D. G. Bailey, C. T. Johnston, 'Single Pass Connected Components Analysis', *Proceedings of Image and Vision Computing New Zealand 2007*, pp. 282–287, Hamilton, New Zealand, December 2007.

Single Pass Connected Components Analysis

D.G. Bailey, C.T. Johnston

Institute of Information Sciences and Technology, Massey University,
Private Bag 11222, Palmerston North 4442, New Zealand

Email: d.g.bailey@massey.ac.nz, c.t.johnston@massey.ac.nz

Abstract

The classic connected components labelling algorithm requires a minimum of 2 passes through an image. This paper presents a modification of this algorithm that allows the resolution of merged labels to be deferred. This enables the subsequent data analysis step to be combined with the labelling procedure, with the result that connected components can 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 the algorithm ideally suited for processing streamed images on an FPGA or other embedded system with limited memory. It is shown that an FPGA-based design could use 1 clock cycle per pixel with a small overhead to manage merging of "U" shaped components. It is demonstrated that the worst case overhead is 20% of the image, although for typical images the overhead is less than 1%.

Keywords: Connected Components, Image Segmentation, FPGA, Streamed Processing

1 Introduction

Connected components labelling is an important step in many image processing 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, connected components labelling is used to assign each region a 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 algorithm [1] requires two raster-scan passes through the image. In the first pass, when an object pixel is encountered, the 4 neighbours (assuming 8-connectivity between object pixels) that have already been processed are examined (see Figure 1) If one of those is already labelled, the label is copied to the current pixel. If none are labelled, a new label is assigned to the current pixel. However, a complication occurs when a "U" shaped object is encountered. 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, it is usual to defer the relabelling process so that all of the merged labels may be changed at once. A merger table is therefore used to record such mergers. When processing the next line it is important to use the correct label, so the merger table is usually constructed as a look-up table, with the label recorded for the previously processed pixels looked up to derive the correct (merged) label for a connected component. In the second pass through the image, the merger table is used to relabel all of the pixels within the image consistently.

The preprocessing operations (typically filters and point operations) are ideally suited for stream-based processing without image buffering (apart from row caching for local filters). This makes them an ideal candidate for implementation using an FPGA or other embedded processor where memory resources are limited. If the connected components algorithm could be implemented within a stream-processing framework then it may be possible to eliminate the need for image buffering altogether. This would allow an FPGA based system to process a progressively scanned image directly from the camera without any off-chip buffering.

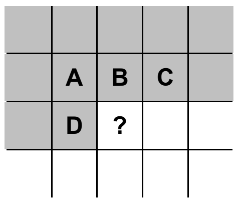

Figure 1: A label is assigned to the current pixel based on already processed neighbours.

While several high-speed parallel algorithms exist for connected components labelling (see for example the review in [2]), such algorithms are very resource intensive, requiring massively parallel processors. This makes them less suitable for FPGA-based implementation because of the bandwidth bottleneck in reading in the image data.

Crookes, Benkrid, et al. [3,4] have implemented a resource efficient iterative algorithm on an FPGA. This uses very simple local processing, but requires an indeterminate number of passes to completely label the image. This makes such an algorithm unsuited for real-time processing. The iterative nature of the algorithm also requires the intermediate images to be buffered between passes.

Jablonski and Gorgon [5] have implemented the classic two-pass connected component labelling on an FPGA. In doing so, they were able to take advantages of the parallelism offered by FPGA-based processing to gain considerable processing efficiencies over a standard serial algorithm. However, their two-pass algorithm still requires the image to be buffered for the second pass, and requires two clock cycles per pixel plus a small overhead for region merging.

To achieve a single pass streamed algorithm the features of interest for each component or region must be extracted while performing the connected components analysis [6]. This removes the need for producing a labelled image and saves having to perform the second, relabelling pass. It is also necessary to perform merging and relabelling on the fly to ensure that consistent results are obtained.

The remainder of this paper describes the details of a new, single pass algorithm that combines the connected components labelling step with the subsequent analysis step. Section 2 describes the basic architecture of our solution, and describes each of the components. Section 3 describes an issue with successive chains of mergers, and presents a solution how this can be resolved at the end of each row (during the horizontal blanking period for real-time processing). The worst-case, and typical processing times and resource requirements are discussed.

2 Single pass algorithm

The single-pass connected components algorithm has four main blocks, as shown in figure 2:

1. The neighbourhood context block provides the labels of the four pixels connected to the current pixel being processed.
2. The label selection block selects the label for the current pixel based on the labels of its neighbours.
3. A merger control block is required to update the merger table whenever two objects are merged.
4. The data table accumulates the raw data from the image required for calculating the features of each connected component. Since the image data is not retained, it must be accumulated for each connected component as the image is scanned.

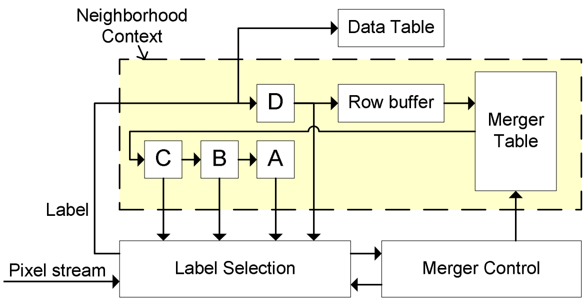

Figure 2: Basic architecture of the single pass algorithm.

2.1 Neighbourhood context

The neighbourhood context is implemented in much the same manner as a window filter. The neighbouring pixel labels are stored in registers A, B, C, and D. These are shifted along each clock cycle as the window is scanned across the image. Since the resultant labels are not saved, the labels from the previous row must be cached using a row buffer. The labels from the cache must also be looked up in the merger table to correct the label for any mergers since the pixel was cached.

2.2 Label selection

The label for the current pixel follows that of the classic connected components algorithm [1]:

- Background pixels are given a label of 0.
- If all of the neighbouring pixels are background, then a new label is assigned.
- If only a single label is used among the labelled neighbours, that label is also assigned to the current pixel.
- If two different labels are used among the neighbours, then this indicates a merger condition. For reasons that will be explained later, we assign the smaller of the two labels to the current pixel.

In practise, the selection logic may be simplified by considering the neighbourhood pixels in a particular order [7]. A merger will only occur if pixel B is background and the label of C is different from either A or D. In all other cases, the labels will already be equivalent from prior processing.

2.3 Merger control

The merger table is used as a look-up table on the output of the row-buffer. This ensures that the correct label is used for any labels that have been stored in the row buffer which have subsequently been merged.

Whenever a new label is created, a new entry is added to the merger table pointing to itself. This avoids the need for initialising the merger table prior to processing. To keep the merger table up to date, whenever a merger occurs, the label that is replaced should subsequently point to the merged label.

This means that in addition to reading from the merger table every clock cycle, there is also the need to write on some clock cycles. Since only one access may be made to memory each clock cycle, this requires either running the merger table RAM at twice the clock frequency, or using dual-port RAM with one port for the read and one for the write.

When a merger occurs, the new label must replace all occurrences of the old label within the neighbourhood. This requires a multiplexer on the input of register B. Register A does not require a multiplexer because B is always a background pixel whenever regions are merged.

Timing considerations also require a multiplexer on the input of register C. The next value for C is being read at the same time that the merger table is being updated. Therefore the value read from the merger table will reflect the label before the merger. So if the next value for C is not the background label, the newly merged label should also be copied to C.

It is essential that the correct label be selected to represent the region when a merger occurs. Consider the "W" shaped object in Figure 3(a). First the left two branches merge, and then the result is merged with the right branch. After a series of such mergers, it may be necessary to recursively look up the cached label multiple times to determine the correct new label. At first glance it would appear best to use the label from the left branch (from register A or D) as the merged label as this would avoid such a chaining effect. However, a counter-example for this is given in Figure 3(b). On the first merger, region 1 would be merged with region 2, and the label 2 (the left label) would represent the region. On the second merger, the 1 is changed to a 2 from the merger table, and label 2 is merged left into label 3. At the third merger, the 1 is again changed to a 2, which is merged into label 4. However label 2 is no longer valid as it has just been merged with label 3, requiring a second lookup to get the correct result.

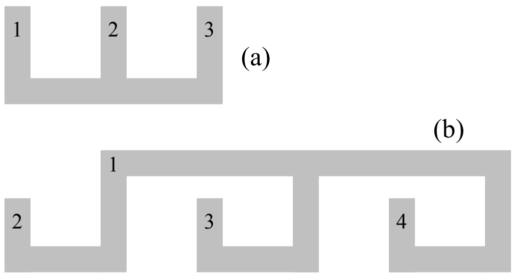

Figure 3: Chaining effect of multiple merges: (a) 1=>2; 2=>3; subsequent label 1s require multiple merger table lookups. (b) Merging left counter-example: 1=>2; 1(2)=>3; 1(2)=>4 resulting in an incorrect merger table.

To avoid this problem, we select the smaller of the two merged labels to become the new label. Since the image is scanned in a raster order, the smaller label will represent the object closer to the top of the image.

Consider the case where the smaller label is on the right (Figure 4(a)): the region labelled 2 will always begin lower than region 1, so there can be no more mergers with region 2 on the current scan-line. In Figure 4(b)-(d) the smaller label is on the left in the first merger, so there could potentially be additional mergers. Because the smaller label is on the left, all instances of the larger label will be changed to the smaller by the merger lookup table before the next merger. For example, in case (d), label 3 will be correctly linked to label 1.

The chaining effect still occurs when the larger label is on the left, but this does not need to be resolved until the end of each row. Resolution of chaining will be described in section 3.

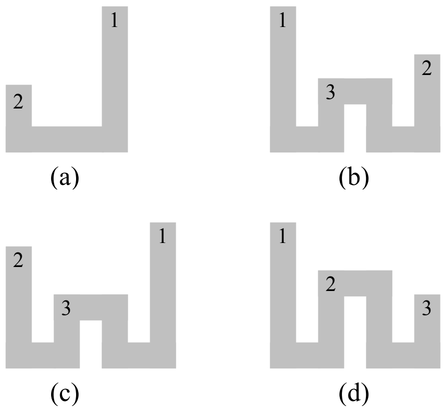

Figure 4: Merging scenarios: (a) the smaller label on right; (b) – (d) the smaller label on the left in the first merger.

2.4 Data table

As each pixel is processed, the data table is updated to reflect the current state of each region. The data that needs to be maintained depends on the features that are being extracted from each connected component.

If only the number of regions is required, this can be determined with a single global counter. Each time a new label is assigned, the counter is incremented, and when two regions merge, the counter is decremented. For measuring the area of each region, a separate counter is maintained for each label. When two regions are merged, the counts are combined. For determining the centre of gravity of the region, the sums of the x and y coordinates are maintained. At the end of the image, these can be divided by the area to give the region centre. To obtain the bounding box of each region, the extreme coordinates of the pixels added are recorded. Other, more complex, features may be determined in a similar manner by accumulating the appropriate raw data.

Each update of the data table requires a read of the existing data followed by a write of the updated data. However, by caching the data for the most recent label and deferring the write until the next background pixel is encountered, at most one memory access of the data table is required per clock cycle. This enables a single-port RAM to be used for data storage. The timing for this process is illustrated in Figure 5. When a new region is encountered, the read is not required, and the new data entry is created. As shown in the right part of Figure 5, there is sufficient time for merging the data of merged regions on the fly because the merged region will always be at least 3 pixels wide.

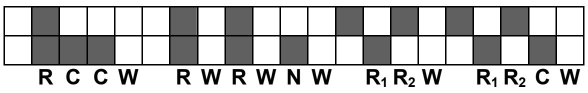

Figure 5: Timing for data table updates. **R** = Read, **C** = from Cache, **N** = create New value, **W** = Write.

2.5 Table sizes

Both the merger and data tables require one entry for each label used. The worst case for the number of objects possible in an $M \times N$ image is illustrated in Figure 6 and requires

$$Number = \text{ceil}\left(\frac{M}{2}\right) \times \text{ceil}\left(\frac{N}{2}\right) \quad (1)$$

objects. In practise the actual number of labels required is significantly less than this, with the number required depending on the expected region size and shape.

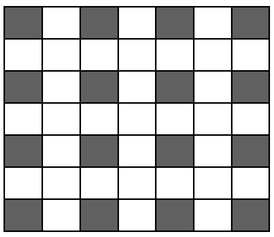

Figure 6: Worst case number of objects.

3 Handling chaining

As described in section 2.3, selecting the smaller label to represent a merged pair of regions allows the resolution of chains of mergers to be deferred. Such a chain is illustrated in Figure 7, where the vertical sub-regions are labelled independently until the last row where they are ultimately connected. At the end of this row, the merger table for labels 4 and 3 contain incorrect values of the final label for the composite region. Any such chains must be unlinked before the next row of the image is processed when the labels 3 and 4 will be looked up in the merger table.

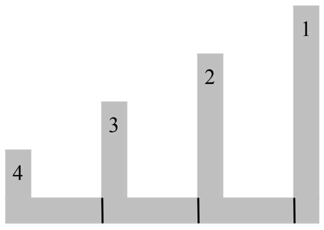

Figure 7: Merger chain: 4=>3; 3=>2; 2=>1.

The simple approach would be to search the whole merger table at the end of each line to resolve such chains. However, as shown in section 2.5, the table is potentially quite large requiring significant time for the search and update.

The search can be avoided by recording only the new mergers for each row. Only the mergers with a smaller label on the right (Figure 4(a) and the second merger in Figure 4(c)) need to be recorded because, as explained earlier, the other mergers resolve themselves by storing the minimum label in the row buffer. Therefore, since all of the mergers that must be resolved have a smaller label on the right, they can be processed most efficiently by considering them from the right first (in the reverse order that the mergers occurred). Pushing the relevant mergers onto a stack, and then popping them off the stack to process them achieves this.

Resolution of mergers requires 3 clock cycles per stacked entry:

1. The index (the larger of the merged pair) and merger target (the smaller of the pair) are popped off the stack.
2. 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.

Since the merger table is dual-port, these three steps can be pipelined with a throughput of one merger per clock cycle (step 2 uses the read port, and step 3 uses the write port of the RAM). In the pipelined implementation, step 2 needs to be modified because it may be trying to read the value that is currently being written in step 3 of the previous cycle:

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. Otherwise lookup the target to obtain the resolved target.

This is illustrated with a more complex example in Figure 8 where multiple separate chains merge. The chain stack and merger table before and after processing are shown in Figure 9.

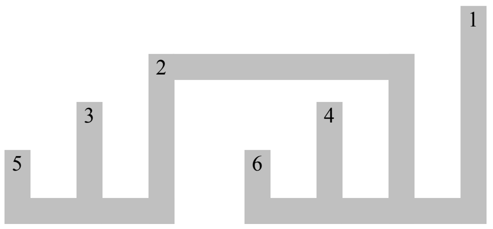

Figure 8: A more complex unchaining example

Stack

①	2->1
②	4->2
③	6->4
④	3->2
⑤	5->3

Merger table

Index	Target	Updated target
1	1	
2	1	① No change
3	2	④ ->1
4	2	② ->1
5	3	⑤ ->1
6	4	③ ->1

Figure 9: The stack at the end of processing Figure 8, showing how the merger table is updated through unchaining. The circled numbers indicate the order in which the mergers are processed.

3.1 Unchaining resources

In the worst case, every merger may need to be stacked, and there may be up to $N/2$ mergers in the longest chain. The chain stack therefore needs to be able to hold this many entries. The data width of the stack is twice that of the merger table because it needs to hold both an index and its target.

Processing the stack using the pipelined procedure would require $N/2+2$ clock cycles in the worst case. However this overhead will not apply to every line because the index region (with the larger label) always starts on a lower row than the target region so a chain of length *k* will by necessity occupy at least *k* rows.

When processing streamed data from an analogue camera, the horizontal blanking period is typically 20% to 25% the length of the active portion. For most images, this blanking period would provide ample time for the unchaining process. Unfortunately, the worst case unchaining time can be up to 50% of the processing required for the active portion of the row. As the unchaining must be completed before the next row can be processed, this may result in the loss of data.

If one line has an excessive number of chained mergers, the next few lines will by necessity require fewer. Therefore this problem may be overcome by using a FIFO buffer on the streamed input to enable the next row to be delayed. To avoid loss of data, the maximum length of the FIFO needs to be 30% to 50% the length of a row, depending on the length of the horizontal blanking period.

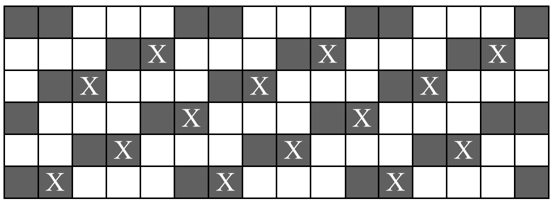

Figure 10: Worst case total chained mergers. The X mark pixels where mergers occur.

It can be shown that the worst case total number of chained mergers would require on average $N/5$ merger resolutions per line (Figure 10). This means that a horizontal blanking period with a length of 20% of the active portion is sufficient to handle the worst case image. For practical images the unchaining overhead would be significantly less than this.

3.2 Examples

To test the algorithm, a series of random images with complex shapes were created by filtering a uniform noise image with a Gaussian filter and thresholding. One example of such an image is shown in Figure 11. The number of regions was determined, and checked using a standard blob counting algorithm.

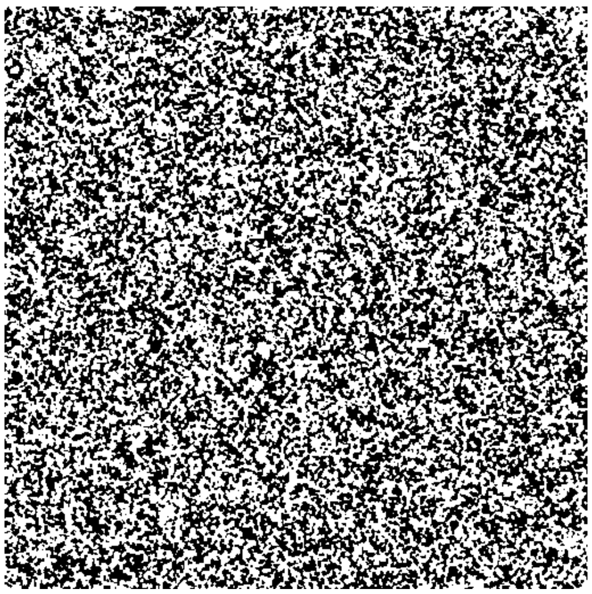
Figure 11: Random test image.

Within the 512x512 image shown in Figure 11, 1112 regions were detected, with 4304 labels required. The difference between these is the number of mergers. The ratio of 4 labels per region detected was typical for random images (over a wide range of filter widths from $\sigma = 3$ through $\sigma = 20$).

For the image in Figure 11, a total of 2210 mergers were stacked for unchaining. For this image, it was slightly higher than usual. Typically about half of the mergers required stacking for unchaining, although there was a considerable variation from image to image. The number of mergers stacked averaged 4 per row, with a maximum of 13 on some rows. This is significantly smaller than the worst case of 256 per row for this size. Such a small number can easily be processed within the horizontal blanking interval of the input video stream, suggesting that the input FIFO would not be necessary for processing typical images encountered in most applications.

The size of stack actually used depended considerably on the average object size, with the larger objects requiring fewer mergers, hence fewer mergers to be stacked. For the image above, the overhead for unchaining was less than 1% of the number of pixels, and this improved with larger object sizes.

4 Conclusions

A novel approach to combining region analysis with connected components labelling is presented. The method described avoids the need for buffering the image, enabling efficient implementation on an FPGA or other embedded system with limited memory.

The actual memory requirements depend strongly on the number and complexity of the regions expected within the image. In a real environment, there may also be a number of small noise regions that must also be processed, using both data and merger table space. For the examples tested, an 8k data and merger table would be more than adequate. The labels would therefore require 13 bits, requiring the merger table to be this wide. The data table width would depend strongly on the features being extracted from the regions. The row buffer for this example would need to be 512 x 13 bits, to hold the labels from the previous row. The size of the unchaining stack would be insignificant (16 x 26 bits).

Since the algorithm only requires one pass through the image, it is approximately twice as fast as the standard 2-pass algorithms. The algorithm is amenable to pipelining, enabling an FPGA implementation with a throughput of 1 clock cycle per pixel, with typically less than a 1% overhead for unchaining mergers each row.

We are currently in the process of implementing this design on an FPGA, and also considering ways of optimising the design further to reduce the relatively large requirements for the data and merger tables.

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.

References

[1] A. Rosenfeld and J. Pfaltz, "Sequential Operations in Digital Picture Processing", *Journal of the ACM*, **13:**(4) 471-494 (1966).

[2] 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).

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

[4] 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).

[5] M. Jablonski and M. Gorgon, "Handel-C implementation of classical component labelling algorithm", in *Euromicro Symposium on Digital System Design (DSD 2004)*, Rennes, France, 387-393 (31 August - 3 September, 2004).

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

[7] K. Wu, E. Otoo, and A. Shoshani, "Optimizing connected component labelling algorithms", in *Medical Imaging 2005: Image Processing*, **SPIE 5747:** 1965-1976 (April 2005).