Vol.1 page 68-75 2015
DOI:10.14456/randk.2015.7

Contour generation for multivalued streamed images using crack codes on a GPU

M. Usman Butt[a], John Morris[b,*], Nitish Patel[a] and Morteza Biglari-Abhari[a]

[a]Electrical and Computer Engineering, The University of Auckland, New Zealand
[b]Faculty of Engineering, Mahasarakham University, Thailand

(Received 9 February 2015; accepted 10 March 2015)

Abstract - We describe an efficient GPU algorithm which extracts multiple contours from an image. The algorithm uses *crack codes* to generate contours which sit logically between adjacent image values; it works scan line by scan line and it can generate multiple contours in parallel with an image streamed directly from a camera. Whilst specifically targeted at detecting object contours in stereo disparity maps, it can also be used for general segmentation with a trivial change to the code generating the crack code masks. Using a 480 ALU 1.4 GHz nVidia GPU, it can generate ~ 25000 contours from a real 2048 × 768 resolution 128 level disparity map image in ~ 29 ms if the contours are further processed in the GPU or ~ 39 ms if contours are transferred to the host - typically ~ 40 times faster than an OpenCV CPU implementation.

Keywords: Image contour generation, GPU, crack codes.

1. Introduction

Intelligent safety systems should be able to anticipate hazards. Careful drivers pay more attention to a pedestrian standing on the curb (but not yet moving) than a postbox. Recognizing shapes from 3D contours is one key step to detecting a potential hazard. Here, we describe a first step by detecting high resolution contours in *real-time* from stereo systems. Conventionally contours have been represented by a *chain code*, containing a starting point and a sequence of unit vectors specifying the next pixel in either a four- or eight-neighborhood along the boundary. The chain based boundary representations include Freeman chain codes (H. Freeman, 1961), crack codes (Wilson and Batchelor, 1990) and mid-crack codes (Shih and Wong, 1992). Several contour generation algorithms have been described by Rosenfeld for pixel to vector conversion (Rosenfeld, 1978). Sobel exploited chain codes to extract region boundaries in binary images (Sobel, 1978). Morrin (Morrin *et al*.,1976) used chain-link encoding for monochrome image compression. These methods, following border pixels in the entire image, require memory for the whole image. The buffer size can be reduced by searching for object boundaries in raster scan mode requiring a buffer for only few lines but this increases algorithmic complexity. Cederberg used a raster scanned approach to chain coding (RC-code) for image compression (Cederberg, 1979). Instead of building the Freeman chaincode, the RC-code works in four directions using a 3 × 3 window, requiring a buffer of two scan lines instead of the entire image, with a series of 20 templates to generate and extend the chain fragments. Chakravarty (Chakravarty, 1981) developed a single-pass algorithm for converting region boundaries in a grey level image into chain-coded line structures by convolving a 3×3 window in a raster fashion and used a 2×2 subwindow to further link these chain-coded lines to complete contours. Pavlidis (Pavlidis, 1978) developed a two-pass algorithm for boundary tracing which encodes the relationship between adjacent lines into a Line Adjacency Graph and requires only one scan line in the buffer at a time. Another two-phased algorithm requiring two buffered scan lines generates chain codes from a runlength coding representation from a 3 × 2 window and preserves connectivity information between runs (Kim *et al*., 1988). These algorithms handle binary images only. Contour generation on multivalued images has been hardly discussed. A naive approach transforms a Δ grey level image into Δ binary images by thresholding and contouring the binary images: this is trivial but slow. Zingarett *et al*.(1998) converted a rasterized multivalued image into crack run-length codes with a high level of complexity. Khan *et al*.(2009) developed an efficient code, 'Salmon', which, simultaneously, extracts multiple contours in the Cyclopæan disparity maps generated by SDPS (Gimel'farb, 2002), but it does not work with general multivalued images. All these approaches on conventional CPUs can not meet real-time demands due to the large number of simple operations when the number of grey levels, Δ, is multiplied by the basic complexity, *O*(*WH)*. Several raster-scan based parallel algorithms - applicable to binary images only - on specialized hardware architectures have been reported (Li, *et al*.,1989; Chia, *et al*., 2003; Chen *et al*., 1993).

We designed and implemented a GPU-based contour generation algorithm in CUDA for high resolution multivalued images. Our approach uses crack run codes from which we build sets of line segments which are then

*Author for correspondence: j.morris@auckland.ac.nz

linked to form a contour. Some contour properties, *e.g.* perimeter length, area and higher moments, can be calculated in the GPU in parallel as the line segments are generated (Leu, 1991). We evaluated our approach by streaming pre-recorded image from a host to a GPU connected through a PCI-express bus. Eventually, we will combine this system with a real-time stereovision GPU code already developed (Kalarot and Morris, 2010) to enable contour maps of 3D scenes to be generated in real-time.

2. Our approach

2.1 Overall data flow

After loading a block of scan lines, we process each image in two stages. The first, highly parallel stage, all Δ contours are generated in parallel. However, long runs of horizontal contours generate data dependencies because data from previous columns may be required at the run ends and long sequences of short segments lead to many serial operations to follow lists. Initially, a single line scan time latency is introduced because windows from two adjacent scan lines are needed to generate the 'masks' which control contour processing. In the first stage, the mask is determined for each 2×2 window of adjacent pixels, this mask then selects the operations to be applied to the existing state. A 2×2 window has 16 different paths (patterns) for cracks running through it, labelled with a 4 bit tag (see Table 1). Sets of horizontal and vertical predecessors and line segments are initialized when a crack is initially detected, *e.g.* in the Top Start or Start Run XY patterns (see Table 1). These hold details of the start of each line segment and are carried as necessary from scan line to scan line as the contours are built. The simplest masks are those without corners (Inside, Top|Bottom Run, Vertical Left|Right, Outside): no changes are needed, the predecessors (if any) for contours are retained for the next window either horizontally or vertically. When a corner is detected (*e.g.* Top End), a line segment is built using data from the predecessors and the current position and saved to the *Global Memory*. If necessary, a new one created and its predecessor data stored. When segments which belong to the same contour meet at a corner, indices for their left and right segments are added to the contour lists: the left and right branches may be followed to create the full closed contour.

The second stage links all the line segments belonging to each contour and transfers them to the host. This stage requires potentially several Mbytes of contour data to be transferred but could be overlapped with computation of the next set of contours in a streaming system.

2.2 Data structures

As they are received from the camera, blocks of scan lines are transferred to *Global Memory* on the GPU. Images are logically padded with a top and left row of zero pixels: this simplifies the code and avoids some cycles that would be needed to check boundaries. It also ensures that there are $W \times H$ crack positions in a $W \times H$ pixel image. Further, to allow contours to be checked against those generated by [illegible] overwritten to zero. This results in a negligible loss of information, < 0.1%, in the high resolution images that our algorithm can handle, but host application code must be aware that contours reaching boundaries are artificially 'closed' by the zero boundary pixels. In practice, this simply means that the image is effectively ~ 0.1% smaller.

2.2.1 Masks

Masks are stored temporarily in the registers for each thread: after determining the mask, the thread will process it following the actions in Table 1.

Since Start Run XY and Top Start have the same horizontal and vertical cracks, both are assigned to mask 1.

Table 1. Patterns, corresponding 4 bit codes, actions and reassigned masks

Pattern	Name	4 bit Code	Action Order	Mask
	Inside	0000		
	Outside	1111		
	Top Start	0001	$p(H), p(V)$	1
	Start Run XY	1110	$p(H), p(V)$	$(14 \rightarrow)$ 1
	Top End	0010	$H,\ V^p \mapsto H,\ p(V)$	2
	End Run X	1101	$H,\ H \mapsto V^p,\ p(V)$	13
	Top Run	0011		
	Bottom Run	1100		
	Bottom Start	0100	$V,\ V \mapsto H^p,\ p(H)$	4
	End Run Y	1011	$V,\ H^p \mapsto V,\ p(H)$	11
	Vertical Left	0101		
	Vertical Right	1010		
	End Run X	0110	7, 14	6
	End Run Y1	1001	13, 11	9
	End Run XY	0111	$V,$ $V \mapsto H^p,$ $H,$ $V^p \mapsto H,$ E?	7
	Bottom End	1000	$V,\ H^p \mapsto V,\ H,\ H \mapsto V^p,$ E?	8

2.2.2 Line Segments

The line segments maintain the end points of horizontal or vertical segments of each contour - as in run-length codes, avoiding the need to store each point in a contour. This also means that patterns in Table 1 with blank actions are ignored in processing as the predecessor remains unchanged. A link to the next line segment is also stored with each structure to allow the final processing step to link segments into a contour. The number of line segments is potentially very large, so these are stored in the *Global Memory*, where a large block is pre-allocated for line segments. The starting point of end segment is held in the horizontal and vertical predecessors and only saved to *Global Memory* at the end of each line segment to reduce expensive accesses to the *Global Memory*. The space needed for line segments (and the size of the index

addressing them) depends on image size and complexity - potentially $O(HW\Delta)$ entries, but in practice much less than this for typical images. In our experiments, a block of space for 7×10^5 line segments (~ 40 Mbytes) sufficed for 2048×768 images

2.2.3 Predecessors

Δ horizontal predecessors, *hp*, and Δ horizontal line segment predecessors, *ahp*, are stored in 16-bit *Shared Memory*. For each of W columns, Δ vertical, *vp*, and vertical line segment predecessors, *avp*, are stored in *Global Memory*. *hp* and *vp* require 16 bits for images with dimensions $< 2^{16}$, but *ahp* and *avp* are line segment indices and may require 16-32 bits depending on total image size (and therefore number of expected line segments).

2.2.4 Contour lists

Contours are stored as line segment index pairs: one for the left and one for the right 'branch'. The line segments themselves are linked lists.

2.3 Processing

2.3.1 Parallel crack detection

We assume the image contains Δ distinct pixel 'intensities'[1] and want to create $\Delta - 1$ contours (ignoring the one around zero intensities). Once the image is loaded into the GPU, $W(\Delta - 1)$ threads are spawned to determine the crack codes for each 2×2 window. The crack codes are 4 bit binary patterns: with '1' bits for a pixel with intensity greater than or equal to the target contour label and '0' bits elsewhere - see Table 1. This definition of the masks will lead to contours that surround *all* pixels with intensity *greater than or equal* to the value which labels the contour. If we want to *segment* an image with contours surrounding an area with a specific value, we simply change the masks to be defined relative to a value *equal* to some target value. Note that the crack code addresses - and the addresses in the generated contours - correspond to the intervals *between* pixels, not the pixels themselves - see Figure 1: examples of generated mask codes are circled.

The crack codes determine the processing steps at each window. When there are no corners in a window, *e.g.* Inside, Outside, . . ., no action is required, the predecessors (if any, *e.g.* for Top Run and Bottom Run) are not changed and the next window below is processed.

2.3.2 Line segment linking

To form a contour, the line segments are formed into a linked list. In our lineby- line algorithm, multiple lists of line segments associated with many contours may be 'active' at any time requiring careful housekeeping to ensure small segments are joined correctly to form the final contour. This is more complex than conventional approaches which track the whole of one contour through the full 1For one important application - contouring depth images from stereo systems - the pixel values are actually disparities encoding depths. image before starting on a second one. Two line segments are associated with every corner and they must be linked in the right order. Line segment pointers are initialized to *N_S*. As the lists are generated, the pointers are updated but the pointer in the head remains *N_S*.

Contours can be classified as

- *boundaries* - contours which enclose values greater than the contour label value - or
- *holes* - contours which enclose values smaller than the label value.

When a corner is encountered, one line segment is terminated and another is created: *e.g.* in Top End, the horizontal line segment ends, so its data is updated and a new vertical segment is created. In these cases, one segment must be linked to another in the right order. Symbols in Table 1 indicate which of the two segments is the new 'end' of the contour linked to the preceding line segment. These conventions ensure that boundaries are linked in the clockwise direction and holes in the anti-clockwise direction.

For corners, End Run XY and Bottom End, it is necessary to determine whether the contour ends here and its branches should be added to the contour list structure. A sequential processing step is introduced here: we trace the right branch, if it reaches the current point, then the contour is complete. If it reaches an *N_S* pointer, then the contour is still 'open' and it will be closed in some later scan line. The steps to link two contours (one boundary and one hole) are shown in Figure 1, where mask codes are shown in circles and line segment indices in squares. In the notation for each diagram, links are denoted as l^{index}_{vlh}. Note that in Figure 1(c), a second list of segments starts to be formed, which is determined to be a part of a hole by mask 7.

2.4 Pseudocode

Examples of two of the mask processing steps follow. A full description of each step and the CUDA code is available (Butt *et al.*, 2014).

2.4.1 End Run XY(Crack Mask 7)

In this mask the vertical line segment is given priority. The current line segment index, n, is incremented to point to an empty line segment space, n++. Its start is taken from the vertical predecessor in the current column, x, *i.e.* $(x, vp[x])$ and the end from the current position, (x, y). It is linked to the previous horizontal segment index taken from vertical line segment predecessor, $avp[x]$. The vertical line segment is saved as { start $(x, vp[x])$, end (x, y), link $avp[x]$ } at line segment index, n. Using skip lists expands this to { start $(x, vp[x])$, end (x, y), link avp[x], ultimate link $uvp[x]$ }.

[1] For one important application - contouring depth images from stereo systems - the pixel values are actually disparities encoding depths.

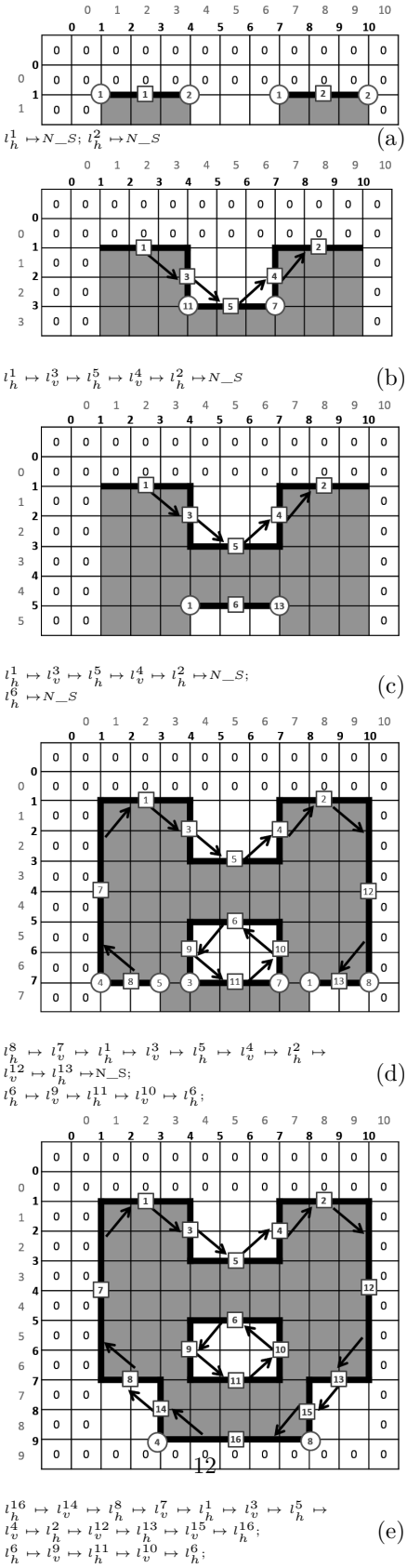

Figure 1. Line segment linking and contour forming Red circled symbols indicate the crack currently directing contour formation. Squares contain line segment indices.

The horizontal segment is formed similarly: the line segment index is incremented to point to a space for the line segment and the horizontal predecessor and the current position provide the x coordinates of it. Using *ahp*, the link in the vertical line segment at the start of the current horizontal one is updated to point to this horizontal segment. Since, two segments are joined here, we have to check whether they connect fragments of different contours or close a contour. We trace back through the predecessors of the vertical line segment. If this trace leads to the current horizontal index, n_h, then it is a closed contour.

In Figure 1(d) at location (7, 7), the links follow

$$l_v^{10} \mapsto l_h^6 \mapsto l_v^9 \mapsto l_h^{11} \mapsto l_v^{10}$$

and the contour is closed here. Otherwise, the horizontal line segment is linked to vertical line segment: see the arrow in Figure 1(b),

$$l_h^5 \mapsto l_v^4 \mapsto N\_S$$

and the contour is not complete. Algorithm 1 shows pseudocode for End Run XY, crack mask 7.

Note in Figure 1(d) that all the line segments of the inner hole are linked counterclockwise.

Algorithm 1 End Run XY: Crack Mask 7,

```
1:  n_v = n + +
2:  Write VLS: LS[n_v] := (j, vp[j], j, y, avp[j])
3:  n_h = n + +
4:  Write HLS: LS[n_h] := (hp, y, j, j, N_S)
5:  Link HLS to VLS i.e. LS[n_h].a := n_v (LS[n_h] ↦ LS[n_v])
6:  if ahp == N_S then
7:      avp[hp]=n_h
8:  else
9:      LS[ahp].a = n_h (LS[ahp] ↦ LS[n_h])
10:     Follow predecessors from VLS[n_v].a link
11:     if Link = n_h found then
12:         Closed contour
13:     else
14:         Contour still open
15:     end if
16: end if
17: clearVP(*vp,* avp, j) and clearHP(hp, ahp, tx)
```

2.4.2 Bottom End(Crack Mask 8)

This forms line segments in the same way as mask 7, but segments are linked in the opposite direction. The index of the previous horizontal line segment is taken from the vertical line segment predecessor *avp*[*x*] and its link is updated to point to the current vertical line segment. If ahp 6= *N_S*, the horizontal line segment is linked to the previous vertical line segment index taken from the horizontal line segment predecessor *ahp*. Since, two line segments are formed here, we have to check whether they connect fragments of different contours or close a contour. We trace back through the predecessors of the horizontal line segment. If this trace leads to the current vertical index, n_v, then it is a closed contour. In Figure 1(e), the links follow

$$l_h^{16} \mapsto l_v^{14} \mapsto l_h^{8} \mapsto l_v^{7} \mapsto l_h^{1} \mapsto l_v^{3} \mapsto l_h^{5} \mapsto l_v^{4} \mapsto l_h^{2} \mapsto l_v^{12} \mapsto l_h^{13} \mapsto l_v^{15} \mapsto l_h^{16}$$

and the contour is closed here. Otherwise, the current vertical line segment is linked to the horizontal line segment. See the arrows in Figure 1(d),

$$l_h^{8} \mapsto l_v^{7} \mapsto l_h^{1} \mapsto l_v^{3} \mapsto l_h^{5} \mapsto l_v^{4} \mapsto l_h^{2} \mapsto l_v^{12} \mapsto l_h^{13} \mapsto N\_S$$

and the contour is not complete. In Figure 1(e), the segments of the contour (boundary) are linked clockwise. Detailed pseudocode for crack mask 8 is presented in Algorithm 2.

Algorithm 2 `Bottom End`: Crack Mask 8,

1: $n_v := n++$
2: Write VLS: $LS[n_v] := (j, vp[j], j, y, N\_S)$
3: Link LS at location $avp[j]$ in LS array to n_vi.e. $LS[avp[j]].a = n_v$ $(LS[avp[j]] \mapsto LS[n_v])$
4: $n_h := n++$
5: Write HLS: $LS[n_h] := (hp, y, j, y, N\_S)$
6: Link VLS to n_h i.e $LS[n_v].a = n_h$ $(LS[n_v] \mapsto LS[n_h])$
7: **if** $ahp == N\_S$ **then**
8: $avp[hp] := n_h$
9: **else**
10: $LS[n_h].a := LS[ahp]$ $(LS[n_h] \mapsto LS[ahp])$
11: Follow predecessors from $HLS[n_h].a$ link
12: **if** Link $= n_v$ found **then**
13: Closed contour
14: **else**
15: Contour still open
16: **end if**
17: **end if**
18: $clearVP(^*vp, ^*avp, j)$ and $clearHP(hp, ahp, tx)$

2.5 Skip lists

Initial experiments showed significant detriments of serialization when tracing lists to determine whether contours were complete or not - see results in Section 4 later. To improve performance, we added skip lists by storing the ultimate predecessor with line segments. This adds an extra predecessor for each segment, copied from the segment to which a newly created segment is linked. The extra storage significantly decreased contour generation time by ~ 13% for typical 'real' images - see Table 2 and Table 3.

3. GPU Implementation

We targetted the family of nVidia GPUs which consist of *Streaming Multi- Processors* (SMPs) which all share a large *Global Memory* - generally several gigabytes and easily able to contain several high resolution images. Individual SMPs contain blocks of ALUs (typically 32) and share a faster, smaller *Shared Memory* or L1 cache (64kb for the GTX 480). Each SMP also has a 32K 32-bit register file. Thread instruction units can handle more threads than the number of physical ALUs and are able to rapidly switch uncompleted instructions to replace stalled instructions.

In our implementation, the inherent parallelism is $O(\Delta\ WH)$ - or $> 10^5$ - for a high resolution image. The limited *Shared Memory* capacity constrains the portion of an image and the number of vertical predecessors in the *Shared Memory*.

The host spawns $\Delta - 1$ (contours) x W (columns) threads, which are managed through CUDA. For multivalued images, all grey levels are mapped to multiple and independent thread blocks as compared to one thread block for binary images. All thread blocks process, concurrently, at a given time and each block independently processes one grey-level value assigned to it at a time.

Table 2. Processing times for CPCC and MCC for multivalued chequer board images

Multivalued Chequer board							
Size	Time, ms					Total Contours	Speedup
	CPU - CPCC			GPU			
	T'hold	Contour	Total	MCC	MCC*		
128 grey levels							
128	254	701	955	29	25	214	38
64	247	686	933	29	25	496	37
32	267	734	1001	29	25	880	40
16	253	788	1040	29	26	3550	40
8	249	821	1070	31	27	7122	40
4	258	854	1112	36	32	13186	35
2	243	943	1189	57	53	23761	22
1	261	1026	1287	145	140	40081	9
256 grey levels							
128	497	1373	1870	48	40	342	47
64	519	1392	1911	48	40	1079	48
32	509	1473	1982	48	40	1270	50
16	502	1496	1998	48	41	4895	49
8	502	1485	1987	49	42	4478	48
4	514	1568	2082	53	46	7626	46
2	518	1610	2128	69	61	13163	35
1	492	1710	2193	137	125	21833	18

Image size: 2048 × 768
Size for chequer boards - square size in pixels;

Table 3. Performance of CPCC, SC and MCC algorithms for multivalued images

Stereo Disparity Maps from SDPS					
Method	**Contour Count**	**T'hold**	**Cracks**	**Contour**	**Total**
		Time (ms)			
Scene 1					
CPCC	24481	249	-	822	1071
MCC	24481	-	7.5	26	33.5
MCC*	24481	-	7.5	21.9	29.4
SC	55963	-	-	159	159
Scene 2					
CPCC	25947	246	-	830	1076
MCC	25947	-	7.5	32.5	40
MCC*	25947	-	7.5	27.5	35
SC	57868	-	-	170	170

MCC* results are for MCC with skip lists
Image size: 2048 × 768
Contour values: 1 - 127
Note that Salmon uses some known characteristics of SDPS disparity maps to reduce noise and thus produces different contour counts.

4. Results

We compared two fast contour extraction algorithms: OpenCV Border Following (OBF) (Suzuki,1985) and Salmon (Khan *et al*., 2009) algorithms (SC). OBF algorithm is an optimized OpenCV library code which extracts the contour in three ways: chain code (CC), points of chain code (PCC) representing coordinates of each chain code, corner points of chain code (CPCC) representing coordinates of corners in chain code. They were implemented on a 3.07 GHz Intel Quad Core i7 CPU with 6 GB RAM, with our GPU code running on a GTX 480 GPU. In contrast, the GTX 480 GPU runs at 1.4 GHz frequency with 1.5 GB 1.9GHz local RAM. The constraints that the GTX 480 places on implementation have been discussed previously in Section 3.

Experiments used three binary and three multi-valued images.

4.1 Binary Images

We used three high resolution synthetic images of size 2048 × 768 pixels of increasing complexity: a chequer board (CB), horizontal lines (HL) and vertical lines (VL) (alternate black and white rows or columns). These images also provide a trivially generated ground truth for validating the code as the number of expected contours is known. We varied the square or line thicknesses to alter the number of expected contours and vary the effect of image complexity on computation time - a consequence of the sequential part of the algorithm. Table 4 reports generated contour numbers and times. Contour counts match expectation for all algorithms. It should be noted that the images with multiple small squares are a pathological case in which there are extremely large numbers of corners between pixels and very long contours - with, in the final case of single pixel squares, contours running the length of the image. This causes the final sequential linking step of our GPU algorithm to require the maximum number of cycles; all the links have to be discovered sequentially. In less artificial images, the segments surrounded by contours are typically much larger with longer run lengths, fewer line segments and fewer cycles in the sequential step. The reverse situation, in which line segments running the length of the image, so that there is a very small number of segments per contour and negligible time required for linking, is shown in the horizontal and vertical line images - see Table 4.

Performance of our GPU algorithm, labelled 'Multivalue Crack Code' (MCC) or MCC*, when skip lists were included, is compared with OpenCV routines in Table 4 for objects with various shapes and sizes. Decreasing the object size increases the number of corners and also increases the sequential component of the computation.

Table 4. MCC vs other algorithms for synthetic binary images

Binary Images						
Size	**Time, ms**					**Total Contours**
	CPU			GPU		
	CC	PCC	CPCC	MCC	MCC*	
Chequer board						
128	12	16	14	10	10	14
64	12	14	13	11	11	117
32	15	17	14	14	14	611
16	20	22	26	27	25	2729
8	34	40	35	76	69	11593
4	75	100	84	309	264	47753
2	197	256	263	1206	1040	193611
1	664	882	887	4801	4132	778384
HL						
128	13	13	15	10.3	10.3	3
64	13	13	15	10.3	10.3	6
32	16	14	22	10.4	10.4	12
16	14	16	18	10.4	10.4	24
8	17	19	20	10.5	10.5	48
4	21	26	19	10.7	10.7	96
2	29	40	28	11	11	192
1	45	67	43	12	12	383
VL						
128	11	12	11	10	10	8
64	12	14	12	10.7	10.7	16
32	13	15	14	10.8	10.8	32
16	16	18	17	11	11	64
8	17	22	19	11.2	11.2	128
4	26	31	25	11.6	11.6	256
2	41	48	36	12.5	12.5	512
1	56	80	54	15	15	1023

4.2 Multivalued Images

To test our algorithm on multivalued images, we used three sets of high resolution images:

- two synthetic 2048×768 images of a chequer board (one with $\Delta = 128$, the other with $\Delta = 256$) in which neighbouring squares have different 'intensities' and complexity increasing (as with the binary images) by decreasing the square size;
- a 256 grey level 1024 × 768 image from a real scene - Figure 2(a); and
- two 2048×768 SDPS disparity maps one containing a single human and the other with three - see Figure 2. These images represent real applications of our algorithm where the contours outline objects being tracked.

For the first two sets of synthetic images, we compared with CPCC only, because 'Salmon' relies on some properties of SDPS disparity maps (Khan *et al.*, 2009) and is compared with our GPU algorithm in the third pair of images. We compared only OpenCV's CPCC option here as it does not generate every neighbouring pixel and thus is closest to our algorithm. In all cases, contours were generated ignoring the zero 'intensity' values representing backgrounds. Results are in Table 2. For the multivalued chequer board images, our algorithm is faster: 9 to 38 times faster for 128 grey levels and 18 to 47 times for 256 grey levels. CPCC is very slow because it binarizes an image Δ times before extractimg contours. Results for a 'general' multivalued image with 256 grey levels (Figure 2(a) - left image of the stereo pair in 2(b)) are in Table 6. Both algorithms produce the same contour counts, but CPCC is ~ 10 times slower. Results for SDPS disparity images are in Table 3. Speedups over 30 (36 for scene 1 and 31 for scene 2) against the CPU alone implementation were achieved. Adding skip lists (MCC*) reduced the overall time by ~ 13% compared to MCC. Salmon is faster than CPCC, but still ~ 5 times slower than our algorithm in both scenes. The extra space consumed by the list head in the skip list implementation adds to the time to transfer contours to the host. The extra time is dependent on the number and complexity of the contours in the image. For the real scenes tested, this was approximately 0.6 ms - dependent on the number and complexity of the contours in the image. The contour data is large for typical real scenes and takes ~ 10 ms to transfer it to the host: if the contours were processed in the GPU, then this cost can be avoided.

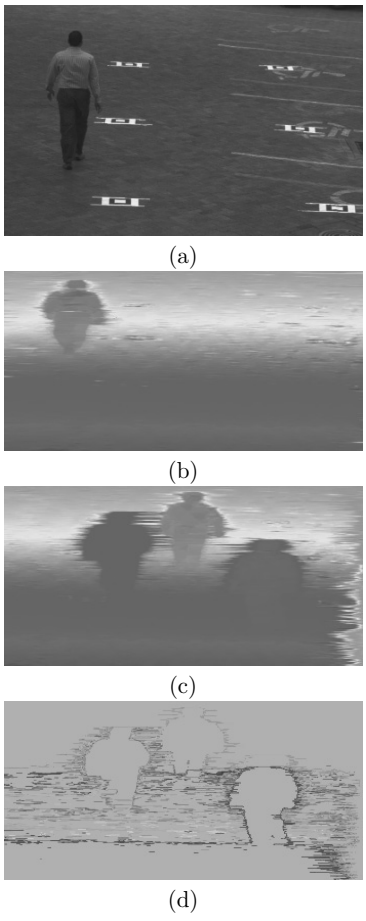

Figure 2. False colour SDPS disparity maps: (a) original left image for (b); (b) single person; (c) three people; (d) contours computed for three people map: only 25 of the full set of 128 contours are shown for visualization. Note these maps have artefacts which are typical of dynamic programming stereo images but the object outlines are clearly identified - the noise is mainly in occluded regions - and contours

5. Conclusions

We designed an algorithm to generate multiple image contours and implemented it using CUDA for a GPU. It handles images - assumed streaming scan line by scan line from a camera or other streaming source *e.g.* stereo disparity maps (Kalarot and Morris, 2010). It was verified with a set of synthetic ground truth images and validated with real images from a stereo system. The problem is a challenging one, because there is an inherent data dependency: although most image columns can be processed by independent threads, contours that inconveniently run along scan lines cause threads to block waiting for data from previous columns. Despite this, it can process high resolution real images (~ 2 Mbytes) with over 100 'intensity' values at ~ 30 fps and is therefore adequate for many real-time applications. It is typically an order of magnitude faster than CPU based code.

We will extend our work to: (1) using a wave style of computation to limit the overall cost of horizontal scans by allowing threads to start processing following scan lines

(or images) when horizontal scans cause subsequent column threads to block; (2) integrating our approach with a GPU implementation of SDPS stereo matching(Kalarot and Morris, 2010) leading to real-time contour extraction; (3) computing the contour properties such as centroid, perimeter, area and higher (shape) moments and (4) use these high resolution depth contours to enhance our previous work on object detection and tracking (Butt and Morris, 2011) to permit real time tracking of multiple objects in dense scenes.

References

Butt, M. U. and Morris, J. 2011. Precise Tracking using High Resolution Realtime Stereo. In: Image and Vision Computing New Zealand (IVCNZ2011), Auckland, New Zealand. IVCNZ, pp. 143–148.

Butt, M. U., Morris, J., Patel, N. and Biglari-Abhari, M. 2014. Contour generation for multivalued streamed images using crack codes on a GPU. The University of Auckland, Tech. Rep.

Cederberg, R. L. 1979.Chain-link coding and segmentation for raster scan Devices. Computer Graphics and Image Processing 10 (3), 224–234.

Chakravarty, I. 1981. A single-pass, chain generating algorithm for region Boundaries. Computer Graphics and Image Processing 15 (2), 182–193.

Chen, L. T., Davis, L. S. and Kruskal, C. P. 1993. Efficient parallel processing of image contours. Pattern Analysis and Machine Intelligence, IEEE Transactions on 15 (1), 69–81.

Chia, T.-L., Wang, K.-B., Chen, L.-R. and Chen, Z. 2003. A parallel algorithm for generating chain code of objects in binary images. Information Sciences 149 (4), 219–234.

Freeman, H. 1961. On the encoding of arbitrary geometric configurations. Electronic Computers, IRE Transactions 2, 260–268.

Gimel'farb, G. L. 2002. Probabilistic regularisation and symmetry in binocular dynamic programming stereo. Pattern Recognition Lett, 23 (4), 431–442.

Kalarot, R. and Morris, J. 2010. Implementation of symmetric dynamic programming stereo matching algorithm using CUDA. In: 16th Korea-Japan Joint-Workshop on Frontiers of Computer Vision, pp.141–146.

Khan, T., Morris, J., Javed, K. and Gimelfarb, G. 2009. Salmon: Precise 3D contours in real time. Dependable, Autonomic and Secure Computing, IEEE International Symposium on, vol. 0, pp. 424–429.

Kim, S.-D., Lee, J.-H. and Kim, J.-K. 1988. A new chain-coding algorithm for binary images using run-length codes. Computer Vision, Graphics, and Image Processing 41 (1), 114–128.

Leu, J.-G. 1991. Computing a shape's moments from its boundary. Pattern Recognition 24 (10), 949–957.

Li, H. F., Jayakumar, R. and Youssef, M.1989. Parallel algorithms for recognizing handwritten characters using shape features. Pattern recognition 22 (6),

Morrin, I. *et al*. 1976. Chain-link compression of arbitrary black-white images. Computer Graphics and Image Processing 5 (2), 172–189.

Pavlidis, T. 1978. A minimum storage boundary tracing algorithm and its application to automatic inspection. IEEE Trans. Sytems, Man, and Cybernntics 8 (1), 66–69.

Rosenfeld, A. 1978. Algorithms for image/vector conversion. In: Proceedings of the 5th annual conference on Computer graphics and interactive techniques. ACM, pp. 135–139.

Shih, F. Y. and Wong, W.-T. 1992. A new single-pass algorithm for extracting the mid-crack codes of multiple regions. Journal of Visual Communication and Image Representation 3 (3), 217–224.

Sobel, I.1978. Neighborhood coding of binary images for fast contour following and general binary array processing. Computer graphics and image processing 8 (1), 127–135.

Suzuki, K. A. S. 1985. Topological structure analysis of digitized binary images by border following. Computer Vision, Graphics, and Image Processing 30, pp. 32–46.

Wilson, G. and Batchelor, B. 1990. Algorithm for forming relationships between objects in a scene, Computers and Digital Techniques. IEE Proceedings E 137(2), 151–153.

Zingaretti, P., Gasparroni, M. and Vecci, L. 1998. Fast chain coding of region Boundaries. IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (4), 407–415.