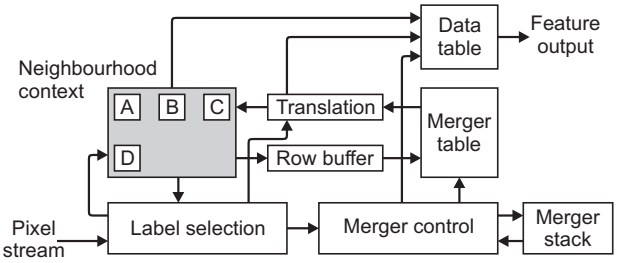

Figure 6: Connected components analysis computational architecture [17].

Whenever two regions merge, the equivalence is stored in the merger table. This acts as a lookup table on the output of the row buffer to update any labels which have changed due to mergers later in the row. The merger table therefore is implemented using dual-port memory. One port is used to perform the lookup, and the other used to update the table when mergers occur. The merger stack is required to efficiently manage chains of mergers by recording key mergers in a stack structure, allowing chains of mergers to be unlinked through a backward scan during the horizontal blanking period. Using a stack enables only the critical elements within the merger table to be updated, saving time.

To reduce the number of labels required, it is observed that the number of active labels depends on the width of the image rather than the area, which requires an aggressive reuse of labels [18]. In the implementation here, regions are relabelled each row, requiring a translation between the labels used on the previous row and the labels used on the current row. This is again implemented using a second dual-port memory.

Finally, the data table contains the feature data being extracted for each object. It is indexed by the label of the current pixel, with the entry updated to include the current pixel within the region. Whenever two regions merge, the corresponding entries in the data table are also merged. With relabeling used for each row, two data tables are required, one for the previous row, and one for the current row. When a region propagates from the previous row, the entry within the previous row data table is combined with the region on the current row. At the end of each row, the roles of the two tables are swapped. Any data not transferred corresponds to completed regions, enabling them to be passed to the classification process at that stage [17, 19] (rather than waiting to the end of the image).

3.9. Use software for software tasks and hardware for hardware tasks

The final principle acknowledges that not all algorithms will map well to a hardware implementation. Functions which are only used occasionally, or have dynamically variable loops, complex control sequences, or large amounts of complex, primarily sequential code are best implemented in software. Many high level vision tasks fall into this category. If implemented in hardware, each step in the sequence would require its own hardware, much of which will be idle for a significant proportion of the time. Such tasks can be implemented more efficiently in software, which is also easier to program.

In many cases, the processor may be implemented as a soft core using the FPGA's programmable logic. The major FPGA vendors provide intellectual property blocks and development environments (MicroBlaze from Xilinx, and Nios II from Altera) for implementing such processors and integrating them with the rest of the design.

4. Results

The connected components analysis algorithm (consisting of connected components labelling and extraction of data from each region) has been implemented on an FPGA as detailed in [17, 20, 21]. The final implementation reduced the memory requirements over a port of the standard software algorithm by over 350 times and improved the latency by more than a factor of two [21]. The resources required were very modest: 4 block RAMS, 600 slice flip-flops, and 1757 LUTs on a Xilinx Virtex 2 [17]. With a clock speed of 40 MHz, it was able to process up to 100 frames per second at VGA resolution [17].

This result demonstrates the key benefits of properly transforming a software algorithm rather than simply porting: appropriate use of parallelism can significantly lower the clock rate, while at the same time reducing the latency. Careful transformation can also significantly reduce the resource requirements and mitigate the effects of limited memory bandwidth.

5. Summary and conclusions

The operations within the software based image processing environment will have been optimised for serial implementation within that environment. Simply porting that algorithm onto the FPGA will generally give relatively poor performance, because it will still be predominantly a serial algorithm. For many low-level image processing operations, the serial algorithm may have a relatively simple transformation to make it suitable for parallel hardware. However, for many intermediate-level operations (such as connected components labelling) the underlying algorithm may need to be completely redesigned to make it more suitable for FPGA implementation.

Despite efforts to make programming FPGAs more accessible and more like software engineering [2], efficient FPGA programming is still very difficult. This is because programming FPGAs is hardware design, not software design. Although the languages may look like software, the compilers use the language statements to create the corresponding hardware. Concurrency and parallelism is implicit in the hardware that is built, and the compilers have to build additional control circuitry to make them execute sequential algorithms. Consequently, sequential algorithms ported from software will by default run sequentially, usually at a significantly lower clock speed than high end CPUs. Relatively simple changes to the algorithm, such as pipelining and loop unrolling enable it to exploit some of the parallelism available, and this is

often sufficient to compensate for the lower clock speed of FPGAs. However, the algorithm is still largely sequential unless it has been specifically adapted for parallel execution. The software-like languages are in reality hardware description languages, and efficient design requires a hardware mindset.

While many of the more recent compilers are able to perform simple transformations automatically to a greater or lesser extent, many of the transformations that result in significant efficiency gains require a significant design effort, and cannot be easily automated.

Several key transformation principles have been illustrated through a simple connected components analysis algorithm. The resulting implementation exploits parallelism to enable the complete application to run at the pixel clock rate, significantly reducing power. The memory requirements are reduced to the extent that the image can be completely processed on-the-fly as it is streamed from the camera, without requiring any frame buffers. Consequently, the latency is also significantly reduced, with the feature data available within one row after the end of each component.

During the algorithm transformation process the designer must consider both the algorithm and the computational architecture. The algorithm is first analysed to determine the underlying structure. The individual operations are then transformed to share a compatible processing mode to enable them to be combined efficiently. Both the algorithm and the architecture are then optimized to reduce both the resources required and the latency. This process of design transformation is iterative and needs a human in the loop who understands the detailed operation of the whole algorithm and is thinking in terms of efficient hardware design.

References

[1] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," *IEEE Computer,* vol. 40, no. 3, pp. 50-57, 2007.

[2] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?" *IEE Electronics & Communication Engineering Journal,* vol. 14, no. 4, pp. 165-173, Aug 2002.

[3] D. Pellerin and S. Thibault, *Practical FPGA programming in C*. Upper Saddle River, New Jersey, USA: Parson Education, 2005.

[4] T. Bollaert, "Catapult synthesis: a practical introduction to interactive C synthesis," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer, 2008, pp. 29-52.

[5] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design & Test of Computers,* vol. 23, no. 5, pp. 375-383, 2006.

[6] A. Rosenfeld and J. Pfaltz, "Sequential operations in digital picture processing," *Journal of the Association for Computing Machinery,* vol. 13, no. 4, pp. 471-494, 1966.

[7] A. J. McCollum, C. C. Bowman, P. A. Daniels, and B. G. Batchelor, "A histogram modification unit for real-time image enhancement," *Computer Vision, Graphics and Image Processing,* vol. 42, no. 3, pp. 387-398, 1988.

[8] M. Weinhaudt and W. Luk, "Memory access optimisation for reconfigurable systems," *IEE Proceedings - Computers and Digital Techniques,* vol. 148, no. 3, pp. 105-112, 2001.

[9] D. G. Bailey, "Image border management for FPGA based filters," in *6th International Symposium on Electronic Design, Test and Applications*, Queenstown, New Zealand, 2011, pp. 144-149.

[10] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Combining data reuse with data-level parallelization for FPGA targeted hardware compilation: a geometric programming framework," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* vol. 28, no. 3, pp. 305-325, 2009.

[11] D. G. Bailey, "Raster based region growing," in *6th New Zealand Image Processing Workshop*, Lower Hutt, New Zealand, 1991, pp. 21-26.

[12] I. E. Abdou and W. K. Pratt, "Quantitative design and evaluation of edge enhancement / thresholding edge detectors," *Proceedings of the IEEE,* vol. 67, no. 5, pp. 753-763, 1979.

[13] ISO, "JPEG 2000 image coding system - part 1: core coding system," ISO/IEC 15444-1:2000, 2000.

[14] C. T. Johnston, D. G. Bailey, and K. T. Gribbon, "Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation," in *Image and Vision Computing New Zealand*, Dunedin, New Zealand, 2005, pp. 422-427.

[15] K. Appiah, A. Hunter, P. Dickenson, and J. Owens, "A run-length based connected component algorithm for FPGA implementation," in *International Conference on Field Programmable Technology*, Taipei, 2008, pp. 177-184.

[16] J. Trein, A. T. Schwarzbacher, B. Hoppe, K. H. Noffz, and T. Trenschel, "Development of a FPGA based real-time blob analysis circuit," in *Irish Signals and Systems Conference*, Derry, Northern Ireland, 2007, pp. 121-126.

[17] N. Ma, D. Bailey, and C. Johnston, "Optimised single pass connected components analysis," in *International Conference on Field Programmable Technology*, Taipei, Taiwan, 2008, pp. 185-192.

[18] V. Khanna, P. Gupta, and C. J. Hwang, "Finding connected components in digital images by aggressive reuse of labels," *Image and Vision Computing,* vol. 20, no. 8, pp. 557-568, 2002.

[19] J. Trein, A. T. Schwarzbacher, and B. Hoppe, "FPGA implementation of a single pass real-time blob analysis using run length encoding," in *MPC-Workshop*, Ravensburg-Weingarten, Germany, 2008, pp. 71-77.

[20] C. T. Johnston and D. G. Bailey, "FPGA implementation of a single pass connected components algorithm," in *IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, 2008, pp. 228-231.

[21] D. G. Bailey, C. T. Johnston, and N. Ma, "Connected components analysis of streamed images," in *International Conference on Field Programmable Logic and Applications*, Heidelberg, 2008, pp. 679-682.