END OF INSERTION DETECTION IN COLONOSCOPY VIDEOS

Avnish Rajbal Malik

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2009

APPROVED:

Jung Huan Oh, Major Professor
Bill Buckles, Committee Member and Associate Dean of the College of Engineering
Yuan Xiaohui, Committee Member
Krishna M. Kavi, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Michael Monticino, Dean of the Robert B. Toulouse School of Graduate Studies

Malik, Avnish Rajbal. End of Insertion Detection in Colonoscopy Videos. Master of Science (Computer Science), August 2009, 59 pp., 10 tables, 20 figures, references, 27 titles.

Colorectal cancer is the second leading cause of cancer-related deaths behind lung cancer in the United States. Colonoscopy is the preferred screening method for detection of diseases like Colorectal Cancer. In the year 2006, American Society for Gastrointestinal Endoscopy (ASGE) and American College of Gastroenterology (ACG) issued guidelines for quality colonoscopy. The guidelines suggest that on average the withdrawal phase during a screening colonoscopy should last a minimum of 6 minutes.

My aim is to classify the colonoscopy video into insertion and withdrawal phase. The problem is that currently existing shot detection techniques cannot be applied because colonoscopy is a single camera shot from start to end. An algorithm to detect phase boundary has already been developed by the MIGLAB team. Existing method has acceptable levels of accuracy but the main issue is dependency on MPEG (Moving Pictures Expert Group) 1/2. I implemented exhaustive search for motion estimation to reduce the execution time and improve the accuracy. I took advantages of the C/C++ programming languages with multithreading which helped us get even better performances in terms of execution time. I propose a method for improving the current method of colonoscopy video analysis and also an extension for the same to make it usable for real time videos. The real time version we implemented is capable of handling streams coming directly from the camera in the form of uncompressed bitmap frames. Existing implementation could not be applied to real time scenario because of its

dependency on MPEG 1/2. Future direction of this research includes improved motion search and GPU parallel computing techniques.

ACKNOWLEDGEMENTS

I would sincerely like to thank Dr. JungHwan Oh for letting me have this opportunity to learn about various image and video processing methodologies in the field of medical videos. His timely advice and suggestions with his experience were of a great importance to help me complete this project.

Also, I would like to thank other members of my lab, Praveen Karri, M.J. Kumara and Ruwan Navarathna for their advice in logic and product development. I would especially like to thank M.J. Kumara for his wonderful work in development of the affine model, one of the key modules of this product.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF ILLUSTRATIONS

Page

CHAPTER 1

INTRODUCTION

Colorectal cancer is the second leading cause of cancer-related deaths behind lung cancer in the United States [1]. As the name implies, colorectal cancers are malignant tumors that develop in the colon and rectum. The survival rate is higher if the cancer is found and treated early before metastasis to lymph nodes or other organs occur [5]. A brief overview of the colonoscopy can be understood as follows. Colon consists of 6 parts.

1 - cecum, 2 - ascending colon, 3 - transverse colon, 4- descending colon, 5 - sigmoid, 6 - rectum.

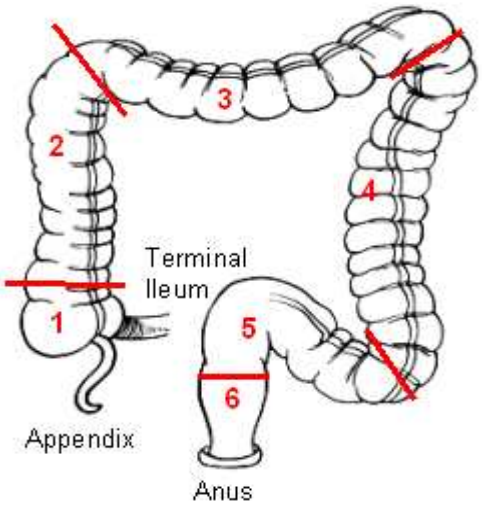

Figure 1.1Various parts of the colon.

Colonoscopy is done with the help of a flexible colonoscope which is a cable with a camera attached at its tip as seen in Figure 1.2.

Figure 1.2 Colonoscope.

The cable is inserted through the anus into a patient's body. A typical colonoscopy procedure consists of two phases namely insertion phase and withdrawal phase. Doctors analyze the condition of the patient in the withdrawal phase usually. According to the set standards the minimum withdrawal time should be at least 6 minutes for a colonoscopy procedure to be classified as a good one. The typical automated procedure for insertion and withdrawal time estimation is based on the idea described in Figure 1.1 Here a video taken by an endoscope which can be in either MPEG (Moving Picture Experts Group) 1/2 [7][8] format is used for preprocessing. Here unwanted parts from the start and the end of the video which are irrelevant to the procedure are removed.

Next part is to find the motion vectors in this video for every block in a frame. There might be irrelevant motion vectors which might play a misleading role when calculating forward and backward direction of the camera, such vector information is removed. After this is done the affine model equation is applied to the data available by motion vector estimation to estimate the dolling (forward or backward) camera motion. The metrics include insertion time, withdrawal time, speed of camera etc. [12][13]. These metrics are important to analyze how well the doctor studied the patient.

In this thesis I propose a method for the motion vector estimation which saves the execution time of the entire procedure and also makes the system independent of any encoding of video. This is because the system is capable of handling the uncompressed data in the form of bitmap images. The remainder of this thesis discusses the methodology used and the experimental results.

CHAPTER 2

EXISTING IMPLEMENTATION

For the purpose of finding the end of insertion in colonoscopy videos, the research team had previously developed a working version. Even though the implementation consists of various other complex parts, I concentrated on the end of insertion module. The previous implementation for end of insertion detection can be understood from the figure below.

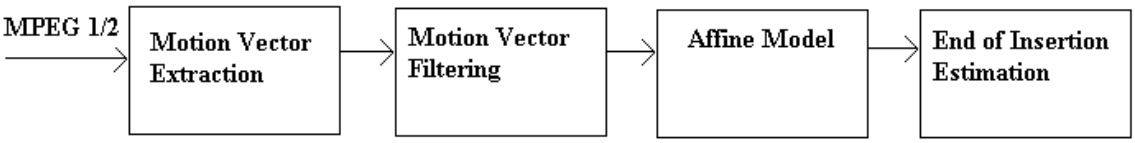

Figure 2.1 Current implementation.

The current implementation relies completely on the motion vector information already present in the MPEG (Moving Picture Experts Group) stream; a library written in JAVA provides the functionality of decoding these motion vectors. A filtering step removes any unreliable motion vectors; this filtered data of motion vectors is then passed on to the affine model for the dolling camera motion values. These values are then plotted cumulatively against the frame numbers and the maximum point of this value is selected as the end of insertion. My improvement is in the module of motion vector extraction, where the aim is to develop a new search technique to reliably calculate the motion vectors and remove the dependency on MPEG 1/2.

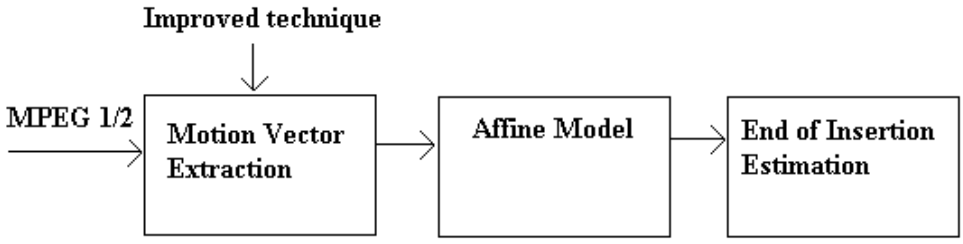

Figure 2.2 Improved modules.

As seen in Figure 2.2 I consider the module of motion vector extraction for improvement. This was chosen for two reasons, the first being the scope for improvement in this and the second being the time taken by this part. As compared to the other modules, the motion vector extraction part takes the largest amount of time. I intended to improve this method in terms of execution time while keeping the accuracy same. As per our experiments which are covered in later sections, I found that the accuracy can still be improved further in time less than even for reading the motion vectors from the file. Also as mentioned before, this eliminated the dependency on MPEG standard, so that this module could even be applied to the real time scenario. Also as will be explained in later sections, I also applied methods like multithreading to increase the accuracy even further. However the multithreading aspect depends purely upon the number of CPU cores available for use, but considering the fact that most of the processors today are at least dual core, I expect at least 2 threads to run on a given system.

CHAPTER 3

MOTION VECTOR ESTIMATION

Typically the type of algorithm chosen for MPEG (Moving Picture Experts Group) encoding is manufacturer dependent [7-8], but always there is a tradeoff between the quality and the file size of the video. The main aim behind this is to compress the video while maintaining acceptable quality. My method takes frames at a rate of 2 Frames per second rate, actually two frames are chosen at a time, in which one is the reference frame and the second is the current frame, thus I can say four. This ratio or rate was chosen as a result of the experiments I conducted for finding the best tradeoff ratio. These experiments can be studied in the experimental results section. I divide the entire image into a number of blocks of 16X16 pixels. The reference image is divided according to the search area I wish to use; currently it is set to 32X32 pixels. A search area of 48X48 pixels was chosen initially for the sake of improved accuracy by it took almost 50% more time as compared to the 32X32 version. However the improvement in accuracy was not more than 20%.

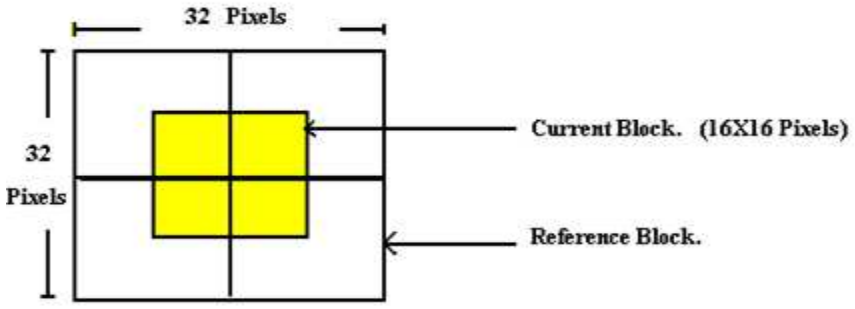

Figure 3.1 Current and reference blocks.

I store the current and reference image in the form of two dimensional arrays. I search for the current block by sliding it in a window of 32X32 pixels. Every pixel consists of 3 values, namely Red, Green and Blue, here I initially took an average of these value and used it for comparison. But later on I found that using all the 3 values lead to better accuracy with no significant change in execution time. I start the comparison from the upper left corner; the result of this comparison is the sum of differences of all 256 pixels. This number is stored as an integer and I move the current block one pixel to the right, I again follow the same procedure. This process continues for all rows and columns with a displacement of 1 pixel each time because of the exhaustive nature of the search. So now for every block of 16X16 pixels there will be 256 X 3 comparisons. So totally there can be 16X16 locations where this current block can be searched, because as can be seen in Figure 3.1 the block can be shifted 8 pixels in any direction. For all such comparisons the minimum value is chosen as the candidate for best match. Here unlike MPEG which selects the best candidate only if it is below a certain threshold value [7], I am bound to find the best match. This is because I do not intend to compress this frame but I want to get a direction of the movement of the current block. Even though the exact extent of movement may be outside the search area, I can still get a clue of its direction. To reduce this kind of direction guess I choose the reference and current frame to be only 1 frame distant. The window size of 32X32 pixels is chosen after much experimentation for determining best tradeoff policy between accuracy and speed of execution.

3.1 Preprocessing Step

Before I can actually start to conduct motion vector search, there is an important module for video preprocessing, here I actually make use of libraries available from the MPEG standards [8]. This library generates an exe called getframes.exe which can be adjusted according to our requirements. Here the code sample below shows exactly how this library is being used.

```
1. strcpy(tmp3,".\\getframes.exe ");
2. strcat(tmp3,tr);
3. strcat(tmp3," -start 1 -frameskip 30 -format bmp -out ");
4. strcat(tmp3,tmp2);
5. system(tmp3);
```

Here as can be seen in the first line the location of this library exe is stored followed by appending of the destination folder where the images will be stored. After this comes the important part of skipping of frames (Line 3), which is currently 30, followed by the output format of .bmp images, as I want to get uncompressed images for easy and fast processing. On line 5 the system call is issued after setting all the requirements. The output of this is a bitmap image with naming done as per frame ID; the output images generated are 24 bit in color depth with a resolution of 720X480 pixels. This process is repeated twice so as to get the reference frame as well. Here the only difference will be in line number 3, where instead of 1 I have a 0. So now the output will be images 0, 1, 30, 31, 60, 61……., Thus I have a pair of images for processing which is 1 frame distant. Followed by this is the code for sorting these images based on their

frame ID's, so that the data can be prepared in the form of pointer queue for processing. The next part is a bit tricky, the images obtained even though uncompressed, but contain hexadecimal values, I need to convert them to integers so that they can be handled using integer variables in C. Every image that is read is sent for conversion and after conversion of their R, G & B values to integers, they are stored in a 720X480 array. One more important thing is that the older versions of C for example the Borland C/C++ does not allow this big size for an array, hence I made use of the Visual Studio platform. Thus finally I have 2 arrays of 720X480 pixels, one for the current block and another for the reference block.

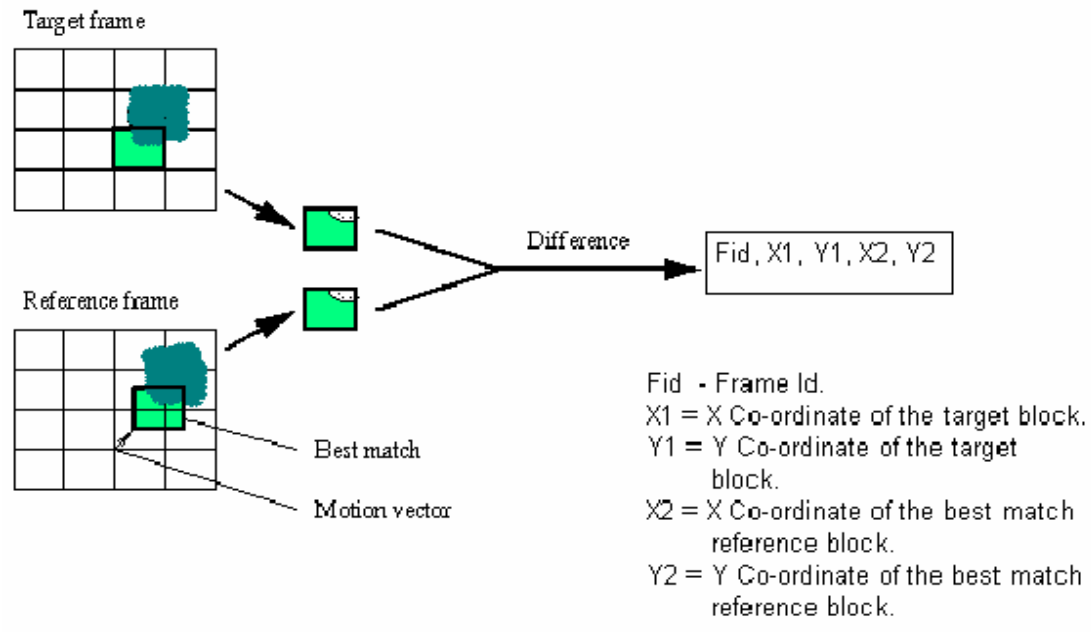

Figure 3.1.1 Search method and output.

3.2 Motion Search.

As can be seen in Figure 3.1.1 the two frames, current and reference frame are divided into a number of blocks of 16X16 pixels each [6][9]. Every block is then searched in its corresponding search area in the reference frame; the output of this is stored in a text file in the format depicted in Figure 3.1.1. As noted earlier the search area is currently set to 32X32 pixels which provide best results for our purpose. The code in figure 3.2.1 shown below is responsible for this part in the actual implementation.

Figure 3.2.1 Motion search.

```
1. for(starty = mb;starty < height - mb;starty = starty + mb + mb)
      2. for(startx = mb;startx < width - mb;startx = startx + mb
+ mb)
          {
                flg = 0;
                rctr=0;

                //loop to consider black blocks
3. while(curr_r[starty][startx]_r < 10 && curr[starty][startx]_g <
10 && curr[starty][startx]_g < 10 && curr_r[starty+mb-
1][startx+mb-1] < 10 && curr_g[starty+mb-1][startx+mb-1] < 10 &&
curr_b[starty+mb-1][startx+mb-1] < 10)
                {
                      startx = startx + mb + mb;
                      if(startx >= width - mb)
```

```
                        {


                                flg = 1;

                                break;

                        }

                }

//if the block is black skip this block and go to the next block


      if(flg == 1)

            break;


      4.for(y = starty - sa;y < starty + sa;y++)

            5.for(x = startx - sa;x<startx + sa;x++)

                  {


                        ans = 0;

                        R1 = 0;

                        G1 = 0;

                        B1 = 0;

                        R2 = 0;

                        G2 = 0;

                        B2 = 0;


                6. for(u=y,offy = 0;u < y + mb;u++,offy++)

                        7. for(v = x,offx = 0;v < x +
```

```
 mb;v++,offx++)
                          {
                      R1 = R1+curr_r[starty+offy][startx+offx];
                      G1 = G1+curr_g[starty+offy][startx+offx];
                      B1 = B1+curr_b[starty+offy][startx+offx];
                      R2 = R2+reff_r[u][v];
                      G2 = G2+reff_g[u][v];
                      B2 = B2+reff_b[u][v];

//finding Euclidean distance to compare two pixel values in current
and reference frame

ans = ans + sqrt((R2-R1)*(R2-R1)+(G2-G1)*(G2-G1)+(B2-B1)*(B2-B1));
                          }
                                res[rctr].ans = ans;
                                res[rctr].x = x;
                                res[rctr].y = y;
                                rctr++;
                    }

                          writeresult(startx,starty,file);
```

Here as can be seen, the first loop (lines 1, 2) takes care of the current frame which is stored in a two dimensional array called “curr_r, curr_g & curr_b”, it decides the number of skips to make. Here I do not consider all the blocks for motion estimation, the reason for which will be explained in the later sections. The next while loop (line 3) takes

care of the blocks which don't need to be considered as they are black, the significance of this will also be explained in the later sections. The next important part is the loop at lines 4 & 5 this actually decides the search area in the reference frame. This is decided by a variable called "sa" which is set to 32 while the macro block size is decided by "mb" variable which is set to 16. After this the next loops at 6 & 7 are responsible for the actual comparison. After loops 6 & 7 are done loops 4 & 5 shift the search area by 1 pixel to right and later down. Later loops 1 & 2 repeat the entire procedure for the next block in the current frame. The results for all the matches are stored in an array called "res" which is later passed to another function for deciding the best candidate. Here the lowest value is chosen and the results are written to a text file in proper format as depicted in Figure 3.1.1.

3.3 Selection of Best Candidate

Best candidate is selected by selecting the minimum value; there is an issue however which needs to be resolved, that is what if there is more than one best candidate which has the same value. Here in this case a match that is closer to the original current block is selected as the best candidate. This is also helpful when the area of search is entirely of the same color as that of the block. Here choosing a displacement of zero is advisable so that it does not mislead the later part of the project in determining the camera motion. This is done by the following code snippet in the project. In the actual implementation I found that for most of the cases the reference and current block have same color properties entirely. So to save time I directly compare the result obtained from

first loop which selects the minimum difference with the centre of the search area where it all began, if that is true the center is decided as final position of match. This saves some iteration and also is based on actual colonoscopy video observations [13].

```
            for(i=0;i<rctr;i++)
{
      if(res[i].ans < f)
      {
            f = res[i].ans;          x2 = res[i].x;
            y2 = res[i].y;
      }
}

if(f == res[rctr/2].ans)
{
      f = res[rctr/2].ans;
      x2 = x;
      y2 = y;
}
```

CHAPTER 4

SKIPPING OF BLACK BLOCKS

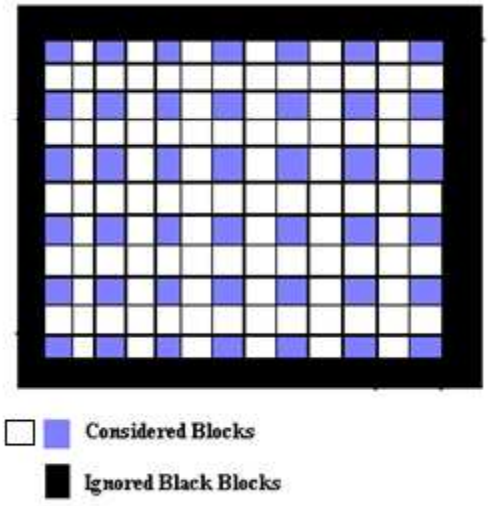

Figure 4.1 Skipped border or black blocks.

I also noted that in some videos made by different manufacturers there was a black area around the video, the video is played in a window in this black area. This black area is at different locations in videos from different endoscope manufacturers. The problem was to avoid searching this black area because if a current black block is searched in the reference area which is also entirely black, then every match becomes a good match. The question is to select one from them, if I choose the displacement of 0, 0 i.e. the same location as that of the current block, the results were accurate but it took more time. This can be understood from Figure 4.1. I placed a threshold value to decide to begin searching, if the value of R, G and B was less than 10 independently; I ignore this block thus saving time. I place a neutral motion vector for this particular block, so

that it does not contribute to the DCM value calculation. The earlier implementation relied on the motion vector information of P-Frames in the MPEG (Moving Picture Experts Group) video which failed to identify this feature. I deal with only dolling (Backward or Forward) camera motion. All the values like the size of block, size of search area, skipping of blocks, threshold value for considering a block as black etc. are kept variable for easy fine tuning of the project. Also there was an additional step performed in the preprocessing module in the older version of this system which included conversion of MPEG2 to MPEG1 [12] [13]. This was because the code was dependent on motion vectors from MPEG1 videos. This step was removed in the new version to save time further. This also improved the current version. More details about how actually the two candidate endoscopes generate the video can be seen in Figure 4.2.

(a) Fujinon Camera (b) Olympus Camera

Figure 4.2 Different endoscopes.

```
while(curr_r[starty][startx]_r < 10 && curr[starty][startx]_g < 10 &&
curr[starty][startx]_g < 10 && curr_r[starty+mb-1][startx+mb-1] < 10
&& curr_g[starty+mb-1][startx+mb-1] < 10 && curr_b[starty+mb-
1][startx+mb-1] < 10)
                {
                        startx = startx + mb + mb;

                        if(startx >= width - mb)
                        {
                                flg = 1;
                                break;
                        }
                }
```

The part of the code which takes care of this is shown above. Here I do not search the entire block for common pixel value, instead I choose two opposite corners of a block. If they both are same and the value is less than the threshold of 10 I break the process and start with the next block. After this is done for the entire frame next frame in the processing queue is served. When all the frames are analyzed for motion estimation, I get a text file MV.TXT which stores all the information of motion data. This file is next given to the affine model which in contrast to the previous version which had a MATLAB exe is implemented in C\C++.

CHAPTER 5

THE AFFINE MODEL

As noted in section 4, for the real time end of insertion affine model was made in C, the basics of the affine model [13] [16] can be understood from the Figure 7.1.1. I just deal with the dolling camera motion for our purpose because only this can detect the extent to which camera goes. Also for the end of insertion detection other camera motions are irrelevant. The actual implementation of the affine model is based on the equation as follows. The forward or backward camera motion is on the Z axis.

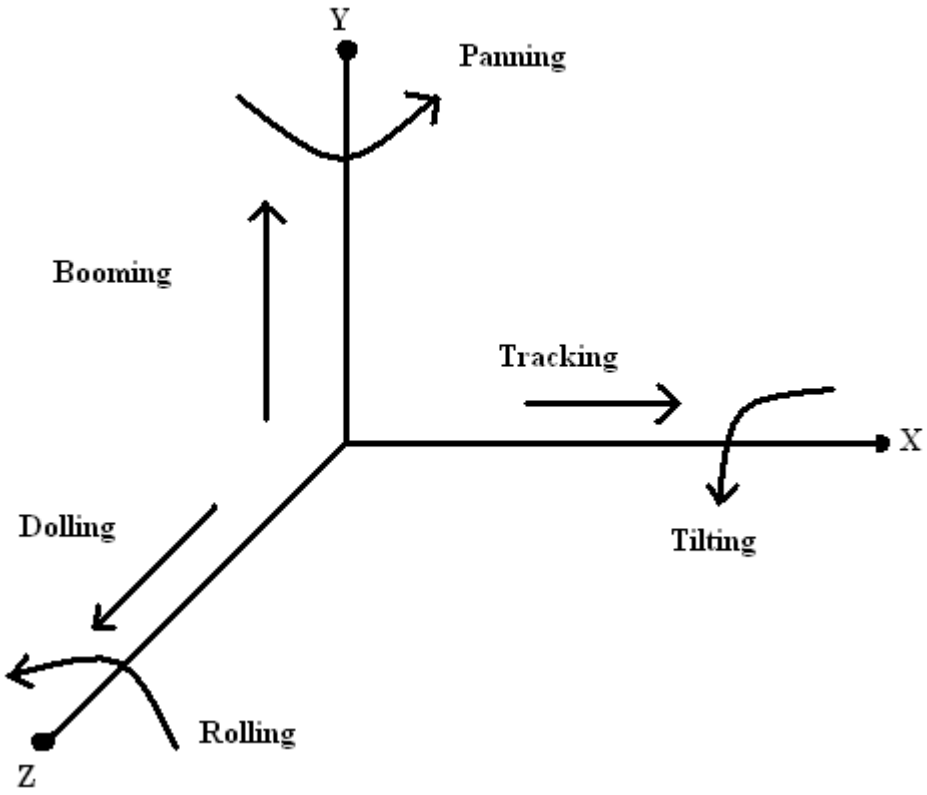

Figure 5.1.1 Camera Motions.

The standard equation for the camera motions can be defined as in (1).

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} a_1^{zoom} & b_1^{roll} \\ -b_2^{roll} & a_2^{zoom} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c^{pan} \\ d^{tilt} \end{pmatrix} + \frac{1}{z} \left[\begin{pmatrix} a_1^{dolly} & 0 \\ 0 & a_2^{dolly} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c^{track} \\ d^{boom} \end{pmatrix} \right] \quad (1)$$

(u, v) is the motion vector of a macro block located at position (x, y) of each frame, z is the depth of the real world. Scalar coefficients concerned with camera motions are as follows,

$$a_1^{zoom} \quad a_2^{zoom} \quad b_1^{roll} \quad b_2^{roll} \quad c^{pan} \quad d^{tilt} \quad a_1^{dolly} \quad a_2^{dolly} \quad c^{track} \quad d^{boom}$$

Endoscope cameras do not have zoom-in and zoom-out functions, Thus our equation is reduced to,

$$a_1^{zoom} = 0,\ a_2^{zoom} = 0$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{a_1^{dolly}}{z} & b_1^{roll} \\ -b_2^{roll} & \frac{a_2^{dolly}}{z} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c^{pan} + \frac{c^{track}}{z} \\ d^{tilt} + \frac{d^{boom}}{z} \end{pmatrix} \quad (2)$$

Let,

$$a_6 = \frac{a_2^{dolly}}{z}$$

$$a_1 = c^{pan} + \frac{c^{track}}{z}, \qquad a_2 = \frac{a_1^{dolly}}{z}, \qquad a_3 = b_1^{roll}, \qquad a_4 = d^{tilt} + \frac{d^{boom}}{z}, \qquad a_5 = -b_2^{roll},$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} a_2 & a_3 \\ a_5 & a_6 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a_1 \\ a_4 \end{pmatrix} \quad (3)$$

Dolling Camera Motion (*DCM*),

Rolling Camera Motion (*RCM*),

Horizontal Camera Motion (*HCM*=Panning + Tracking)

Vertical Camera Motion (*VCM*=Tilting + Booming)

$$DCM = \frac{1}{2}(a_2 + a_6), \qquad HCM = a_1,$$

$$RCM = \frac{1}{2}(a_3 - a_5), \qquad VCM = a_4$$

$$a_1 = c^{pan} + \frac{c^{track}}{z}, \quad a_2 = \frac{a_1^{dolly}}{z}, \quad a_3 = b_1^{roll}, \quad a_4 = d^{tilt} + \frac{d^{boom}}{z}, \quad a_5 = -b_2^{roll}, \quad a_6 = \frac{a_2^{dolly}}{z}$$

As I can see that this kind of implementation is computationally expensive, so in order to speed up, the earlier implementation of MATLAB was converted to C in this implementation. The difference in the Real time version and the Post processing version

is that the Affine model was called only once in the post processing version. After all the files were tested for motion vectors and all the values were stored in a file, an .exe created for Affine in MATLAB was called. Calling an .exe of MATLAB in C is computationally very expensive. Thus in the Real Time version where the Affine model had to be called after every frame of motion detection, this was not feasible. The code was generated in C and then embedded with the motion detection code in the form of a function. The sample implementation can be seen in the code below,

Figure 5.1.2 Affine Model.

```
double affine(double u,double v,double x,double y){
        double temp[3];
        double a[]={0,0,0};
        double b[]={0,0,0};
        double pan,tilt,div,rot;  // motion parameters

        temp[0]=1;
        temp[1]=x;
        temp[2]=y;

         getLeastSquare(temp,u,a);  // obtain leastSquare for a
        getLeastSquare(temp,v,b);  // obtain leastSquare for b
```

```
        // calculating motion parameters


                pan=a[0];

                tilt=b[0];

               div=(a[1]+b[2])/2;

               rot=(b[1]-a[2])/2;


        return (div-1); // returning the calculated DCM


}


void getLeastSquare(double numList[3],double scalar, double *result){


               double largestIndex=0;

               double leastSquare[3];

               double b[3];

               double list[3];

               list[0]=numList[0];

               list[1]=numList[1];

               list[2]=numList[2];


      if(list[0]<list[1])
```

```
                largestIndex=1;

        if(list[1]<list[2])
                largestIndex=2;

                list[0]=1/list[0];

                list[1]=1/list[1];

                list[2]=1/list[2];

                b[largestIndex]=scalar;

          leastSquare[0]=list[0]*b[0];
          leastSquare[1]=list[1]*b[1];
          leastSquare[2]=list[2]*b[2];
    result[0]=leastSquare[0];
    result[1]=leastSquare[1];
    result[2]=leastSquare[2];
}
```

CHAPTER 6

DCM VALUE GRAPH AND END OF INSERTION

When the affine model is applied I get the DCM (Dolling Camera Motion) values in the form of floating point numbers which can be plotted as a graph to detect the end of insertion shown if Figure 6.1.

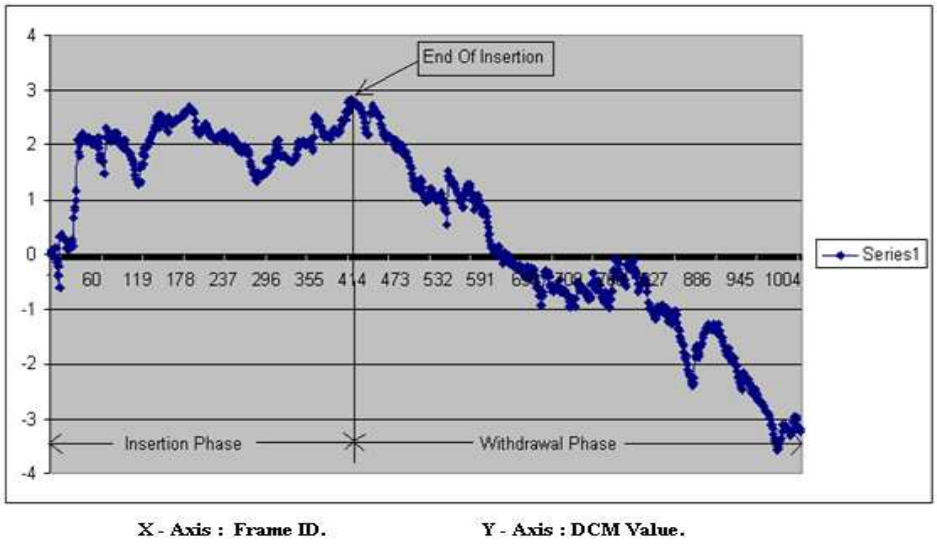

Figure 6.1 DCM value graph.

Typically the end of insertion is reached when the doctor reaches appendix or terminal ileum which can be seen in Figure 6.2. Usually when this region is reached, the doctor confirms this by his voice as well. Thus I have another way to prove that the results obtained in this method comply with those of doctors. Although the results can never be 100% accurate because even after reaching the appendix, the doctor might want to study the appendix as well. He might go more forward and then backward in the

course of time. This will lead to a change in the peak of the DCM value shown as Figure 6.1. Thus it is normal to have accuracy differ by a few seconds. I detect insertion phase time, withdrawal phase time and no camera motion time. Other values are speed and variances of camera motion. By all these metrics an idea about the efforts taken by the doctor can be estimated [12] [13].

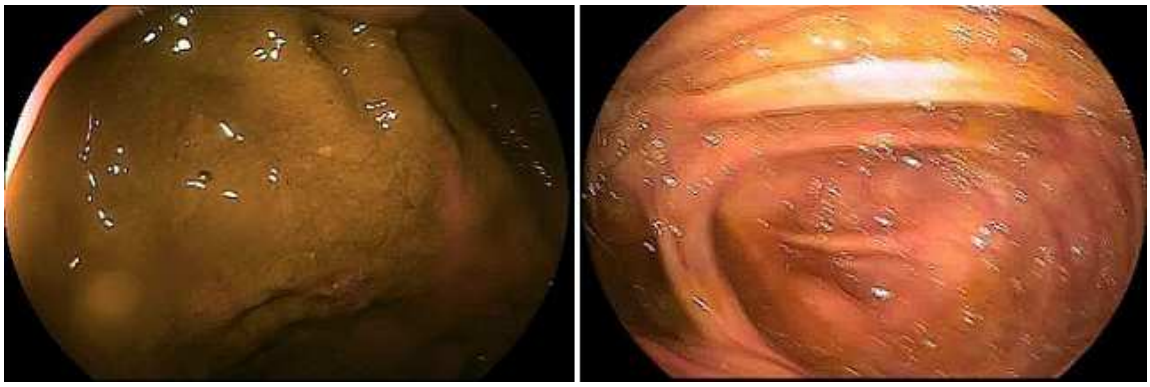

Figure 6.2 Appendix.

Also this solves one more conflict, for example if the same video is given to many doctors and they are asked to tell the end of insertion. It might happen that there is a difference in opinion of these doctors. To solve these kinds of ambiguities, results from our experiments can be used. Thus there is a way to distinguish between different colonoscopy procedures.

CHAPTER 7

MULTITHREADING FOR THE POST PROCESSING

Present desktop computers are equipped with a minimum of dual core processors nowadays. I made a special version for such machines; a special version was required because I cannot assume that the system running this project is a multi core CPU. The post processing version without multithreading is necessary. This is because if I do multithreading on a single core machine like the Intel Pentium 4 or 3 then instead of speed up, it will increase the execution time because of many context switches. I made a version which has 8 threads made specifically for the dual core and the quad core CPU's [18-20]. The actual breakup of the input data can be seen in the Figure 7.1.

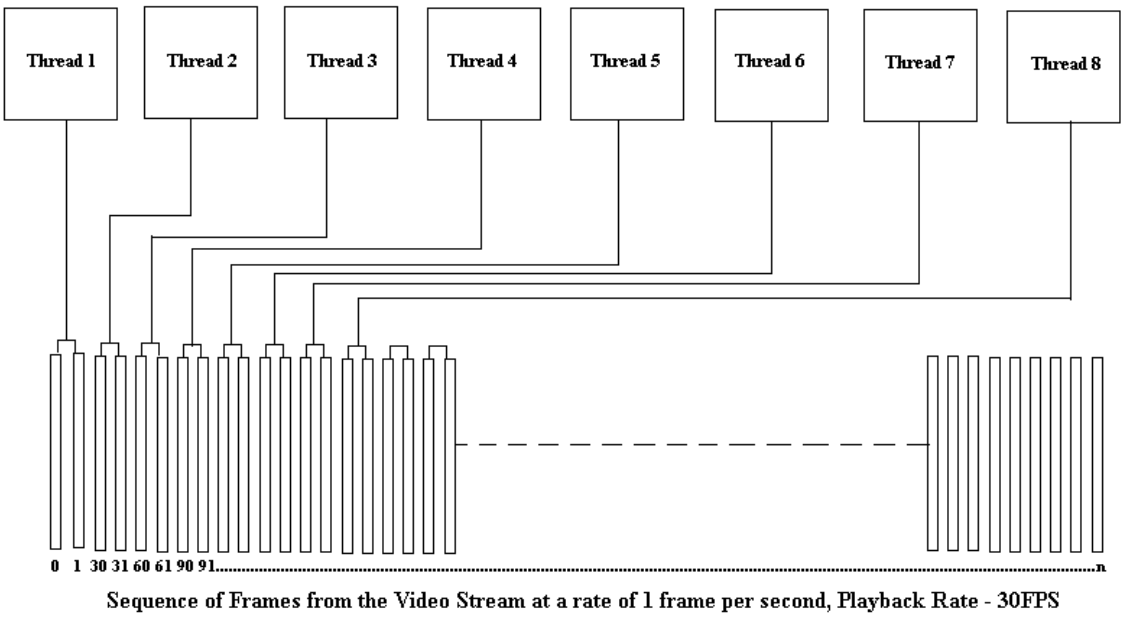

Figure 7.1 Multithreading of motion vector search.

Here as can be seen a pair of two frames is given as input to each thread, one is the current frame and another is the reference frame, the frames were extracted at a rate of 1 frame per second. Thus next thread takes 30-31 if the thread 1 took 0-1 and so on at a distance of 30 frames. This specific number of 8 threads was a tradeoff ratio between performance and context switch times. The comparison of this multithreaded version in terms of the run time can be seen in the experimental results section. I tested this program on 2 configurations 1st one was an Intel Core 2 Duo, 3 GHz with 4 GB of RAM while other one was Intel Xeon Quad core (Dual Processors) with 4 GB of RAM and 250 GB RAID hard drive. The results were excellent when this program was run on the second machine. On the first machine it was about twice as faster as compared to the single thread version. This improvement can be utilized in two ways, first one is to gain more speed like discussed above and the second option is to keep the speed same while increase the accuracy. This is by increasing the frame rate from 1 frame to 2 frames or 3 frames per second. When I tested the performance with more frame rate, the accuracy was increased to about 10% for ever frame increase. Using this power to reduce the execution time resulted in about 50% improvement, thus the best idea was to increase the speed instead accuracy. Also if I would have increased the accuracy and the program was made to run on a single CPU core machine, it would have taken twice the long time as compared to the previous version.

CHAPTER 8

OTHER METHODS INVESTIGATED

The methods described above were not finalized until I experimented with some other search techniques and compared the results with the ground truths. For some cases the accuracy was good but it took very long time for processing, on the other side some techniques were extremely fast but the accuracy was unacceptable. I discuss some of these techniques and the reason for them not being used in the following sub sections.

8.1 Variations in Search Area

The first idea that was used is to have a full exhaustive search of 16X16 pixel blocks in an area of 48X48 pixels. This provided the most accurate results but it took so long that even a highly configured system could not do it faster than twice the video length time [13]. As can be seen in Figure 8.1.1, the total number of comparisons was 1024 for every block.

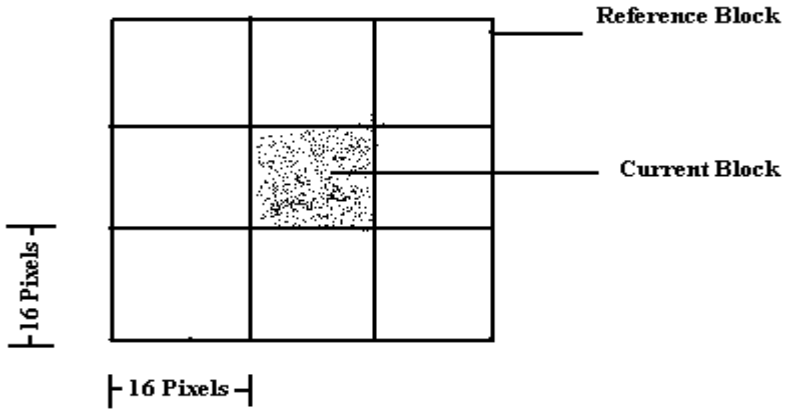

Figure 8.1.1 48X48 Pixels search area.

The second variation did not make an exhaustive search, but it actually made a 9 block search, as can be seen from Figure 8.1.1, the central current block was searched in all its adjacent blocks and the nearest match was selected. This method performed most poorly, with accuracy which was unacceptable. However, this version is so fast that it could detect end if insertion in a 1 hour video, in just around 5 minutes.

The third variation was to go to a long extent in only straight paths; this idea can be understood from the Figure 8.1.2. Here the central current block is searched in a total of 33 locations; this is somewhere in between the two methods described earlier.

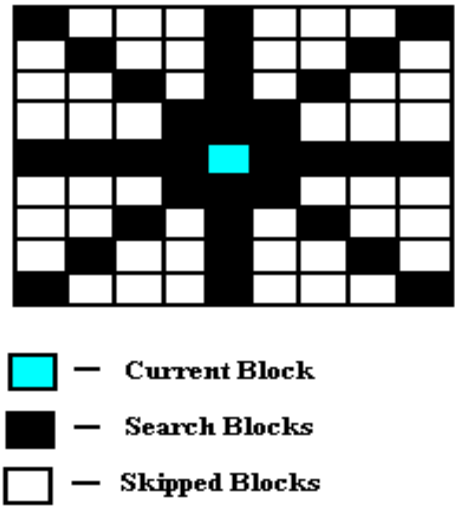

Figure 8.1.2 Straight path search.

However the results were good but I wanted something which could do even better. I also tried a method where all the blocks were considered in Figure 5.3.2, at a displacement of 1 block or 16 pixels. Finally I implemented the 4 block or 32X32

exhaustive searches, which gave very good results and also took less time. So it was like, I found out the most favorable tradeoff between the time taken for execution and the accuracy, which can be later seen in the experimental results section.

The results and the trade off ratios can also be understood from the table 8.1.1**.** I also experimented with various block sizes. I started with 4X4 pixels up to 32X32 pixels with varying accuracies and time. Small block sizes helped in faster execution especially when the exhaustive search was used, because initially the idea was to have for example 3X3 blocks search area for exhaustive search.

Table 8.1.1 Comparison of the search patterns.

	48X48 Exhaustive Search	9 Block Search	Straight Path Search	81 Block Search	32X32 Exhaustive Search
Total Searches	1024	9	33	81	256
Speed	4X Slower	25X Faster	8X Faster	3X Faster	1X
Accuracy	95%	55%	74%	80%	90%.

So this means that if the block size is 16X16 pixels then the search area would become 48X48, but if 8X8 block is considered the search area is reduced to ¼, that is

24X24 pixels. After a lot of experimentation I finalized 16X16 pixels to be the best block size.

8.2 Frame Skipping

Another factor which affected the performance was the distance between the current frame and the reference frame. For example if the frames considered are 0-1, then 0 is the reference frame and 1 is the current frame, it played an important role on how much distance should be allowed to get better motions estimation. If the distance is 1 as mentioned above, there is less chance of getting accurate motions if our search area is small, for example if I use 9 block search then definitely there will be match available but that might go wrong in predicting the DCM value. So a balance between the search method and frame distance had to be decided. From experiments I found that the results were best if the frames were 2 frame distant, but also it required the best search algorithm that is 48X48 exhaustive search. This is because the chances of finding better true and accurate motions increase as I go in increasing order of the frame distance. I found out from the MPEG motion vectors that the maximum extent to which a block in a frame can move is around 42 [7], but if I had used 42X42 searches, then the time taken is very high. Also as I had decided the 32X32 exhaustive search methods already to be the best candidate, I concluded from experiments that the best skip is 1, an example as can be seen in the Figure 8.2.1. A sample test of 48X48 search area with different frame distances is shown. Here as can be seen the 2 frame skip is the best for 48X48 search areas, thus 32X32 for 1 frame distance.

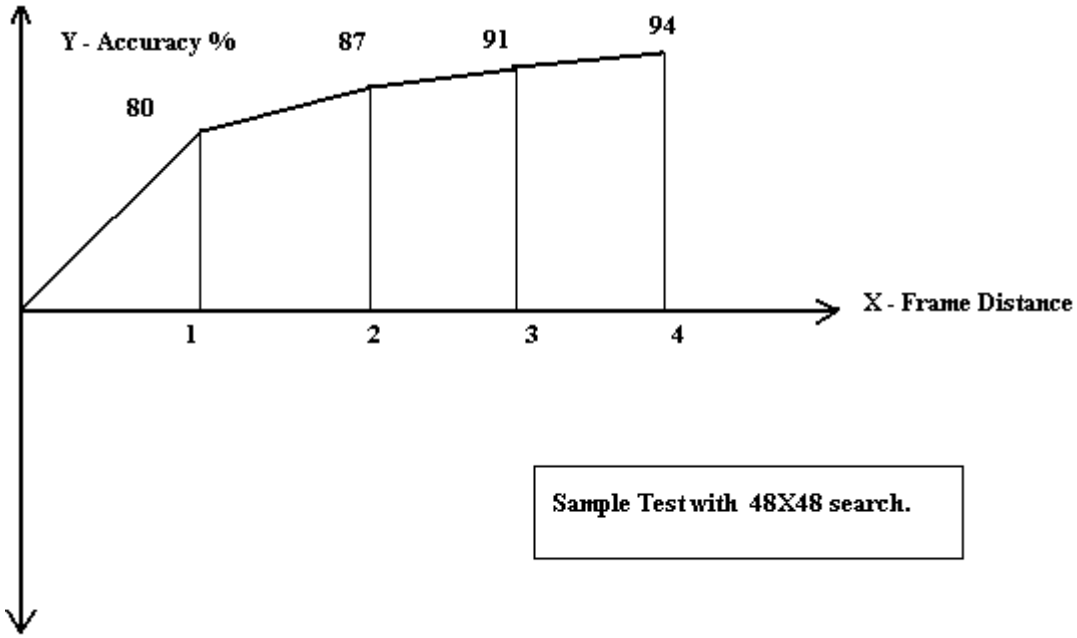

Figure 8.2.1 Frame skip test with 48X48 searches.

This solved one more problem, typically in MPEG only P and B frames are encoded using the motion vector approach. If many "I" frames come in between, the motion for that portion cannot be estimated correctly, this results in reduced accuracy. Also the previous implementation only reads the motion vector values from the MPEG stream [13]. Thus it had no way to alter the accuracy by fine tuning, just like what I did in this implementation. This also made the old implementation more prone to the encoding standards of the Endoscope camera. In other words this method might give accurate results for one manufacturer of camera, while less accurate for the other one. Camera manufacturers decide the encoding standard to be used. The MPEG standard does not define the coding mechanism, and it accepts a stream as MPEG only if it can be played using the MPEG standard decoder player [7][8]. The encoding process is left to the

manufacturer. The race for more information in less disk space gives rise to competent algorithms which work very well for the compression domain, but are futile for our purpose. So in other words, our application is camera manufacturer independent, this can handle even the streams from the new coding techniques that will be released in future, for example our target is the “High Definition” cameras which have a huge resolution starting from 1024X768 pixels.

CHAPTER 9

REAL TIME END OF INSERTION DETECTION

A modification to the current modules makes it usable if the video is streaming or coming directly from the endoscope. The old version and the new version discussed earlier cannot handle live streaming videos [12]. The modification is to consider the stream as a stream of bitmap images and then to consider 1 frame every second, for example frame 1 becomes the reference frame while frame 2 becomes the current frame, after this, 30 frames are skipped to make 31 as reference frame and 32 as current frame. Another difference is that the affine model is called after processing of every frame followed by the quality metrics module. This is done to detect current end of insertion, here I maintain a threshold of 1 minute, if the value currently detected as end of insertion does not change for one minute, and I declare this as time for end of insertion with the frame ID. The working can be better understood from Figure 9.1 and the typical error rates from table 9.1.

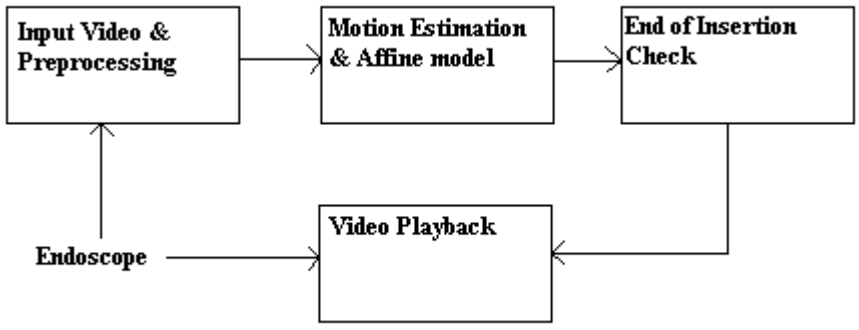

Figure 9.1 Real Time End of Insertion.

Table 9.1 Error rates in real time end of insertion.

Video Name	Ground Truth (EOI)	Real time Version (EOI)	Total Error Count.
Camera 1			
1.mpg	06:02	04:16	0
2.mpg	05:21	05:34	0
3.mpg	15:25	15:22	0
4.mpg	18:06	20:32	0
5.mpg	31:13	29:28	1
6.mpg	05:14	05:12	0
7.mpg	11:16	12:26	0
8.mpg	03:47	05:24	0
9.mpg	26:01	23:46	2
10.mpg	12:59	15:32	1
Difference		01:23	
Camera 2			
1.mpg	08:32	06:50	0
2.mpg	08:18	04:30	0
3.mpg	09:06	11:08	1
4.mpg	05:55	07:36	0
5.mpg	08:05	14:46	1

Table continued on next page.

Table 9.1 continued.

Video Name	Ground Truth (EOI)	Real time Version (EOI)	Total Error Count.
6.mpg	04:26	04:30	0
7.mpg	07:25	08:56	0
8.mpg	12:05	10:56	1
9.mpg	06:17	07:20	1
10.mpg	01:54	04:00	0
Difference		02:11	

As can be seen in Table 9.1, that if the threshold of 1 minute is used the typical error generated is 1, that means only once the end of insertion was detected incorrectly, however as soon as the new correct value is received, the old value is replaced. For our test purposes the maximum error was 2 for only 1 video.

In the real time version the Affine model was implemented in C/C++ as compared to the older version which was in MATLAB. This gave me two advantages, one is the speedup I get by using C while the other is to embed this code directly in the code for motion vectors estimation which is also in C. One of the problems which the old version had was that it had separate exe files for motion and affine model. If I had to make a real time version from that, it would have taken very long time as after every frame processing another exe had to be called from the program. This takes a long time as compared to calling a function in the same program. This process continues until the

video is over, however this real time processing comes at the price of reduced accuracy as I cannot go beyond 1 FPS (Frame Pair per Second). The post processing version is more accurate as compared to the real time version because of 2 FPS rate, but at the cost of more execution time. I also implemented a video player which can display the stream of video from the endoscope and when the earlier discussed program detects the end of insertion the time starts to flash on the video. Also I noted one more feature which takes care of number of times the end of insertion is detected falsely. Recall that I declare a point as end of insertion only when it does not change for 1 minute threshold. If there is an event that a new point is declared as end of insertion after this one minute threshold, I increase the error count by 1. Typically the numbers of errors were 0 and 1 for most of the cases. The algorithm for this can be seen as follows,

Let *CADCM* represent the current accumulated DCM value since the beginning of the procedure. '*Current DCM'* is the DCM value of the current frame and '*previous DCM'* is the DCM value computed before the current DCM value. '*Max. DCM*' keeps track of the maximum DCM value seen so far. Initialization of these variables is performed before the following computation starts. The current time is the timestamp of the current image given the zero timestamp of the image at the start of the procedure.

Loop until the end of video is detected

1. Compute *current DCM*
2. *CADCM = CADCM + current DCM*
3. If *CADCM > Max. DCM*, then

- *Max. DCM* = *CADCM*, and
- the temporary End of insertion time = the current time

4. Otherwise,
   - Compute a difference between the temporary End of insertion time and the current time.
   - If this difference is greater than a threshold (T_E), I take the temporary End of insertion time as the End of insertion time
5. The previous DCM = the current DCM;

I use 1 minute for the threshold, T_E. If the value currently detected as end of insertion does not change for one minute, I declare this as time for current end of insertion with the frame ID.

9.1 Multithreading for the Real Time End of Insertion

I developed an algorithm for the real time end of insertion. The basic difference between this algorithm and the one for post processing is that the post processing version assigns a thread to a pair of frame. At any given time the number of frames being processed will be equal to the number of threads spawned in the system [18-20]. This is in turn equal to the number of CPU cores available in the system, for example, single core, dual core, quad core, dual quad core, etc. The algorithm can be understood as follows.

Assume that I select one pair of frames (i.e., reference frame (f_i) and current frame (f_{i+1})) to process per second, and there are ***m*** number of threads. One frame size is 720 x

480 pixels, and it is divided into a number of blocks (16 x 16 pixels). Since I am not processing the boundary blocks and corner blocks, the total number of blocks (***B***) to be processed is 1,200. Our threading algorithm is as follows;

1. Compute a number of blocks (***b***) to be processed in each thread, $\boldsymbol{b} = \boldsymbol{B} / \boldsymbol{m}$,
2. Divide the total number of blocks, ***B,*** into ***m*** numbers of groups, g_1, g_2, … $g_{\boldsymbol{m}}$, in which each group has ***b*** numbers of blocks,
3. A reference frame (f_i) and g_1 of current frame (f_{i+1}) are given as inputs to the first thread, and a reference frame (f_i) and g_2 of current frame (f_{i+1}) are given as inputs to the second thread, and so on, and
4. After all threads generate their outputs, I combine them into one result.

I have implemented two versions of the phase boundary detection in a real-time environment, which are with and without this CPU multithreading. I implement the algorithms using C language [18-20]. Thus I assign all the available threads to only one frame, to speed up process of finding motion vector for this frame in contrast to speeding up the whole process which I do in the post processing version. Actually I could have done this in the Post processing version as well, but this method cannot make use of the pipelining of CPU processing and disk access. Here the motion extraction part consists of 3 phases, first one being reading current and reference images from the disk, second being computing motion vectors and finally writing the results back to the disk. Here it happens that when the disk access is being made, all the CPU threads are idle, but in case of the post processing, when one thread is busy reading data from the disk, another might be busy in computing motion vectors. So I have much higher efficiency in the post

processing version. In the real time version, the scenario is different, because I just have one pair of frame for processing, and the next frames are yet to arrive, so to maximize efficiency I need to assign all the available threads to quickly process this pair of frames. The details can also be understood from the Figure 9.1.1 below.

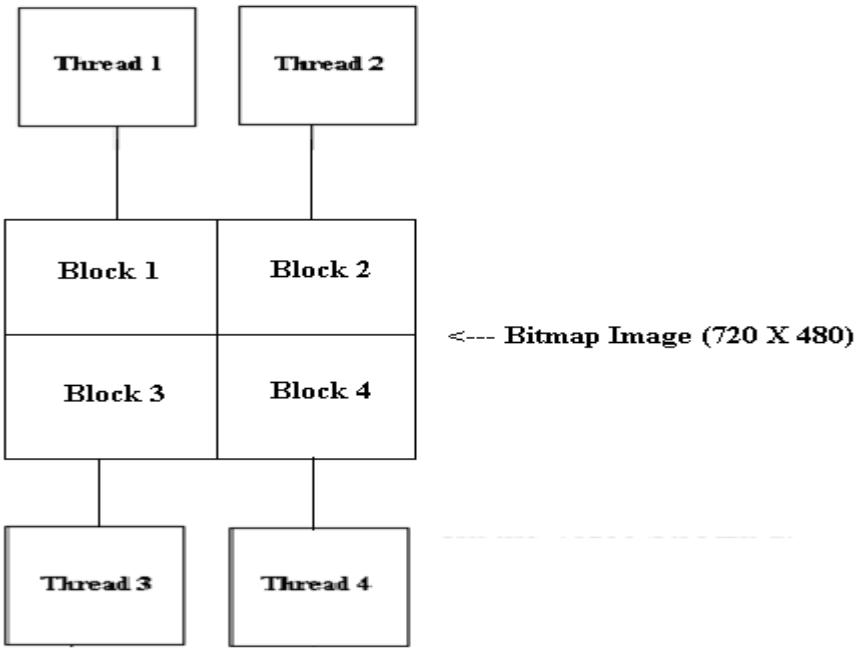

Figure 9.1.1 CPU multithreading for the real time.

Each block here can consist of many 16X16 macro blocks which are processed as a group by the assigned thread. This is actually grouping the total macro blocks in the image based on the number of threads available. All the available results are then collected from each thread and written to the motion vector file, which is later processed by the affine model and then the end of insertion detection eventually.

CHAPTER 10

EXPERIMENTAL RESULTS

Table 10.1 shows the comparison of execution time of the system with old version of motion vector estimation with the new version. Here I see an important property, for the camera 2 which is Olympus endoscope, I was able to get more than twice the performance because of the border blocks and black block skipping. This is because Olympus camera generated videos have more black or dead area as compared to the Fujinon. MPEG based previous implementation [12] [13] could not make use of this feature, hence it was much slower. For the Fujinon videos I was able to get around 1.8 times speed up on an average. Table 10.2 shows the accuracy comparison as compared to the ground truth found by doctors for the end of insertion in terms of time. The candidates are 20 videos from 2 different endoscope camera manufacturers. As can be seen from table 2 the accuracy has not been changed much but the execution time is reduced to almost half as a result of this method. If looked carefully at the table 2 it is seen that for some cases the results are even better as compared to the old program. This speedup was also possible because of the implementation done in C/C++ as compared to java in the old version of the same. The table 10.3 shows the end of insertion detection results for the real time version of the program. Here it can be seen that I have achieved the speed up at the expense of multithreading. However, important point to be noted is that the accuracy is still better than the accuracy results for the previous implementation. So in other words I can summarize the results like Old version (Lowest Accuracy), Real

Time Version (Medium Accuracy) and New Version (Highest Accuracy). So in many ways the new implementation is better than the previous version. The system is not binding to use only the above two mentioned endoscope camera manufacturers. As the methods explained before are capable of fixing the search area in any type of video.

Table 10.1 Comparison of execution times.

Video Name	Old Version (Execution Time)	New Version (Execution Time) Without Multithreading	Speed Up (X Times)
Camera 1			
1.mpg	45:12	24:57	1.81162325
2.mpg	1:12:10	43:00	1.67674498
3.mpg	1:16:01	46:00	1.65434756
4.mpg	3:53:13	2:11:26	1.77616941
5.mpg	2:12:09	1:25:46	1.54563538
6.mpg	39:40	20:48	1.90705128
7.mpg	1:16:38	43:45	1.75788262
8.mpg	36:08	18:01	2.00555042
9.mpg	2:09:50	1:19:03	1.77324387

Table continued on next page.

Table 10.1 continued.

Video Name	Old Version (Execution Time)	New Version (Execution Time) Without Multithreading	Speed Up (X Times)
10.mpg	1:14:49	43:44	1.71477901
Camera 2			
1.mpg	58:42	24:33	2.3910387
2.mpg	57:56	24:39	2.35023665
3.mpg	1:03:00	30:58	2.06017005
4.mpg	1:11:07	31:44	2.26049618
5.mpg	1:42:09	38:47	2.65375617
6.mpg	40:21	19:18	2.09067358
7.mpg	52:08	22:32	2.31360947
8.mpg	1:09:05	28:57	2.41687084
9.mpg	1:08:08	23:05	2.95357918
10.mpg	23:32	09:36	2.45138889
Average			2.42790512

One thing to be noted is that the execution time of the Real time End of Insertion version is same as that of the video length. Here I see that I gain this greater speed up by implementing multithreading.

10.2 Accuracy comparison.

Video Name	Ground Truth	Old Version	Difference	New Version	Difference
Camera 1					
1.mpg	06:02	04:07	01:55	04:16	01:46
2.mpg	05:21	03:45	01:36	05:34	00:13
3.mpg	15:25	07:35	07:50	15:25	00:00
4.mpg	18:06	19:24	01:18	17:58	00:08
5.mpg	31:13	31:51	00:38	30:21	00:52
6.mpg	05:14	04:48	00:26	05:13	00:01
7.mpg	11:16	11:26	00:10	11:27	00:11
8.mpg	03:47	03:49	00:02	03:04	00:43
9.mpg	26:01	09:54	16:07	22:07	03:54
10.mpg	12:59	10:35	02:24	15:02	02:03
Average			03:15		00:59
Camera 2					
1.mpg	08:32	10:48	02:16	10:39	02:07
2.mpg	08:18	05:37	02:41	08:50	00:32
3.mpg	09:06	06:33	02:33	11:08	02:02
4.mpg	08:46	08:24	00:22	05:25	03:21

Table continued on next page

Table 10.2 continued

Video Name	Ground Truth	Old Version	Difference	New Version	Difference
5.mpg	08:05	10:09	02:04	10:22	02:17
6.mpg	06:10	06:37	00:27	04:54	01:16
7.mpg	08:50	09:39	00:49	08:57	00:07
8.mpg	12:05	14:57	02:52	10:56	01:09
9.mpg	06:17	07:36	01:19	07:21	01:04
10.mpg	01:54	01:53	00:01	01:48	00:06
Average			01:32		01:24

Table 10.3 Real Time End of Insertion.

Video Name	Ground Truth (EOI)	Real time Version (EOI) Difference	Errors
Camera 1			
1.mpg	06:02	01:46	0
2.mpg	05:21	00:13	0
3.mpg	15:25	00:00	0

Table continued on next page.

Table 10.3 continued.

Video Name	Ground Truth (EOI)	Real time Version (EOI) Difference	Errors
4.mpg	18:06	00:08	0
5.mpg	31:13	00:52	1
6.mpg	05:14	00:01	0
7.mpg	11:16	00:11	0
8.mpg	03:47	00:43	0
9.mpg	26:01	03:54	2
10.mpg	12:59	02:03	1
Average		00:59	
Camera 2			
1.mpg	08:32	02:07	0
2.mpg	08:18	00:32	0
3.mpg	09:06	02:02	1
4.mpg	08:46	03:21	0
5.mpg	08:05	02:17	1
6.mpg	06:10	01:16	0

Table continued on next page.

Table 10.3 continued

Video Name	Ground Truth (EOI)	Real time Version (EOI) Difference	Errors
7.mpg	08:50	00:07	0
8.mpg	12:05	01:09	1
9.mpg	06:17	01:04	1
10.mpg	01:54	00:06	0
Average		01:24	

I also tested the program for difference in the execution time for different computer configurations as I need to set the minimum system requirements for anyone to run this project. The comparisons can be seen in Table 10.4. As can be seen the important factor here is the BUS speed of the motherboard and the processor speed. This is because in our application the main bottleneck is the data transfer rate between the RAM and the Disk. The bitmap images are large in size; it is not possible to store many of them at a time in the processing queue. This testing was done for the post processing version and not the real time version. So in other words in order to run the real time version Computer 1 configuration is the optimum which is in column 2. The graphics card however does not play any kind of significant role in this, but may be required for future versions. I am

trying to focus into the new generation graphics computing languages which make use of these cards apart from the gaming.

Table 10.4 Computer configuration comparison.

Video	Computer 1 Post Processing.	Computer 2 Post Processing.	Computer 3 Post Processing.
Test1.mpg	13:35	34:56	50:12
Test2.mpg	21:16	1:01:03	1:36:15
Test3.mpg	22:06	1:03:14	1:39:36
Test4.mpg	12:10	29:48	48:18
Test5.mpg	11:20	26:52	47:01
CPU	Intel Pentium Core 2 Duo 3.0 GHz	Intel Pentium Core 2 Duo 1.86 GHz	Intel Pentium D Processor 3.0 GHz.
RAM	3.00 GB	2.00 GB	2.00 GB
RAM BUS	800 MHz	533 MHz	533 MHz
SYSTEM BUS	1333 MHz	1066 MHz	800 MHz
HARD DRIVE	500 GB 7200 RPM	750 GB 7200 RPM	750 GB 7200 RPM
OS	Windows XP SP2	Windows XP SP3	Windows XP SP2
GRAPHICS CARD	NVIDIA 8800GTX, 750 MB	NVIDIA GTX 280, 1 GB	NVIDIA 8800GTX, 750 MB
L2 CACHE	4 MB	2 MB	1 MB

A comparison of different frame rates for this program can be seen in table 10.6, here I used different frame rates because of the speed up I achieved. The multithreading performance can be seen in table 10.5. After experimentation I found that the frame rate can be taken to 2 FPS increasing the accuracy. For the real time version the accuracy could be increased by taking 1 FPS into consideration, without this I was bound to have 0.5 FPS rate.

Table 10.5 Multithreading execution time comparison (FPS – Frame Pair per Second).

Video	1 FPS Run Time (Core 2 Duo)	1 FPS Run Time (Core 2 Quad)	1 FPS Run Time (Intel Xeon, Dual Quad Core)
1.mpg	17:57	11:21	06:35
2.mpg	31:38	22:49	13:18
6.mpg	14:35	09:07	05:52
8.mpg	13:21	09:01	06:47
10.mpg	33:18	22:35	14:32
0103.mpg	16:23	11:32	9:13
0154.mpg	15:45	12:27	8:40
0183.mpg	6:13	4:58	3:37
0118.mpg	19:36	15:01	12:02
0152.mpg	12:24	7:37	5:18

Table 10.6 Comparison of different frame rates (FPS – Frame Pair per Second).

Video	Ground Truth	1 FPS Difference.	2 FPS Difference.	3 FPS Difference.
1.mpg	06:02	01:46	01:22	00:00
2.mpg	05:21	00:13	00:04	00:11
6.mpg	05:14	00:01	00:01	00:00
8.mpg	03:47	00:43	00:02	00:09
10.mpg	12:59	00:19	01:46	01:07
0103.mpg	08:32	02:45	01:02	01:43
0154.mpg	08:50	00:07	00:24	00:21
0183.mpg	01:54	00:14	00:04	00:14
0118.mpg	08:46	01:09	00:47	00:32
0152.mpg	06:10	00:24	00:24	00:43
Average	**Difference**	00:46	00:36	00:30

Table 10.7 Computer 1: Core 2 Duo 3.0 GHz, 3 GB RAM, 500 GB eSATA (FPS – Frame Pair per Second).

Video	1 FPS Run Time	2 FPS Run Time	3 FPS Run Time
1.mpg	17:57	31:22	46:10

Table continued on next page.

Table 10.7 continued.

Video	1 FPS Run Time	2 FPS Run Time	3 FPS Run Time
2.mpg	31:38	56:43	1:15:21
6.mpg	14:35	26:56	40:12
0103.mpg	16:23	29:53	51:33
0154.mpg	15:45	28:22	45:54
0183.mpg	06:13	12:18	19:18
0118.mpg	19:36	39:42	1:03:41
0152.mpg	12:24	23:01	35:11
8.mpg	13:21	25:09	31:52
10.mpg	33:18	1:01:18	1:09:52

Table 10.8 Computer 2: Intel Xeon Quad Core (2 CPU's) 3.0 GHz, 3GB RAM, 160 GB RAID Drive. (FPS – Frame Pair per Second).

Video	1 FPS Run Time	2 FPS Run Time	3 FPS Run Time
1.mpg	06:35	15:12	21:36
2.mpg	13:18	27:52	42:18
6.mpg	05:52	13:11	19:11
0103.mpg	09:13	17:22	24:38

Table continued on next page

Table 10.8 continued.

Video	1 FPS Run Time	2 FPS Run Time	3 FPS Run Time
0154.mpg	08:40	16:01	23:27
0183.mpg	03:37	05:45	08:01
0118.mpg	12:02	21:48	32:01
0152.mpg	05:18	12:00	17:23
8.mpg	06:47	11:30	18:48
10.mpg	14:32	28:23	43:11

Thus considering table 10.6, 10.7 and 10.8, I decided to choose 2 FPS as ideal for our application. This was the best trade off ratio between accuracy and execution time. However as we see in table 10.8, the multithreaded aspect of our application can make use of the higher end CPU's better. Thus in future when available computation power increases I can even think of higher frame rates.

CHAPTER 11

FUTURE ENHANCEMENTS

I am also considering improving the search method technique; the candidates are 3 Step and 2 D Logarithmic search. This is to reduce the execution time taken and also to achieve the same or better accuracy level. I also considered another fact that the accuracy is affected if the frames considered for motion estimation consists of blurred images. I wish to do a preprocessing step where on clear frame are allowed to be examined for motion vectors. For example if the current pair considered for motion vectors say 10-11 have either 10 or 11 as blurry, I ignore this pair completely. However this will introduce an additional extra step, but will always generate accurate results.

The future work in this research is taking a direction where I am trying to make use of Graphics cards for the processing [23][26][27]. Graphics cards can be seen as collection of tiny processing elements with less power as compared to the CPU [26]. The typical memory available to high end graphics cards starts from 512MB which is dedicated and inbuilt into the graphics card processor. This skips the bottleneck of the data transfer speeds created by the North Bridge in the motherboard. This implementation has been started with the new generation GPU language called CUDA (Compute Unified Device Architecture) [21][22][24][25] developed as open source by NVIDIA Corporation. The first version of the implementation is done. A sample of normal speedup obtained by using the graphics card for cumulative addition of a string of number is shown in Figure 9.1. The GPU card used for this purpose was the NVIDIA

8800 GTX, this processor has 128 Processing elements of 600 MHz clock frequency each, with 750 MB of dedicated DDR3 SDRAM.

The latest version of graphics card available from NVIDIA is the GTX 280, with 240 Parallel processing elements and 1 GB of DDR3 SDRAM. The approach I made use of can be described by Figure 11.1. PE stands for processing elements.

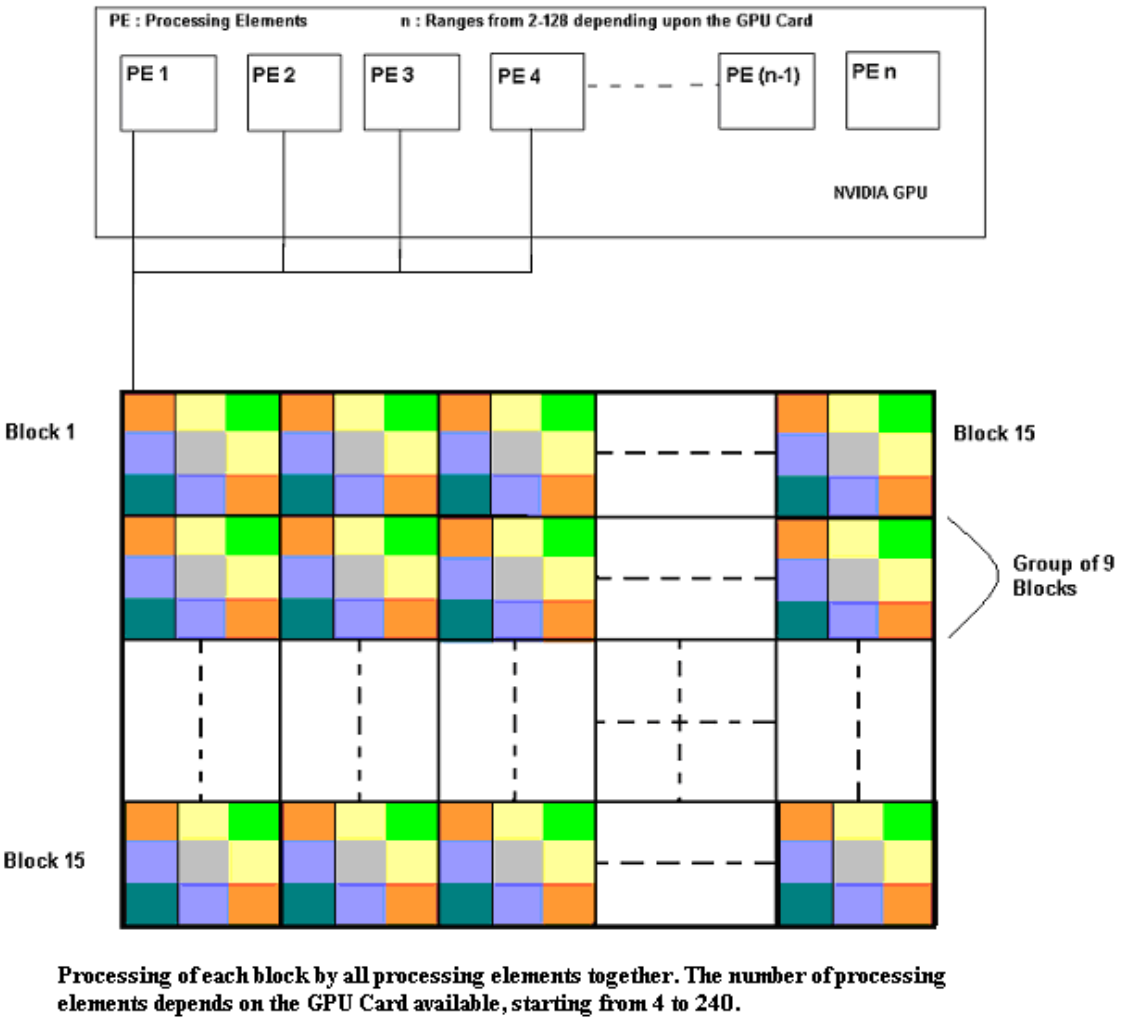

Figure 11.1 Sample of GPU implementation.

REFERENCES

[1] *American Cancer Society*. "Colorectal Cancer Facts & Figures 2008." http://www.cancer.org/docroot/STT/content/STT_1x_Cancer_Facts_and_ Figures_2008.asp, February 2nd 2009.

[2] Jung Hwan Oh, Malik Avnish Rajbal, Jayantha Kumara Muthukudage, Wallapak Tavanapong, Johnny Wong, Piet C. de Groen , "Real-Time Phase Boundary Detection in Colonoscopy Videos," *ISPA 2009* (Accepted for Publication on June 2009).

[3] A. Pabby, R. E. Schoen, J. L. Weissfeld, R. Burt, J. W. Kikendall, P. Lance, E. Lanza, and A. Schatzkin. "Analysis of Colorectal Cancer Occurrence During Surveillance Colonoscopy in the Dietary Prevention Trial," *Gastrointestinal Endoscopy*, vol. 61(3), 2005, p.385 – 391.

[4] D. K. Rex, J. H. Bond, S. Winawer, T. R. Levin, R. W. Burt, and D. A. J. et al. "Quality in the Technical Performance of Colonoscopy and the Continuous Quality Improvement Process for Colonoscopy: Recommendations of the U.S. Multi-Society Task Force on Colorectal Cancer," *American Journal Gastroenterol*, 97(6): 2002, p.1296 – 1308.

[5] D. K. Rex, J. L. Petrini, T. H. Baron, A. Chak, J. Cohen, S. E. Deal, B. Hoffman, B. C. Jacobson, K. Mergener, B. Pertersen, M. A. Safdi, D. O.

Faigel, and I. M. Pike. "Quality Indicators for Colonoscopy," *Gastrointestinal Endoscopy*, vol. 63, 2006, p.S16 – S26.

[6] Morimoto, C.; Burlina, P.; Chellappa, R. "Video Coding Using Hybrid Motion Compensation," *Proceedings., of International Conference on Image Processing, 1997*. Volume 1, 26-29 Oct. 1997 p. 89 – 92.

[7] http://www.mpeg.org/MPEG/video/, February 24th 2009.

[8] http://en.wikipedia.org/wiki/MPEG1/, February 24th 2009.

[9] Yatabe, Y.; Fujimoto, M.; Sodeyama, K.; Komi, H. "An MPEG2/4 dual codec with sharing motion estimation," *IEEE Transactions on Consumer Electronics*, Volume 51, Issue 2, May 2005 p.660 – 664.

[10] R. L. Lagendijk and J. Biemond. "Basic Methods for Image Restoration and Identification", Chapter 3.5, Academic Press, 2000, p. 125 – 139.

[11] J. Caviedes, S. Gurbuz. "No-Reference Sharpness Metric Based on Local Edge Kurtosis," *In Proc. IEEE, International Conference on Image Processing*, 3: June 2002, p. 53 – 56.

[12] J. Oh, S. Hwang, Y. Cao, W. Tavanapong, D. Liu, J. Wong and P. C. de Groen. "Measuring Objective Quality of Colonoscopy," *IEEE*

Transactions on Biomedical Engineering. (Accepted for publication on July 2008).

[13] S. Hwang, J. Oh, J. Lee, P. C. de Groen, Y. Cao, W. Tavanapong, D. Liu, and J. Wong. "Automatic Measurement of Quality Metrics for Colonoscopy Videos". *Proc. of ACM Multimedia 2005*, Singapore November 2005, p. 912 – 921.

[14] B. P., G. M., and G. F., "A unified approach to shot change detection and camera motion characterization," in *IEEE Trans Circuits and Syst. for Video Technol.*, 1999, p. 1030 – 1044.

[15] X. Cao and P. N. Suganthan, "Video Shot Motion Characterization based on Hierarchical Overlapped Growing Neural Gas Networks," in *Multimedia Systems*, October 2003, p. 378 – 385.

[16] H. Yi, D. Rajan, and L.-T. Chia, "Automatic Extraction of Motion Trajectories in Compressed Sports Videos," in *ACM Multimedia 2004*, New York, NY, October 2004, p. 312 – 315.

[17] B. Furht, J. Greenburg, and R. Westwater. "Motion Estimation Algorithms for Video Compression," *Kluwer International Series in Engineering & Computer Science*, 1996.

[18] C. Kyriacou, P. Evripidou, P. Trancoso. "Data-Driven Multithreading Using Conventional Microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Volume 17, Issue 10, Oct. 2006 p.1176 – 1188.

[19] http://community.edc.intel.com/t5/Multicore-Virtualization-Blog/Multithreading-a-Software-Application-for-Performance-on-Multi/bc-p/110;jsessionid=BA0B1BC84BCBEDF17ECA882C03EB798C, February 16th 2009.

[20] J.G. Holm, S. Parkes, P. Banerjee. "Performance evaluation of a C++ library based multithreaded system," *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Volume 1, 7-10 Jan. 1997 p. 282 – 291.

[21] http://www.nvidia.com/object/cuda_develop.html, March 12th 2009.

[22] Zhiyi Yang, Yating Zhu, Yong Pu. "Parallel Image Processing Based on CUDA," *Computer Science and Software Engineering, 2008 International Conference ,* Volume 3, 12-14 Dec. 2008, p.198 – 201.

[23] Wei-Nien Chen, Hsueh-Ming Hang. "H.264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA)," *Multimedia and Expo, 2008 IEEE International Conference,* June 23 2008-April 26 2008, p.697 – 700.

[24] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Yao Zhang, Volkov, V. "Parallel Computing Experiences with CUDA," *Micro, IEEE,* Volume 28, Issue 4, July-Aug. 2008, p.13–27.

[25] Seung In Park, Ponce, S.P., Jing Huang, Yong Cao, Quek, F. "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture," *Applied Imagery Pattern Recognition Workshop, 2008*, AIPR '08. 37th IEEE 15-17 Oct. 2008, p.1 – 7

[26] Buck, Ian "GPU Computing: Programming a Massively Parallel Processor," *Code Generation and Optimization, 2007*, CGO '07. International Symposium on 11-14 March 2007, p.17 – 17.

[27] Qihang Huang, Zhiyi Huang, Werstein, P., Purvis, M. "GPU as a General Purpose Computing Resource," *Parallel and Distributed Computing, Applications and Technologies, 2008*, PDCAT 2008. 9th International Conference on 1-4 Dec. 2008, p.151 – 158.