Center for Embedded and Cyber-Physical Systems
University of California, Irvine

Improved Canny Edge Detector for Parallel Color Video Processing

Xiangdong Che, Rainer Dömer

Technical Report CECS-24-01
April 29, 2024

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

xiangdc2@uci.edu
http://www.cecs.uci.edu

Improved Canny Edge Detector for Parallel Color Video Processing

Xiangdong Che, Rainer Dömer

Technical Report CECS-24-01
April 29, 2024

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

xiangdc2@uci.edu
http://www.cecs.uci.edu

Abstract

Canny edge detection [1] is a well-known algorithm for filtering out edges in grey-scale images. In this report, we describe our improvement of the original implementation by Mike Heath [3] for processing color video streams efficiently. We extend the single image filter application to process a stream of video frames with three color channels. Further, we parallelize the process by slicing the image frames vertically and horizontally for separate processing. The increased parallelism can significantly increase the processing speed and reduce the memory usage for small embedded processors. Our experiments show that image quality closely matches the original frames, resulting in a smooth-playing video.

Contents

1 Introduction **1**
1.1 Original Edge Detection 1
1.2 Sliced Edge Detection 1

2 Improvements **2**
2.1 Color Images Input 2
2.2 Slicing Approach 2
2.3 Quality Issues and Solution 2
2.3.1 Missing Pixels at Boundaries Between Slices 2
2.3.2 Inconsistency in Edge Images 4

3 Experiments and Results **4**
3.1 Color Image Input 4
3.2 Slice Approach 6
3.2.1 Impact of Fixed Threshold on Edge Images 6
3.2.2 Impact of The Number of Slices on Edge Images 6

4 Conclusion **7**

References **7**

List of Figures

1 SystemC model of canny_seq.c . . . 1
2 Example of 2x2 slicing . . . 1
3 New arrays that store RGB information . . . 2
4 New arrays which store RGB information . . . 2
5 Slicing approach . . . 3
6 Edge image with slicing boarders . . . 3
7 Edge image after fix . . . 3
8 Lines that were removed . . . 4
9 Loops that were modified . . . 4
10 New boundary check . . . 4
11 Edge image with inconsistent thresholds . . . 5
12 Edge image after fix . . . 5
13 Lines that were removed . . . 5
14 Chart of differences on 100 color frames . . . 5
15 Chart of differences on 100 frames with consistent thresholds . . . 6
16 Chart of difference among 4 ways of slicing on 100 frames . . . 6

List of Tables

1 Comparison between using color image input and grey-scale image input 5
2 Comparison between original edge images and 1x1 slice approach 6

Improved Canny Edge Detector for Parallel Color Video Processing

X. Che, R. Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
xiangdc2@uci.edu
http://www.cecs.uci.edu

Abstract

Canny edge detection [1] is a well-known algorithm for filtering out edges in grey-scale images. In this report, we describe our improvement of the original implementation by Mike Heath [3] for processing color video streams efficiently. We extend the single image filter application to process a stream of video frames with three color channels. Further, we parallelize the process by slicing the image frames vertically and horizontally for separate processing. The increased parallelism can significantly increase the processing speed and reduce the memory usage for small embedded processors. Our experiments show that image quality closely matches the original frames, resulting in a smooth-playing video.

1 Introduction

The Canny Edge Detector [1] is a widely used algorithm for edge detection in image processing. It was developed by John F. Canny in 1986 but has since undergone several adaptations and optimizations, one of which we used was implemented by Mike Heath [3].

1.1 Original Edge Detection

The original version we use in EECS222 [2] and ECPS203 [4] class that was adapted from the source code from Mike Heath [3]. This version of the detector has a single-threaded implementation where data is passed via local variables on the stack. There are 8 sequential processing stages and 6 of them (blurX, blurY, derivative_x_y, magnitude_x_y, non_max_supp, apply_hysteresis) are in the DUT module, as shown in Figure 1.

Figure 1: SystemC model of canny_seq.c

1.2 Sliced Edge Detection

Figure 2: Example of 2x2 slicing

We have made two improvements to canny_seq.c. The first change is that the algorithm can take color images in ppm format instead of greyscale images

in pgm format and then process information in three color channels of RGB to improve the quality of edge images. The second change is the image processing stage which not only allows the program to run in parallel but also reduces the memory usage during processing images to 1 / (M*N) where M and N are parameters that can be adjusted as needed. For example, input images are sliced into 4 pieces as shown in Figure 2.

2 Improvements

We describe our changes in detail in the following sections.

2.1 Color Images Input

To maintain RGB information across all three color channels, we use six additional arrays, with three designated for storing input images and the remaining three for corresponding edge images. To make the program more organized, we implemented two arrays for ef cient storage of these images, as illustrated in Figure 3.

```
int main(void)
{
    unsigned char imageRed[SIZE];
    unsigned char imageGreen[SIZE];
    unsigned char imageBlue[SIZE];
    unsigned char edgeRed[SIZE];
    unsigned char edgeGreen[SIZE];
    unsigned char edgeBlue[SIZE];
    unsigned char* imageRGB[3];
    unsigned char* edgeRGB[3];

    ...

    imageRGB[0] = imageRed;
    imageRGB[1] = imageGreen;
    imageRGB[2] = imageBlue;
    edgeRGB[0] = edgeRed;
    edgeRGB[1] = edgeGreen;
    edgeRGB[2] = edgeBlue;
```

Figure 3: New arrays that store RGB information

After collecting edge images, the output image is manually generated through the data_out() function. This process involves merging images from the three distinct RGB channels, where pixels are marked as "EDGE" if detected in at least one of the three channels, as illustrated in Figure 4.

```
void data_out(unsigned char *edge,
              unsigned char **edgeRGB,
              unsigned int i) {
  ...
  int index = 0;
  for (index = 0; index < SIZE; ++index) {
      edge[index] = NOEDGE;
    if (edgeRGB[0][index] == EDGE ||
        edgeRGB[1][index] == EDGE ||
        edgeRGB[2][index] == EDGE) {
        edge[index] = EDGE;
    }
  }
  ...
  write_pgm_image(outfilename, edge, ROWS,
                  COLS, "", 255) == 0)
}
```

Figure 4: New arrays which store RGB information

2.2 Slicing Approach

To achieve the goal of less memory usage, we use another approach when performing the detection algorithm on images. Instead of calling canny() on the entire image, we divide input images into multiple rectangular slices by M rows and N columns. We then apply canny() to each of the images individually to produce edge images of each slice. Finally, the resulting edge images from each slice are combined together back to one at the end, as illustrated in Figure 5.

2.3 Quality Issues and Solution

Naive slicing results in reduced image quality that we address as follows.

2.3.1 Missing Pixels at Boundaries Between Slices

In the original algorithm, the edge images' boundaries were set to empty by default, and all loops skipped pixels on these boundaries for ef ciency, as tracking an edge off the image's side was unnecessary. However, this practice is problematic with our slicing approach because the pixels on the boundaries of each

```
void canny_slice(unsigned char* image, unsigned char* edge) {

    int i = 0, j = 0;
    unsigned char slicedImage[SLICESIZE];
    unsigned char slicedEdge[SLICESIZE];

    for (i = 0; i < SLICENUM_ROW; ++i) {

        for (j = 0; j < SLICENUM_COL; ++j) {

            int desOffset = 0;
            int srcOffset = (i * ROWPERSLICE * COLS) + j * COLPERSLICE;
            int index = 0;

            /* copy sliced image from src image to buffer*/
            for (index = 0; index < ROWPERSLICE; ++index) {
                memcpy(slicedImage + desOffset, image + srcOffset, COLPERSLICE);
                desOffset += COLPERSLICE;
                srcOffset += COLS;
            }

            canny(slicedImage, ROWPERSLICE, COLPERSLICE, SIGMA, TLOW, THIGH, slicedEdge);

            desOffset = (i * ROWPERSLICE * COLS) + j * COLPERSLICE;
            srcOffset = 0;
            index = 0;

            /* copy sliced edge from buffer to edge*/
            for (index = 0; index < ROWPERSLICE; ++index) {
                memcpy(edge + desOffset, slicedEdge + srcOffset, COLPERSLICE);
                desOffset += COLS;
                srcOffset += COLPERSLICE;
            }
        }
    }
}
```

Figure 5: Slicing approach

Figure 6: Edge image with slicing boarders

Figure 7: Edge image after x

slice are no longer boundaries once reassembled. Obvious white lines can be found vertically and horizontally at the center of the edge image before the x, as shown in Figure 6, compared with the image after the x, as shown in Figure 7.

To x this issue, we removed the lines of code that set boundary pixels to zero in apply_hysteresis() as shown in Figure 8.

```
void apply_hysteresis() {
    ...

    /*
    for(r=0,pos=0;r<rows;r++,pos+=cols){
        edge[pos] = NOEDGE;
        edge[pos+cols-1] = NOEDGE;
    }
    pos = (rows-1) * cols;
    for(c=0;c<cols;c++,pos++){
        edge[c] = NOEDGE;
        edge[pos] = NOEDGE;
    }
    */

    ...
}
```

Figure 8: Lines that were removed

We also changed the starting index of the loop and initialization of pointers in non_max_supp() to ensure it covers every pixel including boundaries, as shown in Figure 9.

```
void non_max_supp() {
    ...

    for(rowcount=0,magrowptr=mag,gxrowptr=gradx,
      gyrowptr=grady,resultrowptr=result;
      rowcount<nrows;
      rowcount++,magrowptr+=ncols,
      gyrowptr+=ncols, gxrowptr+=ncols,
      resultrowptr+=ncols){
      for(colcount=0,magptr=magrowptr,
          gxptr=gxrowptr,gyptr=gyrowptr,
          resultptr=resultrowptr;
          colcount<ncols;
          colcount++,magptr++,gxptr++,gyptr++,
          resultptr++) {

    ...
}
```

Figure 9: Loops that were modi ed

To compensate for not having zero on the boundaries, we add a condition in follow_edges() to check if the pointer is going off boundaries, as shown in Figure 10.

```
void follow_edges(unsigned char *edgemapptr,
                  short *edgemagptr, short lowval,
                  int cols, int r, int c) {
    /* prevent overwrite */
    if( r < 0 || c < 0 ||
        r >= ROWPERSLICE ||
        c >= COLPERSLICE) return;
    ...
```

Figure 10: New boundary check

2.3.2 Inconsistency in Edge Images

In the original algorithm, edges are determined by two thresholds, i.e. LOWTHRESHOLD and HIGHTHRESHOLD. These two thresholds are computed dynamically for every image based on the intensity of pixels in input images. Therefore, after slicing the input images, each small slice will have different thresholds which generate inconsistent edge images as illustrated in Figure 11, compared with the image after the x, as shown in Figure 12.

To x this issue, we removed the lines of code that calculate thresholds and hard-coded LOWTHRESHOLD and HIGHTHRESHOLD values, as shown in Figure 13.

3 Experiments and Results

To evaluate the differences and improvements, we use a tool called ImageDiff to compare images which calculates the number of different pixels between two images and then calculates the percentage relative to its resolution.

3.1 Color Image Input

We compare the edge images by using color images and grey-scale images as input. We observe that the edge images generated using color images as input have 1.5% more pixels on average, as shown in Table 1 and plotted in Figure 14.

Figure 11: Edge image with inconsistent thresholds

Figure 12: Edge image after x

```
void apply_hysteresis() {
    ...

    /*
    for(r=0;r<32768;r++) hist[r] = 0;
    for(r=0,pos=0;r<rows;r++){
        for(c=0;c<cols;c++,pos++){
            if(edge[pos] == POSSIBLE_EDGE) {
                hist[mag[pos]]++;
            }
        }
    }

    for(r=1,numedges=0;r<32768;r++){
        if(hist[r] != 0) maximum_mag = r;
        numedges += hist[r];
    }

    highcount = (int)(numedges * thigh + 0.5);
    r = 1;
    numedges = hist[1];
    while((r<(maximum_mag-1)) &&
          (numedges < highcount)){
        r++;
        numedges += hist[r];
    }
    highthreshold = r;
    lowthreshold = (int)(highthreshold * tlow + 0.5);
    */

    highthreshold = HIGHTHRESHOLD;
    lowthreshold = LOWTHRESHOLD;
}
```

Figure 13: Lines that were removed

	Mismatched pixels	Percentage
Frame 1	71373	1.74%
Frame 2	71150	1.73%
Frame 3	67465	1.64%
Frame 4	67563	1.64%
Frame 5	64927	1.58%
...	...	...
Frame 96	46923	1.14%
Frame 97	49148	1.20%
Frame 98	42253	1.03%
Frame 99	41678	1.01%
Frame 100	38725	0.94%
Average	61910	1.51%

Table 1: Comparison between using color image input and grey-scale image input

Figure 14: Chart of differences on 100 color frames

3.2 Slice Approach

Our image slicing also affects the image quality.

3.2.1 Impact of Fixed Threshold on Edge Images

In our evaluation, we initially compare the original edge images with those generated using the slice approach with a single slice (1x1) as shown in Table 2.

	Mismatched pixels	Percentage
Frame 1	25941	0.631%
Frame 2	25941	0.672%
Frame 3	25941	0.628%
Frame 4	25941	0.720%
Frame 5	25941	0.79%
...	...	...
Frame 96	99825	2.429%
Frame 97	91515	2.227%
Frame 98	106379	2.588%
Frame 99	117203	2.852%
Frame 100	124174	3.021%
Average	50317	1.224%

Table 2: Comparison between original edge images and 1x1 slice approach

We observe that the differences between the two methods increase across different frames, as illustrated in Figure 15.

Figure 15: Chart of differences on 100 frames with consistent thresholds

The reason is that some frames may have complicated backgrounds while others are relatively simple. The program employs xed thresholds, which are hard-coded at the outset, for edge detection across all images. Therefore, as images change, the xed thresholds may result in larger differences in edge detection results.

3.2.2 Impact of The Number of Slices on Edge Images

Here, we treat 1x1 slicing as the reference and compare four different ways of slicing. We observe that as the slice number increases, the differences increase, as plotted in Figure 16.

Figure 16: Chart of difference among 4 ways of slicing on 100 frames

By slicing the input images into smaller pieces, many factors affect the accuracy. For example, there is a risk of missing some global information that can help in accurately detecting edges and the edge may become discontinued resulting in missing edges.

4 Conclusion

In conclusion, this report has introduced an improved version of the Canny edge detection algorithm capable of processing color images, thereby enhancing the accuracy of edge detection. Moreover, the optimized program demonstrates parallelism with lower memory usage without significant compromise in accuracy. However, it is imperative to acknowledge two critical factors influencing the quality of edge images: the fine-tuning of thresholds according to varied input image scenarios and the impact of image segmentation on edge quality. Increasing the number of image slices adversely affects edge image quality.

References

[1] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8, 1986.

[2] EECS 222. https://canvas.eee.uci.edu/courses/sis_course_id:CourseSpace-Section-W24-17360/assignments/syllabus.

[3] Heath. M, Sarkar. S, Sanocki.T, and Bowyer.K. Comparison of edge detectors: a methodology and initial study. In *Computer Vision and Pattern Recognition*, San Francisco, U.S.A, June 1996.

[4] MECPS 203. https://canvas.eee.uci.edu/courses/sis_course_id:CourseSpace-Section-F23-15205/assignments/syllabus.