EFFECTIVE AND ACCELERATED INFORMATIVE FRAME FILTERING IN COLONOSCOPY VIDEOS USING GRAPHIC PROCESSING UNITS

Venkata Praveen Karri, B. Tech.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2010

APPROVED:

Jung Hwan Oh, Major Professor
Bill Buckles, Committee Member
Parthasarathy Guturu, Committee Member
Ian Parberry, Chair of the Department of Computer Science and Engineering
James D. Meernik, Acting Dean of the Robert B. Toulouse School of Graduate Studies

Karri , Venkata Praveen. Effective and Accelerated Informative Frame Filtering in Colonoscopy Videos Using Graphic Processing Units. Master of Science (Computer Engineering), August 2010, 55 pp., 7 tables, 32 figures, references, 22 titles.

Colonoscopy is an endoscopic technique that allows a physician to inspect the mucosa of the human colon. Previous methods and software solutions to detect informative frames in a colonoscopy video (a process called informative frame filtering or IFF) have been hugely ineffective in (1) covering the proper definition of an informative frame in the broadest sense and (2) striking an optimal balance between accuracy and speed of classification in both real-time and non real-time medical procedures.

In my thesis, I propose a more effective method and faster software solutions for IFF which is more effective due to the introduction of a heuristic algorithm (derived from experimental analysis of typical colon features) for classification. It contributed to a 5-10% boost in various performance metrics for IFF. The software modules are faster due to the incorporation of sophisticated parallel-processing oriented coding techniques on modern microprocessors. Two IFF modules were created, one for post-procedure and the other for real-time. Code optimizations through NVIDIA CUDA for GPU processing and/or CPU multi-threading concepts embedded in two significant microprocessor design philosophies (multi-core design and many-core design) resulted a 5-fold acceleration for the post-procedure module and a 40-fold acceleration for the real-time module. Some innovative software modules, which are still in testing phase, have been recently created to exploit the power of multiple GPUs together.

ACKNOWLEDGEMENTS

The writing of a thesis is not possible without the personal and professional involvement of numerous people. Starting with my major professor to friends and family, each and everyone is in some way or the other responsible for encouraging me throughout the journey. I would like to express my profound gratitude and appreciation to Dr. Jung Hwan Oh for providing me the opportunity to work on the most exciting technology fields – healthcare IT – in the market today.

I extend my sincere thanks to my current team-mates at Multimedia Information Group (MIG) – Ruwan Nawaratna and Kumara Jayantha and last but not the least, my ex team-mate and a good friend of mine – Avnish Malik Rajbal, for contributing towards my research through their valuable intellectual inputs. I am also thankful to the National Science Foundation and Mayo Clinic, Rochester, MN for funding the research and providing me with financial support.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Colonoscopy is an endoscopic technique that allows a physician to inspect the mucosa of the human colon. It has contributed to a marked decline in the number of colorectal cancer related deaths [2]. However, recent data suggest that there is a significant (4-12%) miss-rate for the detection of even large polyps and cancers [3, 4]. To address this, I, at the Multimedia Information Group, Dept. of Computer Science and Engineering in collaboration with Endometric Corporation, Ames, IA have been investigating an automated post-procedure quality measurement system by analyzing colonoscopy videos captured during colonoscopy.

Another approach is to inform the endoscopist of possible sub-optimal inspection immediately in order to improve the quality of the actual procedure being performed. To allow immediate feedback, I need to achieve real-time analysis of colonoscopy videos. A fundamental step of quality measurement in either post-procedure or real-time mode is to remove non-informative frames. I call this step informative frame filtering (IFF).

In my thesis, I proposed a new IFF algorithm which is more effective and accurate than previous algorithms. By utilizing the immense power of many-core graphic processing units (GPU) and multi-core central processing units (CPU) design philosophies, I created two IFF modules: one for real-time and the other for post-procedure colonoscopy scenarios.

Code optimizations through NVIDIA CUDA (Compute Unified Device Architecture) for GPU processing and/or CPU multi-threading concepts embedded in the above two design philosophies resulted a 5-fold acceleration for the post-procedure module and a 40-fold acceleration for the real-time module.

CHAPTER 2

BACKGROUND AND CONTRIBUTION

An informative frame in a colonoscopy video can be broadly defined as a frame which is useful for convenient naked-eye analysis of the colon (Figure 2.1). A non-informative frame has the opposite definition (Figure 2.2). The non-informative frames are usually generated due to three main reasons: too-close (or too-far) focus into (from) the mucosa of colon, foreign substances (i.e., stool, cleansing agent, air bubbles, etc.) covering camera lens or rapid movements through the intracolonic space. In general, non-informative frames can be considered out-of-focus frames. My intention is not to increase the sharpness of these frames (these kind of techniques are commonly termed auto-focusing algorithms), but to just filter them out.

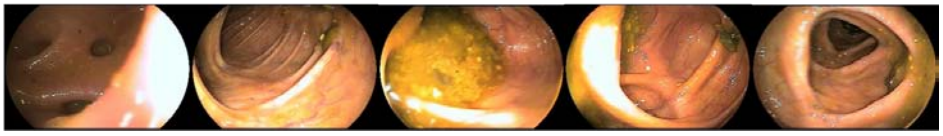


Figure 2.1 Examples of Informative Frames

Figure 2.2 Examples of Non-Informative Frames

Informative and non-informative frames or images can be loosely termed as clear and blurry frames respectively. But, these loose definitions are not sufficient for proper frame filtering in colonoscopy. In a colonoscopy context, my deeper definition of informative and non-informative frames is slightly different from the conventional definitions of clear and blurry

images. I presume that an informative frame in colonoscopy is primarily characterized by curvaceously - circular or semi-circular - connected vivid lines (not just any lines like horizontal or vertical lines or broken lines), because that is the typical content of an informative colon frame (Figure 2.1).

Curvaceous connectivity means more connectivity in diagonal and circular directions. My intention is to retain frames which satisfy this definition and filter out the rest. The best way to completely realize this definition is to first detect the presence of such vivid lines and second measure the amount of curvaceous connectivity they possess. Then with the help of a carefully chosen threshold, identify frames which exhibit more curvaceous connectivity and classify them as informative and vice-versa. Hence, my definition is very different from the rough definition of clear and blurred images which comprise the majority of no-reference blur metric publications. In this thesis, I propose a highly accurate algorithm for IFF (informative frame filtering).

In the real world, colonoscopy primarily has two demands from the computing field. First is to provide automated quality measurement while the procedure is being performed, and the second is to generate a quality report on videos which were already captured and reside on disk. I identified that I need separate real-time and post-procedure software solutions to satisfy these demands, and thus built two separate modules: real-time IFF module and post-procedure IFF module. IFF is only the first step of automated quality measurement. To provide automated quality measurement in real-time for colonoscopy videos which are captured at 30 fps, we have only around 33ms window to process each frame and generate quality metrics.

As already mentioned, there are several steps to generate quality metrics, and if I want to design a good real-time system, I have to make sure all these steps are completed in that 33ms time frame. So, the primary design consideration of real-time IFF module is to consume as much less time as possible below 33ms, leaving the remaining time for other steps to execute. In the post-procedure module, there are a lot of disk-access operations which introduce unnecessary delay in report generation. Half of the total execution time to evaluate colon frames in the post-procedure scenario is consumed for disk-access operations. In this thesis, I propose very fast software solutions for both post-procedure and real-time scenarios by employing CPU (central processing unit) multi-threading and/or GPU (graphic processing unit) co-processing using NVIDIA CUDA (Compute Unified Device Architecture). The main contributions of this thesis can be summarized as follows:

- Several misconceptions are there about the exact definitions of informative and non-informative colon frames. They are often confused with conventional sharp and blurry images. I introduce a new and better definition of an informative colon frame which explains its meaning more clearly
- The previous edge-based algorithm is very inaccurate due to lack of consideration of the deeper meaning of informative colon frames. I propose a new edge-based algorithm with higher accuracy compared to the old one
- IFF is the first among many other steps in automated colonoscopy quality measurement. Based on my new algorithm, I propose a software solution using GPU to evaluate frame quality under real-time constraints

- Disk read operations consume almost half of the total execution time in post-procedure IFF modules. I propose a technique to read and process multiple frames simultaneously using a combination of CPU threads and GPU threads

CHAPTER 3

RELATED WORK

Several techniques have been published on blur detection [12-16]. One is related to my application [12] and has been tested by us, but the results were not satisfactory. According to [12], I find the difference between intensities of adjacent pixels, calculate the variance of all the differences, compute block-wise sum of variances, check for blocks with higher variance, then classify an image with more such blocks as sharp and the rest as blurred. This method failed to suit my needs because:

- Colon frames look alike; almost all of them belong to same color space (brownish red) and relying solely on color intensity difference of adjacent pixels can easily lead to misclassification
- The method in [12] is more applicable to images which stick to conventional definitions of sharp and blurry images; but the definition of informative and non-informative frames is slightly different as mentioned in Chapter 2

The poor accuracy experimental results of this method (Table 9.4), once again prove that the definitions of informative/non-informative colon frames and conventional sharp/blurred images are not synonymous.

The previous informative frame filtering (IFF) method [5] was based on the detected edges within the frame, and I used canny edge detector [17] for generating edge maps. Then the edge map is post-processed, to distinguish the non-informative frames from the

informative ones. In the edge map post-processing, two terms, isolated pixel (IP) and isolated pixel ratio (IPR) were defined for a frame.

An IP is an isolated edge pixel (edge pixel that is not connected to any other edge pixels) in a frame. IPR is computed as a percentage of the number of isolated edge pixels to the total number of edge pixels in the frame. A frame with IPR less than a certain threshold is classified informative and vice versa. Since it was believed according to [5] that non-informative frames would have more isolated pixels, high-sensitive canny parameters were used to expose every detail of the original frame in its edge map. This method is not effective enough because:

- Canny edge detector mostly produced a lot of closed contours irrespective of sensitivity parameters due to its default design; thus it rarely reproduced isolated pixels in the edge map
- Despite low-sensitive edge detection parameters, IPR logic resulted in misclassification of frames because it did not consider the intrinsic meaning of an informative frame

In both Figure 3.1 and Figure 3.2, the IPR of informative frame is more than the non-informative frame, thus resulting in misclassification according to the definition of IPR. I still agree that edge detection is one of the most suitable ways to expose the connectivity present in a frame, but I strongly believe that the choice of edge detector, its sensitivity parameters and method used to post-process the edge map are huge factors and should be carefully chosen to achieve effective filtering. The previous algorithm went wrong in properly considering all these factors.

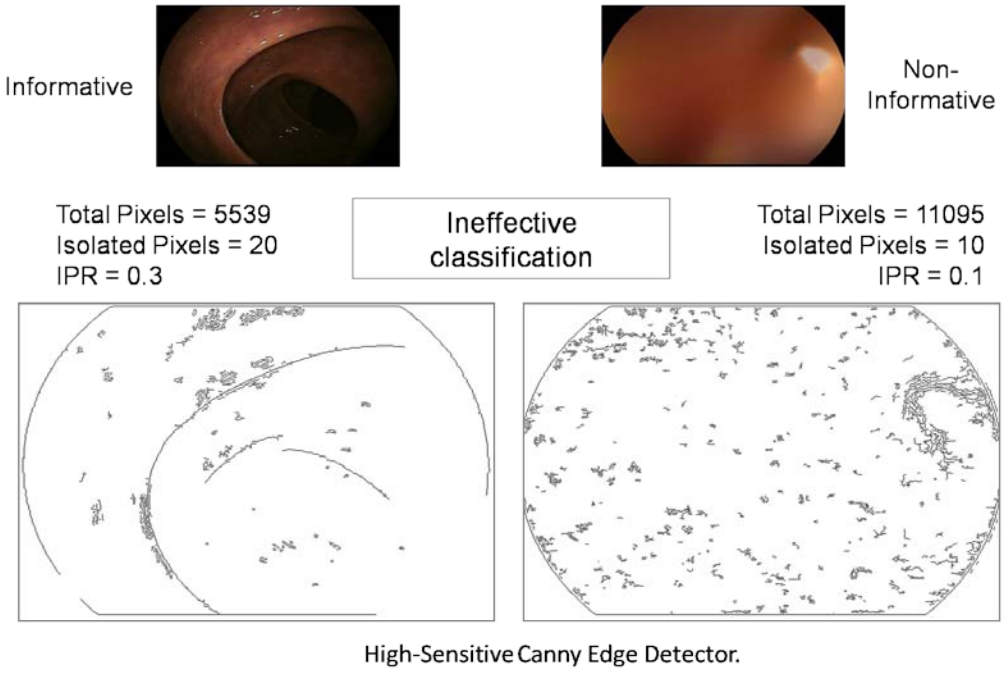


Figure 3.1 Failure of Previous Algorithm with High-Sensitive Canny Edge Detector

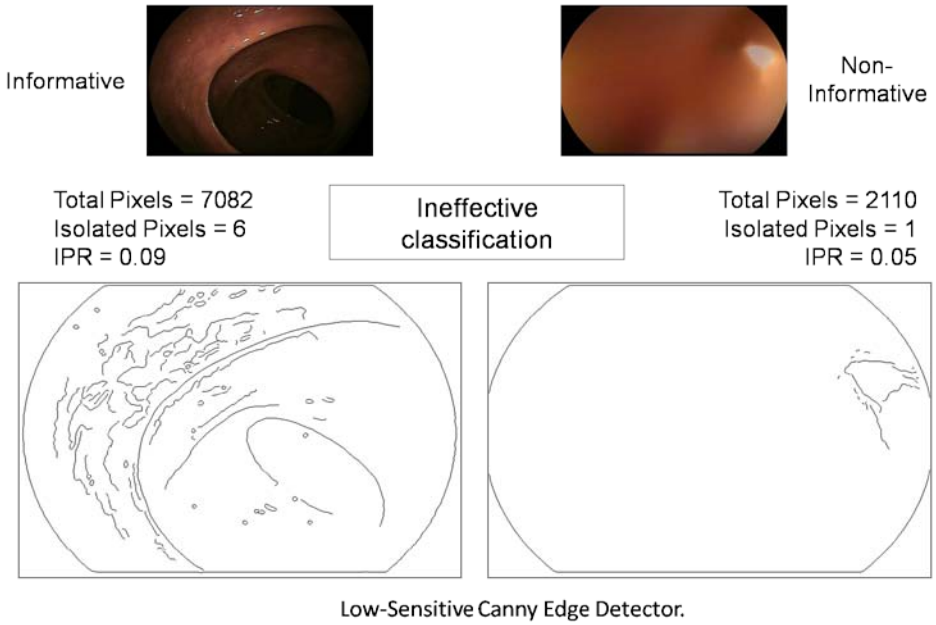


Figure 3.2 Failure of Previous Algorithm with Low-Sensitive Canny Edge Detector

CHAPTER 4

REQUIREMENTS FOR EFFECTIVE INFORMATIVE FRAME FILTERING

As explained in the previous chapters, an informative frame in colonoscopy is primarily characterized by curvaceously – circular or semi-circular – connected vivid lines because that is the typical content of an informative colon frame; and curvaceous connectivity means more connectivity in diagonal and circular directions. The best ways to realize the above explanation are in the following order:

- Find the edges of the frames such that they reflect information as much as possible
- Measure the amount of curvaceous connectivity amongst the pixels of the obtained edge map
- Quantify the portion of frame which is connected to make a decision

I call this edge-detection cum connectivity scheme. A more appropriate and new definition of an informative colon frame would be:

A frame which has vivid lines in its edge map with sufficient amount of curvaceous connectivity amongst these lines (in other words more connectivity in diagonal directions) is defined as an informative frame.

My goal is to retain frames which satisfy this new definition and filter out the rest. I lay two major requirements for effective informative frame filtering (IFF) which can be separated into two phases:

- Phase I: Generate a Polarized Edge Map

  Choose an edge detector with optimal thresholds, such that its edge map:

  - Leaves a small hint of connected edge pixels in the edge map if there is any real information
  - Polarizes Informative and Non-informative frames as much as possible, so that post-processing the edge map for subsequent classification becomes much easier

- Phase II: Estimate the Amount of Connectivity in the Edge Map

  Design a method to post-process the edge map, such that it calculates the amount of connectivity possessed by the edge map, and also quantify the portion of edge map which is sufficiently connected.

4.1 Phase I: Polarization in IFF with Edge Detectors

It has been discussed that a polarized edge map needs to be generated which leaves a small hint of connected edge pixels in the edge map if there is any real information and which polarizes informative and non-informative frames as much as possible, so that post-processing the edge map for subsequent classification (in Phase II) becomes much easier. A few examples of good and bad polarization can be discussed. Generally, conventional edge detectors with low sensitivity parameters do the job of classification without any additional efforts for those colon frames which lie on the extremes i.e., extremely informative or extremely non-informative (Figure 4.1).

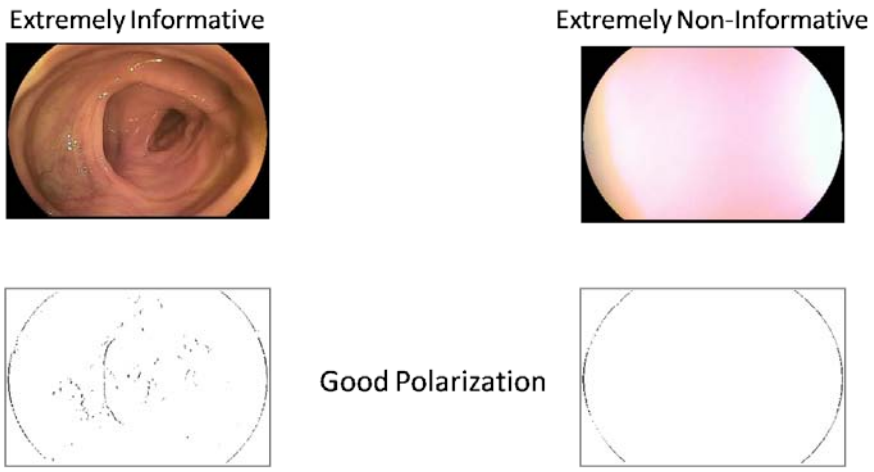


Figure 4.1 Example of Good Polarization

However, ineffective polarization may occur when frames do not belong to either extreme, like the ambiguous ones in Figure 4.2. Such frames comprise more than 50% of a typical colonoscopy video. Another problem might be incorrect polarization, where the edge detector does make a polarization but makes it in places where it is not necessary (Figure 4.3).

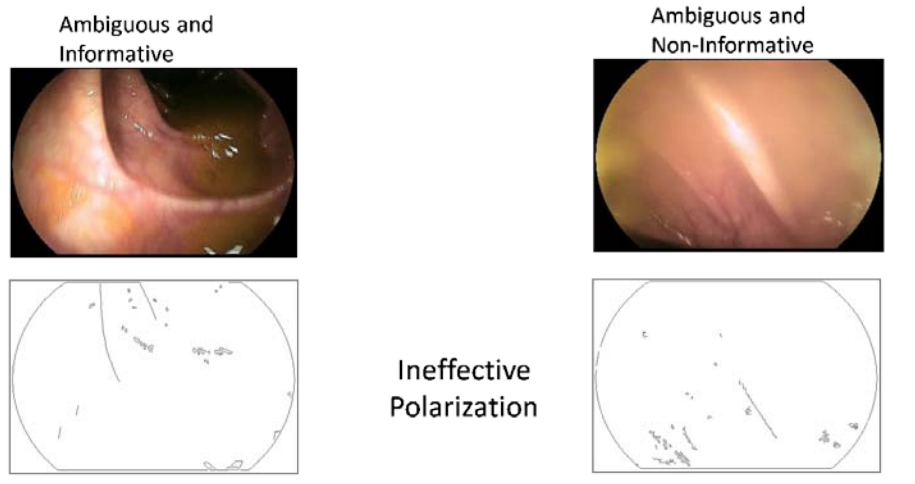


Figure 4.2 Example of Ineffective Polarization

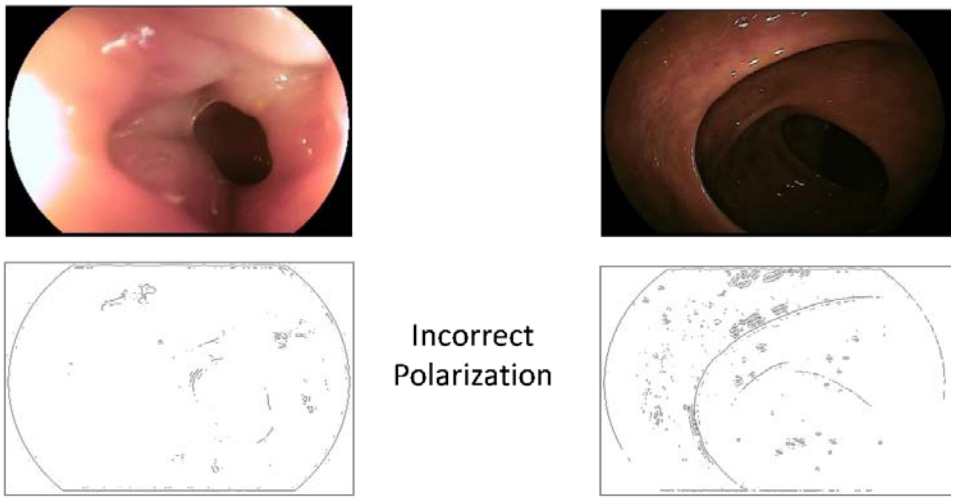


Figure 4.3 Example of Incorrect Polarization

Finally, increasing the sensitivity of edge detector even slightly might lead to noise which totally leaves us with no scope for polarization (Figure 4.4).

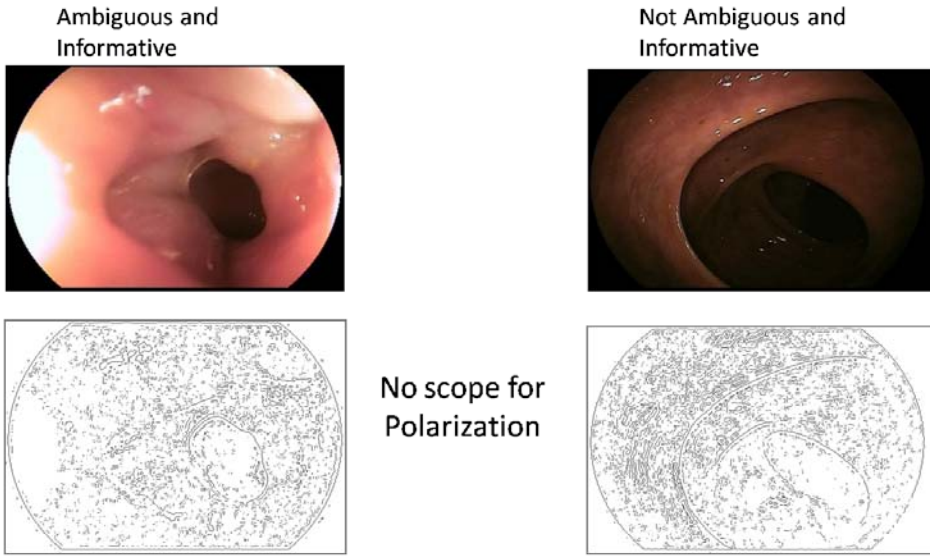


Figure 4.4 Example Showing No Scope for Polarization

So, from this analysis, I observed the following:

- Ability to polarize a frame in its edge map is an important factor in post-processing the edge map to make the subsequent decision as informative or non-informative
- Using a high-sensitive edge detector totally rules out the possibility of polarizing frames in the Phase I of IFF
- Using low-sensitive parameters for edge detection produces an edge map with the display of only minimal features of the colon, but at least leaves us with a chance to polarize the frames
- Although low-sensitive parameters are considered, the type of edge detector used still determines whether the polarization is proper, incorrect or ineffective

And, I can infer the following:

- Choose an edge detector which polarizes effectively
- There is no point in using an edge detector which reproduces connected edge maps irrespective of presence of connectivity in the original frame. (e.g. closed contours of canny)

4.2 Phase II: Concept of Connectivity in an Edge Map

Let us recall the definition of an informative frame once: An informative frame in colonoscopy is primarily characterized by curvaceous connectivity (meaning more connectivity in diagonal directions) because that is the typical content of an informative colon frame.

From the above definition, the operations of Phase II would be

- Stage 1: Calculate the amount of connectivity
- Stage 2: Quantify the portion of edge map which is connected

4.2.1 Connectivity Map Generation: Phase II-Stage 1

From the above definition, colon images are considered more informative or connected if there are more diagonal connections in their edge map. So, if I give the same weight to both adjacent and diagonal connections in an edge map, it will not properly quantify the amount of diagonal connectivity a block possesses during Phase II-Stage 2 of the IFF algorithm requirements. Hence, I give diagonal connection twice the weight of an adjacent connection. This is the first requirement of connectivity map generation.

The connectivity of a pixel (say 'q') in an edge map is calculated based on its connection to only four of its neighboring pixels (immediate right, immediate bottom, immediate diagonal left, immediate diagonal right; these are shown with red arrows in Figure 4.5(b)). This is done to avoid redundancy (redundancy is marked with light green arrows in Figure 4.5(b)) in the calculation of cumulative connectivity when traversing from top left to bottom right corner of the image. This is the second requirement. To satisfy both these requirements an optimal non-redundant connectivity mask is designed (Figure 4.5(a)).

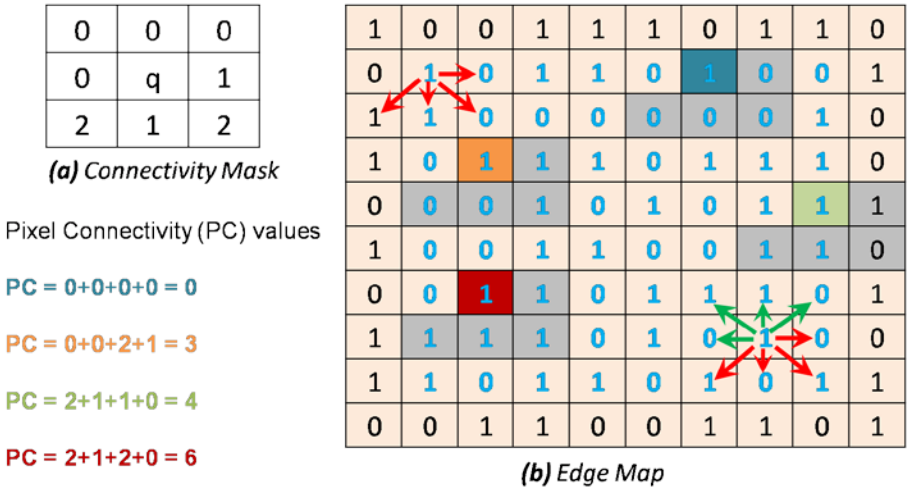


Figure 4.5 Concept of Connectivity

From Figure 4.5, it is clear that the pixel connectivity values range from 0 to 6. In other words $0 \leq PC \leq 6$. An example of how a connectivity map looks like is shown in Figure 4.6.

2	5	6	0	2	1	0	0
2	2	4	3	1	4	5	1
0	4	1	4	3	6	0	2
6	3	2	1	0	5	4	2
1	0	3	4	6	2	1	6
0	3	4	6	2	4	5	1
6	5	3	2	0	1	4	5
5	2	0	1	4	3	1	4

$0 \leq PC \leq 6$

Figure 4.6 Connectivity Map Illustration

4.2.2 Quantify the Portion of Edge Map which is Connected: Phase II-Stage 2

In order to quantify information, the connectivity map is divided into even number of blocks of size (say 'm'), and checked for the number of blocks which have sufficient connectivity (or information). The block size affects the final decision. An example of how different block sizes look like is shown in Figure 4.7.

Each block holds the square root of sum of individual values in a connectivity map. It is called the block connectivity (BC). Based on a threshold for BC a block is decided as informative. Based on a threshold for the number of informative blocks, a frame is decided as informative.

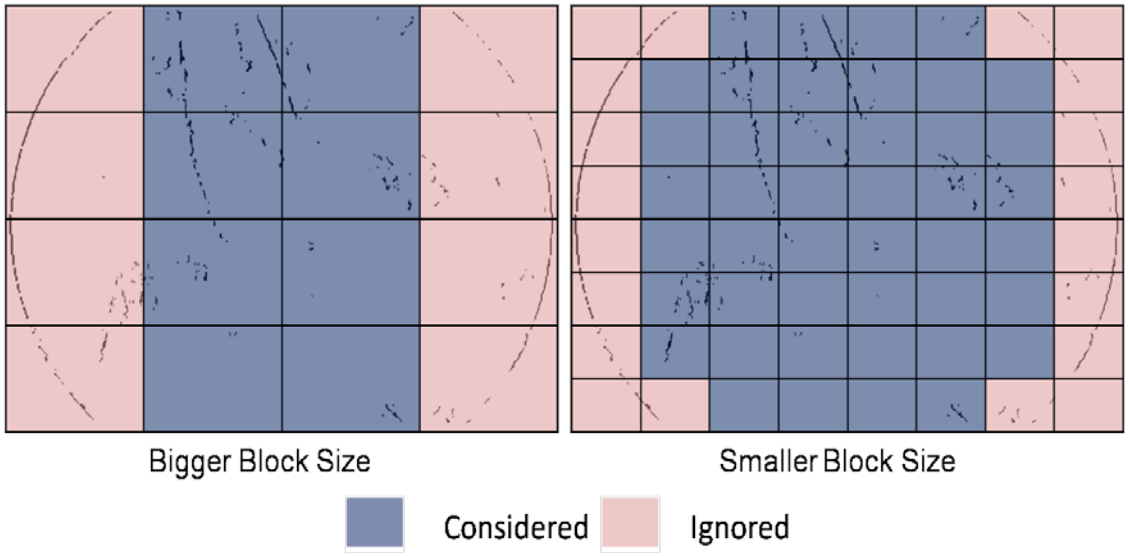


Figure 4.7 Examples of Different Block Sizes to Quantify an Edge Map

A block connectivity graph (Figure 4.8) helps select suitable values for:

- block size, m
- block connectivity (BC) threshold to decide a block as informative
- Number of informative blocks threshold to decide a frame as informative

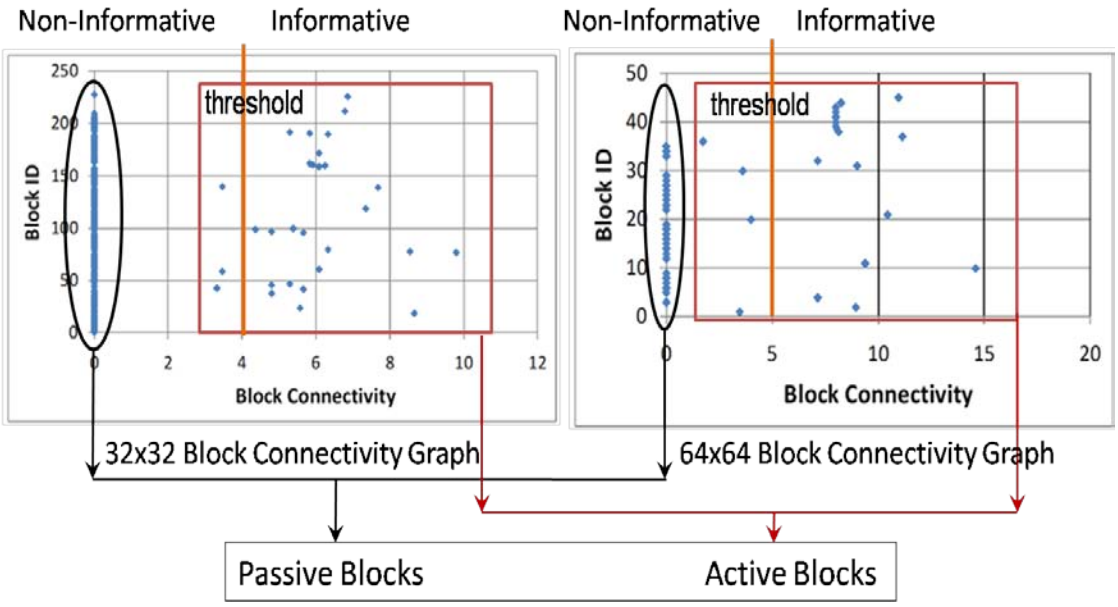


Figure 4.8 Example of Block Connectivity Graph for Different Block Sizes

To make things much better, Figure 4.8 reveals some interesting and important details like:

- Passive blocks (BC=0) mostly dominate active blocks (BC>0) if edge map reproduces only minimal features of a colon frame
- All passive blocks are non-informative blocks, but vice-versa need not be true
- All informative blocks are active blocks, but vice-versa need not be true

CHAPTER 5

PROPOSED ALGORITHM AND COMPUTATIONAL COSTS

The new algorithm is designed based on the requirements mentioned in the previous chapter. While designing the algorithm, several experiments have been conducted (Tables 9.1, 9.2 and 9.3) to choose the right parameters in each phase. A block diagram of the flow of the new algorithm is shown in Figure 5.1.

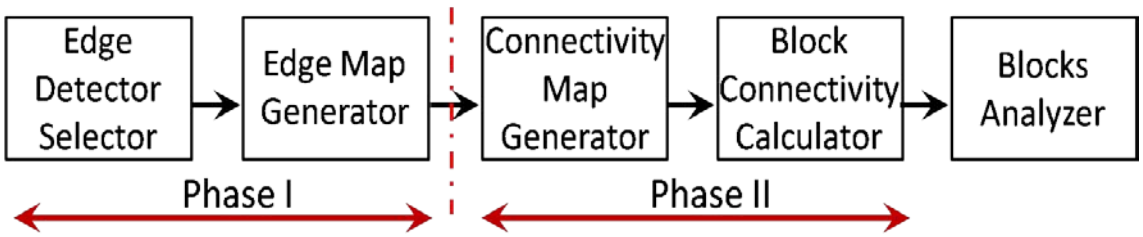


Figure 5.1 Block Diagram of the New Algorithm

5.1 Phase I: Generate Edge Map

To satisfy the first requirement, I need an edge detector which can just leave us with a hint of connected edge pixels in the edge map if there is any real information. Otherwise, it should not output such edges at all. In other words, I need a very low sensitive edge detector such that it can polarize information/connectivity and non-information/non-connectivity. Good polarization is dependent on both sensitivity and type of edge detector chosen to generate the edge map. So, the parameters that can affect Phase-I are:

- Type of edge detector
- Sensitivity of edge detector

Figure 5.2 (below) proves that using either a low-sensitive sobel edge detector or log edge detector improves the chances to get a correct final decision.

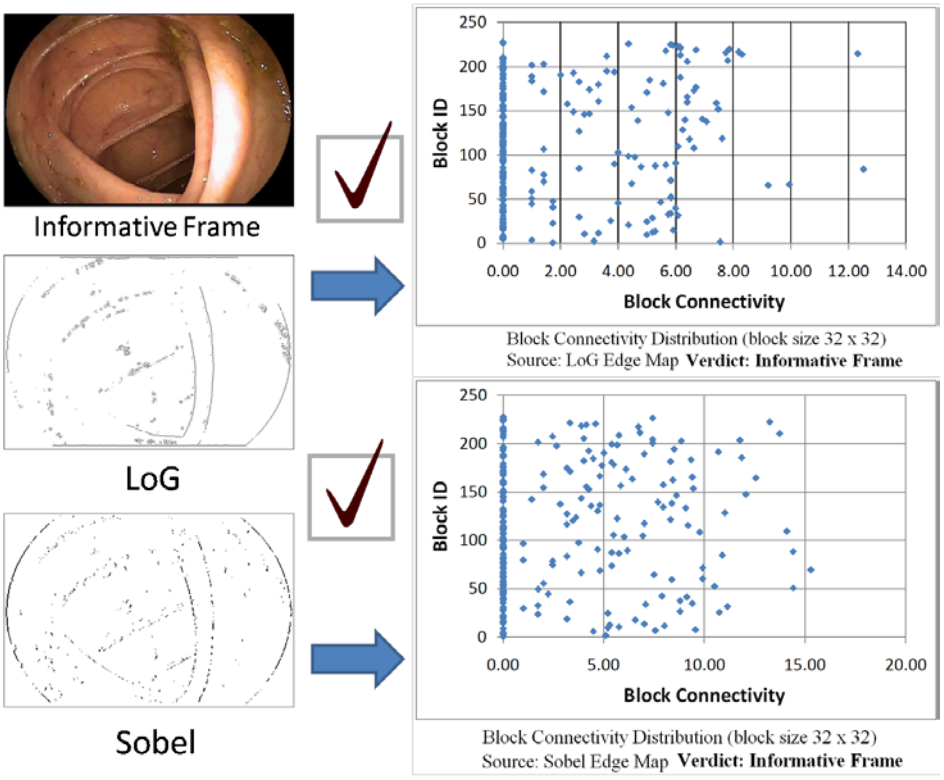


Figure 5.2 Effects of Phase-I Parameters on Extremely Informative Frame

In Figure 5.3 an ambiguous frame is shown which is more inclined towards being an informative frame. The low-sensitive Laplacian of Gaussian (LoG) edge detector leaves every possible chance to make the wrong decision here because of very less number of active blocks in the block connectivity graph. However, the active blocks are slightly more in the block connectivity graph from sobel edge map. So, the chances of classifying the frame as informative are more here, which is what I want.

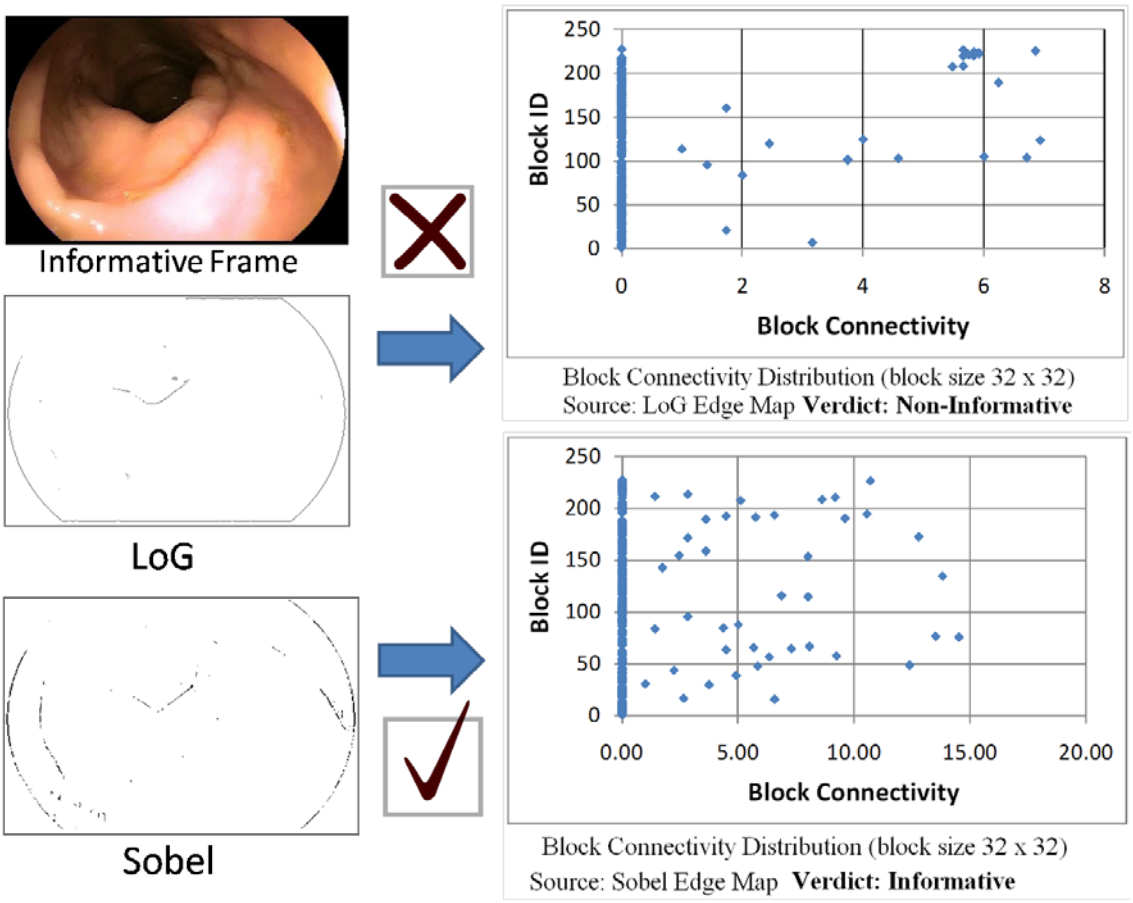


Figure 5.3 Effects of Phase-I Parameters on Ambiguously Informative Frame

Another example is shown in Figure 5.4 where the input frame is ambiguous but more inclined to being non-informative. The block connectivity graph from canny edge map gives more chances to classify the frame as informative because of more active blocks. But, this is not the case with sobel. Hence I can see that sobel is a better choice than other edge detectors.

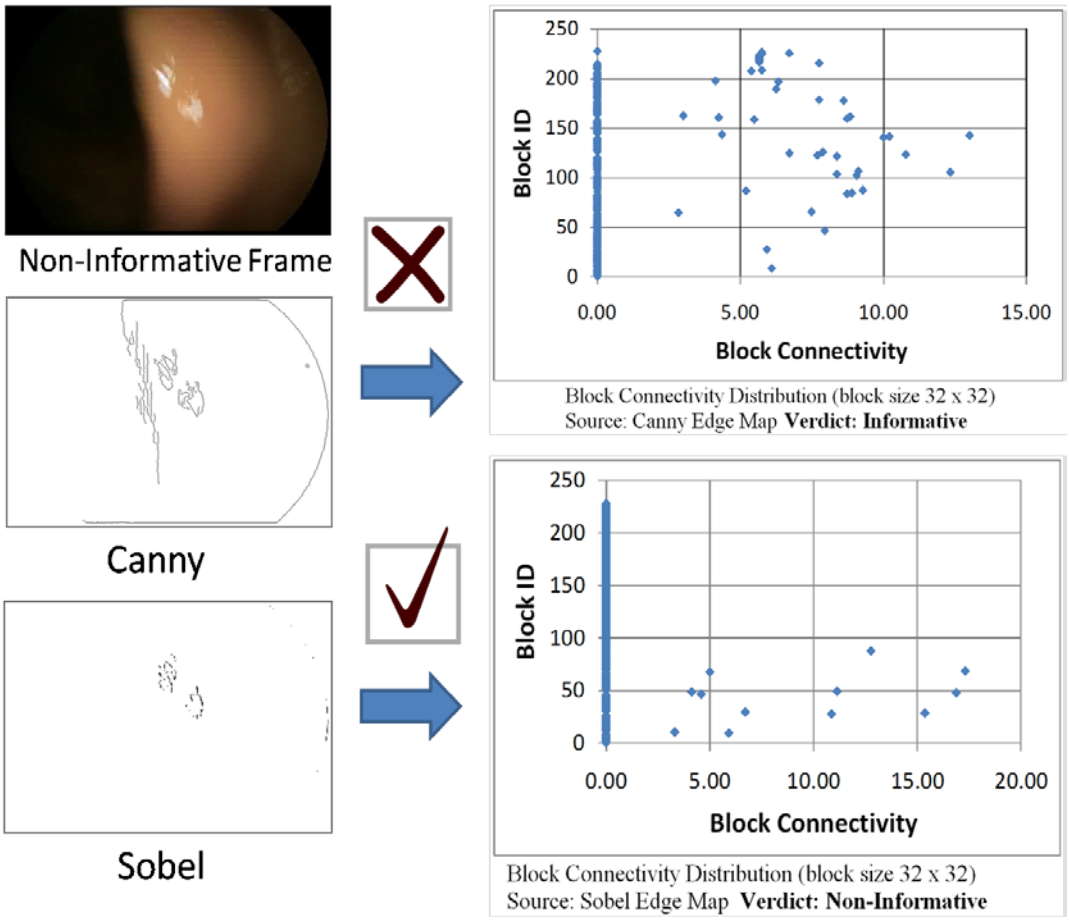


Figure 5.4 Effects of Phase-I Parameters on Ambiguously Non-Informative Frame

The edge map is represented by e(q) such that, e(q) = 0, if it is not an edge pixel and e(q) = 1, if it is an edge pixel.

5.2 Phase II: Generate Connectivity Map to Estimate the Amount of Connectivity

In an 8-connected neighborhood, the connectivity at pixel 'q' by applying the connectivity mask (Figure 4.5(a)), is given by,

$$C_{x,y} = (z_6+2z_7+z_8+2z_9) * e(q). \quad (1)$$

To quantify information, the connectivity map is divided into even number of blocks, and checked for the number of blocks which have sufficient connectivity (or information). The

image is resized to (M_r, N_r), such that the height and width are multiples of block size *m*. So, with a block size *m* x *m* pixels, the total number of blocks will be, $\mu = (M_r \times N_r)/(m \times m)$. The total connectivity of a block, B_i, is given by,

$$B_i = \sqrt{\sum_{x,y=0}^{m,m} C_{x,y}} \text{ , where } i = 1, 2, 3 \ldots \mu. \quad (2)$$

If '€' is defined as the block connectivity threshold, then a block is considered as non-informative if $B_i \leq €$. If a connectivity map has β number of non-informative blocks, I define α as the ratio of non-informative blocks over total blocks, and ϕ as a threshold for non-informative block ratio, then $\alpha = \beta/\mu$; a frame is considered non-informative if $\alpha \geq \phi$ and vice versa.

Figure 5.5 shows how phase II parameters can affect the decision. Several experimental analyses have been conducted to choose the optimal values for Phase-I and Phase-II parameters used in this algorithm (Tables 9.1, 9.2, 9.3). The following are the chosen values: sobel edge detector with threshold 0.33, m = 64, € = 5, ϕ = 0.75. Also the comparison of performance metrics of the new algorithm with others (mentioned in Chapter 3) is shown in Table 9.4.

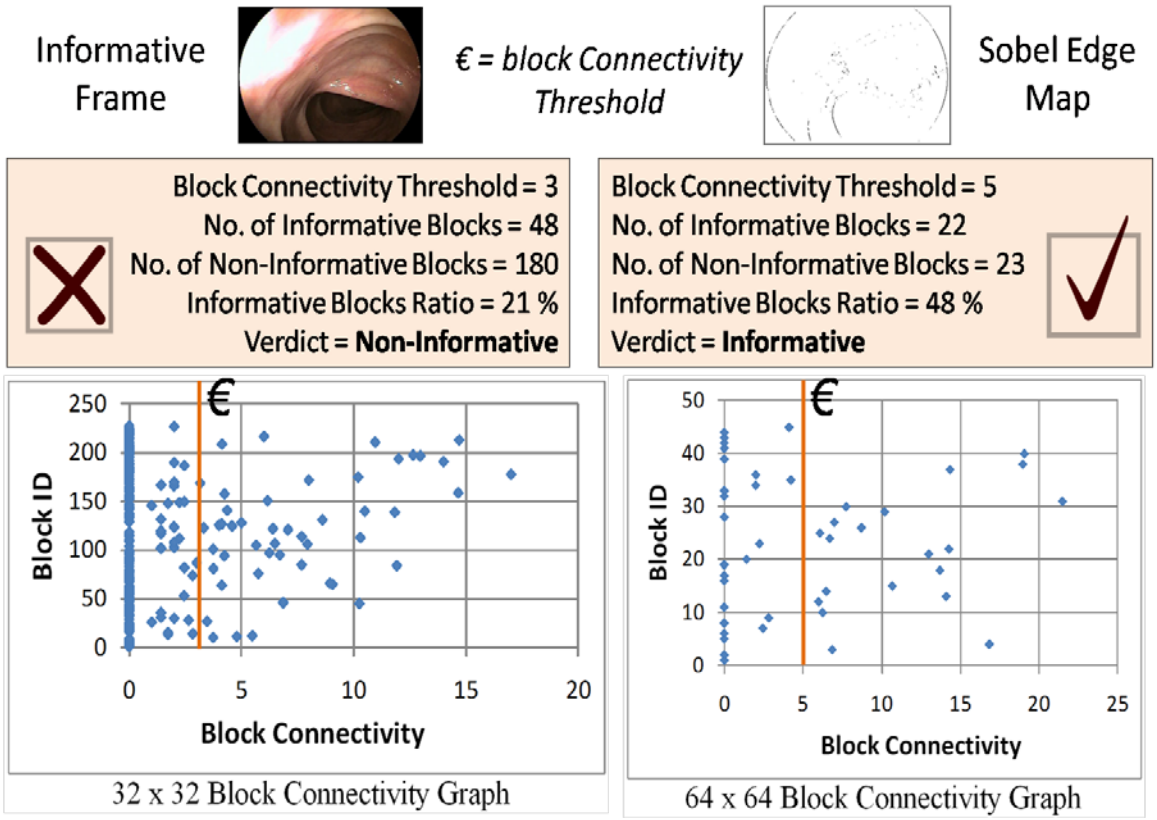


Figure 5.5 Effects of Phase-II Parameters on Informative Frame

5.3 Computational Costs

To identify and tackle the computational costs more easily in Chapter 7, the block diagram has three stages of computation and shows a combination of real-time and post-procedure versions of IFF. From Figure 5.6, the computational cost of IFF algorithm (bottleneck #1) can be 15(Mr x Nr) since Stage 1 has 13(Mr x Nr), Stage 2 has (Mr x Nr), and Stage 3 has (Mr x Nr), which are all numerically intensive sequential iterations. This is a bottleneck for both post-procedure and real-time modules. A modern day GPU like the NVIDIA® GeForce® GTX 280 GPU has 30 Streaming Multi-processors (SMs) with each SM having the ability to handle 1024 threads [18]. I mitigate bottleneck #1 by using CUDA on GTX 280.

The post-procedure version has an overhead operation of reading the frame from disk and storing raw RGB image data in memory, which is not present in the real-time version where input is directly fed from a capture card. I call this disk-read overhead cost (bottleneck #2). A modern day Intel CPU has up to 8 cores; each core having the ability to handle up to 4 threads – making it a total of 32 threads [1]. I mitigate bottleneck #2 by using multithreading on Intel Quad Core @ 3.0GHz.

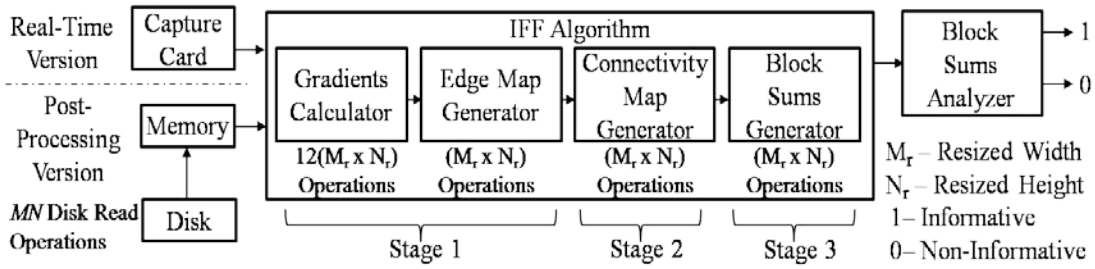


Figure 5.6 Computational Costs in Different IFF Modules

CHAPTER 6

CUDA-BASED COPROCESSOR – GTX 280

I used the EVGA 01G-P3-1280-RX GeForce GTX 280 1GB 512-bit GDDR3 PCI Express 2.0 x16 HDCP Ready SLI Support Video Card for this thesis. This graphics card has 1 GB global memory and 256 KB L1 texture cache. From hardware standpoint, the card is viewed as a combination of 10 Texture/Thread Processing Clusters (TPCs). Each TPC holds 24 KB L2 texture cache and evenly distributes it across three Streaming Multi-processors (SMs). Each SM has 8 scalar processors (SPs), 16 KB of shared memory and a 32 KB register file, which is evenly partitioned amongst resident threads when the device is used for computing.

From programming standpoint, I use the NVIDIA® CUDA™ programming model to run this device in compute mode with CUDA Compute Capability 1.2. CUDA (Compute Unified Device Architecture) views the device as a pool of threads and calls it a Grid. It organizes threads into blocks and delivers them to the SM Controller, which distributes them across the SMs, each restricted with a 1024 thread and 8 blocks limit in a single run and responsible for even distribution of blocks across its 8 SPs.

The unit of parallelism in CUDA (i.e., CUDA threads) is an entity with very small creation cost, resource usage, and switching overhead compared to CPU threads. During its lifetime a thread can access any location in the global memory or texture cache, and is given access to the shared memory space of the SM in which it is running, and also given its own set of registers to use. CUDA threads run data-parallel functions called kernels. After grouping threads into blocks and launching on an SM, they are executed in an SIMT (single instruction multiple threads) fashion within individual blocks, commonly termed as 'warp execution'. The smallest unit of a

warp execution is 32 threads. Figure 6.1 shows a hardware and software point of view of GPU and CUDA.

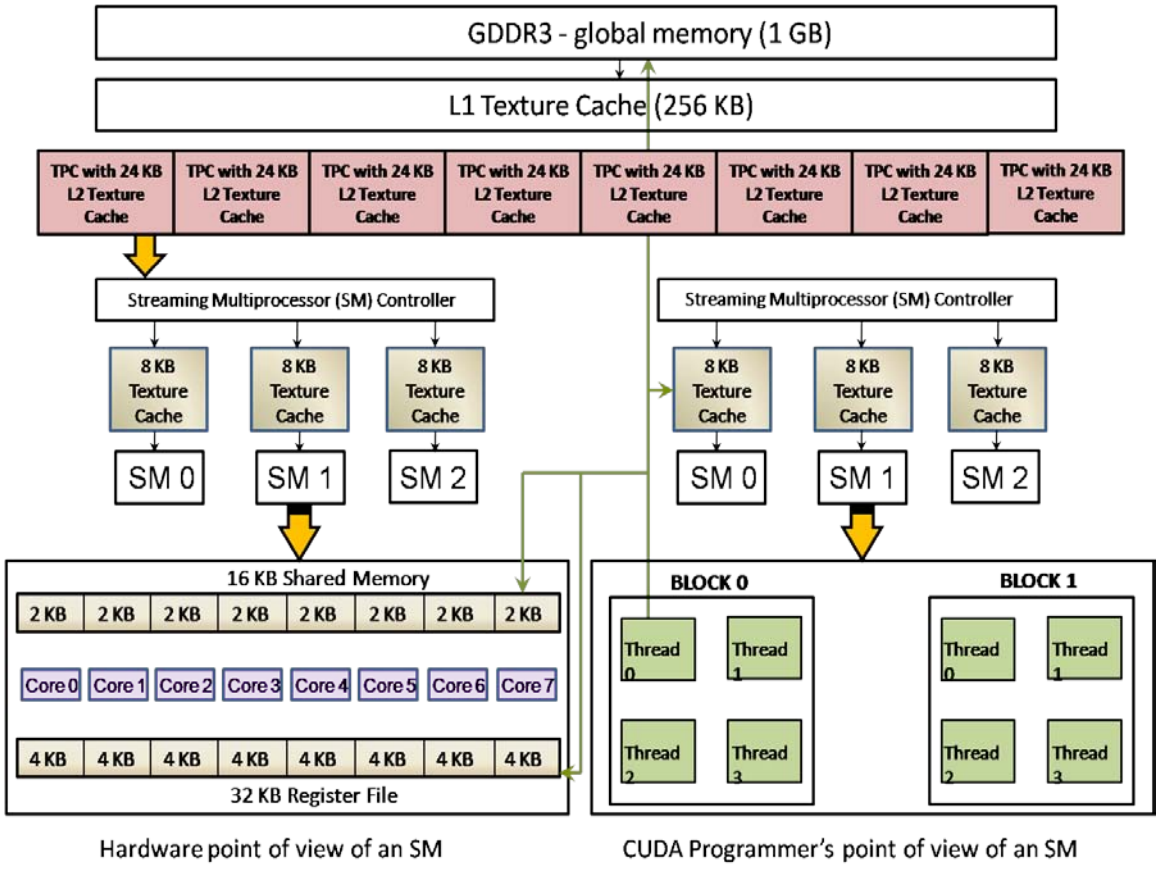


Figure 6.1 Hardware and Software Point of Views of GPU and CUDA

The ultimate goal of an application developer is to obtain best performance from the NVIDIA® CUDA™ architecture using the CUDA Toolkit [11]. Performance is improved by hiding latency during memory accesses, ensuring nominal instruction executions and maximizing memory bandwidth. Instruction Throughput is a performance measure for a compute-bound kernel because the bottlenecks of such a kernel include instruction overhead and latency. Memory Bandwidth Utilization is a performance measure for a memory-bound kernel because effective bandwidth utilization of memory improves speed in such kernels [19].

Throughout the GPU algorithm the number of CUDA threads in x-dimension and y-dimension – TX and TY are chosen 16 and 8 making it 128 threads per block (block size) and 8 blocks per SM. The usage of global memory is limited to only those kernels in which long latency operations like accessing memory with a stride or multiple threads accessing same location are not present.

The shared memory and the usage of register file are controlled. This has ensured 100% device occupancy for all kernels, thus laying a strong platform for further kernel optimizations, for individual IFF algorithm stages. The variables M_r, N_r and *m* introduced in Chapter 5 are designated FRAMEW, FRAMEH and IB_SIZE in the computing platform.

CHAPTER 7

GPU IFF ALGORITHM: HANDLING BOTTLENECK #1

7.1 Stages 1 and 2: Edge Map and Connectivity Map Generation

In both stage 1 and 2 at every step the operation is performed on an image, and the result is an image itself. For example, in stage 1, the input is a grayscale image and the output is an edge map. The operator is a sobel mask. In stage 2, the input is an edge map, the result is a connectivity map and the operator is a connectivity mask. So, in both stages, the intermediate or final result is always another image. The evaluation is done pixel-wise by applying a 3x3 convolution mask in each stage. Which means to evaluate a particular pixel; I need the neighboring pixel data.

Figure 7.1 shows how the pixels in a 3x3 matrix are stored in the global memory. So, when I need to find the gradient at a pixel, I need the values of neighbors which are at least one pixel and at most 4 pixels away from the current pixel on which the operation is performed. In such operations where neighboring pixels that lie at a certain distance from are main pixel, threads need to access with an offset. This operation when done on a global memory is called non coalesced access [19, 21]. Global memory is very slow, so if I start accessing neighboring pixels which lie far away from the reference pixel, that will take a very long time to fetch.

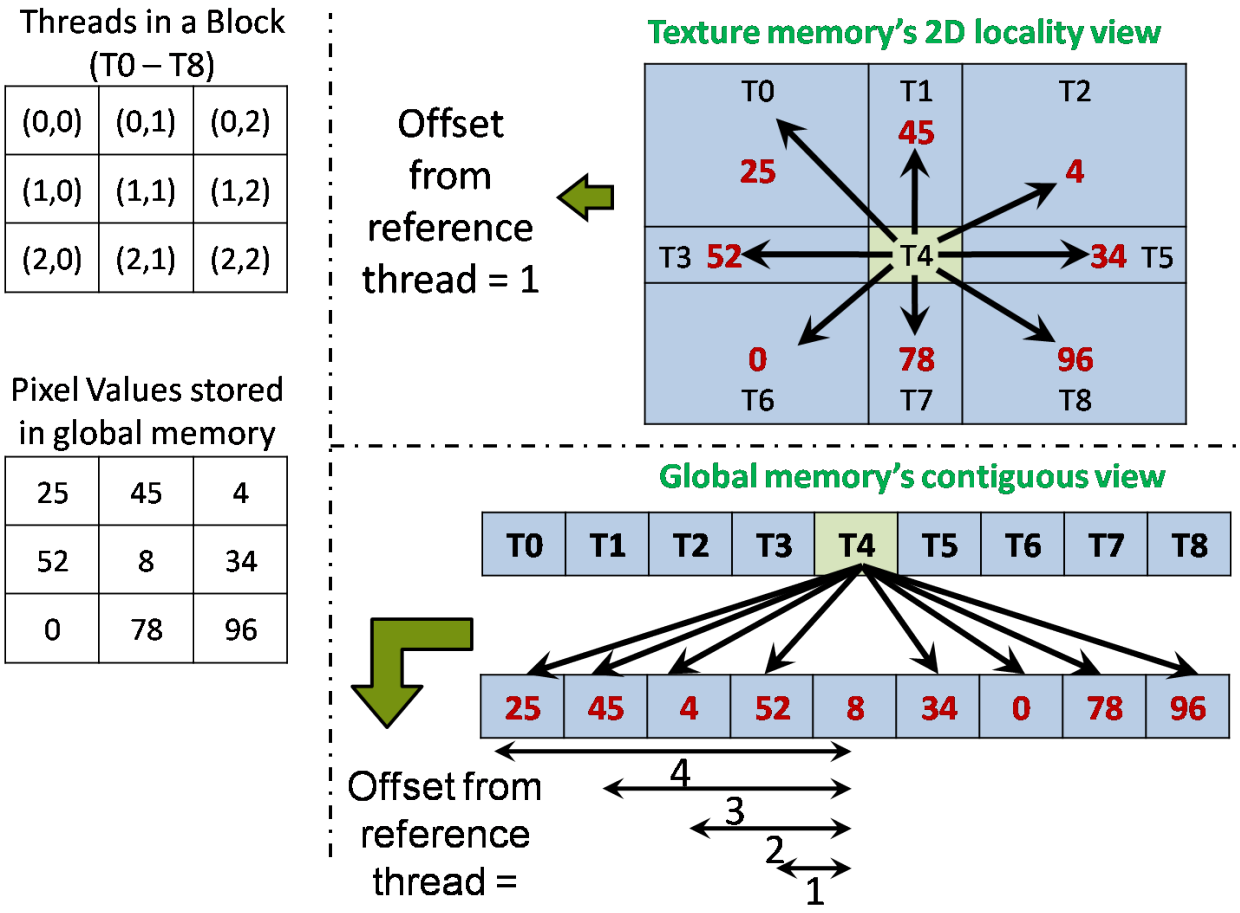


Figure 7.1 Global and Texture Memory Views of a 3 X 3 Matrix of Pixels

Such operations in CUDA (Compute Unified Device Architecture) would enjoy the benefit of high reference of locality in terms of memory access if a memory with better 2D locality fetching is used. This is where texture cache comes into play. Texture cache is a great way of boosting performance in case of 2D locality fetching. GPUs (Graphic Processing Units) are able to exploit texture memory for high performance in such operations in which an offset access (means accessing neighboring pixels) is required. Figure 7.2 shows the performance of memories with respect to offset (also called 'shift').

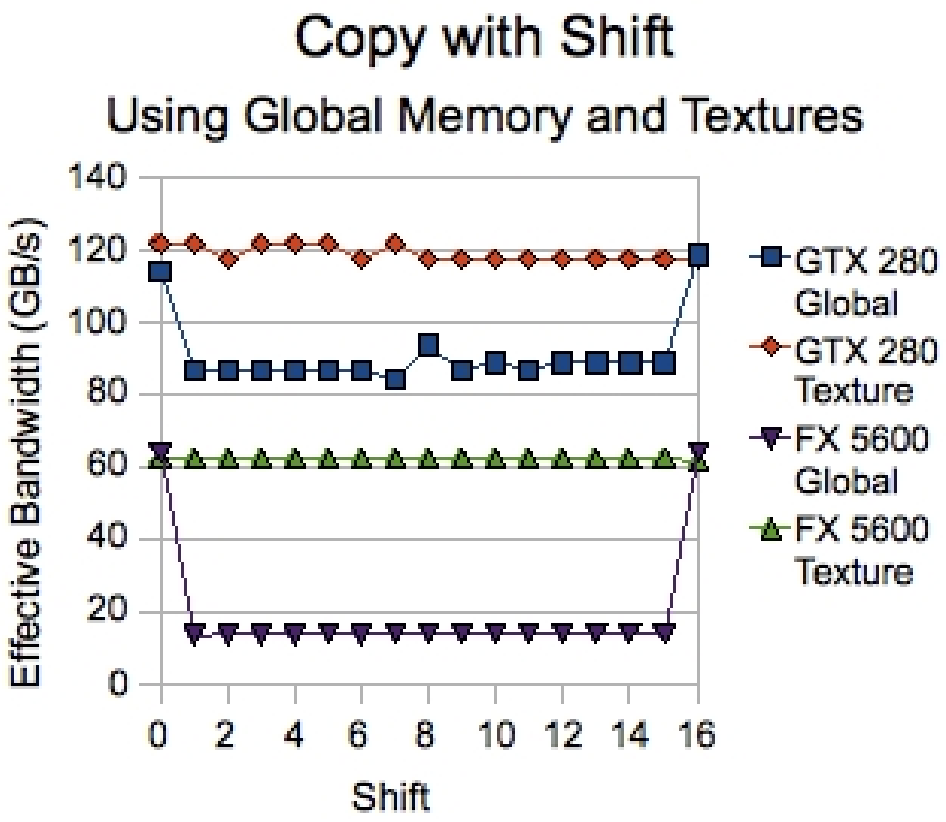


Figure 7.2 Results of Using Texture Memory to Avoid Non Coalesced Global Memory Access (from [19])

Moreover, from Figure 7.3, I can understand that texture cache is relatively faster than global memory access. This shows us that texture cache provides a greater performance when there is an offset copy. Convolution operations are an example where a particular pixel evaluation requires neighboring pixel data. In other words threads have an offset when accessing memory.

Texture memory by default is faster than global memory. So, in convolution operations which obviously involve offset access, texture memory (commonly called texture cache) is preferred to global memory. Another advantage of using texture memory is that the hardware provides automatic handling of boundary cases of the image. When the referenced coordinate

of texture is out of valid range, the hardware clamps the coordinate to the valid range or wraps to the valid range depending on the addressing mode set.

Memory	Location On/Off Chip	Cached	Access Type	Access Speed	Scope	Lifetime
Register	On	n/a	R/W	Fast	1 thread	Thread
Local	Off	No	R/W	Slow	1 thread	Thread
Shared	On	n/a	R/W	Fast	All threads in a Block	Block
Texture	Off	Yes	R	Fast (limited)	All threads + host	Host allocation
Global	Off	No	R/W	Slow	All threads + host	Host allocation

Figure 7.3 Comparisons of GPU Memories

Shared memory is another option to perform convolution, but it is useful only when the offset is large enough. Readers are encouraged to go through the concepts of shared memory in [7, 20] to understand more. I used separable convolution filters for sobel mask calculation on CUDA and normal convolution for connectivity mask calculations.

Figure 7.4 shows how shared memory can improve the performance of CUDA convolution separable kernels in comparison to texture cache with respect to mask radius. In both stages 1 and 2, the consecutive operations are performed on images with mask radius of one. Stage 1 has sobel mask with radius one and Stage 2 has connectivity mask. For a mask

radius as low as one, I need not seek help of shared memory for performance boost, as shown in Figure 7.4. Hence, I opted for texture memory for separable convolution in Stage 1 and for normal convolution in Stage 2.

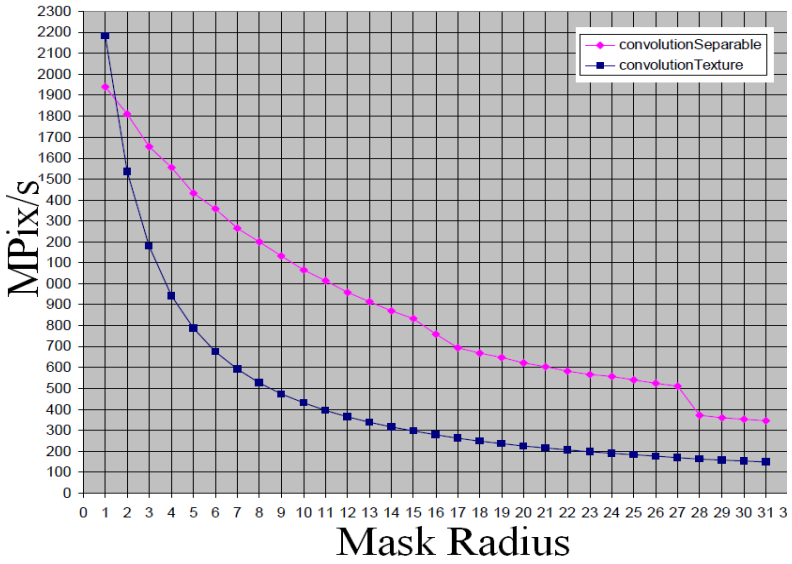
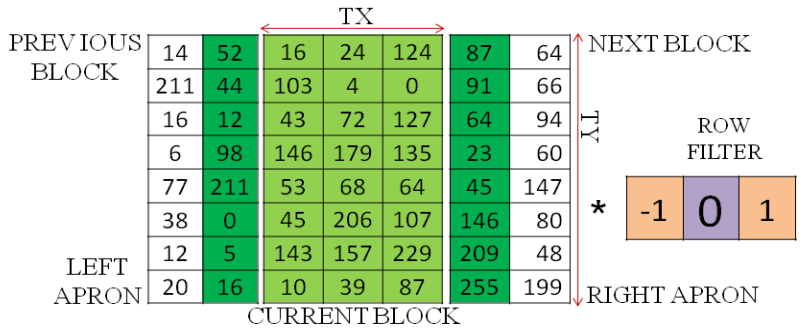


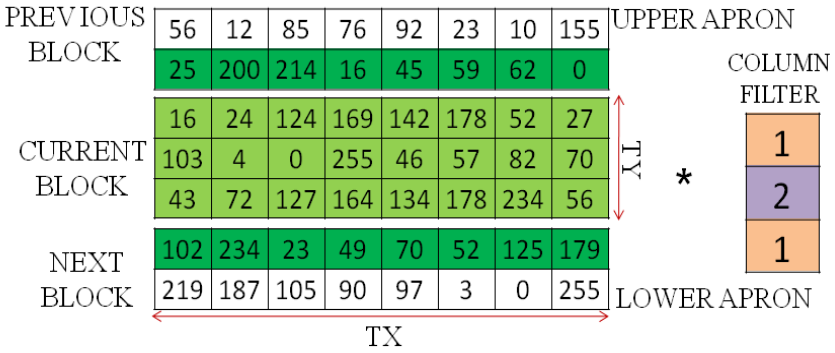
Figure 7.4 Performance of Shared and Texture Memory for Separable Convolution

The calculation of gradient in x-direction using separable row filter and column filter kernels is shown in Figure 7.5 and Figure 7.6. The masks are just reversed for y-gradient calculation during kernel invocation (which will be discussed in Fig. 7.3.1 later). The image arrays are bound to `cu_array`, fed to the kernels and accessed using texture fetches. After calculating the gradients `GradX` and `GradY`, they are fed to the `EDGE_MAP_KERNEL` to generate the edge map output. In this kernel, threads have neither offset access issue nor a problem of multiple threads accessing same data. Hence, the perfect bandwidth utilization is possible in the `EDGE_MAP_KERNEL` simply with the usage of global memory access as illustrated in the code snippet in Figure 7.7(a).




```
__global__ void ROW_FILTER_KERNEL (float *out, float
rMask[3])
{
....
// Unrolled Loop for instruction mix - performance
optimization
float result = tex2D(tex,col-1,row)*rMask[0]+
                  tex2D(tex,col,row)*rMask[1]+
                    tex2D(tex,col+1,row)*rMask[2];
....
}
```

Figure 7.5 Illustration and Kernel for Row Separable Filter




```
__global__ void COL_FILTER_KERNEL (float *out, float
cMask[3])
{
....
float result = tex2D(tex,col,row-1)*cMask[0]+
                  tex2D(tex,col,row)*cMask[1]+
                        tex2D(tex,col,row+1)*cMask[2];
....
}
```

Figure 7.6 Illustration and Kernel for Column Separable Filter

The CONNECTIVITY_MAP_KERNEL operates on the output generated by EDGE_MAP_KERNEL by using the connectivity mask mentioned in section 5.2 and Figure 4.5(a).

So, it has an offset of one again, which justifies the use of texture fetching as illustrated in the code snippet of Fig 7.1.7(b). It has to be observed that in all these kernels (which are compute-bound), for additional performance optimization, I unroll the loops to ensure instruction mix wherever required (which should be ideally done by the compiler) and achieve 100% instruction throughput according to the CUDA Visual Profiler [11].

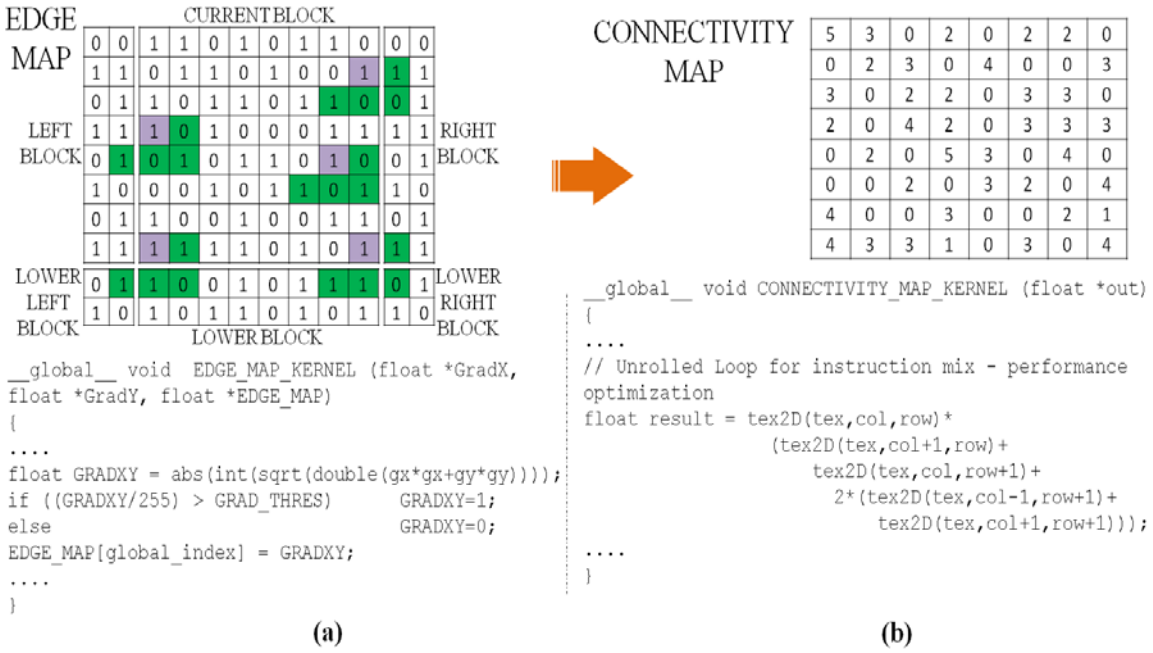


Figure 7.7 Edge and Connectivity Maps – (a) The picture illustrates how an edge map looks like and how the connectivity map is derived from an edge map in a GPU grid – based on Equation (1) a particular pixel (magenta) uses the surrounding pixels (green) to get the connectivity value. The code snippet shows the kernel used for calculation of edge map from the x- and y-gradients obtained from separable convolution. (b) The picture shows how a connectivity map looks like. The code snippet shows the kernel used for calculation of connectivity map from the edge map. The edge map obtained in previous stage is fetched using texture cache and operated on, to get the connectivity map.

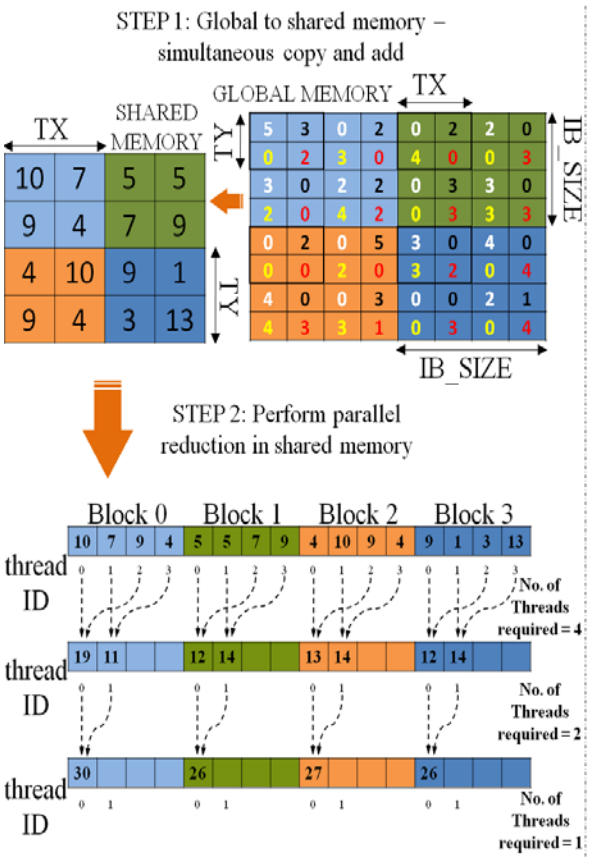
7.2 Stage 3: Block Division and Summation

I define a kernel called SUMMATION_KERNEL which is based on parallel reduction techniques mentioned in CUDA technical training manual [21] and William Kahan's parallel summation [22]. The aim of this kernel is to divide the connectivity map into `IB_SIZE x IB_SIZE` blocks and calculate the block sums, as per the requirements of Stage 3 of Figure 5.6. Although the CUDA block size for this kernel is `TX x TY` (as mentioned in Chapter 6), each block would handle `IB_SIZE x IB_SIZE` elements, because I should remember that the IFF block size is 'm' (i.e., `IB_SIZE` in computing platform).

Hence as the first step, I use "simultaneous copy and add from global to shared" method. In this step, each thread in x-direction handles summation of `IB_SIZE/TX` elements and each thread in y-direction handles summation of `IB_SIZE/TY` elements during global to shared memory loading as shown in Figure 7.8. This is done to condense `IB_SIZE x IB_SIZE` number of elements to 'CUDA block size' number of elements i.e., 128. Hence, this step produces 128 elements on which parallel reduction is performed in shared memory.

In the second step, I use shared memory parallel summation on the 128 elements with each thread handling one or more elements. During the course of parallel summation (Fig. 7.2.1) at every level, the number of threads reduces by a factor of 2. The first level requires 128/2 = 64 threads for parallel summation. It is imperative here that all threads pass the barrier synchronization and thus `__syncthreads()` function is used to ensure this.

For subsequent levels, the threads required are 32, 16, 8, 4, 2 and 1. Threads in all these levels are launched by a single warp instruction which need not be synchronized, because warps operate in SIMT fashion.




```
const unsigned short int MUL_FAC_X = IB_SIZE/TX;
const unsigned short int MUL_FAC_Y = IB_SIZE/TY;
__global__ void SUMMATION_KERNEL (float *d_in, float *d_out)
{
....
// Perform simultaneous copy and add from global to shared
memory: STEP 1
#pragma unroll
for(unsigned short int k=0;k<MUL_FAC_Y;k++)
      for(unsigned short int l=0;l<MUL_FAC_X;l++)
            c +=
d_in[global_index+l*blockDim.x+k*OP_WIDTH*blockDim.y];

sdata[tid] = c;
__syncthreads();

// Perform parallel reduction: STEP 2
if (tid < 64)
      sdata[tid] += sdata[tid + 64];

// Calculation outside a warp: Barrier sync required here
__syncthreads();

// Calculation within a warp: Unrolled and barrier sync
removed
if (tid < 32)
{
sdata[tid] += sdata[tid + 32]; sdata[tid] += sdata[tid + 16]
sdata[tid] += sdata[tid + 8]; sdata[tid] += sdata[tid + 4];
sdata[tid] += sdata[tid + 2]; sdata[tid] += sdata[tid + 1];
}
....
}
```

Figure 7.8 Illustration and Kernel Code Snippet for Stage 3: Values with same color (i.e., white, black, yellow and red) are added and stored in the shared memory. CUDA blocks are shown with bold black borders; IFF blocks are filled with different colors (light blue, green, orange and blue).

This kernel design produces a memory bandwidth of 90GB/s on a GTX280 card for the largest expected video type (1920 x 1080 frame size). Upon testing the algorithm on an arbitrary image matrix (a frame is nothing but a matrix of a values) of size 4096 x 4096 elements (or pixel values), the bandwidth achieved is 132 GB/s on the same card whose designated peak performance is 145GB/s, thus resulting in 92% memory bandwidth efficiency for this memory-

bound kernel. All these bandwidth measurements are reported by the CUDA Visual Profiler which comes with CUDA Toolkit [11].

```
float mask1[3]={-1,0,1}, mask2[3]={1,2,1};

                    // STAGE 1
        // Step 1: Gradient Calculator in x-direction
    // Outputs an intermediate row filtered gradient value
    ROW_FILTER_KERNEL  <<< dimGrid, dimBlock >>> (mask1);
                      // Outputs GradX
    COL_FILTER_KERNEL  <<< dimGrid, dimBlock >>> (mask2);

        // Step 2: Gradient Calculator in y-direction
  // Outputs an intermediate column filtered gradient value
    ROW_FILTER_KERNEL  <<< dimGrid, dimBlock >>> (mask2);
                      // Outputs GradY
    COL_FILTER_KERNEL  <<< dimGrid, dimBlock >>> (mask1);

          // Step 3: Outputs the Binary Edge Map
  EDGE_MAP_KERNEL <<< dimGrid, dimBlock >>> (GRADX, GRADY,
                         EDGE_MAP);

                    // STAGE 2
              // Outputs the Connectivity Map
      CONNECTIVITY_MAP_KERNEL <<< dimGrid, dimBlock >>>
                    (CONNECTIVITY_MAP);

                    // STAGE 3
              // Outputs the Block Sums/Areas
  SUMMATION_KERNEL <<< dimGrid, dimBlock >>> (CONNECTIVITY_MAP,
                         BLOCK_AREAS);
```

Figure 7.9 GPU IFF Algorithm: Flow of Kernel Invocations

7.3 CUDA Kernel Invocations

The complete flow of kernel invocations in the GPU algorithm is presented in Figure 7.9. From this figure I can see that Stage 1 has three steps in it. In the first step, the gradient in x-direction is calculated by calling `ROW_FILTER_KERNEL` and `COL_FILTER_KERNEL`

consecutively with `mask1` and `mask2` as their respective inputs. In the second step, the gradient in y-direction is calculated by calling the same kernels consecutively but with `mask2` and `mask1` as their respective inputs.

In the third step, `EDGE_MAP_KERNEL` is called to generate the edge map. Further in the algorithm, `CONNECTIVITY_MAP_KERNEL` and `SUMMATION_KERNEL` are called in stages 2 and 3 to generate connectivity map and calculate block sums respectively. The block sums are analyzed based on the technique presented in section 5.2 with negligible computational cost. The results of IFF GPU algorithm will be discussed in the real-time module analysis of experimental section.

CHAPTER 8

CPU-GPU COMBO SCHEME: HANDLING BOTTLENECK #2

As discussed in Chapter 5, the bottleneck in post-procedure version is a combination of disk read time and IFF (informative frame filtering) algorithm time. Reading frames from disk is not a numerically intensive operation; instead it is a data-access intensive operation. It should be noted here that the problem is not with inability to allocate buffers in memory to read and hold the data of multiple frames, but it is with the ability to read multiple frames in lesser time. GPU's (graphic processing unit) capabilities do not affect reading multiple frames from disk in a single time frame. I made functions in C code which read frames from disk, store the RGB (Red-Green-Blue) color space data in a buffer and serve it to the IFF algorithm.

With a sequential code design, the time taken to read the frame from disk is almost equal to the time taken to process it using my IFF algorithm on central processing unit - CPU (Table 9.6). According to Table 9.6, sequential CPU code takes 96ms to read from disk and 96ms to process it, making the total execution time around 192ms.

Exploiting the power of GPU to process the frame by using GPU IFF algorithm reduced the algorithm cost to 8ms, but yet the GPU sits idle for 88ms (inefficient GPU occupancy) for the next frame to be served by CPU (Figure 8.1). Sequential code design with algorithm processing on CPU or GPU still does not exploit the multi-core capabilities of a CPU.

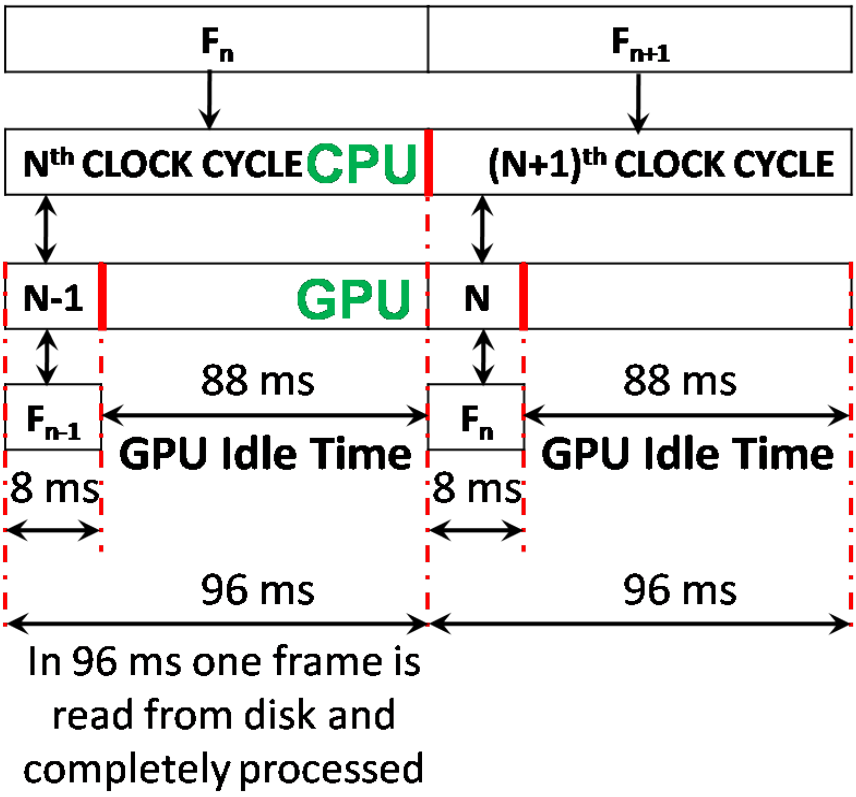


Figure 8.1 Post-Procedure IFF Module: Using Sequential CPU Coding with GPU

In order to ensure maximum GPU occupancy, I explored the multithreading capability of CPU cores. In this way, on an Intel Quad core workstation which has four cores (in other words four hardware threads) and each core with the ability to handle four threads [1], I can generate 16 threads. I assigned thread handlers to the functions made in C, such that 16 threads can simultaneously read 16 frames from disk and serve them to the GPU IFF algorithm. So, in the same 96 ms I can now read 16 frames and serve it to a single GPU. The GPU returns control after receiving the frame data, and takes 128 ms to completely process them all.

During this time, the CPU reads the next set of 16 frames and holds them in a pipeline to serve to the GPU, and is still left with a credit of 32 ms which is utilized to read the subsequent 16 frames.

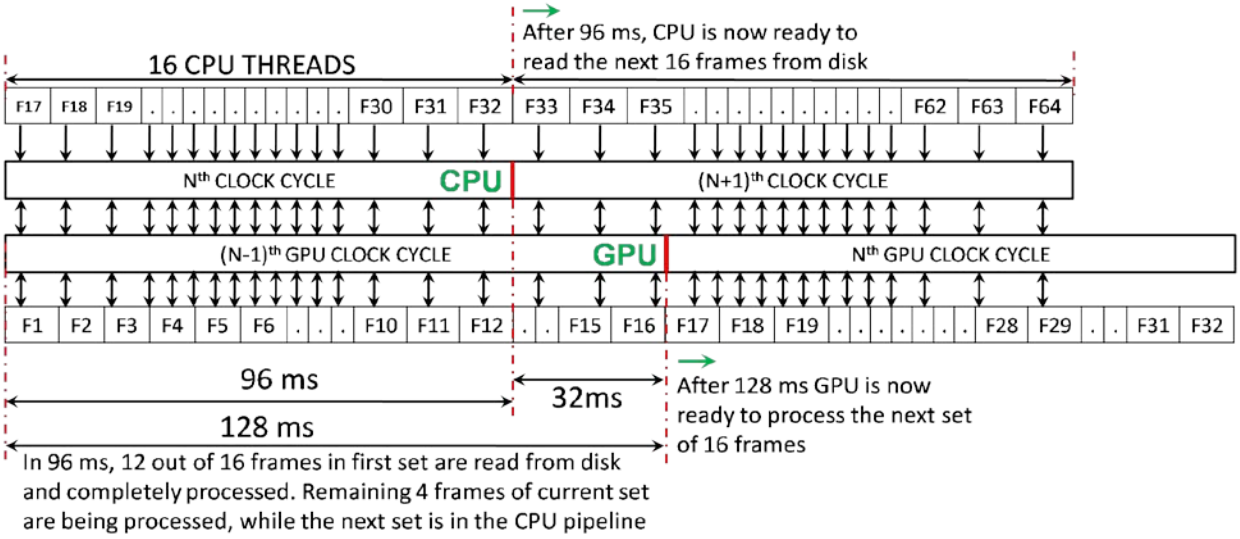


Figure 8.2 Post-Procedure IFF Module: Using Parallel CPU Coding with GPU

This is clearly shown in Figure 8.2. By using such a CPU multi-threaded pipelining scheme coupled with the GPU's massive parallel processing power, I am able to read and process more frames per unit of time (Table 9.7). I made different versions in the post-procedure module exploiting the architectural capabilities of multi-core and many-core processing units in an increasing order. With each version, I added an extra capability that I believed would add to the performance. At this point, I have four implementation versions for the post-procedure IFF module:

- Version 1: Using CPU without multi-threading for disk reads and IFF algorithm
- Version 2: Using CPU without multi-threading for disk reads, and GPU for IFF algorithm
- Version 3: Using CPU with multi-threading for disk reads and IFF algorithm
- Version 4: Using CPU with multi-threading for disk reads, and GPU for IFF algorithm

The execution times and frame processing speeds achieved by running these four versions of post-procedure module on colonoscopy videos are presented in next chapter.

CHAPTER 9

EXPERIMENTAL RESULTS

9.1 Accuracy Results

As mentioned in Chapter 5, several tests have been performed to choose the right parameters for the new algorithm during its design. Performance metrics are calculated based on the following parameters:

- ti = No. of True Informative Frames
- fi = No. of False Informative Frames
- tn = No. of True Non-informative Frames
- fn = No. of False Non-informative Frames

Four performance metrics are chosen to evaluate the performance of the accuracy tests with integrity:

- Precision = ti/(ti+fi)
- Sensitivity = tn/(fi+tn)
- Specificity = ti/(ti+fn)
- Accuracy = (ti+tn)/(ti+tn+fi+fn)

The following parameter sets have been considered for experiments to finalize the algorithm's final parameters (mentioned in Chapter 5):

- Phase I parameters: (thresholds used ensure low-sensitivity)
    - Sobel Edge Detector – threshold = 0.33

- LoG Edge Detector – threshold = 0.008, sigma =2
- Canny Edge Detector – Lower threshold = 0.08, Upper Threshold = 0.2, Sigma =1.2

- Phase II parameters:
  - Set 1
    - Block Size, m = 32 x 32
    - Block Connectivity Threshold, BC_{th} = 4
    - Informative Blocks Ratio Threshold, I_{rth} = 0.25
  - Set 2
    - Block Size, m = 64 x 64
    - Block Connectivity Threshold, BC_{th} = 5
    - Informative Blocks Ratio Threshold, I_{rth} = 0.33
  - Set 3
    - Block Size, m = 80 x 80
    - Block Connectivity Threshold, BC_{th} = 8
    - Informative Blocks Ratio Threshold, I_{rth} = 0.4

The test results of combinations of the above parameters are shown in Tables 9.1, 9.2 and 9.3. It can be noted that sobel edge detector (Phase I parameters) with set 2 (Phase II parameters) show the highest performance metrics.

TABLE 9.1 EXPERIMENTAL RESULTS OF PHASE I CANNY WITH PHASE II – SETS 1, 2, 3

Video ID	Canny											
	32 x 32 block				64 x 64 block				80 x 80 block			
	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC
1_ff_8_3_09	96.71 %	98.31 %	68.71 %	85.91 %	86.74 %	89.87 %	91.81 %	90.69 %	97.20 %	98.52 %	71.05 %	87.01 %
6_ff_8_4_09	96.55 %	99.42 %	39.25 %	81.81 %	80.43 %	93.04 %	69.16 %	86.05 %	94.79 %	99.03 %	42.52 %	82.49 %
8_ff_8_5_09	98.81 %	99.80 %	45.60 %	85.32 %	69.91 %	86.37 %	86.81 %	86.49 %	98.06 %	99.60 %	55.49 %	87.81 %
15_ff_8_5_09	93.60 %	96.88 %	20.86 %	44.68 %	90.94 %	79.69 %	93.05 %	88.86 %	95.06 %	96.88 %	27.45 %	49.20 %
34_ff_8_7_09	96.03 %	98.59 %	46.81 %	76.73 %	67.68 %	68.36 %	90.72 %	77.80 %	92.71 %	96.61 %	58.99 %	80.73 %
49_ff_8_10_09	98.96 %	99.00 %	53.57 %	69.92 %	88.87 %	78.93 %	94.55 %	88.93 %	99.08 %	99.00 %	60.53 %	74.37 %
50_ff_8_10_09	97.70 %	99.07 %	51.90 %	78.65 %	84.71 %	87.59 %	90.11 %	88.68 %	96.56 %	98.45 %	57.05 %	80.53 %
20080729_P06	79.10 %	75.29 %	71.30 %	73.03 %	72.11 %	51.76 %	95.07 %	76.34 %	78.80 %	72.94 %	76.68 %	75.06 %
capture0310	89.36 %	96.48 %	59.15 %	84.04 %	69.47 %	79.58 %	92.96 %	84.04 %	90.20 %	96.48 %	64.79 %	85.92 %
test_4_3201	94.25 %	99.03 %	38.32 %	81.26 %	80.32 %	92.84 %	70.56 %	86.32 %	93.62 %	98.84 %	41.12 %	81.94 %
Average	94.11 %	96.19 %	49.55 %	76.14 %	79.12 %	80.80 %	87.48 %	85.42 %	93.61 %	95.64 %	55.57 %	78.51 %

TABLE 9.2 EXPERIMENTAL RESULTS OF PHASE I SOBEL WITH PHASE II – SETS 1, 2, 3

Video ID	Sobel											
	32 x 32 block				64 x 64 block				80 x 80 block			
	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC
1_ff_8_3_09	100.00 %	100.00 %	57.31 %	82.11 %	97.43%	98.10%	99.71%	98.77%	99.55%	99.79%	64.33 %	84.93 %
6_ff_8_4_09	100.00 %	100.00 %	53.74 %	86.46 %	95.54%	98.07%	100.00 %	98.63%	100.00 %	100.00 %	58.88 %	87.96 %
8_ff_8_5_09	100.00 %	100.00 %	23.08 %	79.44 %	94.29%	98.00%	90.66%	96.04%	100.00 %	100.00 %	32.97 %	82.09 %
15_ff_8_5_09	100.00 %	100.00 %	39.39 %	58.38 %	99.28%	98.44%	98.04%	98.16%	100.00 %	100.00 %	52.94 %	67.69 %
34_ff_8_7_09	98.99%	99.86%	18.96 %	65.71 %	94.18%	96.05%	87.62%	92.49%	97.96%	99.58%	27.85 %	69.31 %
49_ff_8_10_09	100.00 %	100.00 %	37.97 %	60.29 %	98.10%	96.66%	96.80%	96.75%	99.22%	99.33%	47.93 %	66.43 %
50_ff_8_10_09	99.65%	99.90%	38.48 %	73.31 %	96.81%	97.52%	98.64%	98.01%	99.71%	99.90%	47.29 %	77.13 %
20080729_P06	100.00 %	100.00 %	51.57 %	72.52 %	98.65%	98.24%	98.21%	98.22%	100.00 %	100.00 %	54.71 %	74.30 %
capture0310	100.00 %	100.00 %	22.54 %	74.18 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	32.39 %	77.46 %
test_4_3201	100.00 %	100.00 %	53.74 %	86.46 %	95.54%	98.07%	100.00 %	98.63%	100.00 %	100.00 %	59.81 %	88.24 %
Average	99.86%	99.98%	39.68 %	73.89 %	96.98%	97.92%	96.97%	97.57%	99.64%	99.86%	47.91 %	77.55 %

TABLE 9.3 EXPERIMENTAL RESULTS OF PHASE I LAPLACIAN OF GAUSSIAN WITH PHASE II – SETS 1, 2 AND 3

Video ID	LoG											
	32 x 32 block				64 x 64 block				80 x 80 block			
	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC
1_ff_8_3_09	98.85%	99.79%	25.15%	68.50%	95.37%	97.26%	78.36%	89.34%	100.00%	100.00%	28.36%	69.98%
6_ff_8_4_09	100.00%	100.00%	20.56%	76.74%	93.29%	97.87%	71.50%	90.15%	100.00%	100.00%	24.30%	77.84%
8_ff_8_5_09	100.00%	100.00%	16.48%	77.68%	90.37%	97.39%	67.03%	89.28%	100.00%	100.00%	21.43%	79.00%
15_ff_8_5_09	100.00%	100.00%	7.66%	36.60%	98.87%	98.44%	62.21%	73.56%	100.00%	100.00%	15.69%	42.11%
34_ff_8_7_09	98.75%	99.86%	15.28%	64.16%	97.13%	98.87%	52.42%	79.27%	98.44%	99.72%	24.37%	67.92%
49_ff_8_10_09	100.00%	100.00%	27.63%	53.67%	96.57%	95.99%	63.53%	75.21%	100.00%	100.00%	32.89%	57.04%
50_ff_8_10_09	100.00%	100.00%	26.29%	68.09%	94.79%	97.31%	64.09%	82.93%	100.00%	100.00%	31.98%	70.56%
20080729_P06	100.00%	100.00%	45.29%	68.96%	96.07%	95.88%	76.68%	84.99%	100.00%	100.00%	47.98%	70.48%
capture0310	100.00%	100.00%	16.90%	72.30%	83.33%	92.96%	70.42%	85.45%	100.00%	100.00%	22.54%	74.18%
test_4_3201	100.00%	100.00%	18.69%	76.20%	97.22%	99.23%	65.42%	89.33%	100.00%	100.00%	25.23%	78.11%
Average	99.76%	99.97%	21.99%	66.29%	94.30%	97.12%	67.17%	83.95%	99.84%	99.97%	27.48%	68.72%

TABLE 9.4 COMPARISON OF NEW ALGORITHM WITH OLD ALGORITHM AND A THIRD-PARTY ALGORITHM

Video Name	Previous Algorithm				Current Algorithm				Third-Party Algorithm			
	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC	PREC	SENS	SPEC	ACC
1_ff_8_3_09	91.23%	83.71%	79.79%	78.26%	97.43%	98.10%	99.71%	98.77%	93.38%	91.76%	90.04%	90.46%
6_ff_8_4_09	92.25%	86.92%	76.24%	84.52%	95.54%	98.07%	100.00%	98.63%	90.22%	65.61%	91.37%	80.19%
8_ff_8_5_09	87.26%	85.44%	73.58%	81.24%	94.29%	98.00%	90.66%	96.04%	90.88%	89.24%	91.94%	90.70%
15_ff_8_5_09	90.45%	89.62%	78.42%	83.43%	99.28%	98.44%	98.04%	98.16%	95.66%	89.35%	92.10%	89.49%
34_ff_8_7_09	86.14%	90.46%	72.15%	79.07%	94.18%	96.05%	87.62%	92.49%	94.13%	90.07%	87.11%	88.56%
49_ff_8_10_09	88.23%	88.35%	71.84%	80.66%	98.10%	96.66%	96.80%	96.75%	93.34%	86.33%	90.33%	87.45%
50_ff_8_10_09	93.78%	87.01%	77.68%	75.46%	96.81%	97.52%	98.64%	98.01%	93.15%	93.85%	83.17%	90.44%
20080729_P06	90.65%	87.75%	75.72%	81.49%	98.65%	98.24%	98.21%	98.22%	85.68%	88.37%	84.01%	85.63%
capture0310	89.84%	90.26%	74.99%	77.86%	100.00%	100.00%	100.00%	100.00%	89.41%	88.94%	81.19%	85.54%
test_4_3201	88.63%	84.64%	79.11%	86.55%	95.54%	98.07%	100.00%	98.63%	88.22%	84.55%	80.24%	84.11%
Average	89.85%	87.42%	75.95%	80.85%	96.98%	97.92%	96.97%	97.57%	91.41%	86.81%	87.15%	87.26%

These results from Table 9.4 show that my new algorithm (with sobel – set 2 parameters of Chapter 3) is far better in terms of accuracy and effectiveness compared to other algorithms.

9.2 Speed and Acceleration Results

As discussed in Chapter 5, there are two bottlenecks, each one belonging to a different version of the IFF module. In a real-time scenario, where frames coming from a capture-card are processed, the only constraint is the speed of the IFF algorithm. In a post-procedure scenario, apart from the algorithm cost, there is an additional overhead cost of reading the frame from disk to memory and serving it to the algorithm. I presented my solutions to handle the bottlenecks pertaining to both these versions in the previous chapters.

My experiments of different versions of the IFF modules are deployed on a machine having an Intel Quad Core CPU @ 3.0 GHz with 3 GB RAM and an NVIDIA GTX 280 card with 1 GB GDDR3. For convenience, I classify the results as real-time IFF module results and post-procedure IFF module results. Firstly, I present a comparative analysis of GPU and CPU versions of the real-time IFF module – which in other words is nothing but the IFF algorithm alone, without any disk-read operations, on different frame types. By different frame types, I mean, frames having different resolutions due to different video input sources. This analysis is done to present a plausible comparison in the real-time scenario.

I chose six different video input sources with different frame resolutions and fed them to my CPU and GPU IFF algorithm versions. Each algorithm is run over more than a 100 frames of every video type. To accurately measure the elapsed time of the sequence of all CUDA calls in the GPU algorithm, I used `CUDA event` API [19]. The CPU algorithm time is measured using

the high-resolution CPU timer functions – `QueryPerformanceCounter` and `QueryPerformanceFrequency`.

Table 9.5 shows the average processing times taken by the CPU/GPU IFF algorithms to process a single frame belonging to each of these video inputs. A logarithmic graph is plotted (Figure 9.1) to show the acceleration achieved by a GPU compared to the CPU. As seen in Table 9.5 and Figure 9.1, with the increase in the frame size, the CPU processing time increases rapidly. On the other hand, the GPU processing time increases minimally.

TABLE 9.5 REAL-TIME IFF MODULE RESULTS FOR A SINGLE FRAME FROM DIFFERENT VIDEO INPUTS

Video Type	Frame Size	GPU (ms)	CPU (ms)	Speed Up
VGA	640 x 480	6.73	85.76	12.74
DVD	720 x 480	8.00	94.46	11.81
HD 576	720 x 576	9.53	121.99	12.80
XGA	1024 x 768	12.70	236.29	18.61
HD 720	1280 x 720	11.58	268.81	23.22
HD 1080	1920 x 1080	14.88	595.12	40.00

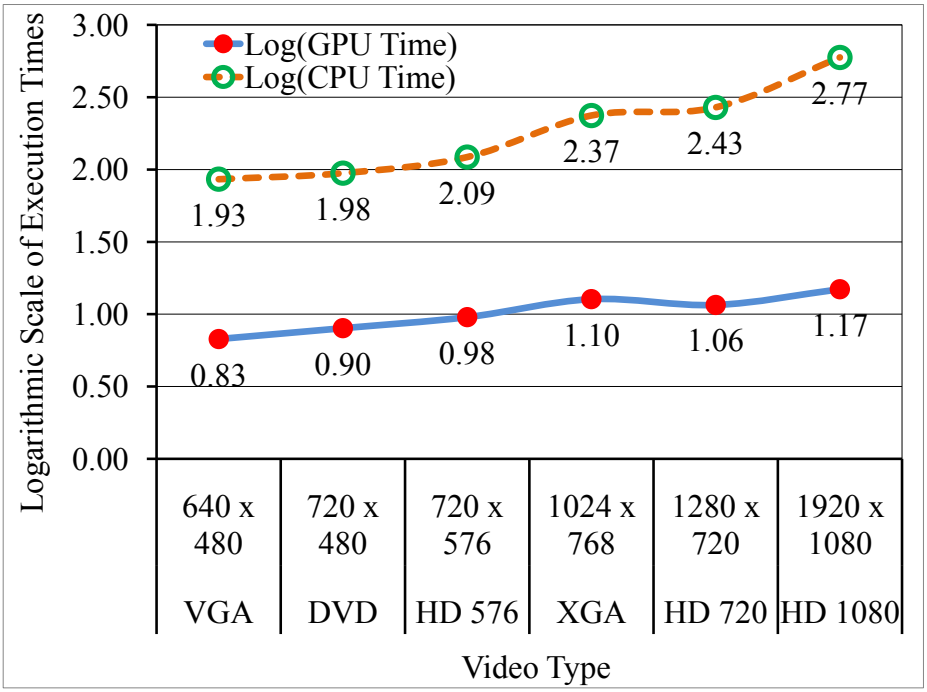


Figure 9.1 Real-Time Iff Module Speed Graph

Secondly, I have tested the post-procedure IFF module described in Chapter 8 and Fig. 5.3.1, on Fujinon videos (frame resolution: 720 x 480) with the four different versions of codes. As mentioned before, this module has an additional 'frame-read from disk' operation. Table 9.6 shows the average disk-read time and average IFF algorithm time on CPU and GPU for a single frame of size 720 x 480 (DVD Resolution). These values show that, on a CPU the algorithm time is almost same as disk-read time, while on a GPU, the algorithm time is very less, almost negligible. Since the disk read time consumes half of the total execution time, even if I reduce the algorithm execution time to a negligible value, the total execution time decreases only to half the original time, in other words, the acceleration is about two-fold. This table is presented to exemplify the fact that disk-read time plays a major role in the total execution time of the post-procedure IFF module.

TABLE 9.6 POST-PROCEDURE IFF MODULE RESULTS FOR A SINGLE FRAME – SHOWS DISK READ AND ALGORITHM TIMES

Frame Size	**Avg. Disk Read Time (ms)**	**Avg. Algorithm Time (ms)**		**Total Execution Time (ms)**	
		CPU	**GPU**	**CPU**	**GPU**
720 x 480	94.14	94.46	8.00	188.6	102.14

The four versions of code I made start from a basic level implementation (Version 1) to an advanced level implementation (Version 4). These versions are targeted at explaining the roles played by CPU in accelerating the post-procedure module's disk-read operations, and by GPU in accelerating the IFF algorithm operations. The versions emerge by permuting and combining the two solutions provided in Chapter 7 and Chapter 8 for the two bottlenecks mentioned at the end of Chapter 5. Currently, in the post-procedure scenario, I am working only on videos with DVD resolution. The four versions are tested on 10 Fujinon videos.

The results are tabulated in Table 9.7. The frame processing speed of each version is calculated as number of input frames over total execution time of that particular version. In this table, the total execution time, based on the version, includes frame reading and loading time, algorithm execution time and for versions using GPU-based IFF algorithm, the memory copy time from CPU to GPU and vice-versa is also included.

TABLE 9.7 POST-PROCEDURE IFF MODULE RESULTS FOR ALL FOUR VERSIONS TESTED ON 10 FUJINON VIDEOS

Video ID	Number of frames	Version 1 - Total Execution Time (sec)	Version 1 - Frame Processing Speed (frames/sec)	Version 2 - Total Execution Time (sec)	Version 2 - Frame Processing Speed (frames/sec)	Version 3 - Total Execution Time (sec)	Version 3 - Frame Processing Speed (frames/sec)	Version 4 - Total Execution Time (sec)	Version 4 - Frame Processing Speed (frames/sec)
1	816	165.03	4.94	89.02	9.17	61.42	13.29	37.21	21.93
2	1277	258.59	4.94	137.67	9.28	86.50	14.76	44.17	28.91
3	1327	268.88	4.94	142.00	9.35	98.17	13.52	50.54	26.26
4	3539	775.36	4.56	411.92	8.59	296.94	11.92	173.56	20.39
5	2381	522.33	4.56	288.75	8.25	192.53	12.37	120.09	19.83
6	731	147.86	4.94	79.48	9.20	56.11	13.03	27.80	26.29
7	1419	287.11	4.94	154.11	9.21	93.45	15.18	63.54	22.33
8	681	137.75	4.94	73.05	9.32	52.05	13.08	25.62	26.58
9	2148	466.20	4.61	257.65	8.34	175.66	12.23	105.39	20.38
10	1344	271.78	4.95	146.43	9.18	104.34	12.88	51.11	26.30
Average Frame Processing Speed			**4.83**		**8.99**		**13.23**		**23.92**

From Table 9.7, I can see that the most unsophisticated version is Version 1. It does not use the multi-core capability of CPU to multi-thread disk-reads and neither does it use the GPU for implementing the IFF Algorithm. This is the most basic version. Version 2 moves a step ahead and leaves the algorithm processing task to a GPU. But it does not use CPU cores to multi-thread the frame serving process yet. However, I see a speed up of around 2x in Version 2 compared to Version 1. In version 3, the CPU cores are exploited. CPU Multi-threading for frame reads is introduced in this version, but the IFF algorithm still runs on CPU. So, the CPU

serves several frames but it processes these frames itself by still maintaining the sequential flow of the algorithm. Version 3 gives a speed up of around 2.8x compared to Version 1. Version 4 exploits both CPU multi-core capability and GPU many-core architecture together. This version uses all possible resources available and achieves a speed up of around 5x compared to Version 1. A graph is plotted showing the speed comparison amongst the four versions of the post-procedure IFF module (Figure 9.2).

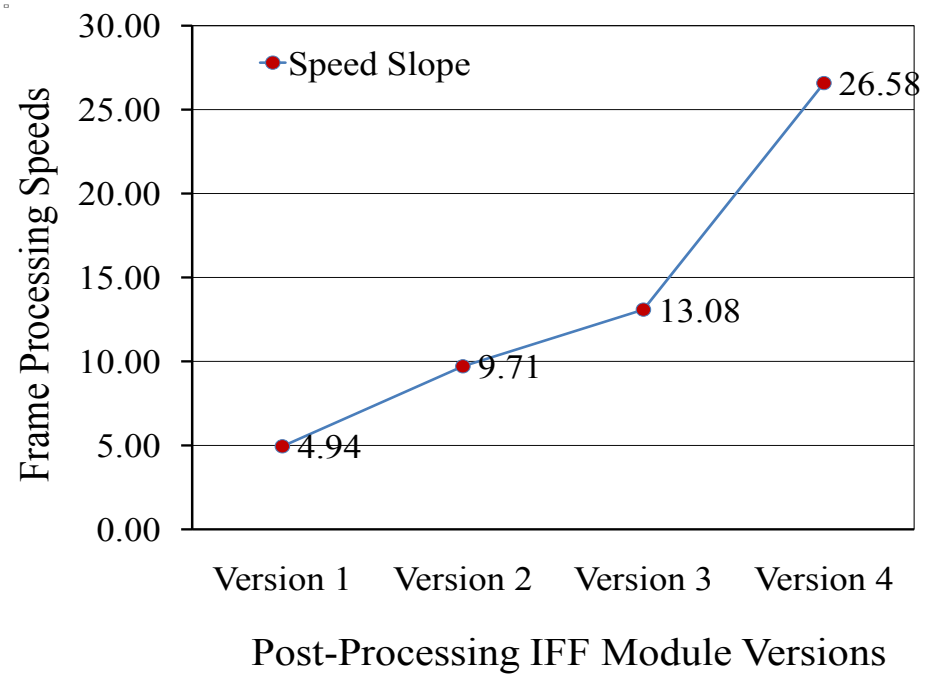


Figure 9.2 Post-Procedure IFF Module Speed Graph

CHAPTER 10

CONCLUSION AND FUTURE DIRECTION

I have two versions for informative frame filtering – post-procedure and real-time. It is clear (from Table 9.7) that, in the post-procedure version the disk read time is almost half of the total execution time. So, although I am able to reduce the algorithm time to negligible value using a GPU, the maximum overall acceleration I can achieve is only two fold (compare Table 9.7, versions 1 and 2). However, when CPU (central processing unit) multi-threading capability is used to read frames from disk to memory and serve these to the GPU (graphic processing unit), I achieved up to a five-fold acceleration (compare Table 9.7, versions 1, 3, and 4).

In the real-time version, all compute constraints reside within the algorithm time because there is no disk read. To estimate the effect on high definition video streams, I used different frame sizes, and calculated that the maximum acceleration I can achieve with a many-core GPU algorithm for future HD (high- definition) 1080 colonoscopy video frames is 40 fold compared to a modern day CPU alone (Table 9.5). The future work will be focused on combining multiple GPUs together to further accelerate colonoscopy video analysis.

I made a module which uses multiple GPUs. NVIDIA's primary requirement is to associate each CPU core with a GPU. It means on a 4 core machine, I can generate only 4 threads and link them with the GPU cards. So, I cannot generate 16 multiple threads like in my CPU-GPU combo scheme (Chapter 8). This module has not shown any significant performance boost yet. So, the future of this research would be to improve the performance with multiple GPUs. Accuracy-wise this algorithm is sufficient enough.

REFERENCES

[1] David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, San Francisco, 2010.

[2] American Cancer Society, "Colorectal Cancer Facts and Figures" 2008, http://www.cancer.org/docroot/STT/content/STT_1x_Cancer_Facts_and_Figures_2008.asp

[3] C. D. Johnson, J. G. Fletcher, R. L. MacCarty, et al, "Effect of slice thickness and primary 2D versus 3D virtual dissection on colorectal lesion detection at CT colonography in 452 asymptomatic adults". AJR American Journal of Roentgenology, 2007.

[4] 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, 2005.

[5] JungHwan Oh, Sae Hwang, JeongKyu Lee, Tavanapong, W., Piet C. de Groen, Johnny Wong, "Informative Frame Classification for Endoscopy Video". J. Medical Image Analysis , 2007.

[6] Yong Han An, JungHwan Oh, Sae Hwang, Wallapak Tavanapong, Piet C. de Groen and Johnny Wong, "Informative-Frame Filtering in Endoscopy Videos". SPIE International Symposium, Medical Imaging, 2005.

[7] NVIDIA CUDA Programming Guide 3.0-beta1, 2009, www.nvidia.com

[8] Yuancheng Luo, R. Duraiswami, "Canny Edge Detection on NVIDIA CUDA". IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008.

[9] Seung In Park, Sean P. Ponce, Jing Huang, Yong Cao and Francis Quek, "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture". 37th IEEE Applied Imagery Pattern Recognition Workshop, 2008.

[10] Zhiyi Yang, Yating Zhu, Yong Pu, "Parallel Image Processing Based on CUDA", International Conference on Computer Science and Software Engineering, 2008.

[11] NVIDIA CUDA Toolkit v3.0-beta1 (includes CUDA Visual Profiling Tool), 2009, www.nvidia.com

[12] Elena Tsomko, Hyoung Joong Kim, “Efficient Method of Detecting Globally Blurry or Sharp Images”. Ninth International Workshop on Image Analysis for Multimedia Interactive Services, 2008.

[13] Frederique Crete, Thierry Dolmiere, Patricia Ladret, Marina Nicolas, "The Blur Effect: Perception and Estimation with a New No-Reference Perceptual Blur Metric". SPIE Electronic Imaging Symposium Conf Human Vision and Electronic Imaging, San Jose, 2007

[14] Zhang Hua, Zhu Wei, Chen Yaowu, "A No-Reference Perceptual Blur Metric by using OLS-RBF Network", IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application, Vol. 1, 2008.

[15] EePing Ong. Weisi Lin, Zhongkaiig Lu, Xiaokang Yang, Susir Yao, Feng Pan, Lijilrn Jiarrg, and Fulvio Moscheni, "A no-reference quality metric for measuring image blur ". Seventh International Symposium on Signal Processing and Its Applications, 2003.

[16] Pina Marziliano, Frederic Dufaux, Stefan Winkler and Touradj Ebrahimi, "A no-reference perceptual blur metric", International Conference on Image Processing, Proceedings Vol. 3, 2002.

[17] J. F. Canny, “A Computational Approach to Edge Detection", IEEE Trans. Pattern Analysis and Machine Intelligence, Nov. 1986.

[18] NVIDIA GeForce GTX 200 Technical Brief, www.nvidia.com

[19] CUDA Programming Best Practices Guide 3.0-beta1, 2009, www.nvidia.com

[20] V. Podlozhnyuk, “Image Convolution with CUDA”, White Paper, June 1, 2007, www.nvidia.com

[21] CUDA Technical Training, Vol. I: Introduction to CUDA Programming, 2008, www.nvidia.com

[22] http://en.wikipedia.org/wiki/Kahan_summation_algorithm