FPGA implementation of a Single Pass Connected Components Algorithm

Christopher T Johnston and Donald G Bailey
Institute of Information Sciences and Technology
Massey University, New Zealand
c.t.johnston@massey.ac.nz, d.g.bailey@massey.ac.nz

Abstract

The classic connected components labelling algorithm requires two passes through an image. This paper presents an algorithm that allows the connected components to be analysed in a single pass by gathering data on the regions as they are built. This avoids the need for buffering the image, making it ideally suited for processing streamed images on an FPGA or other embedded system with limited memory. An FPGA-based implementation is described, emphasising the modifications made to the algorithm to enable it to satisfy timing constraints.

Keywords: Connected Components Analysis, Image Segmentation, Stream Processing, FPGA

1 Introduction

Connected components analysis is an important step in many image analysis applications. There are typically four stages in such algorithms. First the input (colour or greyscale) image is preprocessed through filtering and thresholding to segment the objects from the background. The preprocessed image is usually binary, consisting of a number of regions against a background. Next, each connected group of pixels is assigned unique label, enabling the distinct objects to be distinguished. In the third stage, each region is processed (based on its label) to extract a number of features of the object represented by the region (for example, area, centre of gravity, bounding box, etc). In the final stage, these features are used to classify each region into one of two or more classes.

The classic connected components labelling algorithm [1] requires two raster-scan passes through the image. In the first pass, when an object pixel is encountered, the 4 neighbours that have already been processed (see Figure 1) are examined to determine the label for the current pixel. For a "U" shaped object, each of the branches of the "U" will have a different label, and when they join at the bottom these labels must be merged. One of the two labels will continue to be used, and all instances of the other label need to be replaced with the label that was retained. Since many mergers may occur in processing an image, relabelling is deferred so that all of the merged labels may be changed at once. A merger table records such mergers, and is used to relabel all of the pixels within the image consistently in the second pass.

The preprocessing operations (typically filters and point operations) map well to stream-based processing without image buffering (apart from row caching for local filters). This makes them well suited for implementation on an FPGA where memory resources are limited. Jablonski and Gorgon [2] have implemented this classic two-pass algorithm on an FPGA. This uses stream-based processing, with a small local window in the first pass, and a point operation (the merge table lookup) in the second pass, giving a latency of one frame.

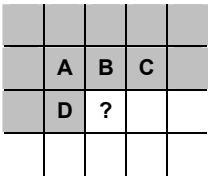

Figure 1: A label is assigned to the current pixel based on already processed neighbours.

Several high-speed parallel algorithms exist for connected components labelling (see for example the review in [3]). While such algorithms give considerable speed improvement over the classic algorithm, this speedup is achieved through massive parallelism. Such parallelism is very resource intensive, requiring a large number of essentially identical processors. There is also the bandwidth bottleneck in reading the image data into the FPGA.

At the other extreme, a resource efficient iterative algorithm has been implemented on an FPGA [4,5]. This uses very simple processing, but requires multiple passes through the image to completely label an image. Being iterative, it requires a frame buffer to store the intermediate image between passes. The number of passes required depends on the complexity of the region shapes, making such an algorithm unsuited for real-time processing.

A modification [6] to the classic two-pass algorithm enables a single pass implementation, eliminating the need for a frame buffer, and significantly reducing the latency. A single pass algorithm must extract the features of interest for each component while determining the connectivity [7]. This removes the need for producing a labelled image and avoids the

0-7695-3110-5/08 $25.00 © 2008 IEEE
DOI 10.1109/DELTA.2008.21

second, relabelling pass. However, it requires merging and relabeling on the fly to ensure that consistent results are obtained.

2 Single pass algorithm

The single-pass connected components algorithm [6] has five main blocks, as shown in figure 2:

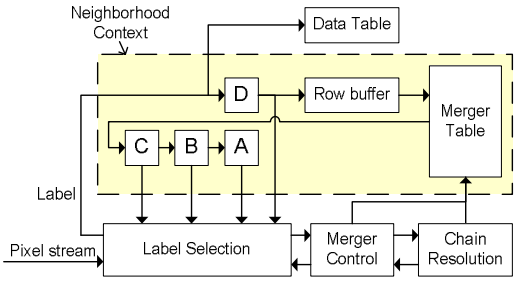

Figure 2: Architecture of the single pass algorithm.

1. The neighbourhood context provides the labels of the four pixels adjacent to the current pixel. A row buffer caches the labels from the previous row. These must be looked up in the merger table to correct the label for any mergers since the pixel was cached.
2. Label selection assigns the label for the current pixel based on the labels of its neighbours. This follows the classic two-pass algorithm:
   - background pixels are labelled 0;
   - if all neighbouring labels are background a new label is assigned;
   - if only a single label appears among the labelled neighbours, that is selected;
   - if the neighbours have two distinct labels, those regions are merged, and the smaller of the two labels is retained and selected.
3. The merger control block updates the merger table when two objects are merged. The larger (redundant) label is modified to point to the smaller label used to represent the region.
4. When a series of mergers occur with the smaller label on the right, it is possible for a chain of mergers to result. Such mergers are pushed onto a chain stack. These chains are then resolved during the horizontal blanking period by visiting them in reverse order, updating the merger table so that each of the old labels points to the smallest label.
5. The data table accumulates the data required to calculate the features of each connected component. As each pixel is processed, the data table is updated to reflect the current state of each region. When regions are merged, the corresponding data table entries are also merged to reflect the combined object.

3 Algorithm mapping

The algorithm described above was implemented onto FPGA hardware in three stages. This enabled the main blocks of the algorithm to be developed and tested independently. In the first stage the main labelling feedback loop, including the merger table, was implemented. The second stage added the stack for handling label chaining. The third stage introduced the data table, initially to enable "blob" counting, and then to measure the area of each region.

We implemented this algorithm on a Celoxica RC100 development board (Spartan-II XC2S200 FPGA) using Handel-C as the hardware description language. Our aim was to operate on a 670 by 480 image on the fly at 60 frames per second (VGA output rates).

To debug and validate the algorithm, a simple test image was created that contained patterns which exercised the merging and chaining logic. The initial labels assigned to each pixel were displayed as different colours, with new labels and mergers highlighted when these events occurred. This enabled a visual verification that the labelling process was operating as expected. The data extracted from each region was verified by placing a breakpoint within the simulator, and using the debugger to examine the merger and data tables. The number of new labels and mergers were also displayed on the RC100 seven segment display.

3.1 Object Labelling

The basic algorithm was implemented as described above. Separate registers were used for the neighbourhood pixels, A, B, C, and D. The row buffer was implemented using a dual-port BlockRAM.

Assuming that the image is filtered prior to connected component analysis, there should be few, if any, noise regions in the image. This allowed us to use a small merger table with only 256 entries. The merger table was also implemented using a dual-port BlockRAM with an early read and late write.

The first implementation did not meet timing constraints for the FPGA. The critical path was the feedback loop for labelling the present pixel. The propagation delay was reduced by calculating the minimum and maximum of A and C in the previous clock cycle. This required reading from the row buffer and merger table one clock cycle earlier and storing it into an extra register, pre-C. Pre-C and B (which on the next clock will be C and A) are then compared to determine the minimum and maximum which are then registered and compared against D in the next clock cycle. A second issue was looking up the value from the row buffer in the merger table. This requires two clock cycles (if both are stored in BlockRAM) or moving the merger table to fabric RAM (RAM implemented using the configurable logic blocks of the FPGA).

3.2 Chain Resolution

Chains within the merger table (such as Figure 3) must be unlinked. Labels 3 and 4 must point directly to label 1 before the next row of the image is processed. Otherwise, when they are read from the row buffer and looked up in the merger table, the incorrect label will be returned.

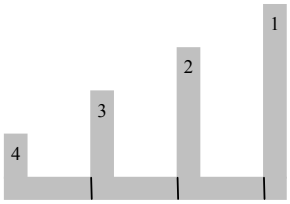

Figure 3: Merger chain: 4=>3; 3=>2; 2=>1.

Potential links (mergers with the smaller label on the right) are pushed onto a stack. Then during the horizontal blanking period, each potential link is revisited in reverse order. Resolution of mergers requires 3 clock cycles per stacked entry. The steps, within a pipelined implementation [6] are:

1. The index (the larger of the merged pair) and merger target (the smaller of the pair) are popped off the stack.
2. If the target is the same as the index from the previous cycle, the resolved target is the resolved target from the previous cycle. This data caching prevents trying to read a value that has not yet been written to the merger table. If the target is different from the previous index, the target is looked up in the merger table to obtain the resolved target.
3. If the resolved target is different from that popped off the stack, the resolved target is saved at the indexed location in the merger table.

With an early read and a late right for the merger table, steps 2 and 3 can be combined into a single clock cycle, reducing the latency.

The stack was implemented using a BlockRAM. While for typical images, the number of mergers requiring unchaining is low, we have set the stack size to 128 mergers, which is about the limit that can be processed during the horizontal blanking period.

In [6] we showed that the number of potential links is typically low even for quite complex images. For comparison, an alternative implementation placed the stack in fabric RAM with 16 entries, reducing the number of BockRAMs required.

3.3 Region Count Data Table

The data table implementation depends significantly on the data required for each region. The simplest analysis is a count of the regions present in the image. The number of mergers is counted, and this is subtracted from the number of labels generated to give the region count.

3.4 Region Area Data Table

A more realistic analysis is to measure the area of each connected component. The region data for this consists of a count of the number of pixels within the region. As only one region is operated on at a time, these can be placed in dual-port BlockRAM. The data table has 256 entries (the same as the number of labels) and is 18 bits wide to accommodate regions that occupy most of the image.

To avoid having to do two reads and a write when a merger occurs (to combine the areas for the two merged regions), the label and partial data for the current region are cached in registers. The area is accumulated for the different scenarios as follows:

- When a new label is created, it is copied into the cached label, and the cached area is set to 1.
- If a merger occurs, the data table entry for the region with the higher label is read and added to the cached area. That data table entry is also cleared to indicate that this data is not valid. The cached label is set to the lower label, as this is the label now associated with the data.
- Otherwise if the current label is the same as the cached label, the cached area is incremented.
- Finally, when a background pixel or the end of the row is encountered, the cached label is used to read the area from the data table. This is added to the cached area, and written back into the data table. The cached label is then set to the background.

At the end of each frame, the data can be read from the data table for subsequent processing. Since the merged regions have been reset to 0, the merger table is not needed to determine whether or not the label is valid.

4 Results

Both the blob counting and area counting implementations give the correct results when applied to the test image, both in simulation and when running on the development board. This image was designed to test the key aspects of the algorithm including: initial labelling, label propagation within a region; label merging, resolution of chains of mergers, and maintaining a data table.

4.1 Resource Utilisation

Table 1 compares the resource requirements for the different implementations. Note that these include the display logic, input pixel logic, and seven segment display logic which are common to all of the implementations.

Table 1: Resources used for each of the connected component analysis implementations.

Resource		Blob counting			Region areas		
		Merger table in fabric RAM	Stack in fabric RAM	Both in BlockRAM	Merger table in fabric RAM	Stack in fabric RAM	Both in BlockRAM
BlockRAMs		3	3	4	5	5	6
Flip-flops		267	271	279	295	299	306
LUT cells used for:	Logic	734	597	613	895	735	751
	RAM	256	16	0	256	16	0
	ROM	7	7	7	7	7	7
	Total	997	620	620	1158	758	758

The row buffer required 2 BlockRAMs because the width of the image is greater than 512 pixels, requiring a 10 bit address for the row buffer. With 256 labels, the row buffer is 8 bits wide. If more labels were needed, this would increase to 4 BlockRAMs to account for the greater bit width.

The merger table required 256 x 8 memory bits. When implemented in fabric RAM, this required 256 LUT cells. Alternatively, with 256 labels the table easily fits within a single BlockRAM. However if 512 labels were used, it would require 2 BlockRAMs because of the increased data width.

With 256 labels, the stack needs to be 16 bits wide to store 2 labels, so nicely fits the width of a BlockRAM. However, as typical images require few stack values, the stack also fits well into fabric RAM requiring only 16 LUT cells. If the number of labels was increased to 512, this would only increase slightly to 18 LUT cells. If processing comb-shaped regions, the stack depth would need to increase significantly, having a moderate impact on LUT resources.

To measure the region areas, the 18 bit width requires 2 BlockRAMs. Increasing the number of labels to 512 would double the BlockRAM requirements. More complex region analysis, for example the centre of gravity, would require a wider data table. This will have a significant impact on the number of BlockRAMs required with 1 BlockRAM needed for each 16 bits of data (with 256 labels). This may limit the complexity of embedded processing.

The 7 LUT cells used for ROM in the above table are used for the 7 segment display decoding.

5 Discussion

The conversion of an existing algorithm to one that can be streamed is a non-trivial task. The mapping of this algorithm onto the FPGA is also non-trivial with the architecture design being a major factor in whether it will meet timing constraints, or fit within the resources available on the FPGA. Both logic and memory type need to be carefully considered in order to meet timing requirements.

The restriction to 256 labels within the image was primarily for demonstration purposes. For relatively simple and clean images this is sufficient. The memory requirements of the merger and data tables will grow in proportion to the number of labels required. This would require a larger FPGA if it was necessary to be able to handle every case. In the algorithm implemented, we also made the assumption that only a few merger chains require resolution.

In summary, we have demonstrated that connected components analysis can be practically implemented on an FPGA without the need for a frame buffer.

6 Acknowledgements

The authors would like to gratefully acknowledge the Celoxica University Programme for generously providing the DK4 Design Suite, and the Xilinx University Programme for providing the ISE Foundation.

7 References

[1] A. Rosenfeld and J. Pfaltz, "Sequential operations in digital picture processing", *Journal of the ACM*, 13(4), 471-494 (1966).

[2] Jablonski, M., Gorgon, and M., "Handel-C implementation of classical component labelling algorithm", in *Euromicro Symposium on Digital System Design (DSD 2004)*, Rennes, France, 387-393 (2004).

[3] H.M. Alnuweiti and V.K. Prasanna, "Parallel architectures and algorithms for image component labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(10), 1014-1034 (1992).

[4] D. Crookes and K. Benkrid, "FPGA implementation of image component labelling", in *Reconfigurable Technology: FPGAs for Computing and Applications*, SPIE vol 3844, 17-23 (1999).

[5] K. Benkrid, S. Sukhsawas, D. Crookes, and A. Benkrid, "An FPGA-based image connected component labeller", in *Field-Programmable Logic and Applications*. Springer Berlin, 1012-1015 (2003).

[6] D.G. Bailey and C.T. Johnston, "Single pass connected components analysis", in *Image and Vision Computing New Zealand*, Hamilton, NZ, (2007).

[7] D.G. Bailey, "Raster based region growing," in *Proceedings of the 6th New Zealand Image Processing Workshop*, Lower Hutt, NZ, 21-26 (1991).