A Graphical Processing Unit Based on Real Time System

by

Khaled M. Alqahtani

Submitted in partial fulfilment of the requirements
for the degree of Master of Applied Science

at

Dalhousie University
Halifax, Nova Scotia
March 2016

To My Parents: Mohammed and Hissah, Family: Hind, Faisal and Norah, & Prof. Joshua Leon

Table of Contents

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The field of real-time image processing focuses on analysing and enhancing images in real time. An effective way to accomplish this is to utilise both the central processing unit (CPU) and the graphics processing unit (GPU) at the same time, to achieve the highest possible performance. The CPU manages tasks such as sequential operations and input-output, while the GPU is used for operations that can be done in parallel. The real-time processing of medical digital images requires the processing of an image with a response guaranteed within a specified time. This time period should be within a couple of seconds or less, so that images can be analysed and manipulated in a real-time medical scenario.

Real-time image processing has gained significance due to its widespread use in communication schemes involving video conferencing, video calls, innovative media, and digital and mobile cameras. Because of this, image identification has recently become an important research topic. The goal of this work is to develop a real-time digital image processing system aimed at feature identification and classification.

This thesis presents a method which seeks to reveal normal and malignant tissues. A method of digital image processing via real-time processing is designed to allow radiologists to detect abnormal tissues or cells, and then determine whether a tumor is a carcinoma or benign. Although radiologist decisions remain the preferred way of detecting non-palpable cancer, the method in this thesis aims to make detection by radiology trainers and radiologists easier and more accurate than is the case with traditional techniques.

The research described in this thesis involves a toolbox with a number of different filters, including classical high- and low-pass filters, as well as various novel morphological filtering tools. Edge and feature detection algorithms have also been added.

This thesis presents the first phase of the removal of noise from noisy images (having a signal-to-noise ratio of 10% or more). Simulated noise of different types is added to original images and then a variety of filters including mean, median, erosion, dilation, opening and closing filters are applied. These filters are used to denoise the original images. Erosion and dilation filters are the two basic filters in the area of mathematical morphology. Although they are usually used at the binary level, in the present research they are used at the grey-scale level. Mean and median filters function in a similar manner, except that median filters preserve more important details in a processed image than is the case with mean filters.

In addition to the above-mentioned filters, Laplacian filters are also utilised to increase the contrast of edges, and thresholding techniques are then applied for a first attempt at feature identification. Although the initial work was done in MATLAB, the algorithm developed here is also implemented using CUDA on graphics processing units, with the goal of implementing the system in real time or near real time. Moreover, some algorithms related to segmentation and automatic identification features have been developed in CUDA.

LIST OF ABBREVIATIONS USED

CUDA	Compute Unified Device Architecture
GPGPU	Graphics Processing General Purpose Computing On Graphical Processing Units
SNR	Signal-to-Noise Ratio
GPU	Graphics Processing Unit
CPU	Central Processing Unit
HPC	High-Performance Computing
MRI	Magnetic Resonance Imaging
SE	Structuring Element
SF	Structuring Function
DIP	Digital Image Processing

ACKNOWLEDGEMENT

First and foremost, it is a pleasure to thank the many people who made this thesis possible. It is difficult to overstate my gratitude to my supervisor, Prof. Joshua Leon. With his enthusiasm, his inspiration, patience, motivation and his great efforts to explain things clearly and simply. His guidance helped me to gain a deeper understanding of the field of digital image processing, Matlab, C#, and CUDA programming. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I could not have imagined having a better advisor and mentor for my master study. It has an honor to be his master student.

Last but not least, I am very grateful to my family for giving birth to me at the first place and supporting me spiritually throughout my life, especially to my mother Hissah Alarifi for her patience and love, which are very important to me.

CHAPTER 1: INTRODUCTION

1.1 Introduction

Real-time image processing refers to the analyses and enhancement of images in almost real-time. In order to achieve the fastest possible performance, image enhancement, our approach, was to develop algorithms that operate on both the central processing unit (CPU) and the graphics processing unit (GPU).

In today's world, digital images are everywhere. Human beings generally understand and perceive more via their eyes than with the other four senses. Much of the information acquired by people is obtained through images. Thus, the human sense of vision and the science of mathematics are two aspects which are combined in the field of image processing.

Digital image processing (DIP) is a widely used technology, with diverse fields of application. Today, digital images have made an impact in almost all areas of technical endeavour. For example, the technical revolution in medicine and particularly in medical image processing plays a decisive role. Image processing is employed in many scientific areas, such as:

- Forensic science: DIP has many applications in this area, such as enhancing an image or a short video clip captured by a surveillance camera.
- Industry: DIP plays a significant role, for instance in detecting defective items on production lines.
- Information processing: DIP is useful for handwriting interpretation. Problems associated with forgery and fraud can thus be addressed via DIP.

- Computer aided diagnosis: Especially in North America, DIP is commonly used in early breast cancer detection.

Digital image processing can be a valuable tool for the purpose of:

- Image segmentation: Dividing an image into meaningful sections can provide a useful approach in some tasks such as monitoring tumor growth or identifying scarred or damaged tissue following a heart attack.
- Image registration or matching: Part of one or more images of the same organ of a patient can be extracted, to study a region of interest regarding human organ or body tissues.

The use of digital images dates back to the 1920s, when newspaper pictures were sent between London and New York via submarine cables, a major breakthrough during that era. The time required to transfer such an image was less than 3 hours; however, quality was the greatest concern of researchers. This problem was addressed by means of a photographic reproduction method based on perforated tape [1, 2].

In the past century image processing has developed rapidly, and has became an inextricable part of countless facets of daily life. DIP is an essential element of numerous areas of application, including prominent fields such as medical, military and industrial machinery applications.

In medicine, image processing is used extensively in processing X-rays and MRI images and in simplifying the images. It can be used by doctors and radiologists to diagnose, detect and recognise serious diseases. Although X-rays represent the oldest such medical technology used to diagnose diseases, a wide range of other technologies have been developed that also require image processing techniques [3]. DIP can be envisaged as involving ten fundamental components (see Figure 1). In this thesis, Only the unshaded processing steps have been used.

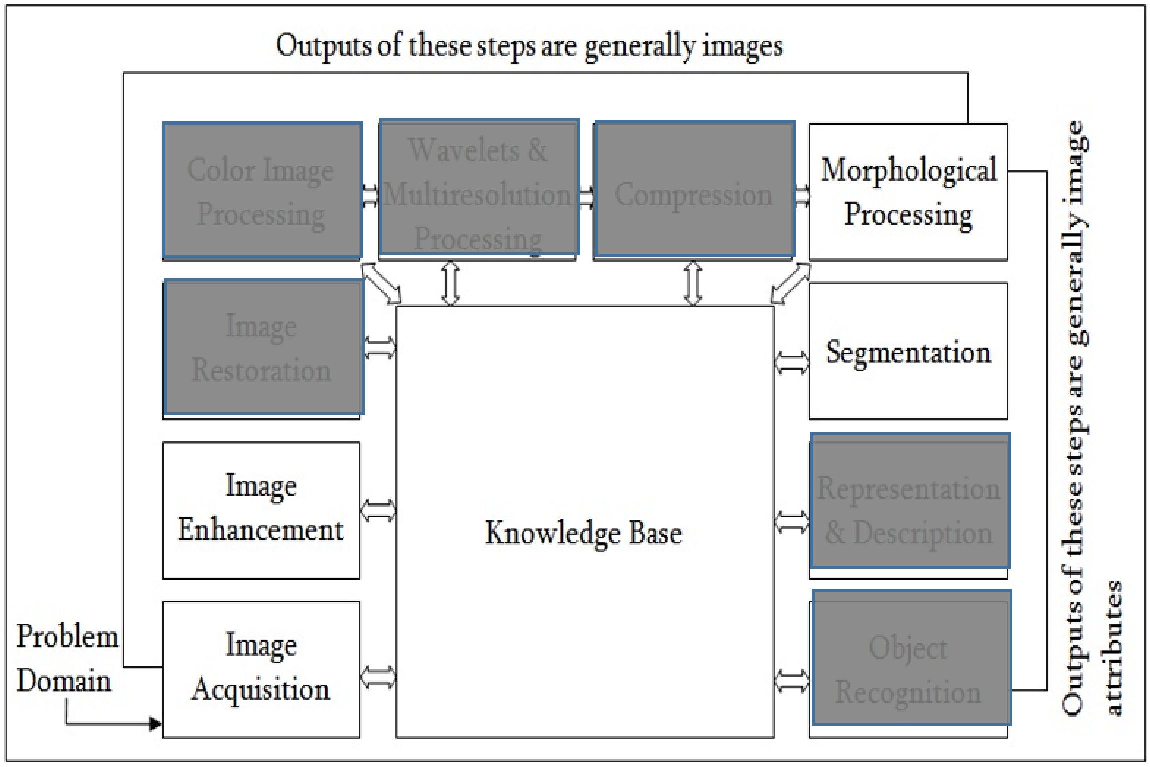


Figure 1*: Flowchart of image processing steps [2].*

In image processing, the user enters an image into a processor, which processes the image and produces a new image (see Figure 2) In the present thesis I describe a software tool I developed which runs the main compute kernels on an Nvidia GPU.

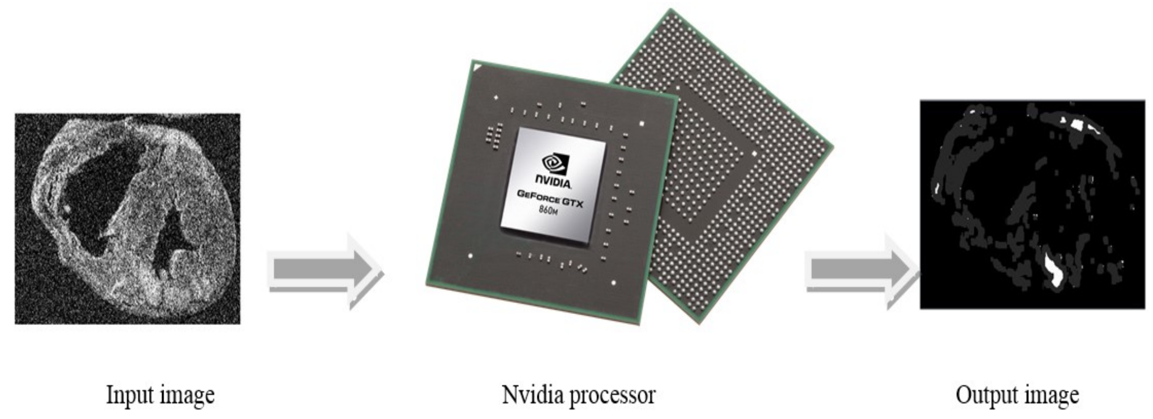


***Figure 2**: The basic technique of image processing.*

The work in this thesis is focused on two areas: 1) morphological processing filters, and 2) image filtering enhancements. Image enhancement is the process of applying filters to an image, resulting in an image where a feature is more apparent and more readily observable than in the original image. The human eye is the judge of image quality and the appropriateness of the filters used. Essentially, the technique of filtering serves to enhance or highlight some obscure details or features of an area of interest. Image filtering enhancement is one of the most attractive and easily applied areas of DIP.

Morphological image processing is a method of extracting an area of interest in an image which is helpful for the illustration and description of an image shape. Mathematical morphological filters are very important DIP tools that are used to address various technical problems. In fact, their outstanding performance is one of the main reasons why they are commonly used in a demanding field such as medicine.

The identification of an area within a digital image at the grey-scale level is a challenging, complex task. For this reason, changing grey-scale to a binary-scale level serves as an approach which can be applied to such problems [4].

1.2 Morphological Filters

There are many difficulties associated with binary image processing applications. Noise easily destroys image quality and texture. This may lead to changes in the image structure and content, resulting in degradation of image clarity, which in turn can pose serious problems for tasks such as image segmentation. Noise in digital images manifests itself in various ways, including Gaussian noise[1] and salt-and-pepper noise[2]. Errors in an image acquisition process are the result of noise and distortion which does not reflect real intensities occurring in the original objects. Figure 3 illustrates degradation of the digital image that can occur due to noise.

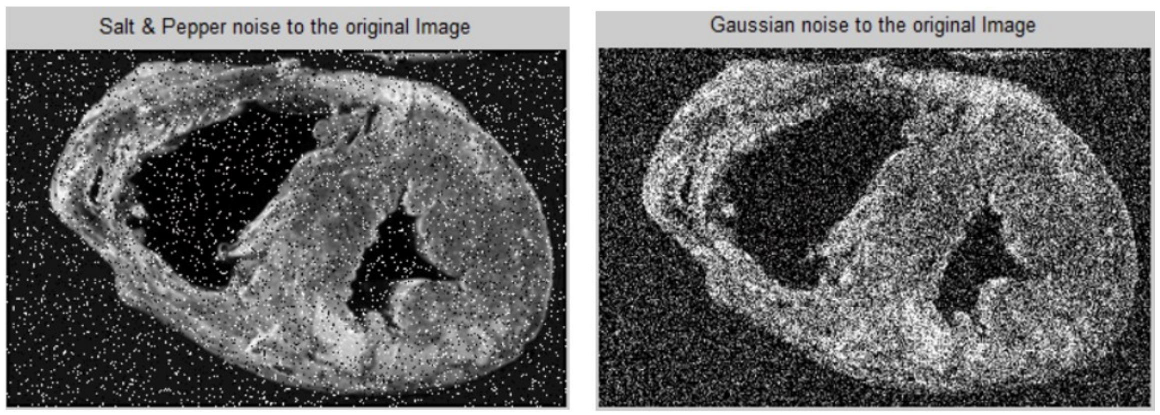

Salt & Pepper noise to the original Image
Gaussian noise to the original Image

***Figure 3**: The image on the left shows the effect of salt-and-pepper noise on the digital image. The image on the right shows the effect of Gaussian noise arising during transmission.*

Digital images may be affected by unwanted noise. Noise arises during the capture, storage, transmission, processing, or conversion of images. Gaussian blur can be used as a solution in order to obtain a smooth grey-scale digital image with less distortion.

[1] Gaussian noise is statistical noise that has a normal probability density function (PDF).
[2] Salt and pepper noise is a noise that occurs in images as black and white pixels.

There are both linear and non-linear approaches to filtering noise. Typical linear filters include Fourier-based low-pass filters, and mean, Gaussian or boxcar filters. Averaging, and median filters are useful approaches for removing noise [5]. Gaussian noise is statistical noise that has a normal probability density function (PDF) (also known as Gaussian distribution). In digital images, Gaussian noise arises during image acquisition and transmission.

There are simple non-linear filters such as the median filter, which are used to reduce noise. There are also much more complex non-linear approaches, such as morphological filters. Morphological image processing filters address noise issues by taking both the shape and structure of the processed image into account. Morphological image processing is a non-linear operation that relies on the ordering of pixel values. A series of non-linear operations are applied to the digital images, whether they are binary or grey-scale images. A key tool used by morphological operations is a structuring element (SE), also referred to as a kernel. It is a combination of sub-images that is applied to an image for some properties of interest. Figure 4 shows various examples of SE that can be used in digital image processing. The origin or center of the SE is always assigned to be 1 [6].

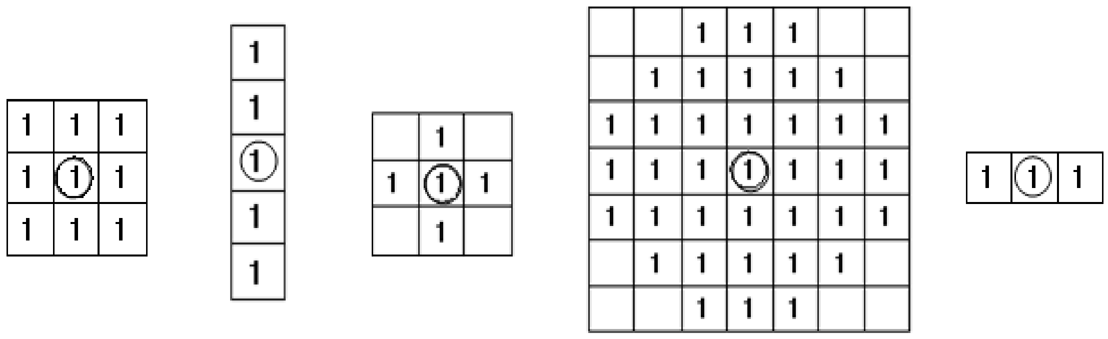


Figure 4*: Some SE shapes, with all centers equal to one.*

The SE interacts with the image to be processed, and is an essential part of any morphological operation, including dilation, erosion, closing, opening, mean and median filtering. It is represented by a matrix of any size or shape. In the present thesis research, the structuring elements used are squares with sizes 3×3, 5×5 and 7×7.

The size of the SE is always much smaller than that of the image being processed. The central value of the kernel, always assigned to be one, defines the neighbourhood where the operation is to take place. Figure 5 shows kernels of different sizes, with zeros or ones around the center. The size of the neighbourhood defines the level of interaction between pixels and hence is in some sense related to the frequency cut-off. The smaller the neighbourhood, the greater the frequency it will pass. In fact, the SE plays the same role as the convolution kernel in linear image filtering.

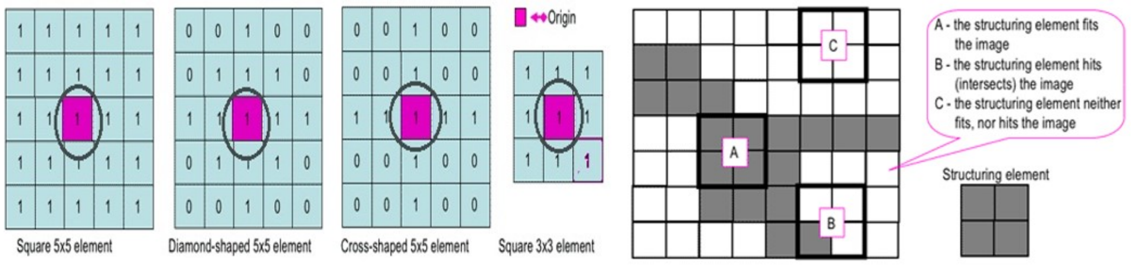

Origin
Square 5x5 element
Diamond-shaped 5x5 element
Cross-shaped 5x5 element
Square 3x3 element
A
B
C
A - the structuring element fits the image
B - the structuring element hits (intersects) the image
C - the structuring element neither fits, nor hits the image
Structuring element

Figure 5: *On the left are 3x3 and 5x5 structuring elements with various values: One is assigned to the origin, with zeros and ones around it. On the right is an illustration of kernel operations.*

Morphological algorithms deal with filters that extract some components from the processed image. These components are helpful for describing and representing the shape or clarity of an image. Morphological image processing is performed by moving the SE center over all of the pixels in the image that is being processed. The size and shape of the SE determine the size and shape of the processed image. The interaction of an image with a SE is described by various morphological operations and algorithms.

Morphological algorithms are used with binary images, and can also be applied to grey-scale images. Binary images are digital images that have no textural content (i.e., grey-scale or colour). The pixels of a binary image have only two possible values: Zero or one, representing the two colours, black and white. Therefore, binary images are referred to as two-level or bi-level images. The digits or bits can be stored in memory as a bitmap, referred to as a bit array [7].

In contrast, the pixels of grey-scale digital images carry intensity information. In this case, black represents the highest intensity and white the lowest intensity, with intermediate intensities being composed of shades of grey.

1.2.1 Dilation

Dilation is one of the two fundamental mathematical morphological operations or filters used in digital image processing. Dilation is basically applied to binary or grey-scale images in order to increase the intensity of bright pixels through correlation with the neighbourhood dark pixels. The dilation operation uses two images: The image being processed, referred to as the active image, and the kernel or structuring element (SE). These images can be represented by sets of pixels. The SE can be different sizes and shapes, which determine a precise modification of the active image. The dilation of the active image, A, by a SE, Z, is denoted by $\boldsymbol{A \oplus Z}$, which depend on the size of the SE. Because *A* and *Z* are two sets in the image being processed (*B^2*), the dilation of *A* by *Z* is defined as in Equation 1.1:

$$\boldsymbol{A \oplus Z = \{b \mid (\hat{Z})_b \cap A \neq \emptyset\}} \qquad \text{Equ1.1}$$

This equation defines the reflection of Z that is obtained from the center of the active image, and is shifted by ***b*.** In other words, the dilation of A by Z is the set of all points b where Z is translated by b, which is contained in ***A*.** The dilation filter generally adds extra pixels to the inner and outer boundaries of the active image A. This increases the size of binary images. Thus, dilation expands and thickens the narrow regions, and changes the features around the edges. Figure 6 shows the steps of the dilation operation.

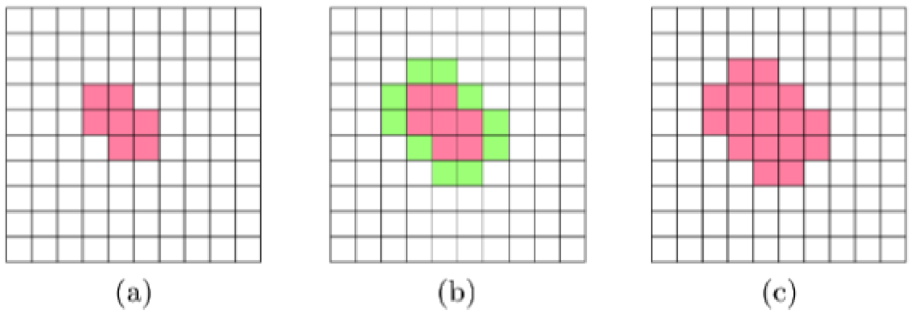


Figure 6: *Illustration of the dilation operation.*

In Figure 6, Panel (a) represents the original image, (b) the image after using the SE of four neighbouring pixels (a cross-shaped SE), and (c) the final result of the dilation operation. A background pixel (zero) is changed to a foreground pixel (one) if and only if the pixel being processed has a foreground pixel as a 4-neighbour. Dilation thus changes the original image by increasing the number of foreground pixels [9].

The dilation of the active image A by the structure element Z can also be performed by displacing all points b in Z which means that the reflection of Z ($\hat{Z}$) and A are overlapped by at least one element. Based on this, Equation 1.2 can be expressed:

$$\boldsymbol{A \oplus Z = \{b \mid (\hat{Z})_b \cap A \nsubseteq \emptyset\}} \quad \text{Equ1.2}$$

An example of dilation of binary images is shown in Figure 7. Here the dilation operation is performed with a SE of size 3×3, set to all ones. The result is shown on the right-hand side of Figure 7. The new pixel values depend upon the pixel values of the neighbourhood that is defined by the SE. Thus, a background pixel is changed to a foreground pixel with a value of 1, if and only if any of the ones of the square SE are contained within the binary image background. If not, then the background pixel remains background, with a value of 0.

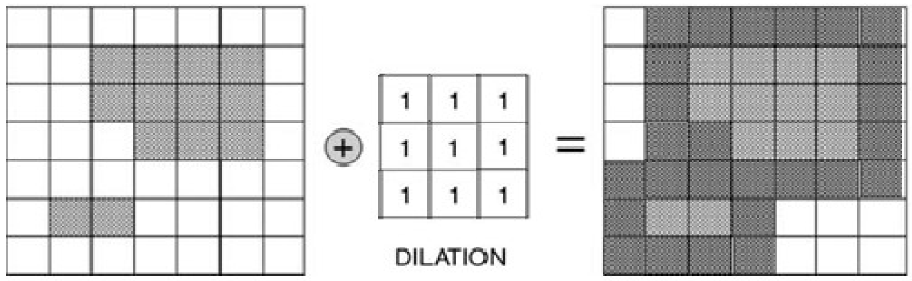


Figure 7: *Dilation with a square structuring element of size 3x3.*

Figure 8 illustrates how the actual operation of dilation occurs via fixed locus, by using the same 3×3 SE. In order to perform dilation on binary images, the origin or center of the SE is superimposed on each pixel, starting at the top left-hand pixel position of the active image. The center of the SE is moved systematically over all of the pixels, until it reaches the bottom right-hand pixel of the active image. The value of the active image pixel being processed becomes one, if any of the neighbourhood pixel values are equal to one. Therefore, the new value of each pixel depends upon the pixel values of the SE neighbourhood. When the center of the SE reaches the last pixel of the active image, some pixels of the SE will be out of range of the border of the image. In this situation, a padding operation must be used, and these pixels are assumed to be set to the value zero [7, 8].

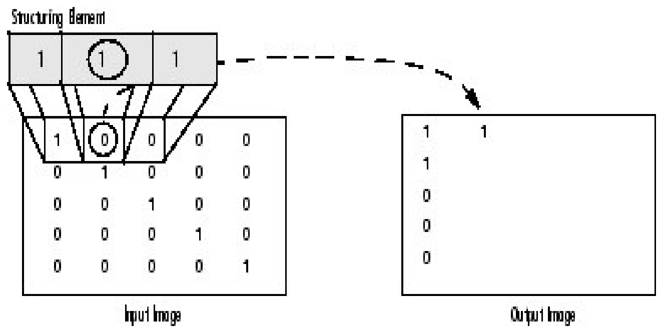


Figure 8*: Morphological dilation of a binary image. (This image is taken from www.mathworks.com) [8].*

In grey-scale morphology, structuring elements have a role similar to that of their counterparts for binary images. A given image is processed to enhance particular properties. The SE used with grey-scale images is also referred to as a structuring function (SF); however, it performs the same operation as in the case of binary images. A grey-scale SF belongs to one of two main categories: Flat (two-dimensional) or non-flat. In this thesis, a flat SF is used for medical images, as described in Chapter 3.

In fact, the use of grey-scale images is very widespread, especially in digital image processing. Most of today's hardware can capture and display 8-bit images. The 8-bit approach permits 256 possible representations of colour gradation. Colour images are most commonly represented by means of the red-green-blue (RGB) system. In comparison to colour images, grey-scale images require less information for each pixel, since shades range only from black to white. In binary images, the luminance is coded by one bit, whereas in grey-scale images, the luminance is represented by n-bits. Grey-scale images are composed of a range of grey shades; where black represents the darkest possible shade and white is the lightest possible shade.

Morphological dilation operations for grey-scale images are similar to those for binary images. The dilation operation of any image, *f*, by any flat SF, *b*, at any pixel location (*x, y*) can be expressed by Equation 1.3:

$$[f \oplus b](x,y) = \max_{(s,t)\in b}\{f(x-s,y-t)\} \qquad \text{Equ.1.3}$$

The two coordinates *x* and *y* represent the location of any pixel in an image. To perform the dilation of a grey-scale image, *f*, the center of a SF, *b,* must be moved over every pixel in the image. The output value of a pixel in the image, *f*, where that pixel coincides with the SE, *b,* is determined by selecting the maximum pixel value of the neighbourhood values for the pixel that is being processed [2].

This process is illustrated in Figure 9, where the original image is dilated by using a square SE, *b*, of size 5×5. Here the changes are clear and noticeable; it can be seen that the original image (shown on the left) has three small black dots, in the middle and lower part of the image, which have been eliminated in the output image (shown on the right). Moreover, the diagonal streaks have been thickened.

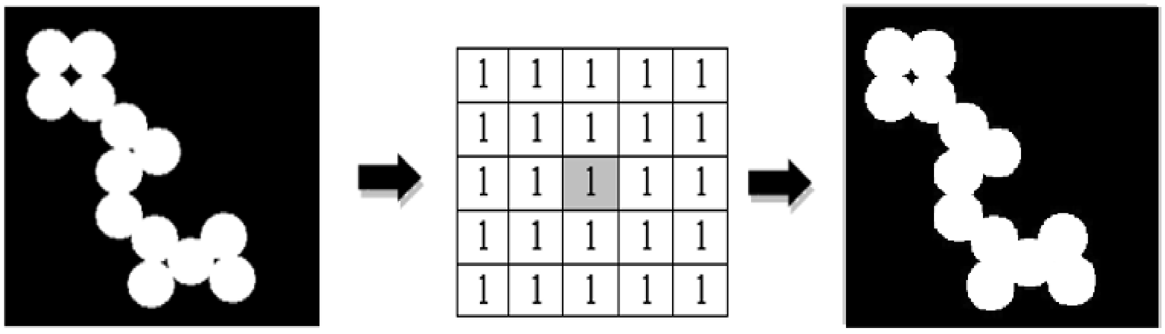


Figure 9*: Illustration of dilation operation on a grey-scale image, with SF of size 5x5.*

If the SF, *b,* is changed to size 3×3, the result is slightly different (see Figure 10). This is obvious in the white circles, where the black pixels in the input image disappear. As described above, the processed image features become thicker. Even though it seems that the dilation operation controls and shapes the result, it should be kept in mind that the prime controller in this process is SF.

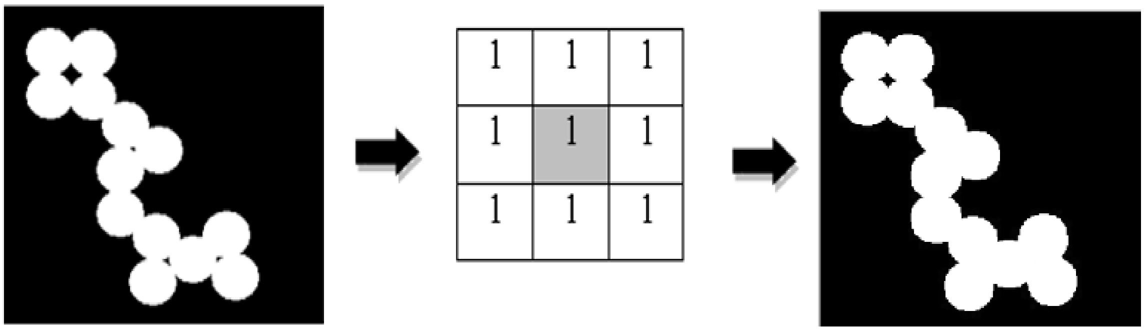
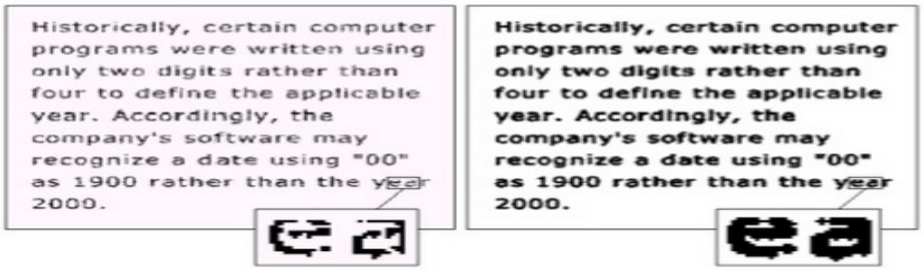
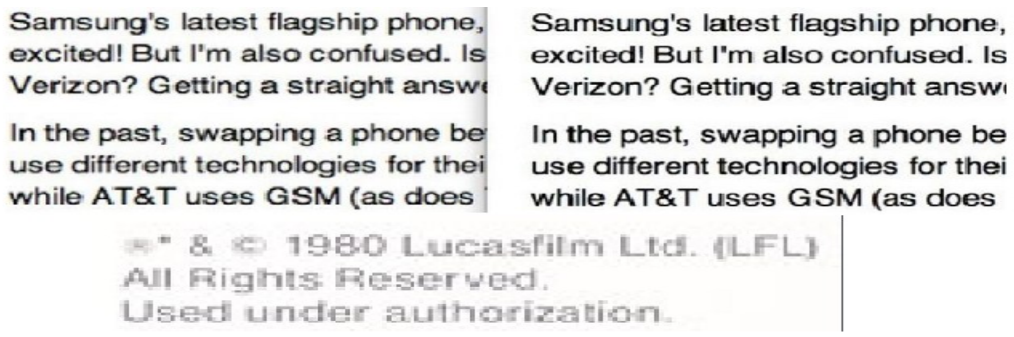


Figure 10*: Illustration of dilation operation on a grey-scale image, with SF of size 3x3.*

There are some other low-pass filter applications of dilation, such as bridging gaps, where dilation is used to enhance broken characters, for example. Figure 11 shows an instance of poor resolution text, as can frequently occur in fax transmissions. Despite the fact that human beings have the ability to read and recognise text with broken characters, devices have real difficulty deciphering such images. Bridging gaps is a solution that can address this type of situation. Figure 11 illustrates a dilation operation where a SE of size 3×3 was used.


Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.



Samsung's latest flagship phone,
excited! But I'm also confused. Is
Verizon? Getting a straight answ

In the past, swapping a phone be
use different technologies for thei
while AT&T uses GSM (as does

Samsung's latest flagship phone,
excited! But I'm also confused. Is
Verizon? Getting a straight answ

In the past, swapping a phone be
use different technologies for thei
while AT&T uses GSM (as does

™ & © 1980 Lucasfilm Ltd. (LFL)
All Rights Reserved.
Used under authorization.

Figure 11*: Illustration of the effect of dilation on a text sample with poor resolution.*

A primary use of dilation is to increase the intensity of bright pixels by correlating each pixel with the neighbourhood dark pixels. As a result of dilation, it can be expected that the size of binary images will increase. As explained in the following section, the opposite happens in the case of erosion. Morphological algorithms are based on the two operations of dilation and erosion.

1.2.2 Erosion

Erosion is the second basic mathematical morphological operation or filter in digital image processing, which employs set theory by considering sets of pixels. These sets represent objects in an image and in morphological operations. The erosion operation, which was originally applied to binary images, is now also used for grey-scale images. Erosion generally decreases the intensity of bright pixels through correlation with neighbourhood dark pixels, as shown in Figure 12. In binary images, erosion shrinks objects in the foreground.

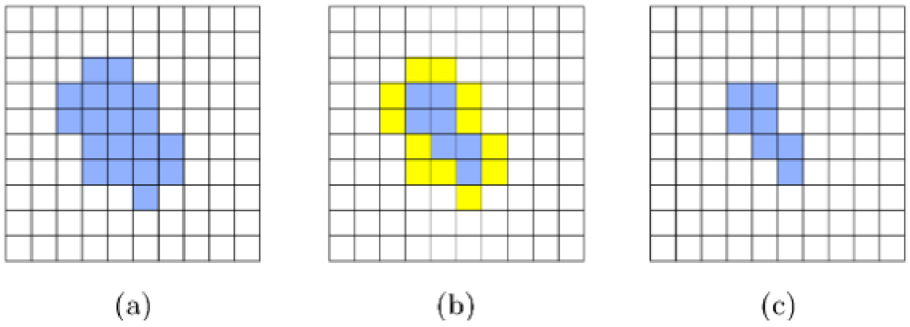


Figure 12*: Illustration of the erosion operation.*

In this figure, panel (a) represents the original image, (b) the image after using the SE of four neighbouring pixels, and (c) the final result of the erosion operation. A foreground pixel is changed to a background pixel if and only if the pixel being processed has a background pixel as a 4-neighbour. Erosion thus changes the original image by decreasing the number of foreground pixels, thus shrinking objects in the foreground [9]. While erosion and dilation have opposite effects, both operations work with two images, or arguments:

1) The input (active) image, whether binary or grey-scale.
2) The kernel or structuring element (SE).

These arguments are represented as sets of objects. Structuring elements of different sizes and shapes determine precise modifications of the active image. The erosion of the active image, *A,* by a SE, *Z,* is denoted by $\boldsymbol{A \ominus Z}$, which produces a new set of images that depend on the size of the SE. Since *A* and *Z* are two sets in *B^2*, the erosion of *A* by *Z* is defined as in Equation 1.4:

$$\boldsymbol{A \ominus Z} = \{\boldsymbol{b} | (\boldsymbol{Z})_{\boldsymbol{b}} \cap \boldsymbol{A^c} = \emptyset\} \qquad \text{Equ1.4}$$

The equation expresses the relationship that the sets of all points b are obtained from the center of the active image A, such that Z is translated by b, that is enclosed in A. Z is assumed to be the SE, A^C is the complement of the active image or set that needs to be processed, A; and $\emptyset$ is an empty set. Since Z must be included in A, Equation 1.4 can be expressed as shown in Equation.5:

$$\boldsymbol{A \ominus Z} = \{\boldsymbol{b} | (\boldsymbol{Z})_{\boldsymbol{b}} \subseteq \boldsymbol{A}\} \qquad \text{Equ1.5}$$

This equation provides another formulation. In fact, the erosion of the active image, A, by the SE, Z, can be performed by displacing all the points of set b over Z. Since the size of the SE affects the size of the processed image, all the points of set b must be located in the SE. Thus, erosion and dilation filters perform opposing operations. Erosion removes some pixels from the inner and outer boundaries of the active image, A. Erosion decreases the size of binary images, shrinking and thinning the narrow regions, and changing the features around the edges.

Figure 13 shows the effect of erosion on a binary image, using an SE of size 3×3. The SE is moved over the binary image so that the center SE pixel coincides with each background or foreground pixel. The result of the operation is shown on the right-hand side of Figure 13. The pixel values of the new image depend upon the pixel values of the neighbourhood that is defined by the SE. Thus, a foreground pixel remains a foreground pixel, with a value of 1, only if the ones in the SE are all contained within the binary image foreground. If not, then the foreground pixel is changed to a background pixel, with a value of zero.

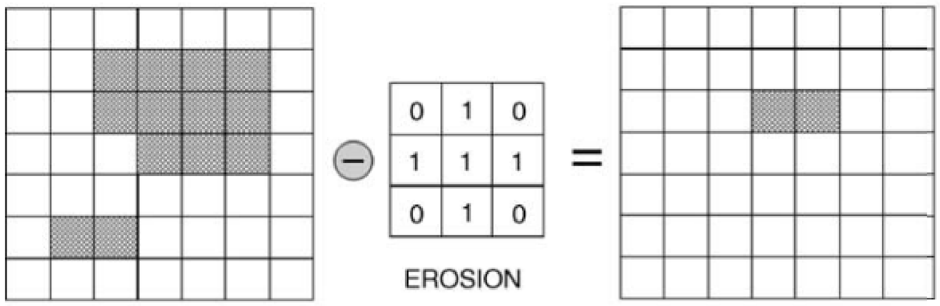


Figure 13*: The effect of erosion on a binary image, with an SE of size 3x3.*

Figure 14 illustrates how the actual operation of erosion occurs via fixed logic, by using the same 3×3 SE. In order to perform erosion on binary images, the origin or center of the SE is placed on each pixel, starting at the top left-hand pixel position of the active image. The center of the SE is moved over all of the pixels, until it reaches the bottom right-hand pixel of the active image. The value of the active image pixel being processed becomes one, if all of the neighbourhood pixel values are equal to one. Otherwise, it remains zero. Therefore, the new value of each pixel depends upon the pixel values of the SE neighbourhood. As it was discussed in dilation, when the center of the SE reaches the last pixel of the active image, some pixels of the SE will be out of range, or on the border of the image. In this situation, a padding operation must be used, and these pixels are assumed to be set to the value zero.

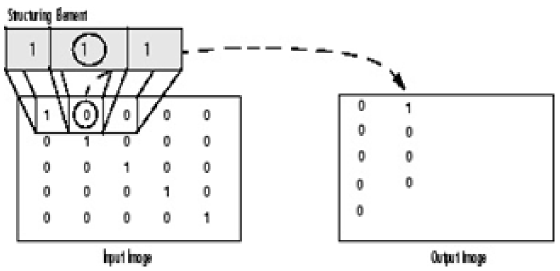


Figure 14*: Morphological erosion of a binary image.*

Erosion operations for grey-scale images are similar to those for binary images. The erosion operation of any image, *f*, by any flat SF, *b*, at any pixel location (*x, y*) can be expressed by Equation 1.6:

$$[f \ominus b](x,y) = \min_{(s,t)\in b}\{f(x+s, y+t)\} \qquad \text{Equ.1.6}$$

In a manner similar to the dilation operation discussed in Section 1.2.1, the two coordinates *x* and *y* represent the location of any pixel in an image. To perform the erosion of a grey-scale image, *f*, the center of a SF, *b*, must be moved over every pixel in the image. The output value of a pixel in the image, *f*, where that pixel coincides with the SF, *b,* is determined by selecting the minimum pixel value of the neighbourhood values for the pixel that is being processed [2].

This process is illustrated in Figure 15, where the original image is eroded by using a square SE, *b,* of size 5×5. Here the changes are noticeable. It can be seen that the original image (shown on the left) has three small black dots, in the middle and lower part of the image; in the output image (shown on the right), these dots have become larger. Moreover, the diagonal streaks have been thinned.

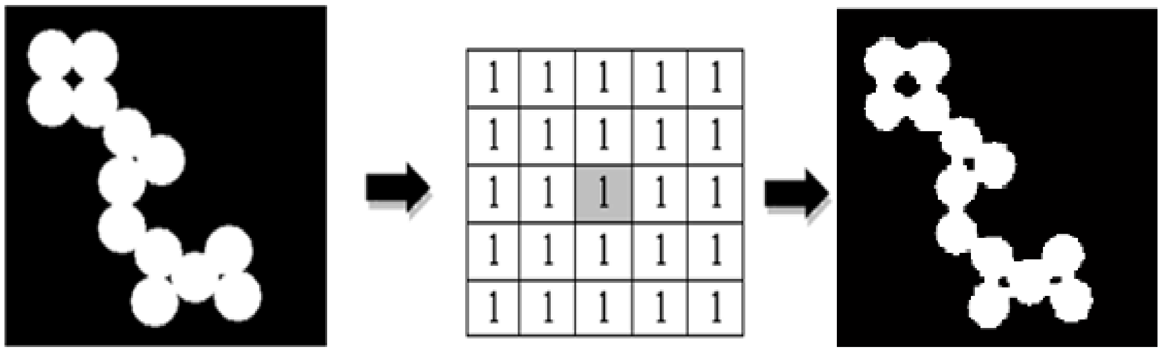


Figure 15: *Illustration of erosion operation on a grey-scale image, with SF of size 5x5.*

If the SF, *b*, is changed to size 3×3, the result is slightly different (see Figure 16). This is obvious in the vicinity of the white circles, where the black pixels have shrunk. As described above, the processed image features become thinner. Even though it seems that the erosion operation controls and shapes the result, it must be kept in mind that the main controller in this operation is the SF.

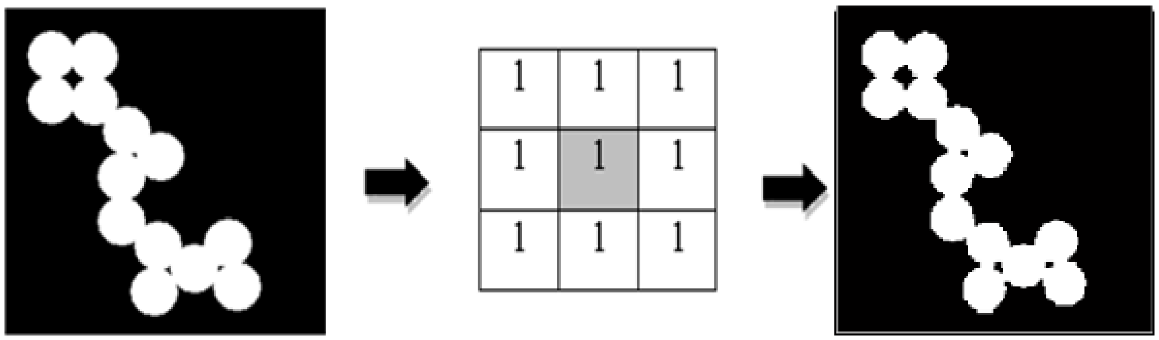


Figure 16: *Illustration of erosion operation on a grey-scale image, with SF of size 3x3.*

For an additional illustration of the effect of using erosion on digital images, a colour image was used, as shown in Figure 17. It can be seen that dark areas of the active image were enhanced, and the original image was eroded. For each pixel, the erosion filter correlates all pixel and luminosity values with the darkest pixel of the neighbourhood pixels. As a result, dark pixels are added to dark areas and the pixel being processed may be changed to a higher value, which creates some noise in the processed image.

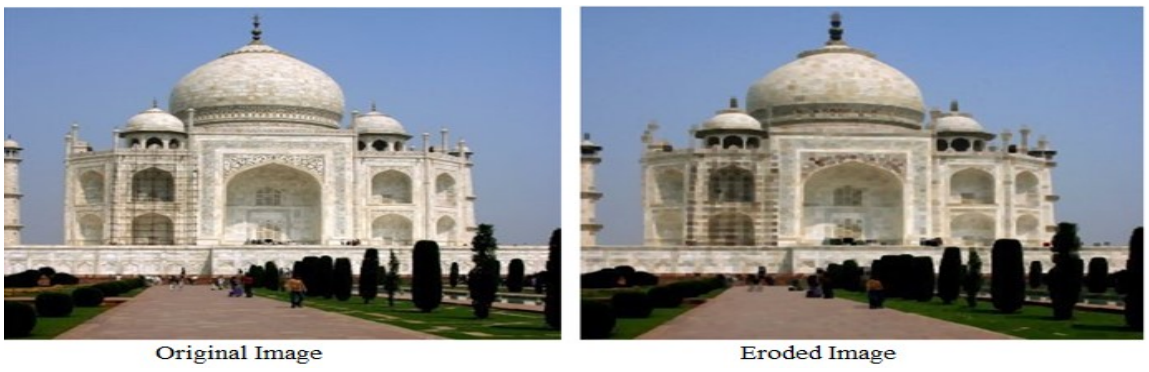


Figure 17: *The effect of using an erosion filter on a colour image.*

Lastly, the primary function of erosion is to decrease the intensity of bright pixels though correlating each pixel with neighbourhood pixels. As a result of erosion, it can be expected that the size of binary images will decrease. As described in this and the previous section, the components in a digital image are generally expanded by dilation and shrunk by erosion. The effects of dilation and erosion filters on digital images are summarised in Table 1.

***Table 1**: Summary of dilation and erosion operations.*

Erosion	Dilation
Depends on the structuring element used.	Depends on the structuring element used.
If all SE values are flat, results tend to be darker.	If all SE values are flat, results tend to be brighter.
Dark details are either enlarged or added.	Dark details are either reduced or eliminated.

1.2.3 Opening Filter

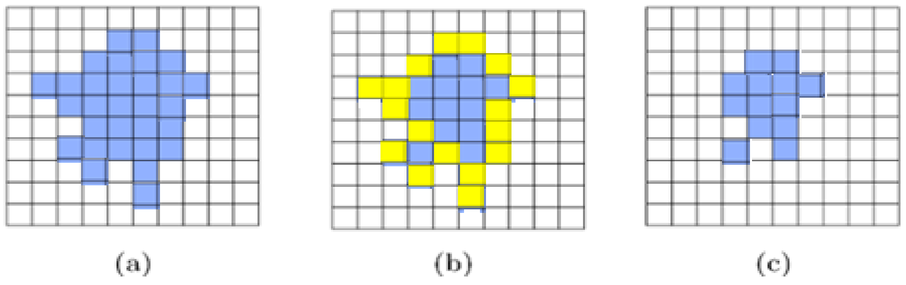
Other filters, such as the opening filter, can be built from dilation and erosion filters. From a morphological perspective, the opening filter is an algebraic operation which can be extended to grey-scale images. The opening operation is a combination of two filters: Erosion followed by dilation. The basic operation of the opening filter is to eliminate small objects and dark pixels from the foreground, while large objects are to remain unchanged. Small objects are removed from the foreground and placed in the background of the processed image. In fact, in its main effect the opening filter is similar to the erosion filter, which tends to eliminate some of the bright foreground pixels on the edges. However, the erosion filter is more destructive than the opening filter [11].

The opening filter generally smooths corners from the inside, removes thin bumps, and breaks straight isthmuses. The opening operation filters a binary or grey-scale digital image, at a scale that is defined by the size of the main controller, which is the SE. The size of the SE is an important factor in removing noisy details and keeping the objects of interest.

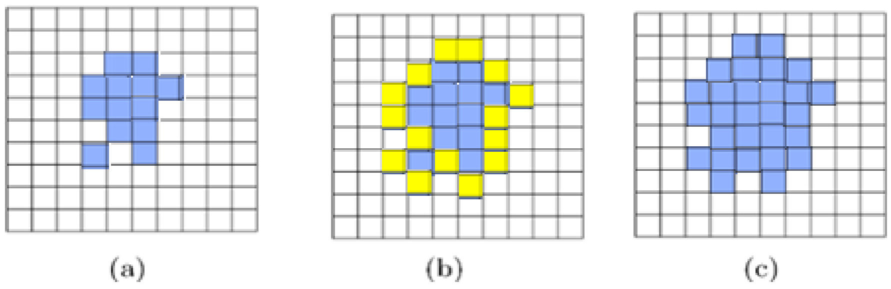
The opening of an active image, *A*, by a SE, *Z*, is denoted by $\boldsymbol{A \circ Z}$ and is dependent on *Z*. The opening operation produces a new set of images, as defined in Equation 1.7:

$$\boldsymbol{A \circ Z = (A \ominus Z) \oplus Z} \qquad \text{Equ1.7}$$

Thus, the opening of *A* by *Z* is simply the erosion of a digital active image, *A*, by SE, *Z*, followed by a dilation of the result by *Z*. In other words, the opening operation applies the erosion filter to the image of interest by using a SE of a specified size, then applies the dilation filter to the result by using the same SE. The opening filter requires two inputs, the SE and an image to be opened. It eliminates stray foreground pixels by moving them into the background. Figure 18 illustrates the steps of opening filter operation. In Phase 1, the erosion filter is applied to the region of interest. Here (a) represents the original image, (b) shows the application of the erosion filter, and (c) shows the result of the erosion operation. In Phase 2, the dilation filter is applied to the result of Phase 1.



Phase 1: Application of erosion filter.



Phase 2: Application of dilation filter to the result of Phase 1.

Figure 18: *Illustration of the opening operation, with a cross-shaped SE of size 3×3. Phase 1 deletes some pixels, and Phase 2 adds pixels.*

Figure 19 provides a geometrical illustration of the details of the operation of the opening filter. Assuming that Z is a flat SE, represented by a rolling ball, and A is a set of pixels, the boundary of the opening filter, $\boldsymbol{A \circ Z}$, is created by all the points Z that are reached along the inner boundary of A as Z is moved inside this boundary. Subsequently, the opening of A by Z can be obtained by taking the union of all Z points that translate and fit into A. ***A***

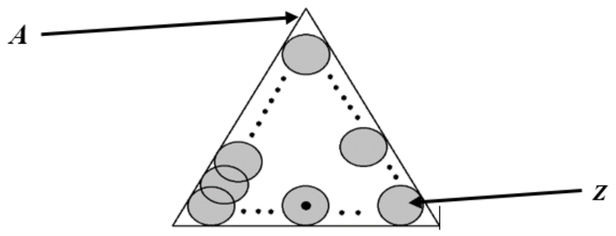


Figure 19: *Geometrical illustration of operation of opening filter.*

In the triangle A, Z is represented by a ball that moves along the inner boundary of A. Since the opening operation on an image starts with the erosion operation, all regions that are smaller than the size of the SE will be eliminated. Once this operation is followed by dilation, any bright regions that are similar to or larger than the size of the SE will retain their original size. The opening operation is expressed as in Equation 1.8:

$$\boldsymbol{A} \circ \boldsymbol{Z} \; = \cup \{(\boldsymbol{Z})_{\boldsymbol{b}} \,|(\boldsymbol{Z})_{\boldsymbol{b}} \subseteq \boldsymbol{A})\} \qquad \text{Equ1.8}$$

Equation1.8 represents the union, ∪, between *A* and *Z*. The opening filter is obtained by taking the union of translates of *Z* into *A*. Combining Figure 19 and Equation 1.8 yields the result shown in Figure 20. The shaded area inside triangle *A* represents the result of applying the opening filter, $\boldsymbol{A} \circ \boldsymbol{Z}$.

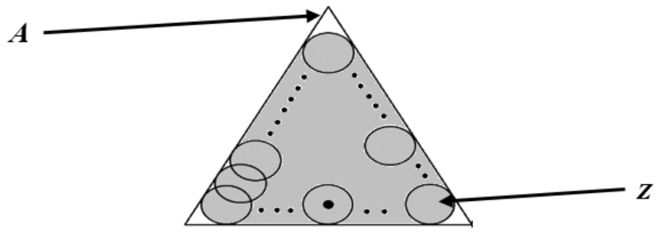


Figure 20*: Illustration of result of opening operation.*

At the grey-scale level, the opening filter operates in a manner similar to its operation at the binary level. Figure 21 illustrates the operation of an opening filter on a grey-scale image. Here it can be seen clearly that the opening filter has removed background noise.

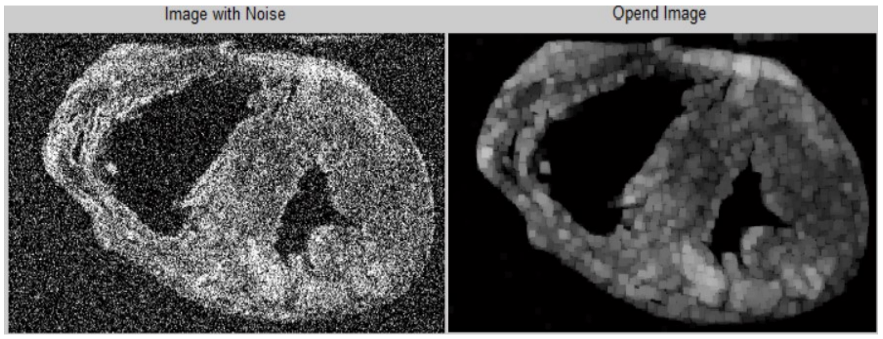


Figure 21*: Illustration of opening operation on a grey-scale image.*

A grey-scale opening filter consists of grey-scale erosion followed by grey-scale dilation. Erosion is used to eliminate the noise associated with undesirable foreground pixels; however, it has the disadvantage that it could potentially affect all foreground pixels. The opening operation overcomes this disadvantage by applying both erosion and dilation to the digital image. In the case of erosion, the position of the SE origin is important, and the result depends upon it. In contrast, the result of the opening operation is independent of the position of the SE origin.

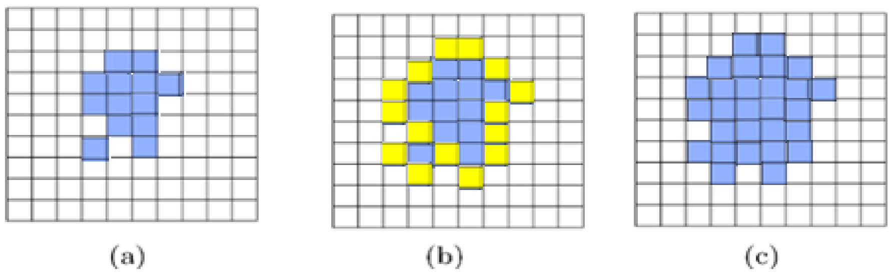
1.2.4 Closing Filter

The closing morphological mathematical operation is one of two dual mathematical filters: Opening and closing. The closing operation can be applied to a binary image, and can be extended to a grey-scale image. The closing filter, derived from the fundamental operations of dilation and erosion, has an effect opposite to that of the opening filter. The closing operation is a combination of two filters: Dilation followed by erosion. It expands the foreground boundaries and bright pixels in the regions examined, and shrinks background pixels and holes in the same regions. In other words, it fills in holes in these regions. The closing filter has an effect similar to that of the dilation filter. However, dilation is more destructive to the boundary shape.

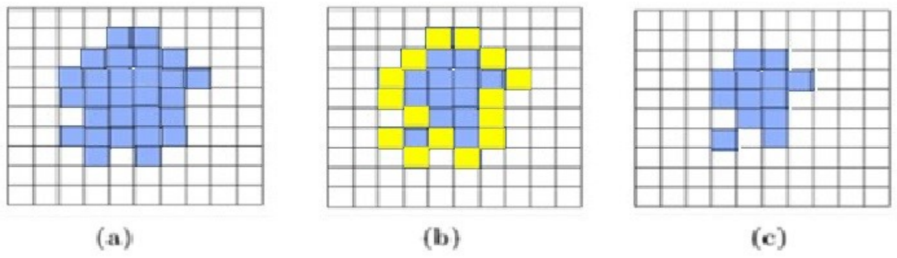
The closing filter is sometimes used to fill in a particular background region of the processed image. The closing filter is defined by the size of the SE. Therefore, the size of the SE is important in removing noisy objects and retaining objects of interest. The closing of a digital image, *A*, by a SE, *Z*, is denoted by $\boldsymbol{A \bullet Z}$, and is dependent on *Z*. The closing operation produces a new set of images, as defined in Equation 1.9:

$$\boldsymbol{A \bullet Z = (A \oplus Z) \ominus Z} \qquad \text{Equ1.9}$$

Thus, the closing of A by a SE, Z, is achieved by the dilation of an active digital image, A, by SE, Z, followed by the erosion of the result by Z. In other words, the closing operation applies the dilation filter to the image of interest by using a SE of a specified size, then applies the erosion filter to the result by using the same SE. Like the other filters, the closing filter requires two main inputs, the SE and an image to be processed. It enlarges the boundaries of foreground pixels and eliminates the background holes. Figure 22 illustrates the steps of closing filter operation. In Phase 1, the dilation filter is applied. Here panel (a) represents the original image, (b) shows the application of the dilation filter, and (c) shows the result of the dilation operation. In Phase 2, the erosion filter is applied to the result of Phase 1. In this illustration, the SE used is cross-shaped, with size 3×3.



Phase 1: Application of dilation filter.



Phase 2: Application of erosion filter to the result of Phase 1.

Figure 22*: Illustration of the closing operation, with a cross-shaped structuring element of size 3x3.*

Figure 23 provides a geometrical illustration of the operation of the closing filter.

Assuming that Z is a flat SE, represented by a rolling ball, and A is a set of pixels, the boundary of the closing filter, $\boldsymbol{A \bullet Z}$, , is created by all the points Z that are reached along the outer boundary of A as Z is moved outside this boundary. Subsequently, the closing of A by Z can be obtained by taking the union of all Z points that translate and fit into $\boldsymbol{A}$**.**

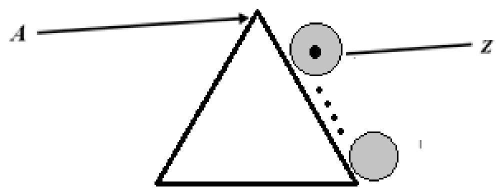


Figure 23*: Geometrical illustration of operation of closing filter.*

In the triangle A, Z is represented by a ball that moves along the outer boundary of A. Since the closing operation on an image starts with the dilation operation, all regions that are smaller than the size of the SE will be enlarged. Once this operation is followed by erosion, any bright regions that are similar to or larger than the size of the SE will be shrunk. The closing operation is expressed as in Equation 1.10:

$$\boldsymbol{A \bullet Z} = \cup \{(\boldsymbol{Z})_{\boldsymbol{b}} \,|(\boldsymbol{Z})_{\boldsymbol{b}} \subseteq \boldsymbol{A})\} \qquad \text{Equ1.10}$$

Equation1.10 represents the union, ∪, between A and Z. The closing filter is obtained by taking the union of translates of Z into A. Combining Equation 1.10 and Figure 23 yields the result shown in Figure 24. The shaded area inside triangle A represents the result of applying the closing filter.

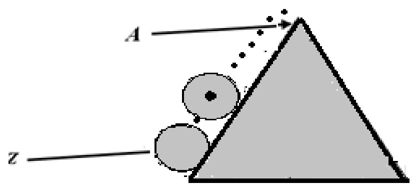


***Figure 24**: Illustration of result of closing operation.*

At the grey-scale level, the closing filter operates in a manner similar to its operation at the binary level. Figure 25 illustrates the effect of applying the closing filter to a grey-scale image. Here it can be seen that the closing filter has removed noise from the image. Although some image details have been lost, the result is still better than the original noisy image.

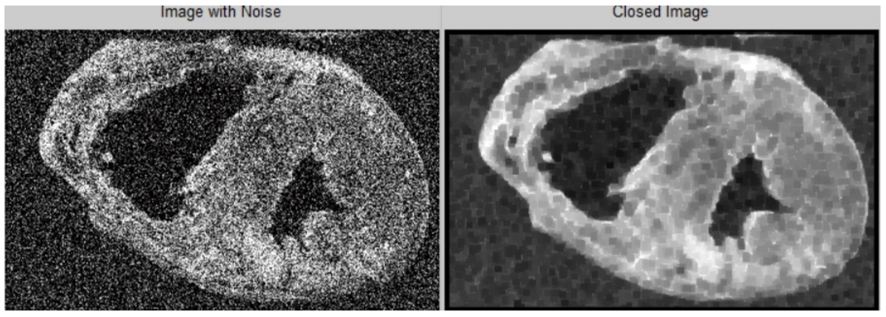


***Figure 25**: Illustration of closing operation on a grey-scale image.*

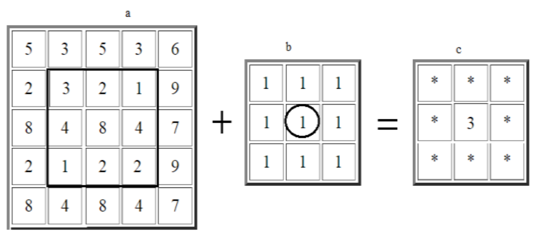
A grey-scale closing filter consists of grey-scale dilation followed by grey-scale erosion. Dilation is used to fill in small background holes; however, it has the disadvantage that it could potentially distort all regions of background pixels. The closing operation overcomes this disadvantage by applying dilation to the digital image, followed by erosion.

1.2.5 Mean Filter

The mean filter is another filter commonly used to reduce noise in medical images. It is used to smooth digital images and to reduce intensity variations between a pixel and its neighbours. In the case of a large mean filter, there are a number of large neighbourhoods, which means intensive smoothing. Mean filtering is performed by moving the SE over all pixels in the active image. A SE of size 3×3 means that each pixel has eight neighbours. The origin of the SE is moved over the image, where the origin fits entirely within the image boundaries. The processing software usually starts at the top left-hand corner of the active image. The value of the pixel being processed will be replaced by the average value of the eight neighbouring pixels, for an SE of size 3×3. The mean filter averages the pixel value intensity in a region of interest. Assuming a matrix of size $m \times n$ (here $m=n$), the mean filter algorithm can be expressed as shown in Equation 1.11 [12].

$$x(i,j) = \frac{1}{mn} \sum_{\mathrm{k} \in \mathrm{m}} \sum_{L \in n} f(k, L) \qquad \text{Equ1.11}$$

The variables m and n represent the size of the SE or kernel. For example, the SE size can be 3×3, 5×5 or 7×7. A convolution operation is used in calculating the average. Figure 26 shows an area of interest from an image. At the top of Figure 26, the array on the left (a) represents the image entered, the central array (b) represents a square SE of size 3×3, and the right-hand array (c) represents the result of applying the mean filter. The matrix (d) below demonstrates the operation of the mean filter.



d

$$g(i,j)=\frac{1}{3\times 3}\begin{bmatrix} 3 & 2 & 1 \\ 4 & 8 & 4 \\ 1 & 2 & 2 \end{bmatrix}$$

Figure 26*: Application of mean filter, with SE of size 3x3.*

Each SE transition produces a single output pixel. The value of the output pixel is calculated by adding together and averaging the image pixel values for each cell. If all the pixel values of the indicated area of array (a) are added together and divided by 9 (for an SE of size 3×3), the result of this averaging is: 27/9=3. The amount of filtering can be controlled by the size of the SE. The larger the SE, the more blurred the result will be. Figure 27 shows the effect of using a mean filter with different kernel or SE sizes.

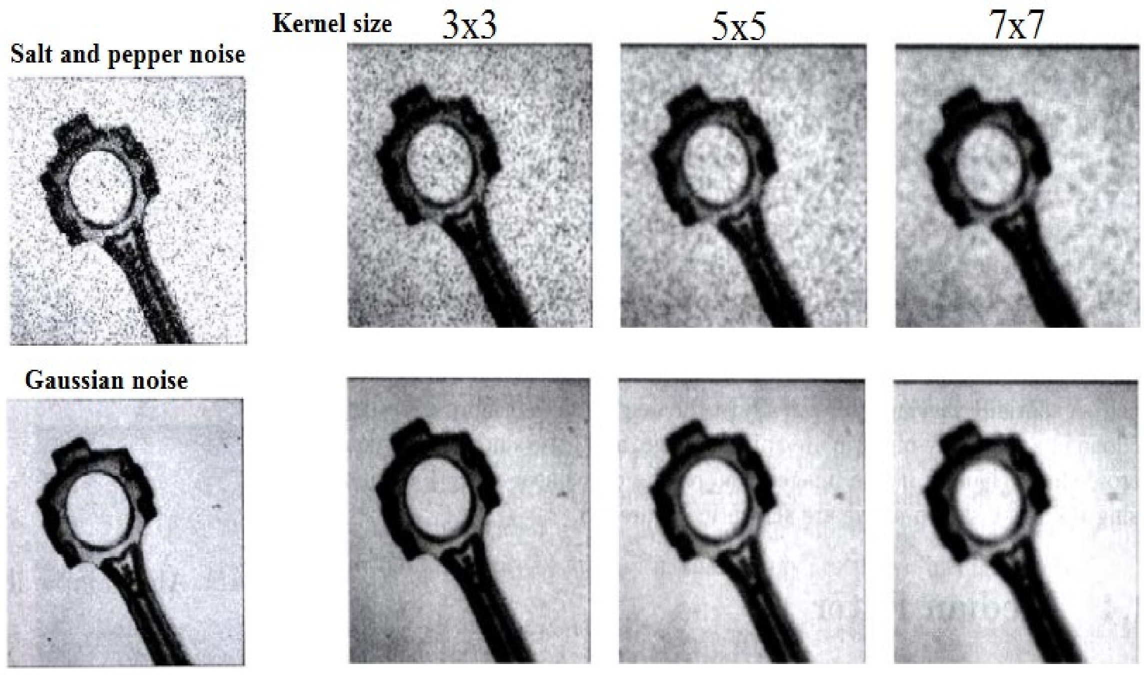


Figure 27*: Illustration of the effect of applying a mean filter with different SE sizes.*

Mean filtering is used to smooth images. Increasing the size of the SE decreases image noise, however image details will be reduced. As shown in Figure 27, if the SE size is large, for example 7×7, the resulting image will be blurred and some image details will be lost. Thus, if the SE is too large, mean filtering will not produce a significant improvement in noise reduction and, furthermore, the image will be very blurred.

A significant variation between the value of the pixel being processed and that of a neighbouring pixel can also have an adverse effect on the result of mean filtering. For example, Figure 28 illustrates mean filtering with a square SE of size 3x3. Here it can be seen that there is a very unrepresentative pixel value (58) which will affect the result of mean filtering; this unrepresentative value arose due to noise.

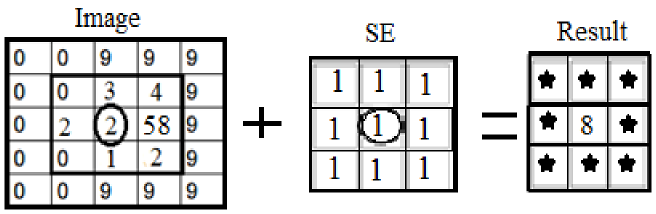


Figure 28: *Illustration of the effect of applying a mean filter to an image that has a single pixel with a very unrepresentative value.*

This unrepresentative pixel value (58) will lead to a blurred result, which creates a problem if the user requires an output image with sharp edges. This problem can be addressed by median filtering, described in the following section; although often more effective than mean filtering in reducing noise, median filtering requires longer computation times.

1.2.6 Median Filter

The median filter is a non-linear spatial filter that is used to reduce noise and smooth images. Smoothing a digital image means reducing the variation in intensity between a pixel and its neighbours. The median filter is well-suited to removing outlier noise such as salt-and-pepper noise, and is more effective than the mean filter in preserving edges.

For two non-linear sets, X and Y, the median value of {X+Y} is not equal to the median value of {X} + the median value of {Y}. For example, as shown in Figure 29, for two sequences of length 3, the median of X is equal to 1, the median of Y is equal to 1, and the median of X+Y is equal to 3.

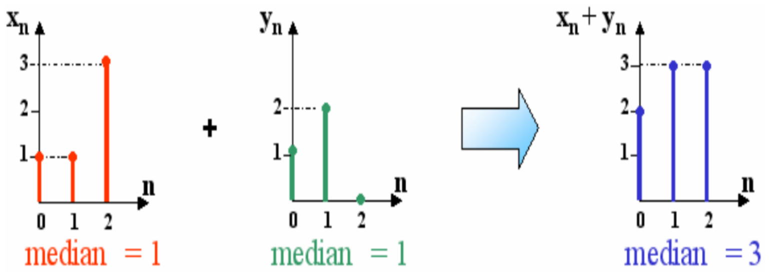


Figure 29*: A non-linear operation: The median of X plus the median of Y is not equal to the median of {X+Y} [19].*

For intensive smoothing, a large median filter is necessary to achieve better results. As in the case of mean filtering, median filtering is performed by moving the SE over all of the pixels in the active image, from the top left-hand corner to the bottom right-hand corner. With a SE of size 3×3, each pixel has eight neighbours. The median operation is started by moving the origin of the SE over the image pixels, where the center SE pixel is completely within the image boundaries. The value of the pixel being processed will be replaced by the median value of the eight neighbouring pixels. In median filtering, the output pixel value is calculated as follows:

1. Sort the values of all eight neighbouring pixels into numerical order, and select the middle value.
2. Replace the value of the pixel being processing with the middle value.

The calculation of the median value used in median filtering is illustrated in Figure 30. First the intensity values of all eight neighbouring pixels are sorted into numerical order: 1, 1, 2, 2, 2, 3, 4, 4, 8. The median value, which is the middle value of the sorted values is then selected: 2. In Figure 30, the array on the left represents the original image, the array in the middle represents a square SE of size 3×3 with all ones, and the right-hand array shows the value of the output pixel obtained by applying the median filter [18].

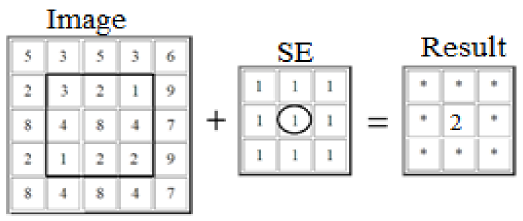


***Figure 30**: Median filtering: Calculation of the value of an output pixel, with SE of size 3x3.*

Within the region of interest, the median filter replaces each pixel value with the median value of the neighbouring pixels. Median filtering is an effective operation for reducing noise and preserving image details. Figure 31 shows the effect of using a median filter with different kernel or SE sizes.

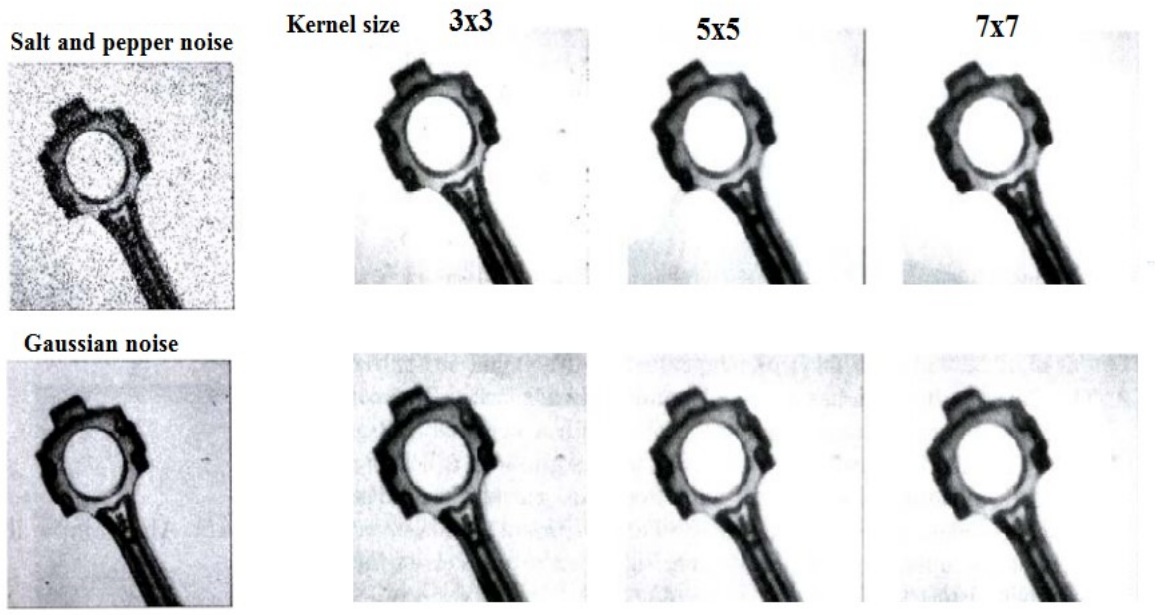


Figure 31: *Illustration of the effect of applying a median filter with different SE sizes.*

Like mean filters, median filters smooth images; however, median filtering is more effective in preserving image details. If the size of the SE is increased, the output will be smoother. The median filter is more effective in improving images and removing noise than other filters are. As shown in Figure 32, in the case of median filtering, the presence of a single unrepresentative pixel value among the neighbouring pixels does not have a significant impact.

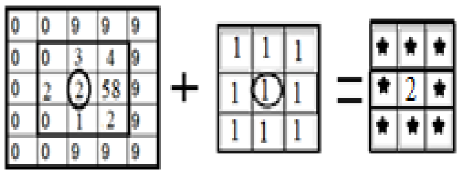


Figure 32: *Illustration of the effect of applying a median filter to an image with a single unrepresentative pixel value.*

Here it can be seen that the unrepresentative pixel value (58) does not affect the result. However, although median filtering is more effective in reducing noise, it requires a longer time to perform computations and produce the processed image than is the case with mean filtering. Figure 33 provides a comparison of the results of applying mean and median filtering to an image.

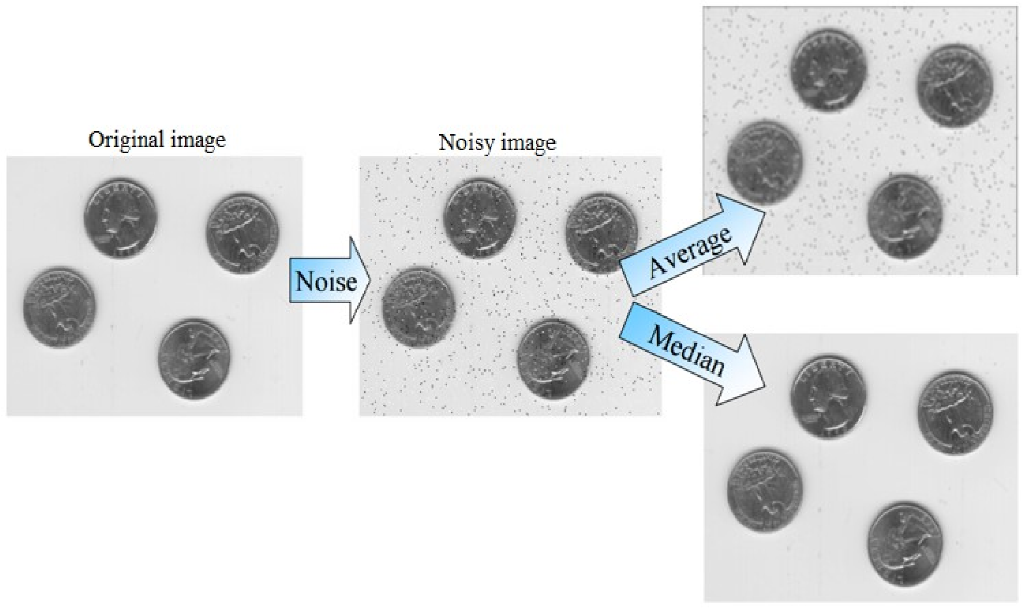


Figure 33*: The results of denoising an image by applying mean (average) and median filtering.*

Figure 33 and Table 2 illustrate the main differences between mean and median filters. The mean filter is computationally simple. It blurs the result, but works with all types of noise. The median filter is an ideal choice for the elimination of salt-and-pepper noise.

Table 2*: Summary of the main differences between mean and median filters.*

Mean filter	**Median filter**
Simple and effective on all types of noise.	Complex, but very effective for salt-and-pepper noise.
Blurs images, causing image details to be lost.	Preserves images detail.

1.3 Graphics Processing Unit/Graphics Processing General Purpose Computing On Graphical Processing Units (GPU/GPGPU)

The first generation of General-purpose computing on graphics processing units programming was carried out using a shader-based framework; programmers had to learn the details of rendering pipelines and graphics programming in order to use this framework. A pipeline is a series of steps used by programmers to transform a 3D model into a 2D representation. Graphics pipelines are implemented by two types of hardware: GPUs and CPUs, with software such as OpenGL and DirectX. Initially, GPGPU algorithms were complex and had to be mapped into pixel lightning or 2D/3D vertex transformations. However, in 2007, Nvidia released the Compute Unified Device Architecture (CUDA) that forms the basis for Nvidia hardware. CUDA utilises an extension of the C programming language, permitting programmers to run programs directly on GPUs. Nvidia cards are designed for and dedicated to the use of GPUs [14].

A GPU is optimised for accelerating graphical computations, and its architecture is designed for processing vast amounts of data. General-purpose computing on graphics processing units (GPGPUs) was not fully supported by all programming languages before 2007. This meant that graphics had to be handled by the controller of the video graphics array (VGA), which organises image pixels and sends them to the output device [14]. Today, GPUs are used in a wide range of applications and devices.

In 2003, the floating point performance of CPUs and GPUs was comparable, however by the end of 2006, when Nvidia G80 cards were produced, GPU processing time was about 60 times faster than that of CPUs (with Intel Core 2 Duo microprocessors) [13]. To explore the speed benefits of GPUs, a non-equispaced fast Fourier transform was run on a GeForce 8800 GTX Nvidia card. The results showed that the acceleration could be boosted by over 400%, which improved thread scheduling, data instructions and memory access patterns. The optimisation of the convolution-interpolation kernel on a GeForce 8800 GTX Nvidia card (GPU) was about 110 times faster than the same kernel that was run on an Intel Core 2 Duo microprocessor (CPU) [14].

This remarkable improvement encouraged researchers to begin processing medical scans such as magnetic resonance imaging (MRI) and X-ray scans on GPUs. However, there was a problem with using C or OpenGL programming languages, which were initially not designed for general-purpose computing. Therefore, in 2007, Nvidia released the CUDA programming language, which involves a new set of instructions and new architecture. The availability of CUDA encouraged researchers to use GPUs as their main computing devices, which has led to the establishment of CUDA as the standardised architectural model and framework of many applications.

GPUs have thus developed from being only used for gaming tasks to being employed for many computational processing applications. Now GPGPU uses the Nvidia processor, which has a device compiler (nvcc), instead of using only the host (CPU) compiler. Modern graphics processing units are designed for high performance and fast processing time. Because these GPUs are priced for consumers, they provide an attractive platform when fast processing time is a primary requirement.

Most modern digital image processing applications in the neuroscience area are computationally rich in image manipulation processing techniques, such as those required to process CT scan, X-ray and MRI images. For example, MRI studies require powerful real-time multiprocessor systems to process the complex interactions of the digital images.

Although these powerful multiprocessor systems may be expensive for researchers, they provide high performance, reduced processing time, and sufficient results in real time.

For such applications, the use of modern improved graphics processing units (GPUs) now far outweighs that of central processing units (CPUs), especially when real-time processing performance is a concern.

The main difference between a CPU and a GPU is that a CPU has fewer arithmetic logic units (ALUs), caches, and control units than is the case with a GPU. This is illustrated in Figure 34.

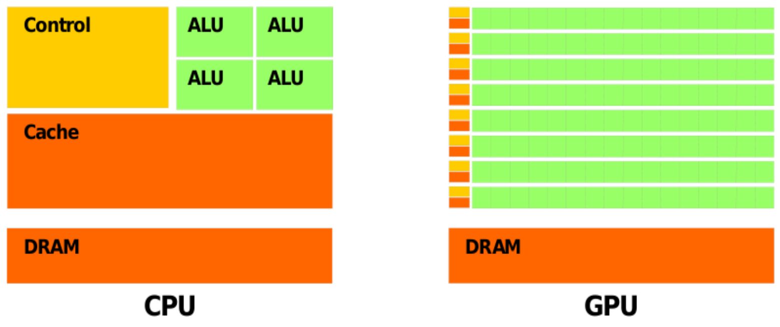


***Figure 34**: Comparison of CPU and GPU architecture [15].*

A GPU contains many ALUs for arithmetic and logical operations. In order to ensure fast processing times, there is a high ratio of ALU operations to memory operations, allowing efficient parallel processing. In fact, fast processing times are a primary requirement for any graphical application. Parallel processing technically means processing a set of pixels and allocating the threads simultaneously. If quickly processed results are a priority, software can be modified to be suited to CUDA, and the modified code can be run in parallel on a GPU card.

With CUDA, many general-purpose image processing applications can use a built-in toolbox, such as the parallel computing toolbox (PCT) or image processing toolbox

(IPT), so as to achieve accurate, high-speed processing [14]. For this thesis, the code has been written with a built-in toolbox in order to accelerate the processing time and take advantage of the resources of the GPU.

Many modern digital medical image processing applications use graphics processing units as a powerful graphics engine. The GPU has outpaced the CPU due to its high-capacity parallel programmable processors, featuring high performance, peak arithmetic capabilities, and fast memory access.

General-purpose graphics processing units (GPGPU) are used to improve the processing time of many applications. GPUs are integrated with a large number of processing units, which provide excellent performance for parallel processing. Two techniques are employed to achieve GPU parallelism:

1. Single instruction, multiple data (SIMD).
2. Single program, multiple data (SPMD).

SIMD is a single instruction stream model, however, it can run on many GPUs at the same time. SPMD is a multiple instruction stream model, and can run many programs on the same CPU systems.

1.4 Application of CUDA

GPU speed has addressed problems of processing complexity and programming capacity. While the general concept of image processing algorithms is to treat or process all pixels in series, the newest technology of Nvidia cards can process pixels in parallel. Because the GPU can process vast numbers of images in parallel, it plays an important role in many medical image processing applications.

The Nvidia hardware is based on Compute Unified Device Architecture (CUDA), which uses a programming language that is an extension of the C language. With the use of CUDA, programmers are able to run programs directly on GPUs. Nvidia provides its own compiler, the nvcc compiler, and programs written for CUDA can be run on Nvidia cards. Programmers can modify programs accordingly, if the main goal is parallelism.

CUDA involves two main parts:

1. The host: This is where software executes on the CPU.
2. The kernel: This is the part on Nvidia cards, where software executes on the GPU.

Below are some examples of the wide range of applications of CUDA:

1. Implementing fast MRI gridding on GPUs via CUDA [28].
2. Performance Study of Satellite Image Processing on Graphics Processors Unit Using CUDA [29].
3. Performance Measurements of Algorithms in Image Processing [30].
4. Parallel Biomedical Image Processing with GPGPUs in Cancer Research [31].
5. Image Processing Application on Graphics processors [32].
6. Neural Network Implementation using CUDA and Open MP [33].

CHAPTER 2: METHODS

2.1 Introduction

Modern graphics processing units (GPUs) have been designed for high-performance single instruction, multiple data (SIMD) processing. Although images can be processed by CPUs, this computing revolution has made GPUs an attractive platform for parallel processing in diverse applications. However, developing efficient general-purpose code for graphics-optimised architectures is a significant challenge.

There are many morphological algorithms used to enhance or improve digital images. They are important tools in the medical world, for example allowing radiologists and other medical professionals to improve the image quality of a large number of images very quickly. The algorithms developed in this thesis, which are implemented on modern Nvidia GPUs, can process many frames within milliseconds, rather than seconds. These algorithms read digital images in a variety of formats and process them within only a few milliseconds. This is accomplished by using:

- Modern graphics cards (originally used for gaming purposes, but now applicable for high-speed general-purpose computations).
- Fast image data processing on a CPU, while the GPU uses multiple-core processors (up to hundreds in modern generations, e.g., the Nvidia GeForce GTX 860M 4GB).

2.2 Noise Suppression

In digital image processing, noise arising during image acquisition or image transmission can easily corrupt images. This makes noise suppression an important concern. There are many types of noise, such as Gaussian and salt-and-pepper noise, which occur in digital images.

In daily life, digital images exist everywhere, in applications such as mobile devices, television, medical scanners (e.g., MRI and CT), and geographical and astronomical systems. Noise has a serious impact on digital images, since it can degrade important details in an image. Thus noise remains a major challenge and concern in many digital image systems and applications. One of the simplest mathematical methods that is used to remove noise from digital images is the median filter, which is one of the most useful filters for treating salt-and-pepper noise. It is a very effective noise removal tool if and only if the noise does not affect the image edges, because the median filter works only on low noise densities. If the effect of noise on the image edges exceeds 50%, image details will not be preserved completely and denoising the image will result in a blurred image [24] [25].

In medical images, noise can randomly affect image pixels, leading to changes in some image pixel information or image brightness. Such noise is inadvertently produced by many electrical machines and devices, for example, device sensors and medical MRI and CT scanners. Due to this unwanted noise, digital image quality and details can easily be degraded and changed. An image restoration process must be applied in order to restore the original image. This is an operation that is applied to a corrupted or noisy image in order to recover the original image [26]. Problems of noise in digital image have encouraged researchers to continue studying the behaviour of noise and its impact on images. In this thesis, noise removal techniques are based on my morphological algorithms. In this research, the aim is to reduce noise and improve the image appearance within a few milliseconds. The algorithms used are shown in the following section.

2.3 Computational method

The methods used in this research were based on real-time digital image processing techniques. In the first phase of the thesis, the work was simulated in MATLAB; the results are presented in Section 3.2. The next phase was to utilise C#, one of the most common general programming languages, as presented in Section 3.3. Lastly, the code was written in CUDA and implemented on a Nvidia card. The primary platform of the program is the CUDA application, which contains the main program and the kernel. The CUDA kernel is a function that is called by the software from the host, and is run simultaneously on the device, in parallel. The main program contains my morphological filters. The interface between C# and CUDA parallel computing is shown in Figure 35. Once the GPU finishes processing the image, it transfers the processed image back to the CPU memory.

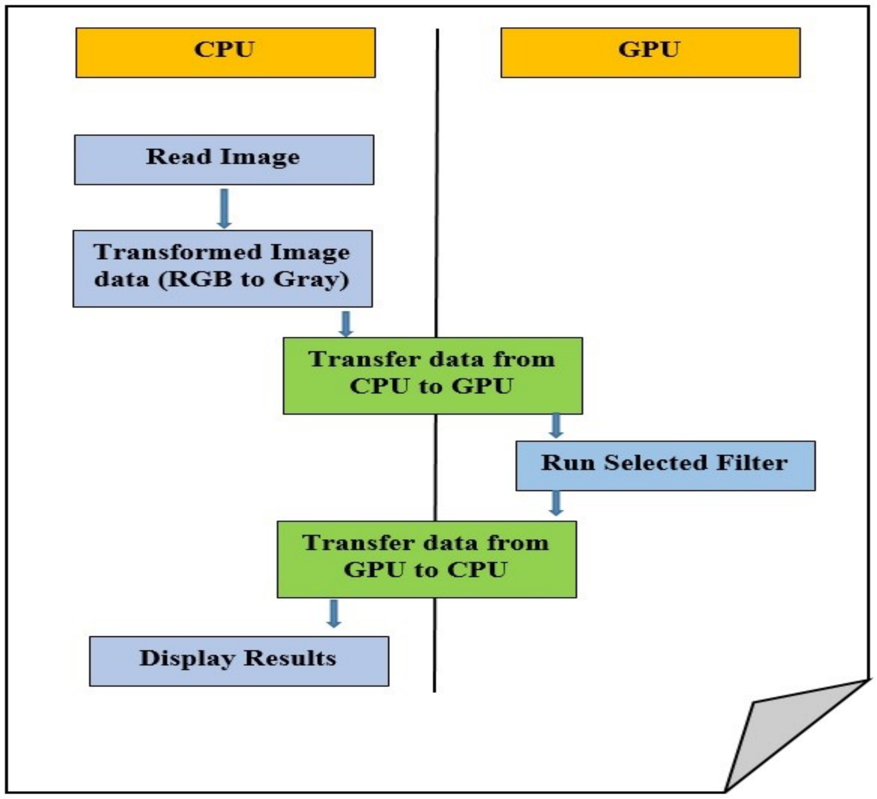

CPU
GPU
Read Image
Transformed Image data (RGB to Gray)
Transfer data from CPU to GPU
Run Selected Filter
Transfer data from GPU to CPU
Display Results

***Figure 35**: Simplified flowchart of the digital image processing.*

Figure 36 shows the image processing steps, starting with image acquisition and ending with the final results (as represented by arrows). In this thesis two-dimensional images were used. Digital images employ four imaging techniques; because of the concern with medical images, this research focused on grey-scale processing. The morphological filters were involved with mathematical algorithms and segmentation. Image segmentation is the technique of dividing a medical image into parts or sets of pixels. This segmentation is used to simplify an image into segments that have meaning and that make analysis easier.

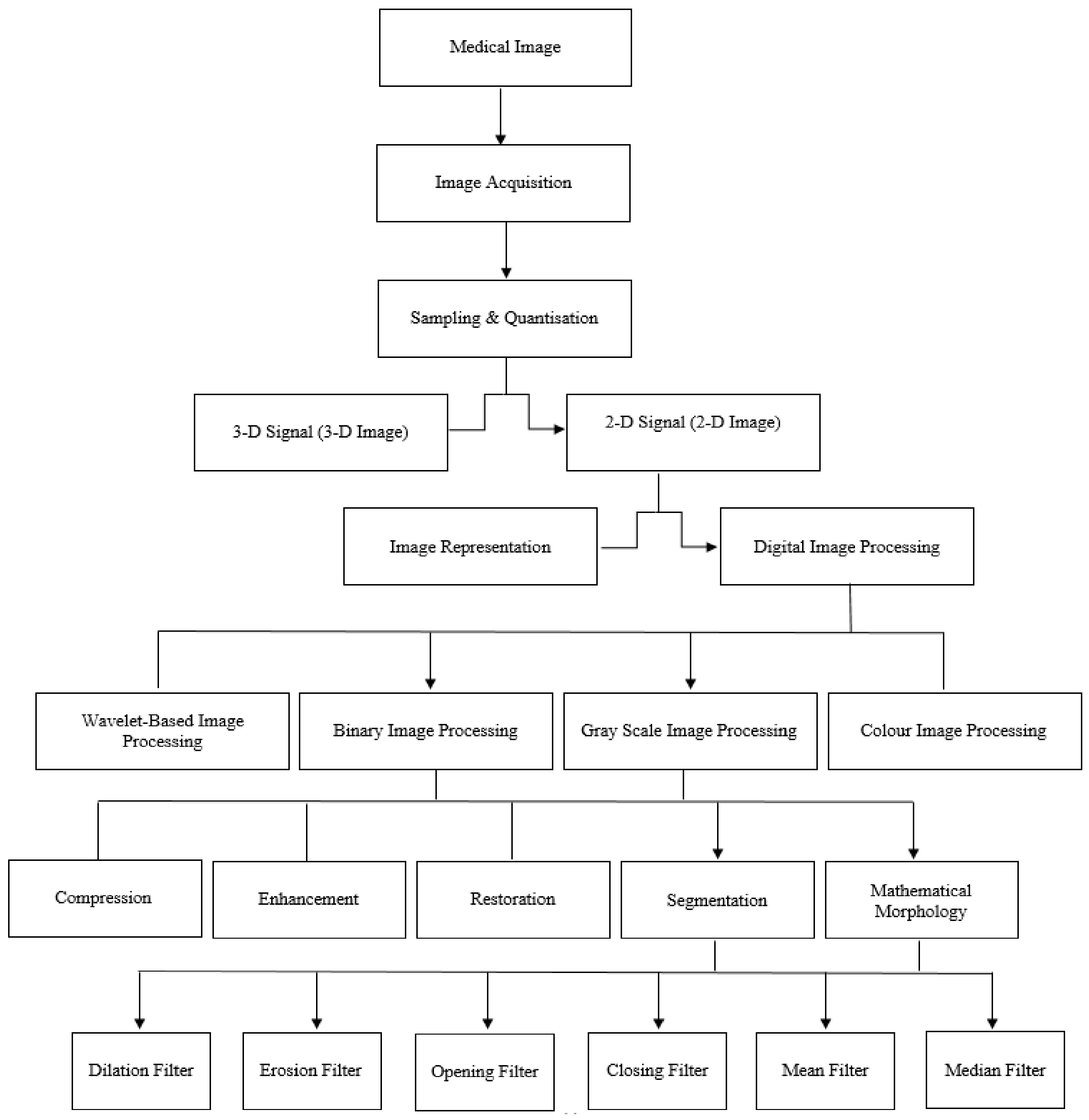


Figure 36: *Image processing steps applied in this research.*

2.4 Implementation

In this research, the algorithms were implemented on CPU and GPU processors. The GPU was used primarily for parallel processing purposes. Table 3 shows the main features of the hardware and software used. The Nvidia Geforce 860M GPU provides 40%

higher performance than that offered by other Nvidia models. This Nvidia CUDA has 1152 cores, which make it faster and provide an advantage over the CPU [34].

***Table 3**: The main features of the hardware and software used in this research.*

Hardware Setup	
CPU	Intel Core i7 – 4710HQ 2.5GHz.
Main RAM	16GB, DDR3
GPU	Nvidia Geforce GTX860M, Intel HD Graphics 4600 chipset.
Graphics RAM	4GB, GDDR5, 128-bit interface
CUDA Cores	1152
CUDA compute capability	3.0
Software Setup	
Operating system	MS Windows 8.1 x64
Compiler	MS Visual Studio professional 2013 x64
CUDA version	CUDA 6.5.14

CHAPTER 3: RESULTS AND DISCUSSION

3.1 Introduction

Most of the modern digital image processing techniques are computationally rich of image manipulation processing techniques such as what is using to process CT scan, X-Ray and MRI images. Lately, many researches and studies especially on the neuroscience area have become a very important topics and too many papers are generated every year! For example, the MRI (Magnetic Resonance Imaging) studies are required a high powered real time multiprocessors systems to process the complexity and the interaction of the digital images. Although these powered multiprocessors systems may be expensive to researchers, these multiprocessors systems provide high performance, less processing time and sufficient results in real-time.

The modern improved graphical processing units (GPUs) have dwarfed the use of central processing units (CPUs) especially when the processing performance in real-time is a concern.

This chapter shows the power of the GPU by comparing CPU sequential (MATLAB and C#) algorithms with GPU computational (CUDA) algorithms on the Nvidia card Geforce GTX 860M. Here it is proved that the speed and performance of digital image processing can be accelerated by applying real-time parallel processing techniques through the use of the GPU as a digital processor.

In the present research, the speed and performance of the proposed approach for processing digital medical images was investigated by using three different programs:

- The original DIPS package was written in MATLAB and implemented on a Lenovo Y50 computer. This served as the standard for subsequent work. The performance of all other software was compared with the MATLAB results.

- Once all of the DIPS algorithms were implemented in MATLAB, a package was developed in C#, capable of reading, filtering, analysing, and displaying images.

- Once all of the algorithms were implemented in MATLAB and C#, a new package was developed in CUDA and implemented in Nvidia. The package was tested with a wide variety of images and was shown to be robust.

This chapter is organised as follows. First the simulation results, which were implemented in MATLAB, are presented in Section 3.2. These results were used as the standard, and the quality of all subsequent work was compared with them to ensure that the algorithms worked. Section 3.3 shows the results obtained from the C# package. Section 3.4 presents the results from the package with the fully implemented CUDA kernel, and Section 3.5 summarises the performance of the three packages. As expected, the CUDA significantly outperformed the C# program, which significantly outperformed the MATLAB implementation.

3.2 Matlab Application

In the first experiment, the simulation was implemented with the MATLAB application, in order to calculate the total processing time for many selected digital images. The main goal of this work was to ensure that my algorithms were implemented correctly and, as mentioned above, to set standards to form the basis of comparison with subsequent work. The processing time required was dependent upon the size of the image and the type of filter used. The processing time results are analysed:

- For the different operations or filters, i.e., erosion, dilation, opening, closing, mean and median filters.
- For structuring elements (SE) of different sizes, i.e., 3×3, 5×5 and 7×7.

The use of the GPU as a digital processor is described, and processing times of the following three methods are compared:

- Simulation and analysis via CPU by using the MATLAB application.
- Processing digital images via CPU by using the C# programming language.
- Processing digital images by CPU using C++ programming language.
- Processing digital images via GPU by using CUDA for real-time parallel processing.

Figure 37 shows the effect of image noise, as processed by the MATLAB application. MATLAB reads the image and processes it, to simulate and demonstrate the characteristics of the digital image.

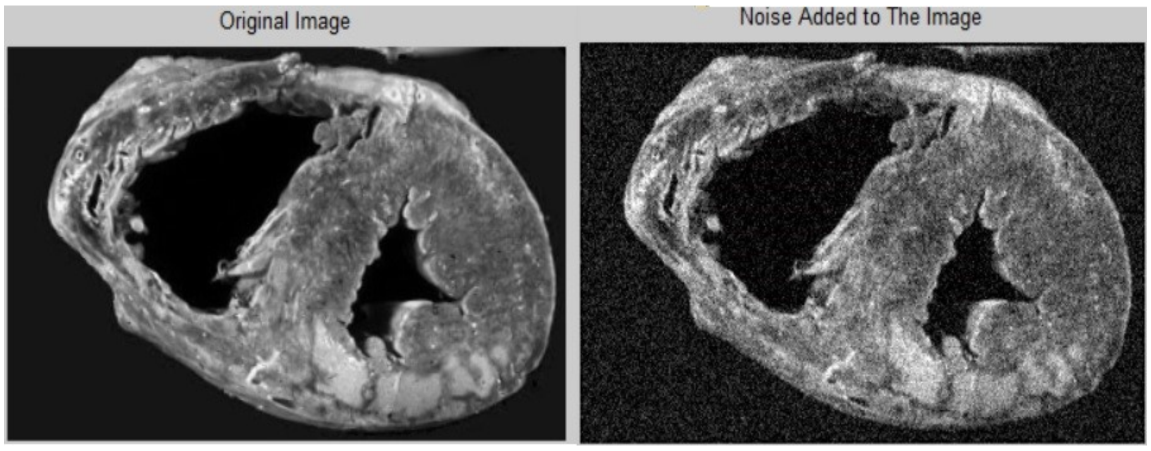


Figure 37*: Illustration of the effects of noise, before processing of the image to reduce these effects.*

In Figure 37, the original image is shown on the left, and the image to which noise has been added is shown on the right. This demonstrates the effect of noise on a digital image, and how noise can reduce the quality of an image. Digital images in particular are exposed to many sources of noise, and the application of morphological filters helps to reduce this distortion. The approach used in this research was tested in the MATLAB environment with a variety of filters, and with SE of different sizes. The following filters and structuring elements were used:

1. Erosion filters with SE of sizes 3x3, 5x5 and 7x7.
2. Dilation filters with SE of sizes 3x3, 5x5 and 7x7.
3. Opening filters with SE of sizes 3x3, 5x5 and 7x7.
4. Closing filters with SE of sizes 3x3, 5x5 and 7x7.
5. Mean filters with SE of sizes 3x3, 5x5 and 7x7.
6. Median filters with SE of sizes 3x3, 5x5 and 7x7.

For example, the grey-scale erosion operation with a square SE of sizes 3×3, 5×5 and 7×7 computes the minimum intensity values for each pixel. The result of applying an erosion filter to a grey-scale image is that bright pixels are reduced and dark pixels are increased.

Figure 38 shows the effect of applying an erosion filter to an image with a dark background. The first image shown is the original image (A). The second image, with salt-and-pepper noise (B), is assumed to be the transmitted image to be processed.

The result of using the erosion filter with a 3×3 square SE is shown in the third image (C); here it can be seen that most of the noise associated with bright pixels has disappeared. A similar effect can be observed when the noisy image is eroded with a 5×5 square SE (D). However, although noise is removed when the image is eroded with a 7×7 SE (E), it can be seen that the rest of the image has been degraded significantly. The optimal SE size is dependent upon the severity of the noise. In the example shown in Figure 38, the 3×3 square SE removed the noise with less impact on the picture details.

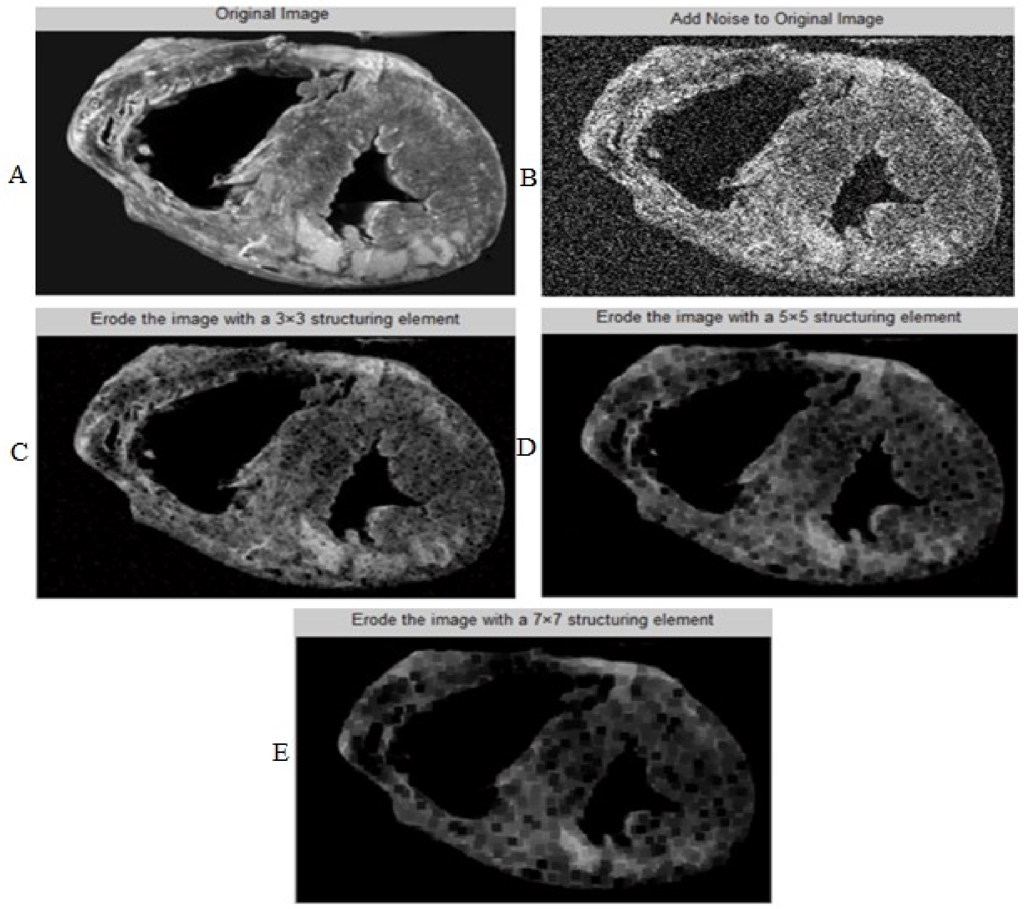

Original Image
Add Noise to Original Image
Erode the image with a 3×3 structuring element
Erode the image with a 5×5 structuring element
Erode the image with a 7×7 structuring element
A
B
C
D
E

Figure 38*: Illustration of the effect of applying an erosion filter with different SE sizes. Image A is the original image, and image B has salt-and-pepper noise. Images C, D and E show the effect of using an erosion filter in terms of keeping the original image details; square structuring elements of size 3×3, 5×5 and 7×7 were applied, respectively.*

The primary effect of an erosion filter is to erode away the boundaries of regions of foreground pixels. Thus, this filter is typically used to erode white pixel noise. This causes areas of foreground pixels to shrink in size, and holes in the foreground to become larger. The erosion operation removes any foreground pixel that is not completely surrounded by other white pixels [10]. Figures 38 and 39 illustrate two useful filters that may be employed to remove noise from a digital image. The erosion and dilation filters are the two basic filters in the area of mathematical morphology. Whereas erosion decreases the intensity of bright pixels which may contribute to noise, the opposite is the case with the dilation operation, which increases the bright pixels. Figure 39 shows the impact of applying the dilation operation. An SE of size 3×3 is generally the most effective SE, that is commonly used for dilation and erosion. However, other sizes could be used. For example, an SE of size 7×7 is quite a large SE, which produces a more extreme effect in terms of dilation and erosion.

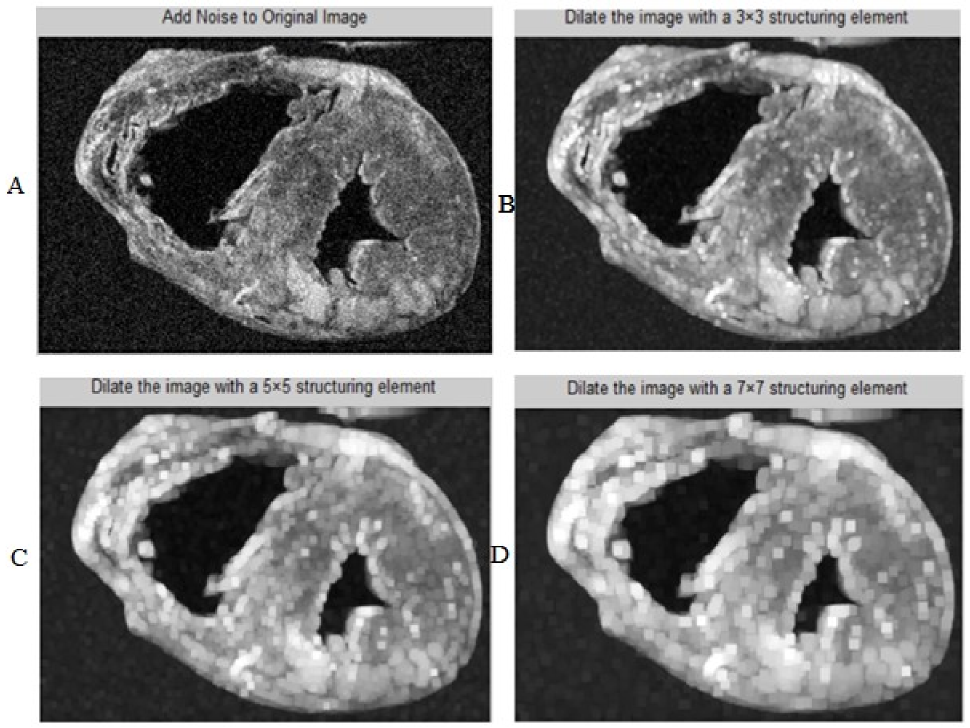

Add Noise to Original Image
Dilate the image with a 3×3 structuring element
Dilate the image with a 5×5 structuring element
Dilate the image with a 7×7 structuring element
A
B
C
D

***Figure 39**: Illustration of the effect of applying a dilation filter with different SE sizes. Images B, C and D show the effect of using a dilation filter with structuring elements of size 3×3, 5×5 and 7×7, respectively.*

Note that in Figure 39, the borders of shapes in images B, C and D have been rounded off. Dilation usually increases the white pixels: This means that a bright region that is surrounded by a dark region will grow in size, while a dark region that is surrounded by a bright region will shrink in size. Therefore, small dark spots will be eliminated and bright spots will become larger.

Some other morphological filters, such as the mean and median filters, provide more accurate operations for dealing with noise. The use of median and mean filters is preferred for the reduction of Gaussian noise. They preserve edges while eliminating noise, and are widely used in real-time digital image processing. Figure 40 illustrates the application of dilation, mean and median filters to a noisy image.

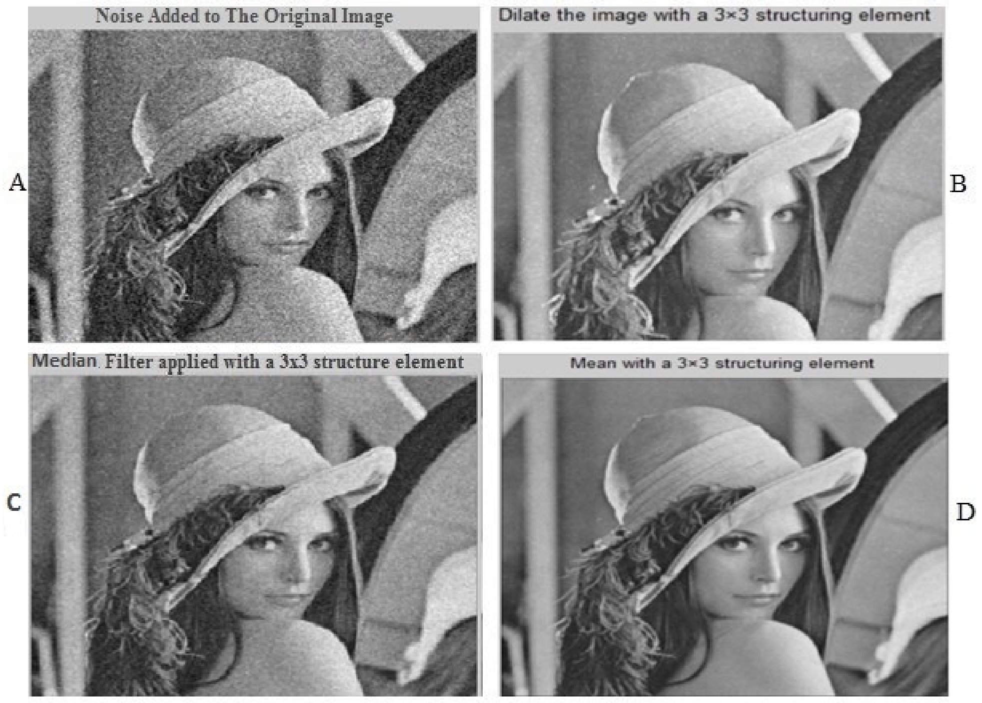

Noise Added to The Original Image
Dilate the image with a 3×3 structuring element
A
B
Median Filter applied with a 3x3 structure element
Mean with a 3×3 structuring element
C
D

***Figure 40**: Illustration of the effect of applying dilation, mean and median filters with a 3×3 square structuring element. Images B, C and D show the effect of using a dilation, median and mean filter, respectively.*

The application of median and mean filters can keep the details of a processed image clear and readable. This is shown in Figure 41, where a detailed image is almost obscured by noise (A). Here, median (B) and mean (C) filtering result in clear images with many details. Prior to the application of these filters, the noisy image is so incomprehensible as to be practically meaningless.

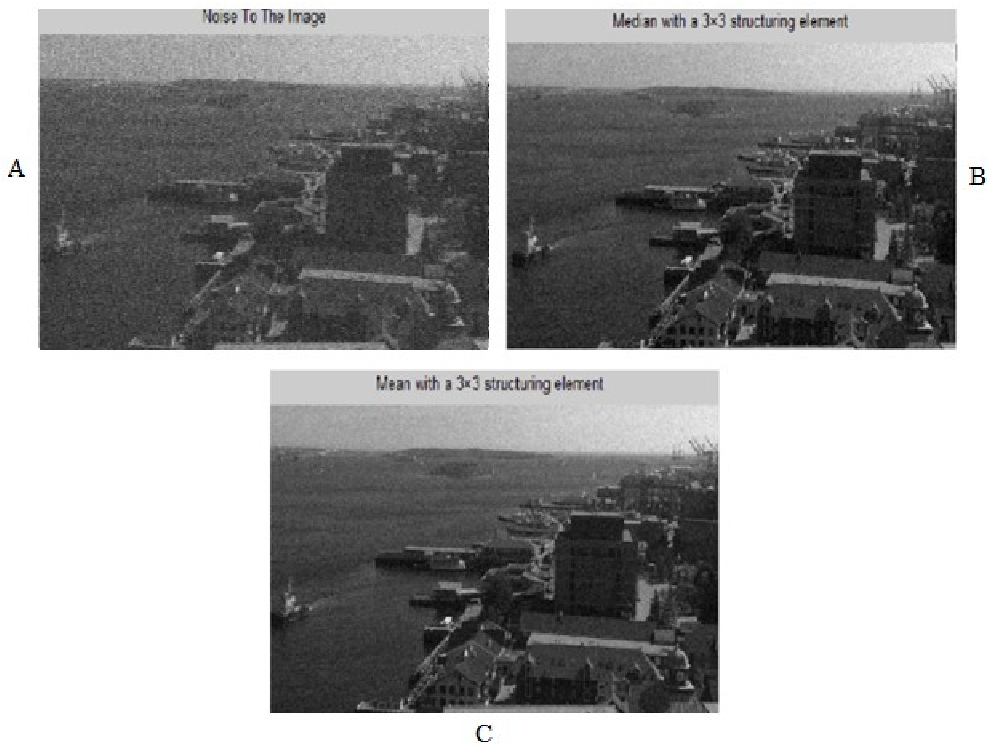

Noise To The Image
A
Median with a 3×3 structuring element
B
Mean with a 3×3 structuring element
C

Figure 41*: Illustration of the effect of applying median and mean filters with a 3×3 square structuring element. A shows the noisy image, and B and C show the effect of using a median and mean filter, respectively.*

While previous sections describe the results of applying morphological filters to digital images in theoretical terms, the present discussion demonstrates the power of using such filters to eliminate noise. For this research, MATLAB was used as a tool for simulation and comparison purposes; the following section describes the work with the C# programming language.

3.3 C# Strategy

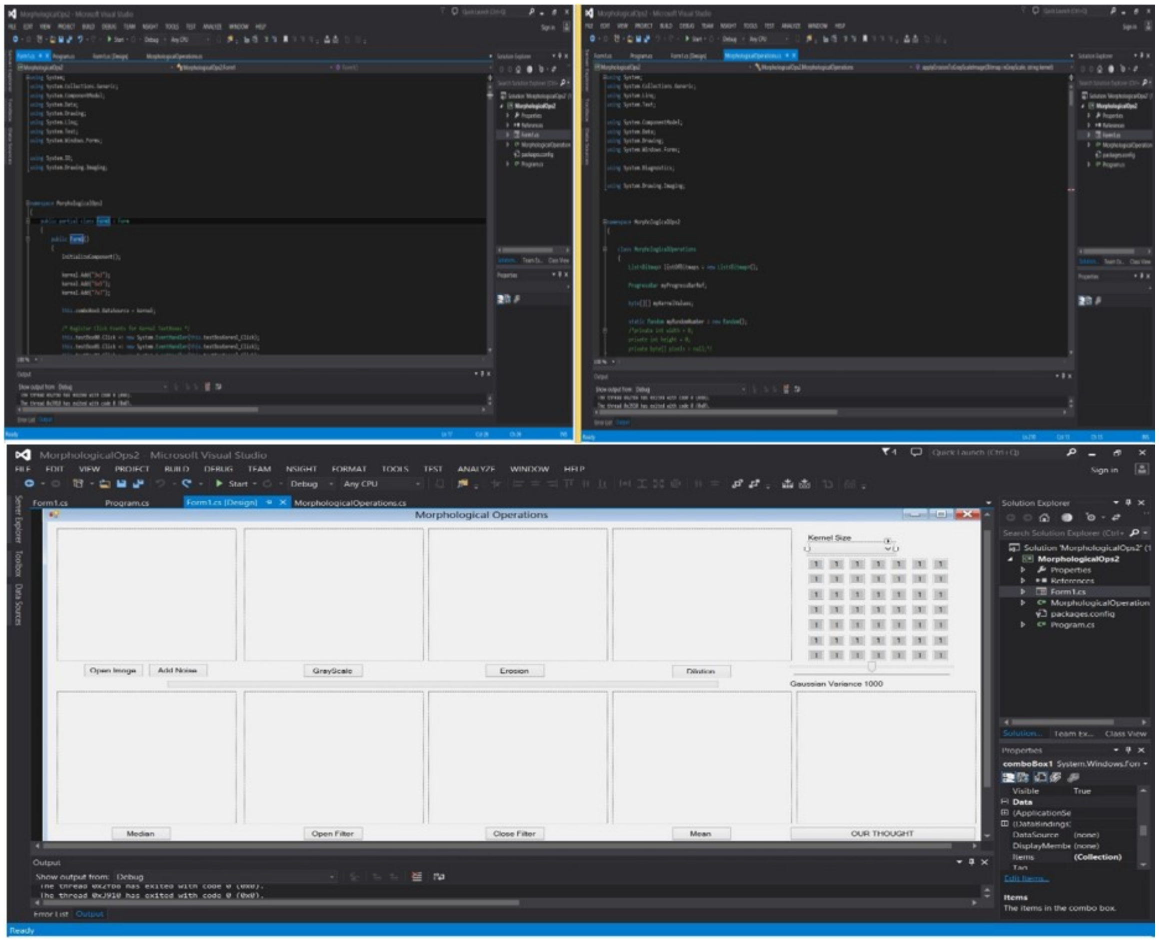
C# and C++ are object-oriented computer programming languages. Whereas C++ is a high-level programming language, C# is intermediate level. C# is an extension of the C language, and includes a graphical operation system and graphical user interfaces (GUI).

In this section, C# is used to process digital images. As described in Chapter 1, morphological image processing is a collection of non-linear operations and techniques that can be used with digital medical images to remove noise, and to enhance, recover, segment and extract features, etc.

In this research, C# was used because it is flexible, powerful and simple. It can be used to create a variety of applications. Based on the code employed, users are able to see when applications begin moving pixels or changing their values. Six morphological methods or operations for digital image processing were addressed: Erosion, dilation, opening, closing, mean and median operations. These methods were used for noise reduction with structuring elements (SE) of equal size. In order to compare the processing time of these six filters, MATLAB, C# and CUDA were used to create applications. For these experiments, the environment of Visual Studio 2013 was employed. The C# language was used to process digital images with a central processing unit (CPU).

The execution time of the C# CPU code was significantly faster than that of MATLAB. Figure 42 shows a snapshot of the code and the application platform. An easy-to-use graphical user interface was created. It incorporates a series of buttons, windows and a drop-down menu, allowing the user to choose the images to be processed and the filters to be used.

This semi-automatic approach permits users to select the filter most appropriate for the image set being processed.

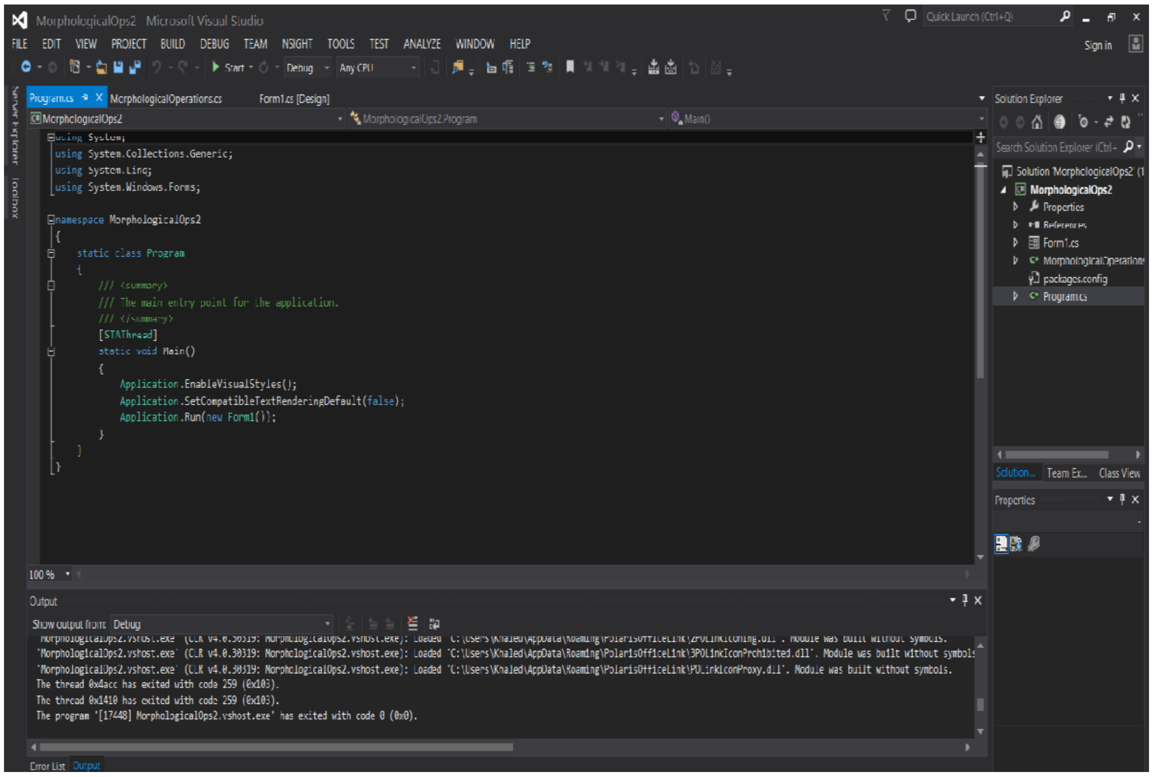


```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using System.IO;
using System.Drawing.Imaging;

namespace MorphologicalOps2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            kernel.Add("3x3");
            kernel.Add("5x5");
            kernel.Add("7x7");

            this.comboBox1.DataSource = kernel;

            /* Register Click Events for Kernel TextBoxes */
            this.textBox00.Click += new System.EventHandler(this.textBoxKernel_Click);
            this.textBox01.Click += new System.EventHandler(this.textBoxKernel_Click);
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;

using System.Diagnostics;

using System.Drawing.Imaging;

namespace MorphologicalOps2
{
    class MorphologicalOperations
    {
        List<Bitmap> ListOfBitmaps = new List<Bitmap>();

        ProgressBar myProgressBar;

        byte[] myKernelValues;

        static Random myRandomNumber = new Random();
        //private int width = 0;
        private int height = 0;
        private byte[] pixels = null;
```

Figure 42: *Snapshot of the codes for the morphological operations, and the main platform of the application.*

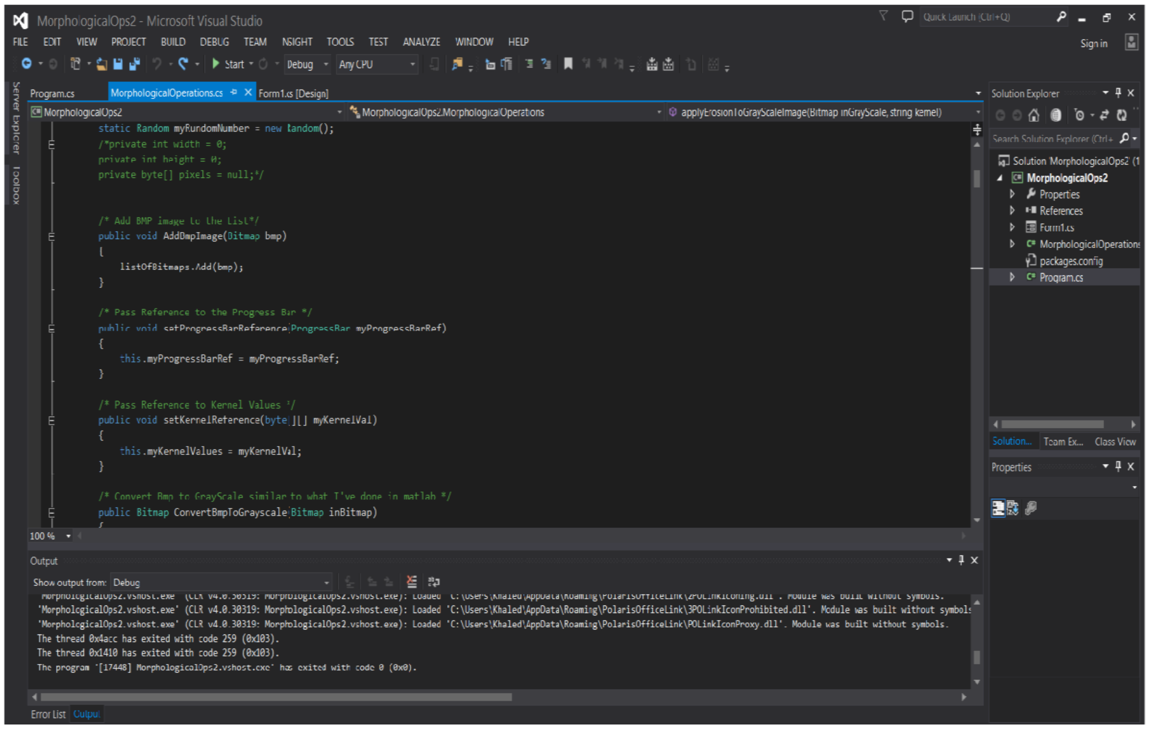
Figure 43 shows the code and application platform in greater detail. Part A of Figure 43 illustrates the main entry or starting point for the medical image processing program. Part B shows a snapshot of the main code, where most of the operations and functions are applied and run, and part C shows the application interface, which represents the platform of the project. This platform enables users to select and apply any of the filters and SE or kernel values.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace MorphologicalOps2
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

```
'MorphologicalOps2.vshost.exe' (CLR v4.0.30319: MorphologicalOps2.vshost.exe): Loaded 'C:\Users\Khaled\AppData\Roaming\PolarisOfficeLink\2POLinkIconing.dll'. Module was built without symbols.
'MorphologicalOps2.vshost.exe' (CLR v4.0.30319: MorphologicalOps2.vshost.exe): Loaded 'C:\Users\Khaled\AppData\Roaming\PolarisOfficeLink\3POLinkIconProhibited.dll'. Module was built without symbols
'MorphologicalOps2.vshost.exe' (CLR v4.0.30319: MorphologicalOps2.vshost.exe): Loaded 'C:\Users\Khaled\AppData\Roaming\PolarisOfficeLink\POLinkIconProxy.dll'. Module was built without symbols.
The thread 0x4acc has exited with code 259 (0x103).
The thread 0x1410 has exited with code 259 (0x103).
The program '[17448] MorphologicalOps2.vshost.exe' has exited with code 0 (0x0).
```

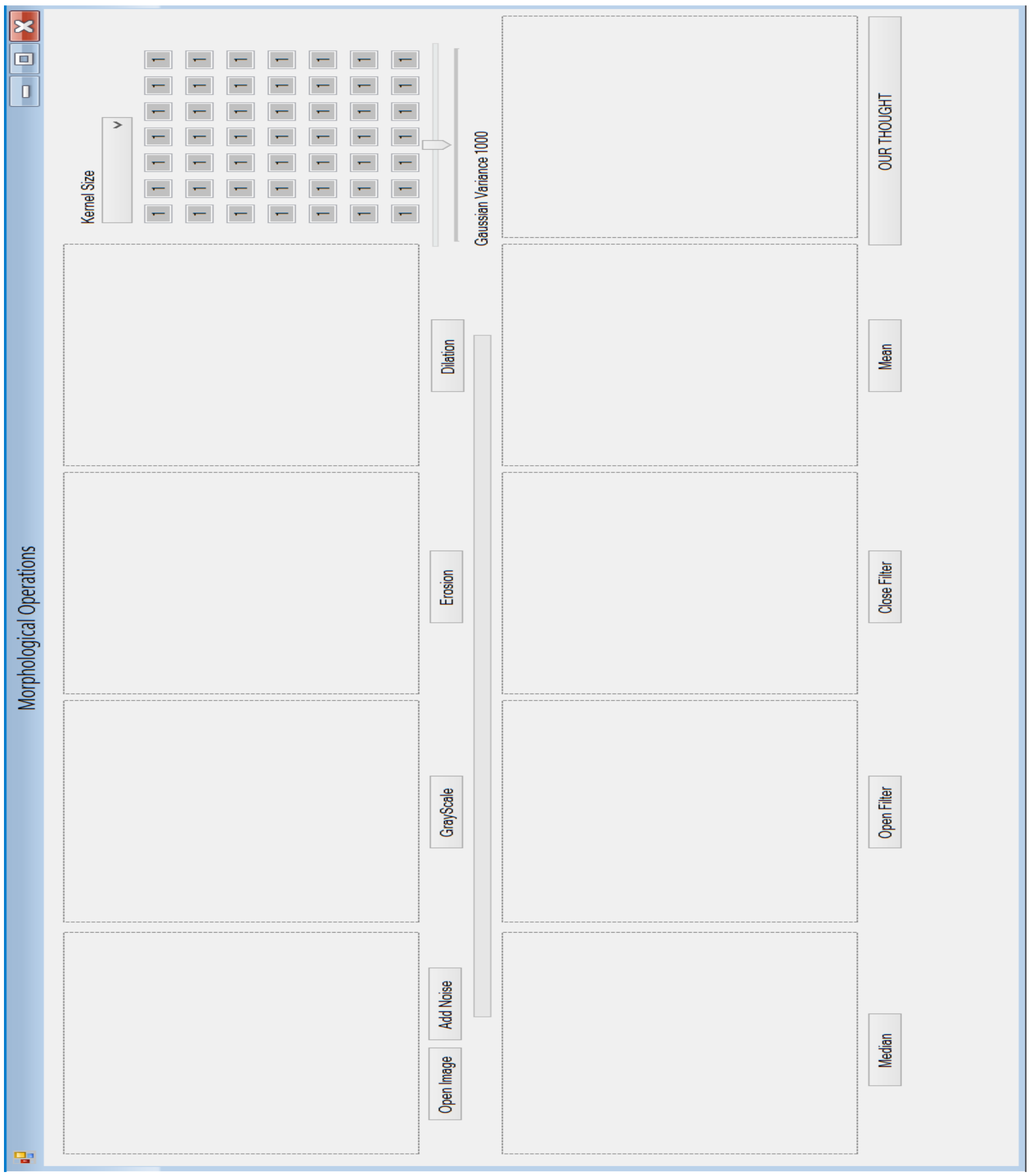
Part A: Snapshot of kernel codes for the morphological operations.



```
        static Random myRundomNumber = new Random();
        /*private int width = 0;
        private int height = 0;
        private byte[] pixels = null;*/


        /* Add BMP image to the List*/
        public void AddBmpImage(Bitmap bmp)
        {
            listOfBitmaps.Add(bmp);
        }

        /* Pass Reference to the Progress Bar */
        public void setProgressBarReference(ProgressBar myProgressBarRef)
        {
            this.myProgressBarRef = myProgressBarRef;
        }

        /* Pass Reference to Kernel Values */
        public void setKernelReference(byte[][] myKernelVal)
        {
            this.myKernelValues = myKernelVal;
        }

        /* Convert Bmp to GrayScale similar to what I've done in matlab */
        public Bitmap ConvertBmpToGrayscale(Bitmap inBitmap)
        {
```

```
'MorphologicalOps2.vshost.exe' (CLR v4.0.30319: MorphologicalOps2.vshost.exe): Loaded 'C:\Users\Khaled\AppData\Roaming\PolarisOfficeLink\2POLinkIconing.dll'. Module was built without symbols.
'MorphologicalOps2.vshost.exe' (CLR v4.0.30319: MorphologicalOps2.vshost.exe): Loaded 'C:\Users\Khaled\AppData\Roaming\PolarisOfficeLink\3POLinkIconProhibited.dll'. Module was built without symbols
'MorphologicalOps2.vshost.exe' (CLR v4.0.30319: MorphologicalOps2.vshost.exe): Loaded 'C:\Users\Khaled\AppData\Roaming\PolarisOfficeLink\POLinkIconProxy.dll'. Module was built without symbols.
The thread 0x4acc has exited with code 259 (0x103).
The thread 0x1410 has exited with code 259 (0x103).
The program '[17448] MorphologicalOps2.vshost.exe' has exited with code 0 (0x0).
```

Part B: Snapshot of main codes for the morphological operations.



Part C: Snapshot of platform for the morphological operations.

Figure 43: *Illustration of the main windows of the project. Parts A and B illustrate the codes and Part C represents the main platform of the application.*

Figure 44 shows a screenshot of the user interface for the C# software. The features provided include filter icons, a SE (kernel) dialog box or drop-down list, and a progress bar. The drop-down list allows users to choose an SE size of 3×3, 5×5 or 7×7. Users can easily manipulate the SE values and apply any selected filter simply by clicking the desired options. The window depicts the original and grey-scale images, as well as the six morphological operations.

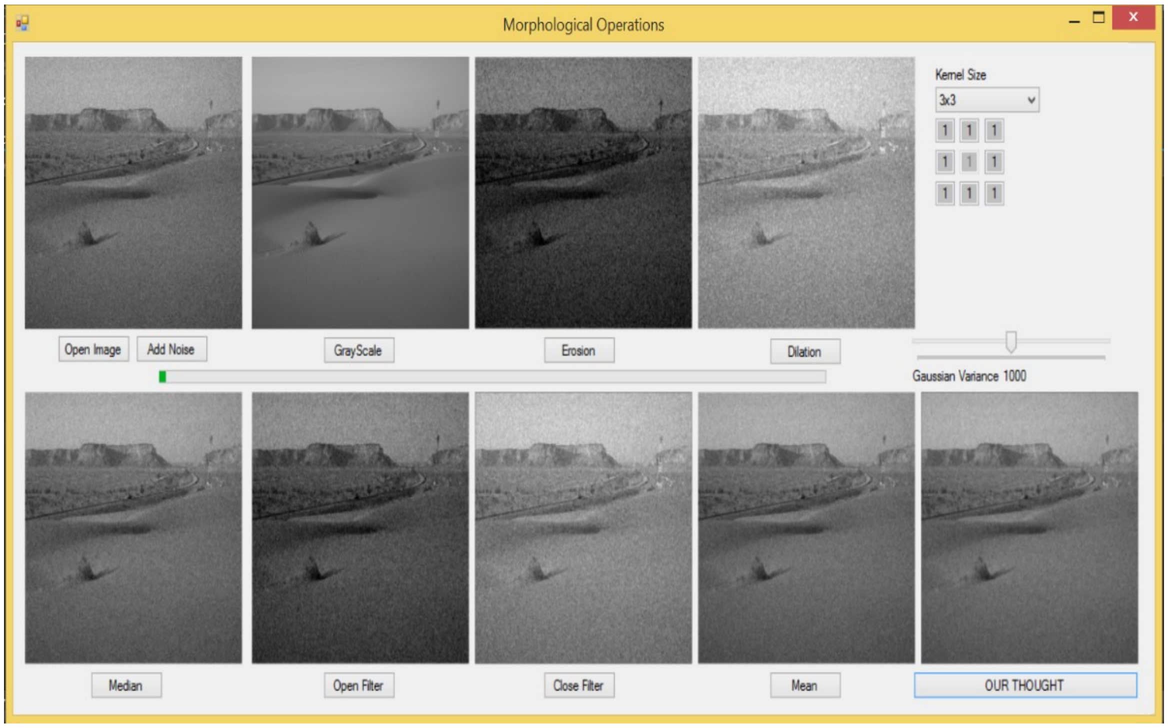
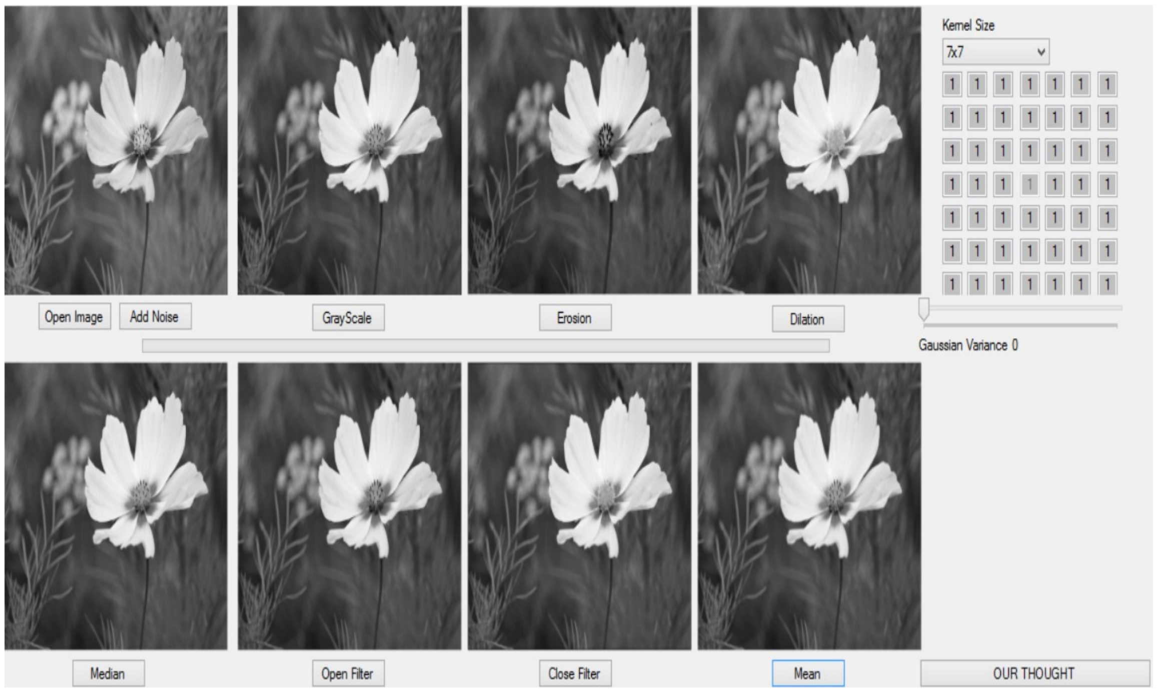


Figure 44: *The morphological operations platform. The top left-hand image is the original image, followed by the grey-scale image. The subsequent images show the effect of using the erosion, dilation, median, opening, closing and mean filters.*

Additive noise in particular, such as Gaussian noise and salt-and-pepper noise, which affect digital images, can be removed by a variety of mathematical methods. Morphological filters were developed in C# for noise reduction in digital images. Figure 45 illustrates the effect of applying the morphological filters to a noisy image. Although the erosion and dilation filters remove noise from the image, the flower stigma appears a little darker than the original grey-scale image in the erosion result, and a little brighter in the dilation result. Erosion reduces noise but makes foreground objects shrink in size, whereas dilation reduces noise but causes foreground objects to increase in size.

In contrast, the opening filter involves an erosion operation followed by a dilation operation. This causes foreground (bright) objects to shrink in size and smooth's the objects contours by removing some pixels from the edges. The opening filter operates similarly to erosion, however, it is less destructive than erosion. Conversely, the closing filter involves a dilation operation followed by an erosion operation: This enlarges foreground (bright) boundaries, but is less destructive than dilation.



1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

Figure 45*: Snapshot of the C# morphological operations platform. The top left-hand image is the original image, followed by the grey-scale image. The subsequent images show the effects of using filters.*

Mean filtering reduces the noise by reducing the amount of intensity variation among pixels. Mathematically, it replaces the value of each pixel in an image with the average value of the neighbouring pixels. Thus a mean filter changes the value of pixels that are unrepresentative of their surroundings. Median filtering is likewise used to reduce noise in a digital image, however it is more effective than mean filtering in preserving useful details of the processed image.

Figure 46 illustrates the difference between mean and median filtering operations. Whereas mean filtering replaces each pixel value with the average value of neighbouring pixels, median filtering replaces each pixel value with the median value of neighbouring pixels. Here it can be seen that median filtering is better suited to preserving image details.

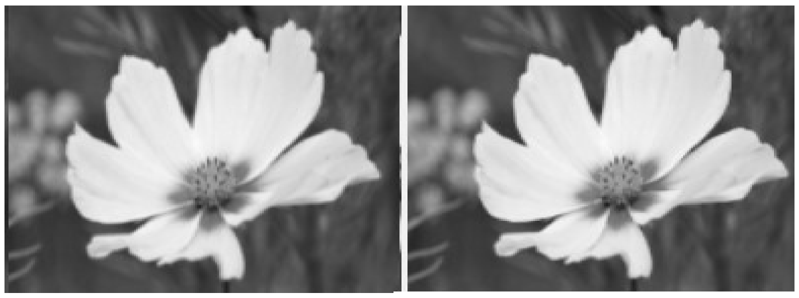


Figure 46*: Illustration of the effect of mean filtering (left-hand image) and median filtering (right-hand image).*

3.4 CUDA Strategy

In this research, digital image processing algorithms were implemented by using Nvidia's CUDA (Compute Unified Device Architecture) technology. General-purpose computations can be executed on a CUDA-capable graphics card in fast parallel processing. A major objective was to determine the performance enhancements achieved by using the image processing algorithms implemented in C# and to compare these with

CUDA. The comparison of the GPU time measurements with the CPU reference implementations shows the benefit of using CUDA in the image processing algorithms.

This research has developed the proposed approach on CUDA, a parallel computing platform and application processing interface (API). Computing performance is increased dramatically by employing the power of the graphical processing unit (GPU). High-level programming languages such as those used with CUDA allow applications to gain efficiency by running the sequential part of their workload on a CPU, while moving sections of code that can be executed in parallel to a GPU. CUDA was developed for applications that are inherently parallel, which can take advantage of the SIMD architecture of the GPU to achieve a significant reduction of overall execution time.

There are two main factors to be considered in relation to digital image restoration with morphological applications: The accuracy of the image restoration and the execution time. The algorithms in the present research deal with reducing the effects of noise arising due to external sources, such as Gaussian noise caused by poor illumination, or salt-and-pepper noise resulting from transmission errors. The use of a Nvidia GPU significantly decreases the total execution time. In order to access the parallel processing units of the video card, the main application transfers the digital image back and forth between the computer RAM and the Nvidia graphics card RAM.

Although transferring images between the computer RAM and the graphics card RAM adds a small overhead in terms of time required, this is still considerably less than the time which would be needed for processing in the CPU. The processing time required by the GPU depends upon the SE size and the Nvidia card used. The present research employed the Nvidia GeForce GTX 860M 4GB and the Intel Core i7-4710HQ for performing digital image processing.

The GeForce GTX 860M has various types of on-chip memory. In this project texture memory, optimised for 2D spatial access patterns, was used to speed up memory access. Figure 47 illustrates the relationship and communication between different types of memory. The texture memory space works as a pointer to an already defined data set. The disadvantage of using texture memory is that it is read-only and cannot be used to write data back to the global memory.

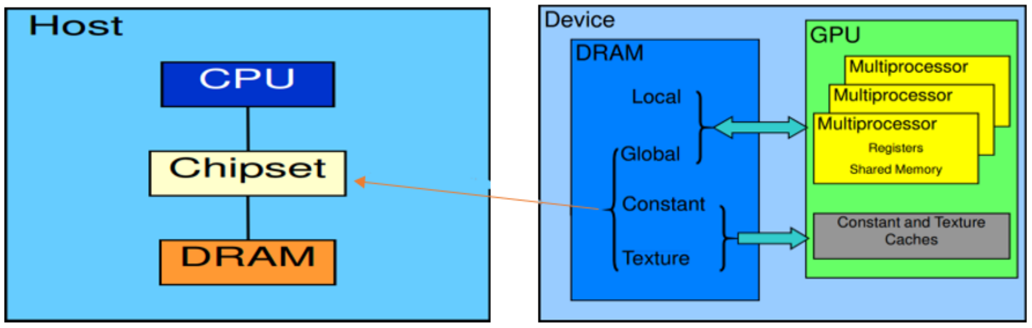


Figure 47*: Illustration of communication between the device RAM and Nvidia RAM.*

Texture memory serves as a container of one or more images, and makes the data available to the GPU. Tables 4 and 5 provide a brief summary of the different types of CUDA memory, illustrating the main features and differences.

Table 4*: Summary of CUDA memory types showing the features of each type of memory.*

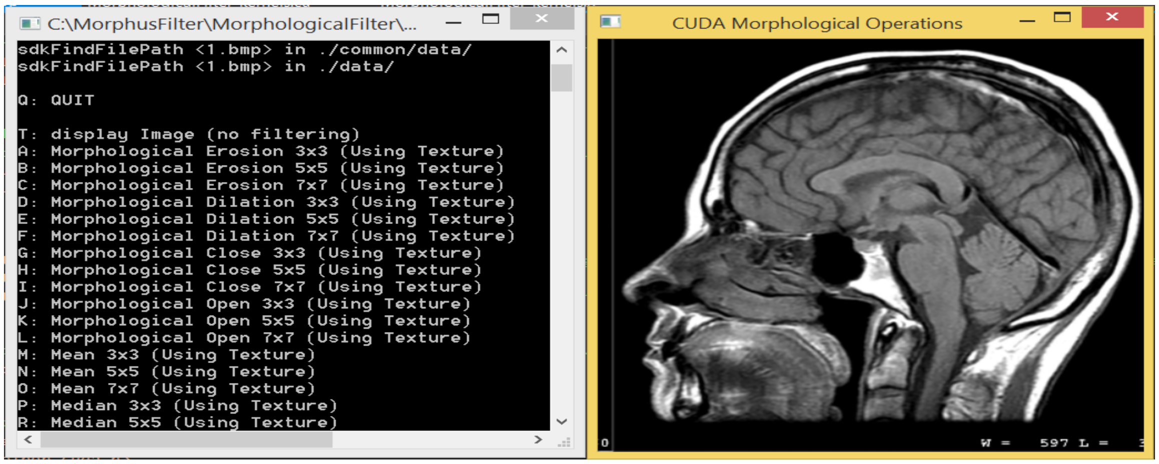
CUDA MEMORY TYPES	FEATURES
GLOBAL MEMORY	# It is slow and uncached. # It requires sequential and aligned 16-byte read/writes to be fast.
SHARED MEMORY	# It is fast but subject to bank conflicts.

	# Permits exchange of data between threads in block.
TEXTURE MEMORY	# Its cache is optimised for 2D access patterns.
	# Data can be accessed by different variables in a single operation.
LOCAL MEMORY	# Is used for whatever doesn't fit into the registers of global memory.
	# It is slow and uncached.
CONSTANT MEMORY	# Its constants and kernel arguments are stored.
	# It is slow but cached.

***Table 5**: Summary of CUDA memory types showing the location, access, scope and lifetime of each type.*

Memory	*Location*	*Cached*	*Access*	*Scope*	*Lifetime*
Local	Off-chip	NO	R/W	One thread	Thread
Shared	On-chip	NO	R/W	All thread in a block	Block
Global	Off-chip	NO	R/W	All threads + Host	Application
Constant	Off-chip	Yes	R	All threads + Host	Application
Texture	Off-chip	Yes	R	All threads + Host	Application

The left-hand image in Figure 48 illustrates the interface menu that permits users to access X-ray images. In this menu there are 20 options from which to choose, labelled sequentially from A to T. For each type of filter, there are three SE size options: 3×3, 5×5 and 7×7. Thus options A, B and C are assigned to the erosion filter; D, E and F to the dilation filter; G, H and I to the closing filter; J, K and L to the opening filter; M, N and O to the mean filter; and P, R and S to the median filter. In addition, the option Q refers to Quit, and T refers to the original image. These options are summarised in Table 6.


C:\MorphusFilter\MorphologicalFilter\...
CUDA Morphological Operations
W = 597 L =

```
sdkFindFilePath <1.bmp> in ./common/data/
sdkFindFilePath <1.bmp> in ./data/

Q: QUIT

T: display Image (no filtering)
A: Morphological Erosion 3x3 (Using Texture)
B: Morphological Erosion 5x5 (Using Texture)
C: Morphological Erosion 7x7 (Using Texture)
D: Morphological Dilation 3x3 (Using Texture)
E: Morphological Dilation 5x5 (Using Texture)
F: Morphological Dilation 7x7 (Using Texture)
G: Morphological Close 3x3 (Using Texture)
H: Morphological Close 5x5 (Using Texture)
I: Morphological Close 7x7 (Using Texture)
J: Morphological Open 3x3 (Using Texture)
K: Morphological Open 5x5 (Using Texture)
L: Morphological Open 7x7 (Using Texture)
M: Mean 3x3 (Using Texture)
N: Mean 5x5 (Using Texture)
O: Mean 7x7 (Using Texture)
P: Median 3x3 (Using Texture)
R: Median 5x5 (Using Texture)
```

Figure 48: *Left: CUDA interface menu. Right: Filtered X-ray image.*

Table 6: *Letters indicating type of filter and size of structuring element size used.*

Operation	*SE= 3x3*	*SE= 5x5*	*SE= 7x7*
Erosion	A	B	C
Dilation	D	E	F
Close	G	H	I
Open	J	K	L
Mean	M	N	O
Median	P	R	S
Quit/Original image		Q/T	

The effect of each of these filters on a noisy image is very different. Figure 49 illustrates the application of each of the morphological filters. The SE or kernel size plays a major role and can affect the result of any filter applied. Therefore, users should select which filter to use depending on their needs.

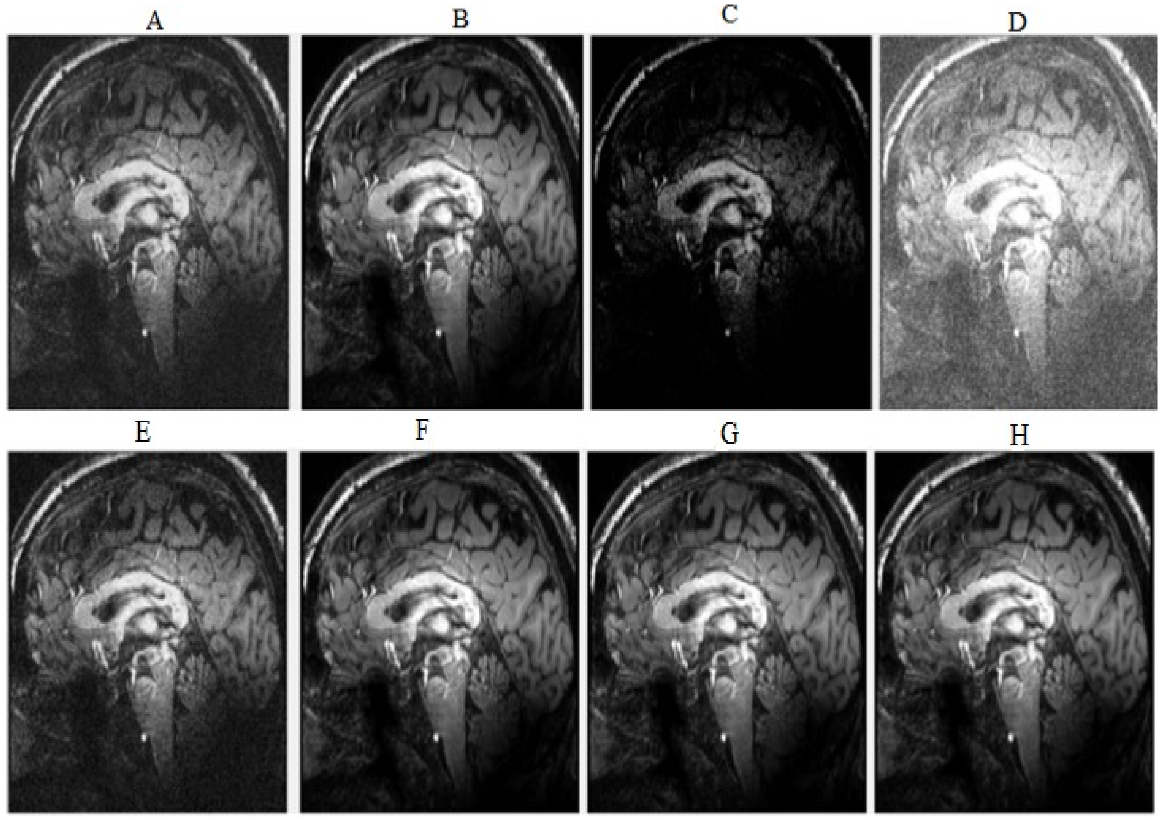


Figure 49*: Illustration of the application of morphological filters to an X-ray of the human head. A shows the input image and B shows the original image with no noise. C shows the effect of applying the erosion filter, D the dilation filter, E the median filter, F the opening filter, G the closing filter, and H the mean filter.*

In Figure 49 C (erosion), in comparison to the original image the dark details have increased and the bright pixels have greatly decreased. In image D (dilation), the dark details have been reduced or eliminated and the bright pixels are more evident. In image E, the median operation has removed noise from the image, which preserving the shape of the edges. In image F (opening), the foreground pixels or regions that have a shape similar to that of the SE have been preserved, while other pixel values have been changed. In image G (closing), the bright foreground pixels on the boundaries have increased, whereas the background pixels have decreased. In image H (mean), the amount of intensity variation has been smoothed and pixel values have become more similar to those of neighbouring pixels.

The CUDA toolbox processes digital images much more quickly than C# or MATLAB. The processing time depends upon the size of the image. For example, an images of size 1024x1024 required between 28.64 ms to 934.44 ms for processing by the Nvidia processor (GPU), depending on the filter used as compared with a times of between 5,446 ms to 16,210 ms for the CPU. The processing times measured will be discussed in detail in the following section. In these experiments, the results demonstrate that the processing speed achieved by using CUDA is much faster than that of C# or MATLAB. Although the use of CUDA requires data to be copied back and forth between the CPU memory and GPU memory, CUDA processing times are an average of 120 times faster than the times measured for the CPU.

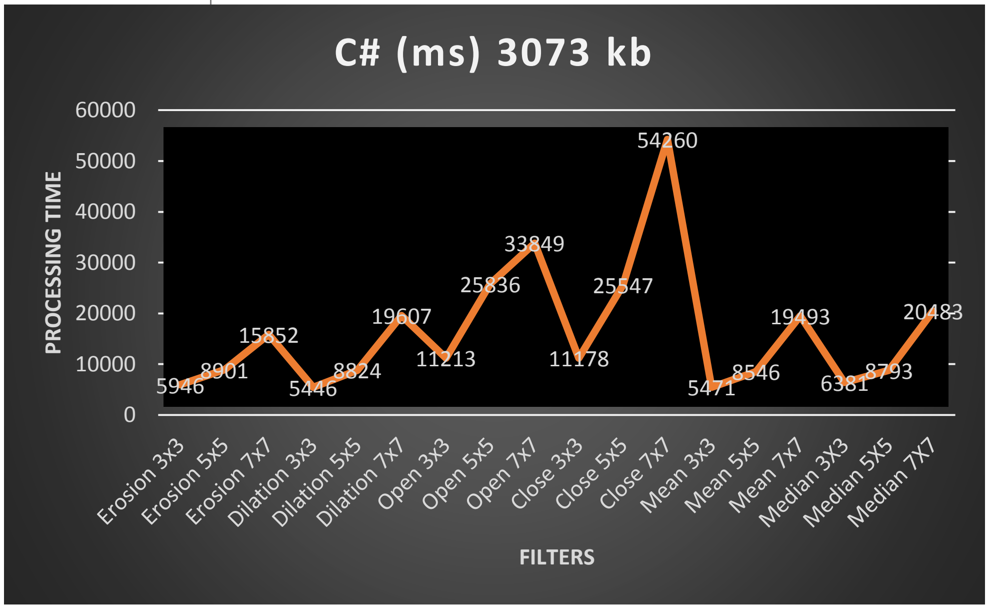
3.5 Comparison of Execution Times

This section compares the processing times measured for the CPU and GPU. The tables below summarise the performance of the CPU (with C#) and GPU (with CUDA). Table 7 shows the processing times for an image of size 3073 kb (1024 pixels × 1024 pixels). This table is followed by graphs that compare the performance of the two approaches in processing an image of size 3073 kb (see Figure 50).

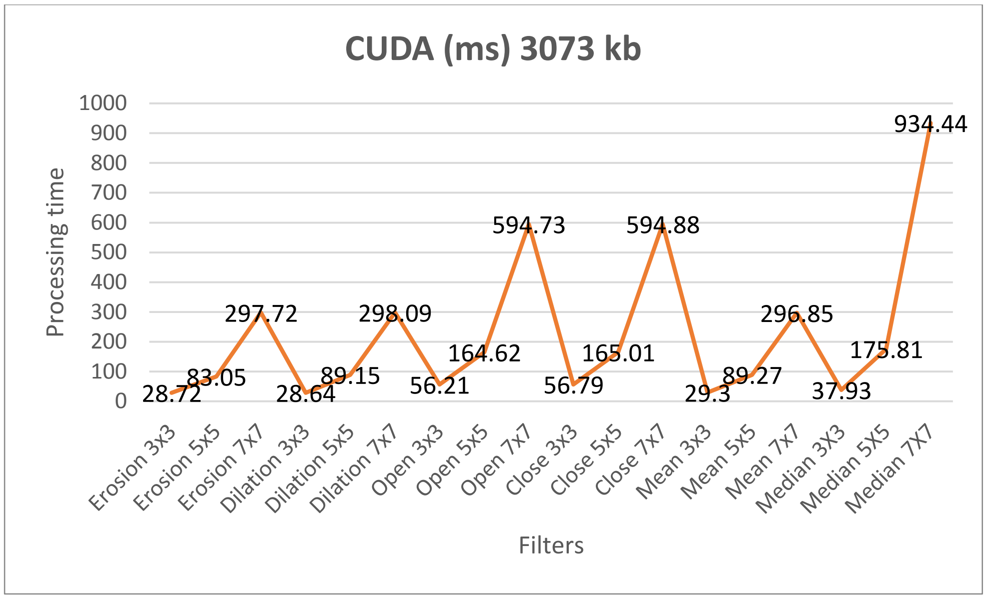
***Table** 7: Comparison of CPU (C#) and GPU (CUDA) processing times, for an image of size 3073 kb.*

Filter size	*C# (ms)*	*CUDA (ms)*
	3073 kb	**3073 kb**
Erosion 3×3	5,946	28.72
Erosion 5×5	8,901	83.05
Erosion 7×7	15,852	297.72

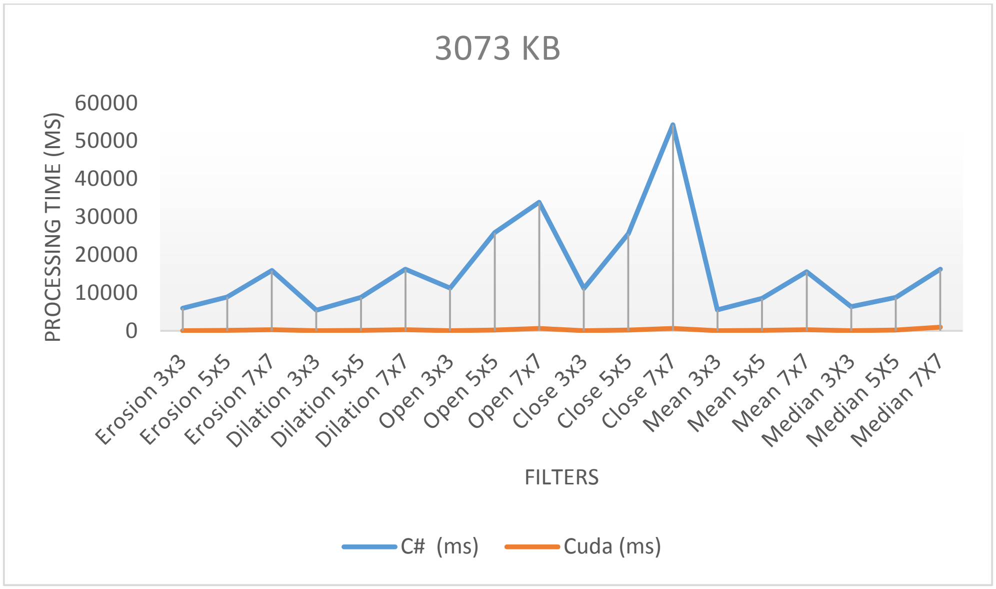
Dilation 3×3	5,446	28.64
Dilation 5×5	8,824	89.15
Dilation 7×7	19,607	298.09
Open 3×3	11,213	56.21
Open 5×5	25,836	164.62
Open 7×7	33,849	594.73
Close 3×3	11,178	56.79
Close 5×5	25,547	165.01
Close 7×7	54,260	594.88
Mean 3×3	5,471	29.3
Mean 5×5	8,546	89.27
Mean 7×7	19,493	296.85
Median 3×3	6,381	37.93
Median 5×5	8,793	175.81
Median 7×7	20,483	934.44



A



B



C

***Figure 50**: Processing times measured for C# (A) and CUDA (B), for an image of size 3073 kb. C compares the performance times of the CPU (C#) and the GPU (CUDA).*

Table 7 shows that the longest CUDA processing time, 594.88 ms, was recorded for the closing filter with the use of a square SE of size 7x7. This was because the closing operation involves the use of two filters, dilation followed by erosion. Results for the opening filter were similar; however, in both cases the performance of CUDA was many times faster than that of the CPU with C#. Table 7 shows that the fastest processing time, 28.64 ms, was recorded for the dilation filter with a SE of size 3x3, and the result for the erosion filter with SE of size 3x3 was very similar.

Table 8 compares the processing times for an image of size 2485 kb, and Figure 51 presents graphs representing the processing time of C# and the performance of CUDA.

Table 8: *Comparison of CPU (C#) and GPU (CUDA) processing times, for an image of size 2485 kb.*

Filter size	***C# (ms)***	***CUDA (ms)***
	2485 kb	**2485 kb**
Erosion 3×3	3,802	23.1
Erosion 5×5	7,346	72.61
Erosion 7×7	12,869	241.08
Dilation 3×3	3,292	21.28
Dilation 5×5	7,105	72.53
Dilation 7×7	15,534	241.61
Open 3×3	6,745	42.63
Open 5×5	14,559	133.03
Open 7×7	26,260	675.24
Close 3×3	6,561	41.16
Close 5×5	16,032	144.69
Close 7×7	28,690	483.01
Mean 3×3	3,270	21.42
Mean 5×5	7,115	72.56
Mean 7×7	15,870	239.67
Median 3×3	3,119	32.97
Median 5×5	8,215	155.67
Median 7×7	16,320	765.12

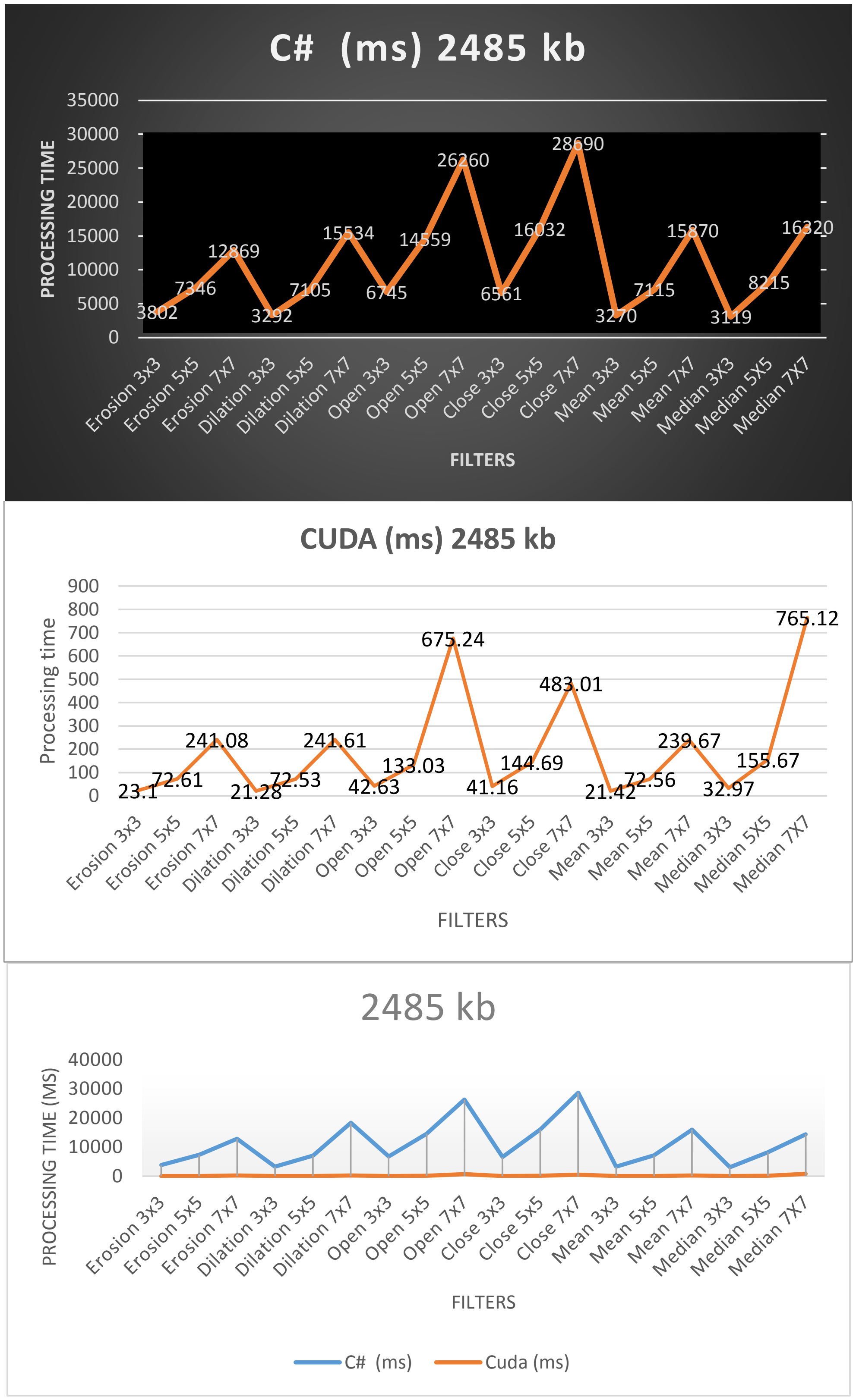


Figure 51: *Processing times measured for C# (top) and CUDA (middle), for an image of size 2485 kb. The bottom graph compares the performance times of the CPU (C#) and the GPU (CUDA).*

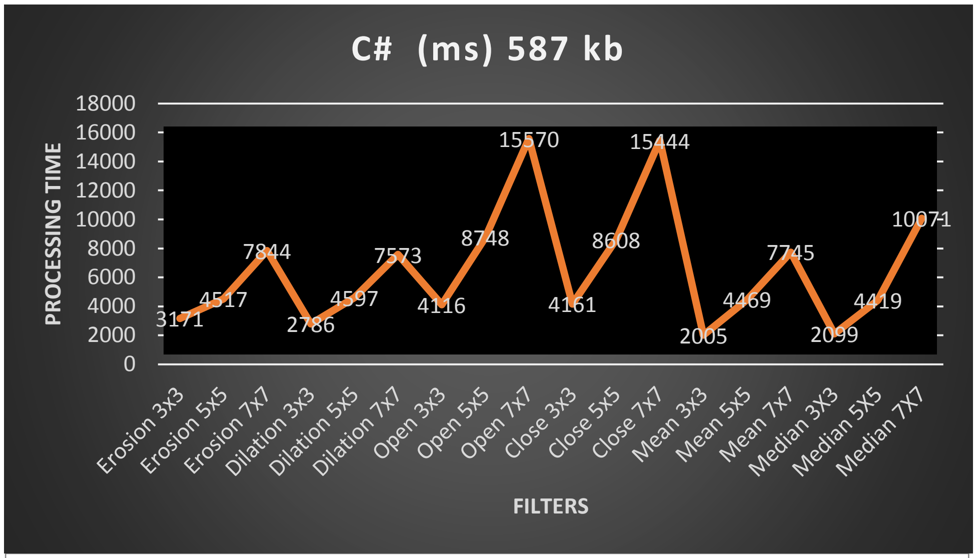
Table 8 shows that the fastest GPU processing time, 21.28 ms, was recorded for the dilation filter with a SE of size 3x3 while it was 3.292 minute with the use of CPU. Table 9 shows the processing times for an image of size 587 kb. This table is followed by three graphs that compare the performance of the CPU and the GPU (see Figure 52).

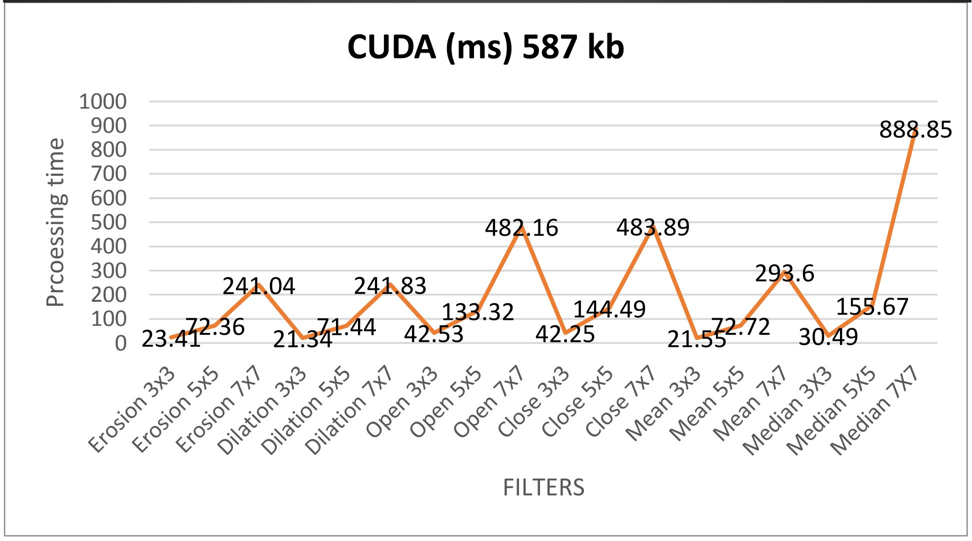
***Table 9**: Comparison of CPU (C#) and GPU (CUDA) processing times, for an image of size 587 kb.*

	C# (ms)	***CUDA (ms)***
Filter size	**587 kb**	**587 kb**
Erosion 3×3	3,171	23.41
Erosion 5×5	4,517	72.36
Erosion 7×7	7,844	241.04
Dilation 3×3	2,786	21.34
Dilation 5×5	4,597	71.44
Dilation 7×7	7,573	241.83
Open 3×3	4,116	42.53
Open 5×5	8,748	133.32
Open 7×7	15,570	482.16
Close 3×3	4,161	42.25
Close 5×5	8,608	144.49
Close 7×7	15,444	483.89
Mean 3×3	2,005	21.55
Mean 5×5	4,469	72.72
Mean 7×7	7,745	293.6
Median 3×3	2,099	30.49
Median 5×5	4,419	155.67
Median 7×7	10,071	888.85

Table 9 shows the processing times for an image of size 587 kb, and Figure 52 compares the performance of the CPU with C# to that of the GPU with CUDA.

Based on the test results, it can be seen that the parallelism used by the GPU leads to much lower processing times, especially for calculations containing many parallel elements. It can be concluded that for the types of digital image processing investigated in this research, use of the Nvidia GPU is very beneficial, because it greatly improves the software efficiency, yielding faster execution times.





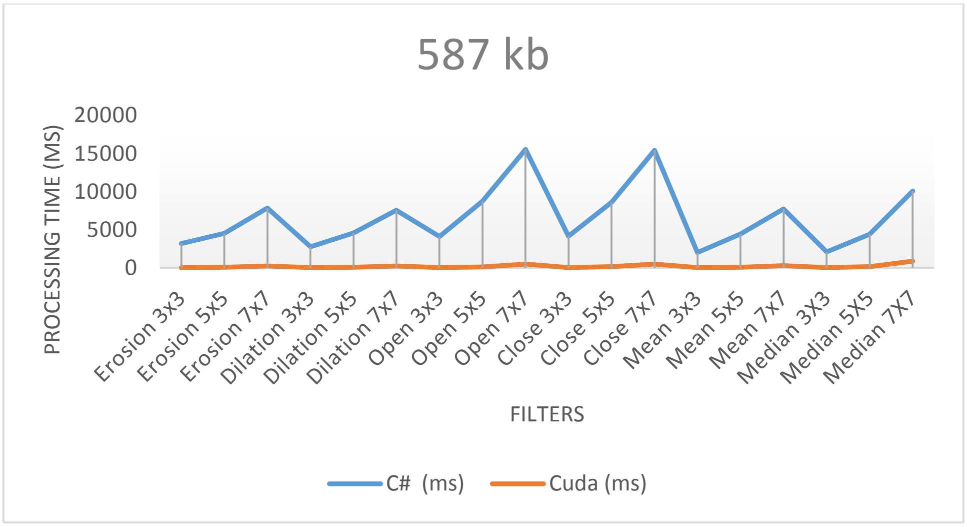


Figure 52*: Processing times measured for C# (top) and CUDA (middle), for an image of size 587 kb. The bottom graph compares the performance times of the CPU (C#) and the GPU (CUDA).*

Table 10*: Ratios between GPU and CPU processing times, for images of size 3073 kb, 2485 kb and 587 kb.*

Image size		**3073kb**	**2485kb**	**587kb**
Run time				
Erosion 3x3	C#			
	GPU	1:207	1:165	1:135
Erosion 5x5	C#			
	GPU	1:107	1:101	1:62
Erosion 7x7	C#			
	GPU	1:53	1:53	1:33
Dilation 3x3	C#			
	GPU	1:190	1:155	1:131
Dilation 5x5	C#			
	GPU	1:99	1:98	1:36
Dilation 7x7	C#			
	GPU	1:66	1:67	1:31
Open 3x3	C#			
	GPU	1:200	1:158	1:97
Open 5x5	C#			
	GPU	1:157	1:109	1:66
Open 7x7	C#			
	GPU	1:57	1:39	1:32
Close 3x3	C#			
	GPU	1:197	1:159	1:98
Close 5x5	C#			
	GPU	1:155	1:111	1:60
Close 7x7	C#			
	GPU	1:91	1:59	1:32
Mean 3x3	C#			
	GPU	1:187	1:153	1:93
Mean 5x5	C#			
	GPU	1:96	1:98	1:61

Mean 7x7	C#			
	GPU	1:66	1:66	1:26
Median 3x3	C#			
	GPU	1:168	1:95	1:69
Median 5x5	C#			
	GPU	1:50	1:53	1:29
Median 7x7	C#			
	GPU	1:22	1:21	1:11

Table 10 shows that even when processing large images, the GPU is much faster than the CPU. It can be seen that in the case of an erosion filter with SE of size 3x3, for a large image (3073 kb), the ratio between the CPU and GPU processing times is 1:207, representing an impressive gain in efficiency.

CHAPTER 4: CONCLUSION

This chapter will provide a summary of the work done, highlighting the innovations of the work, and presenting suggestions for future research. Following a brief overview of the research scope and methodologies, Section 4.1 describes the major contributions in detail and the conclusions of the research. Section 4.2 discusses future work that is related to this field.

This thesis addresses the analysis and enhancement of real-time image processing. In the initial phase, image processing analysis was done on a CPU; this was followed by implementation on a GPU. It was found that the best performance and lowest processing times could be attained by running the image enhancement algorithms simultaneously on the CPU and GPU, in order to leverage the capabilities of the GPU.

Real-time image processing is a key component of all modern communication systems, including media technologies, digital cameras, mobile devices, video conferencing and video calls. Real-time digital medical image processing applications have been developed in this thesis, focused especially on image identification and classification. The methods of digital medical image processing allow radiologists to identify abnormal tissues and subsequently make a diagnosis. Although radiologist decisions remain decisive in detecting non-palpable cancer, the applications developed in this research can help physicians to detect abnormal cells more easily and accurately than is the case with traditional techniques.

4.1 Thesis Contributions

The main contribution of this thesis is the development of a software package implementing a variety of morphological methods on a GPU. The result is a robust package that will help physicians and other detect normal and abnormal body cells. This package is able to execute complex filtering algorithms within milliseconds rather than seconds, yielding improved accuracy and performance. The following three calculation strategies are considered in this thesis.

The initial strategy is the application of mathematical morphological operations in MATLAB, with the aim of analysing digital binary and grey-scale images. In this phase, noise was added to an image in order to simulate the distortion caused by noise in medical images. Various morphological filters were then created, including erosion, dilation, opening, closing, mean and median filters. These filters were then used with structuring elements of different sizes (3×3, 5×5 and 7×7) in order to remove noise so as to restore the original images. Although morphological filters are usually applied at the binary level, in this research these filters have been used at the grey-scale level. In additional to the filters, a thresholding technique was used for the purpose of feature identification.

The second strategy is the implementation of the morphological filters in C#, which is important in order to obtain the results and then compare them with the CUDA application results. This phase processes medical digital images relatively quickly; however, processing images on a CPU is generally time-consuming, especially when the image to be processed is large in size. This problem was addressed by using CUDA.

The third strategy is the utilisation of CUDA on Nvidia graphics processing units, with the goal of using the system in real-time processing. As an example, six morphological filters were created in C# and then developed in CUDA: Erosion, dilation, opening, closing, mean and median filters. Other algorithms were used for segmentation and automatic feature identification. This work is the first investigation of real-time image processing which compares CUDA with the Intel Core i7 CPU, for parallel processing purposes. Based on knowledge of CUDA and the C# language, through calculating the processing time on both cards (Intel Core i7 and GeForce M860), the processing time is reduced and the performance enhanced. Performance improvements are much more extensive when the GPU system is employed, making it easy to select which approach to use. The CUDA system achieved processing speeds ranging from 11 times (for median filter with 7x7 SE) to 207 times (for erosion filter with 3x3 SE) faster on multi-GPU implementations.

4.2 Future work

4.2.1 Improving Performance

This thesis focuses on the total processing times attainable with the utilisation of the new Nvidia card. It can be seen that the processing times were 11 to 207 times faster when the GeForce M860 GPU was used, as compared to the Intel Core i7 CPU. This is based on the fact that with the GeForce M860 it was possible to have the GPU and CPU working constantly together, to make the image processing more efficient. Table 11 shows clearly that the GPU processing times are much faster than the CPU processing times. For an image size of 587 kb, with an erosion filter with SE of size 3x3, the processing time for the GeForce M860 was 23.4 ms, as compared to 3,171 ms for the Intel Core i7. Here it can be seen that the GPU is more than 135 times faster than the CPU, which is a significant achievement in the field of image processing. In order to build on these results, future work should include investigations of state-of-the-art GPUs to determine how they can best be utilised to achieve even more efficient performance with real-time applications.

Table 11: *Ratios between GPU and CPU processing times, demonstrating the speed of the GPU.*

Image size		**3073kb**	**587kb**
Run time			
Erosion 3x3	C#		
	GPU	1:207	1:135
Erosion 5x5	C#		
	GPU	1:107	1:62
Erosion 7x7	C#		
	GPU	1:53	1:33
Median 3x3	C#		
	GPU	1:168	1:69
Median 5x5	C#		
	GPU	1:50	1:29
Median 7x7	C#		
	GPU	1:22	1:11

4.2.2 Applying the CUDA Algorithms and the Methods Developed to Some Medical Images from Hospital Databases

Medical image processing with the use of GPUs has great potential that can dramatically improve methods of digital image processing, to provide clear and accurate results. For this reason, future work will involve using some of the newest Nvidia cards and studying how to maximise their processing times. It must be kept in mind that the accurate performance of graphics cards (GPUs) cannot be maximised in the absence of appropriate software installed and used on them. In fact, the large number of medical images demonstrates the power of CUDA and the benefits of this technology. In this research, all morphological algorithms and methods were analysed first using MATLAB codes. They were then verified through the C# platform and CUDA toolboxes. Most of the future work will involve implementing these applications on real medical images from hospital databases. Also, one of the goals of this thesis is a package with a good interface (like that in C#) but that runs the algorithms on the GPU. The first stage of this goal was achieved and will be continued on future work.

APPENDIX A: SOURCE CODE LISTING FOR C# IMPLEMENTATION

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;

using System.Diagnostics;


namespace MorphologicalOps2
{

    class MorphologicalOperations
    {
        List<Bitmap> listOfBitmaps = new List<Bitmap>();

        ProgressBar myProgressBarRef;

        byte[][] myKernelValues;

        static Random myRundomNumber = new Random();


                public void AddBmpImage(Bitmap bmp)
        {
            listOfBitmaps.Add(bmp);
        }


        public void setProgressBarReference(ProgressBar myProgressBarRef)
        {
            this.myProgressBarRef = myProgressBarRef;
        }


        public void setKernelReference(byte[][] myKernelVal)
        {
            this.myKernelValues = myKernelVal;
        }


        public Bitmap ConvertBmpToGrayscale(Bitmap inBitmap)
        {
            Bitmap outGrayScale = new Bitmap(inBitmap.Width, inBitmap.Height);

            for (int i = 0; i < inBitmap.Width; i++)
            {
                for (int x = 0; x < inBitmap.Height; x++)
                {
                    Color oc = inBitmap.GetPixel(i, x);
                    int grayScale = (int)((oc.R * 0.3) + (oc.G * 0.59) + (oc.B *
0.11));
                    Color nc = Color.FromArgb(oc.A, grayScale, grayScale,
grayScale);
                    outGrayScale.SetPixel(i, x, nc);
                }
            }

            return outGrayScale;
        }


        public Bitmap AddGaussianNoiseToGrayScale(Bitmap inBmp, double mean,
double variance)
        {

            Bitmap myGaussianNoiseImage = new Bitmap(inBmp.Width, inBmp.Height);

            double temp, rundomLog, rundomSinCos;
            int GaussianPixelValue;
            byte colorValueR;
            Color pixelGaussian = new Color();

            byte switch1 = 1;

            rundomLog = 0.0;
            for (int i = 0; i < inBmp.Height; i++)
            {

                this.myProgressBarRef.Value = (int)(((double)i /
(double)inBmp.Height) * this.myProgressBarRef.Maximum);

                for (int x = 0; x < inBmp.Width; x++)
                {
                    Color pixelValue = inBmp.GetPixel(x, i);

                    while (rundomLog == 0.0) rundomLog =
myRundomNumber.NextDouble();
                    temp = Math.Sqrt(-2.0 * variance * Math.Log(rundomLog));

                    rundomSinCos = myRundomNumber.NextDouble();

                    if (switch1 == 1)
                    {
                                                GaussianPixelValue = pixelValue.R
+ (int)(temp * Math.Cos(2 * Math.PI * rundomSinCos) + mean);
                        switch1 = 0;
                    }
                    else
                    {

                        GaussianPixelValue = pixelValue.R + (int)(temp *
Math.Sin(2 * Math.PI * rundomSinCos) + mean);
                        switch1 = 1;
                    }
                    if (GaussianPixelValue > 255)
                    {
                        GaussianPixelValue = 255;
                    }
                    else if (GaussianPixelValue < 0)
                    {
                        GaussianPixelValue = 0;
                    }
                    colorValueR = (byte)GaussianPixelValue;
                    pixelGaussian = Color.FromArgb(255 << 24 | colorValueR << 16 |
colorValueR << 8 | colorValueR);

                    myGaussianNoiseImage.SetPixel(x, i, pixelGaussian);

                    rundomLog = 0.0;
                }
            }
            this.myProgressBarRef.Value = 0;
            return myGaussianNoiseImage;
        }

        public byte[] getImagePlaneFromBmp(Bitmap inBmp, char plane)
        {
            byte[] outPlane = new byte[inBmp.Width * inBmp.Height];
            if (plane.Equals('R'))
            {

                for (int i = 0; i < inBmp.Height; i++)
                {
                    int lineOffset = i * inBmp.Width;

                    for (int j = 0; j < inBmp.Width; j++)
                    {
                        Color pixelColour = inBmp.GetPixel(j, i);
                        int pixelOffest = lineOffset + j;
                        if (plane.Equals('R'))
                        {
                            outPlane[pixelOffest] = pixelColour.R;
                        }
                        else if (plane.Equals('G'))
                        {
                            outPlane[pixelOffest] = pixelColour.G;
                        }
                        else
                        {
                            outPlane[pixelOffest] = pixelColour.B;
                        }
                    }
                }

            }

            return outPlane;
        }
        public Bitmap applyErosionToGrayScaleImage(Bitmap inGrayScale, string
kernel)
        {

           var watch = Stopwatch.StartNew();


            int RowStart, ColStart, KernelStart, KernelEnd;

            if (kernel.Equals("3x3"))
            {
                RowStart = 1;
                ColStart = 1;
                KernelStart = -1;
                KernelEnd = 2;
            }
            else if (kernel.Equals("5x5"))
            {
                RowStart = 2;
                ColStart = 2;
                KernelStart = -2;
                KernelEnd = 3;
            }
            else
            {
                RowStart = 3;
                ColStart = 3;
                KernelStart = -3;
                KernelEnd = 4;
            }

            byte[] grayScalePlane;

            grayScalePlane = this.getImagePlaneFromBmp(inGrayScale, 'R');


            Bitmap erosionImage = new Bitmap(inGrayScale.Width,
inGrayScale.Height);
            byte colorValueR;

            Color pixErode = new Color();
            Color col = new Color();
            Color col2 = new Color();

            for (int y = ColStart; y < inGrayScale.Height - ColStart; y++)
            {

                this.myProgressBarRef.Value = (int)(((double)y /
(double)inGrayScale.Height) * this.myProgressBarRef.Maximum);

                for (int x = RowStart; x < inGrayScale.Width - RowStart; x++)
                {
                    col = inGrayScale.GetPixel(x, y);

                    colorValueR = col.R;
                    for (int a = KernelStart; a < KernelEnd; a++)
                    {
                        for (int b = KernelStart; b < KernelEnd; b++)
                        {
                            if (this.myKernelValues[a - KernelStart][b - 
KernelStart].Equals(49))
                            {
                                col2 = inGrayScale.GetPixel(x + a, y + b);
                                colorValueR = Math.Min(colorValueR, col2.R);
                            }
                        }
                    }

                    int pixvalARGB = 255 << 24 | colorValueR << 16 | colorValueR 
<< 8 | colorValueR;
                    pixErode = Color.FromArgb(pixvalARGB);

                    erosionImage.SetPixel(x, y, pixErode);
                }
            }

            this.myProgressBarRef.Value = 0;
            return erosionImage;



            watch.Stop();
            var elapsedMs = watch.ElapsedMilliseconds;
        }


        public Bitmap applyDilationToGrayScaleImage(Bitmap inGrayScale, string 
kernel)
        {

            var watch = Stopwatch.StartNew();

            int RowStart, ColStart, KernelStart, KernelEnd;

            if (kernel.Equals("3x3"))
            {
                RowStart = 1;
                ColStart = 1;
                KernelStart = -1;
                KernelEnd = 2;
            }
            else if (kernel.Equals("5x5"))
            {
                RowStart = 2;
                ColStart = 2;
                KernelStart = -2;
                KernelEnd = 3;
            }
            else
            {
                RowStart = 3;
                ColStart = 3;
                KernelStart = -3;
                KernelEnd = 4;
            }


            Bitmap dilationImage = new Bitmap(inGrayScale.Width,
inGrayScale.Height);
            byte colorValueR;

            Color pixDilate = new Color();
            Color col = new Color();
            Color col2 = new Color();

            for (int y = ColStart; y < inGrayScale.Height - ColStart; y++)
            {

                this.myProgressBarRef.Value = (int)(((double)y /
(double)inGrayScale.Height) * this.myProgressBarRef.Maximum);

                for (int x = RowStart; x < inGrayScale.Width - RowStart; x++)
                {
                    col = inGrayScale.GetPixel(x, y

                    colorValueR = col.R;

                    for (int a = KernelStart; a < KernelEnd; a++)
                    {
                        for (int b = KernelStart; b < KernelEnd; b++)
                        {
                            if (this.myKernelValues[a - KernelStart][b -
KernelStart].Equals(49))
                            {
                                col2 = inGrayScale.GetPixel(x + a, y + b);
                                colorValueR = Math.Max(colorValueR, col2.R);
                            }
                        }
                    }

                    int pixvalARGB = 255 << 24 | colorValueR << 16 | colorValueR
<< 8 | colorValueR;
                    pixDilate = Color.FromArgb(pixvalARGB);

                    dilationImage.SetPixel(x, y, pixDilate);
                }
            }


            this.myProgressBarRef.Value = 0;

            return dilationImage;

            watch.Stop();
            var elapsedMs = watch.ElapsedMilliseconds;

         }

        public Bitmap applyMedianToGrayScaleImage(Bitmap inGrayScale, string
kernel)
        {
            var watch = Stopwatch.StartNew();


            int RowStart, ColStart, KernelStart, KernelEnd;

            List<byte> pixelList = new List<byte>();

            if (kernel.Equals("3x3"))
            {
                RowStart = 1;
                ColStart = 1;
                KernelStart = -1;
                KernelEnd = 2;
            }
            else if (kernel.Equals("5x5"))
            {
                RowStart = 2;
                ColStart = 2;
                KernelStart = -2;
                KernelEnd = 3;
            }
            else
            {
                RowStart = 3;
                ColStart = 3;
                KernelStart = -3;
                KernelEnd = 4;
            }


            Bitmap medianImage = new Bitmap(inGrayScale.Width,
inGrayScale.Height);
            byte colorValueR;

            Color pixMedian = new Color();
            Color col = new Color();
            Color col2 = new Color();

            for (int y = ColStart; y < inGrayScale.Height - ColStart; y++)
            {

                this.myProgressBarRef.Value = (int)(((double)y /
(double)inGrayScale.Height) * this.myProgressBarRef.Maximum);

                for (int x = RowStart; x < inGrayScale.Width - RowStart; x++)
                {
                    col = inGrayScale.GetPixel(x, y);

                    colorValueR = col.R;

                    for (int a = KernelStart; a < KernelEnd; a++)
                    {
                        for (int b = KernelStart; b < KernelEnd; b++)
                        {

                            col2 = inGrayScale.GetPixel(x + a, y + b);
                            pixelList.Add(col2.R);
                        }
                    }

                    pixelList.Sort();

                    colorValueR = pixelList.ElementAt((pixelList.Count - 1) / 2);
                    pixelList.Clear();

                    int pixvalARGB = 255 << 24 | colorValueR << 16 | colorValueR
<< 8 | colorValueR;
                    pixMedian = Color.FromArgb(pixvalARGB);

                    medianImage.SetPixel(x, y, pixMedian);
                }
            }


            this.myProgressBarRef.Value = 0;

            return medianImage;

            watch.Stop();
            var elapsedMs = watch.ElapsedMilliseconds;
        }



        public Bitmap applyMeanToGrayScaleImage(Bitmap inGrayScale, string kernel)
        {
            var watch = Stopwatch.StartNew();


            int RowStart, ColStart, KernelStart, KernelEnd;

            List<byte> pixelList = new List<byte>();

            if (kernel.Equals("3x3"))
            {
                RowStart = 1;
                ColStart = 1;
                KernelStart = -1;
                KernelEnd = 2;
            }
            else if (kernel.Equals("5x5"))
            {
                RowStart = 2;
                ColStart = 2;
                KernelStart = -2;
                KernelEnd = 3;
            }
            else
            {
                RowStart = 3;
                ColStart = 3;
                KernelStart = -3;
                KernelEnd = 4;
            }
            Bitmap meanImage = new Bitmap(inGrayScale.Width, inGrayScale.Height);
```

```
            byte colorValueR;
            Color pixMedian = new Color();
            Color col = new Color();
            Color col2 = new Color();
            for (int y = ColStart; y < inGrayScale.Height - ColStart; y++)
            {

                this.myProgressBarRef.Value = (int)(((double)y /
(double)inGrayScale.Height) * this.myProgressBarRef.Maximum);

                for (int x = RowStart; x < inGrayScale.Width - RowStart; x++)
                {
                    col = inGrayScale.GetPixel(x, y);

                    colorValueR = col.R;
                    double meanValue = 0;

                    for (int a = KernelStart; a < KernelEnd; a++)
                    {
                        for (int b = KernelStart; b < KernelEnd; b++)
                        {

                            col2 = inGrayScale.GetPixel(x + a, y + b);
                            pixelList.Add(col2.R);
                            meanValue += col2.R;
                        }
                    }


                    colorValueR = (byte)Math.Round(meanValue / pixelList.Count());

                    pixelList.Clear();

                    int pixvalARGB = 255 << 24 | colorValueR << 16 | colorValueR
<< 8 | colorValueR;
                    pixMedian = Color.FromArgb(pixvalARGB);

                    meanImage.SetPixel(x, y, pixMedian);
                }
            }


            this.myProgressBarRef.Value = 0;

            return meanImage;

            watch.Stop();
            var elapsedMs = watch.ElapsedMilliseconds;
        }

    }
}
```

```
namespace MorphologicalOps2
{
    static class Program
    {

        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}




namespace MorphologicalOps2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            kernel.Add("3x3");
            kernel.Add("5x5");
            kernel.Add("7x7");

            this.comboBox1.DataSource = kernel;

            this.textBox00.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox01.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox02.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox03.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox04.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox05.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox06.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox10.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox11.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox12.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox13.Click += new
System.EventHandler(this.textBoxKerenl_Click);
```

```
            this.textBox14.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox15.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox16.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox20.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox21.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox22.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox23.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox24.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox25.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox26.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox30.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox31.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox32.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox33.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox34.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox35.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox36.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox40.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox41.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox42.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox43.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox44.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox45.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox46.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox50.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox51.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox52.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox53.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox54.Click += new
System.EventHandler(this.textBoxKerenl_Click);
```

```
            this.textBox55.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox56.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox60.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox61.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox62.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox63.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox64.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox65.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            this.textBox66.Click += new
System.EventHandler(this.textBoxKerenl_Click);
            kerenlTexBoxes.Add(this.textBox00);
            kerenlTexBoxes.Add(this.textBox01);
            kerenlTexBoxes.Add(this.textBox02);
            kerenlTexBoxes.Add(this.textBox03);
            kerenlTexBoxes.Add(this.textBox04);
            kerenlTexBoxes.Add(this.textBox05);
            kerenlTexBoxes.Add(this.textBox06);
            kerenlTexBoxes.Add(this.textBox10);
            kerenlTexBoxes.Add(this.textBox11);
            kerenlTexBoxes.Add(this.textBox12);
            kerenlTexBoxes.Add(this.textBox13);
            kerenlTexBoxes.Add(this.textBox14);
            kerenlTexBoxes.Add(this.textBox15);
            kerenlTexBoxes.Add(this.textBox16);
            kerenlTexBoxes.Add(this.textBox20);
            kerenlTexBoxes.Add(this.textBox21);
            kerenlTexBoxes.Add(this.textBox22);
            kerenlTexBoxes.Add(this.textBox23);
            kerenlTexBoxes.Add(this.textBox24);
            kerenlTexBoxes.Add(this.textBox25);
            kerenlTexBoxes.Add(this.textBox26);
            kerenlTexBoxes.Add(this.textBox30);
            kerenlTexBoxes.Add(this.textBox31);
            kerenlTexBoxes.Add(this.textBox32);
            kerenlTexBoxes.Add(this.textBox33);
            kerenlTexBoxes.Add(this.textBox34);
            kerenlTexBoxes.Add(this.textBox35);
            kerenlTexBoxes.Add(this.textBox36);
            kerenlTexBoxes.Add(this.textBox40);
            kerenlTexBoxes.Add(this.textBox41);
            kerenlTexBoxes.Add(this.textBox42);
            kerenlTexBoxes.Add(this.textBox43);
            kerenlTexBoxes.Add(this.textBox44);
            kerenlTexBoxes.Add(this.textBox45);
            kerenlTexBoxes.Add(this.textBox46);
            kerenlTexBoxes.Add(this.textBox50);
            kerenlTexBoxes.Add(this.textBox51);
            kerenlTexBoxes.Add(this.textBox52);
            kerenlTexBoxes.Add(this.textBox53);
            kerenlTexBoxes.Add(this.textBox54);
            kerenlTexBoxes.Add(this.textBox55);
            kerenlTexBoxes.Add(this.textBox56);
            kerenlTexBoxes.Add(this.textBox60);
            kerenlTexBoxes.Add(this.textBox61);
            kerenlTexBoxes.Add(this.textBox62);
            kerenlTexBoxes.Add(this.textBox63);
            kerenlTexBoxes.Add(this.textBox64);
            kerenlTexBoxes.Add(this.textBox65);
            kerenlTexBoxes.Add(this.textBox66);

            GaussianVarianceValue = trackBar1.Value;

        }

        string imageSourceDirectory = "c:\\";

        List<string> kernel= new List<string>();
        List<Bitmap> imageList = new List<Bitmap>();
        List<Bitmap> grayScaleList = new List<Bitmap>();

        List<TextBox> kerenlTexBoxes = new List<TextBox>();

        int GaussianVarianceValue;

        private byte[][] getKernelValues(string kernel)
        {
            byte[][] myKernel;
            if (kernel.Equals("3x3"))
            {
                myKernel = new byte[3][];
                myKernel[0] = new byte[3]{
(byte)this.textBox00.Text.ToCharArray().ElementAt(0),

(byte)this.textBox01.Text.ToCharArray().ElementAt(0),

(byte)this.textBox02.Text.ToCharArray().ElementAt(0)};

                myKernel[1] = new byte[3]{
(byte)this.textBox10.Text.ToCharArray().ElementAt(0),

(byte)this.textBox11.Text.ToCharArray().ElementAt(0),

(byte)this.textBox12.Text.ToCharArray().ElementAt(0)};

                myKernel[2] = new byte[3]{
(byte)this.textBox20.Text.ToCharArray().ElementAt(0),

(byte)this.textBox21.Text.ToCharArray().ElementAt(0),

(byte)this.textBox22.Text.ToCharArray().ElementAt(0)};

            }
            else if (kernel.Equals("5x5"))
            {
                myKernel = new byte[5][];
                myKernel[0] = new byte[5]{
(byte)this.textBox00.Text.ToCharArray().ElementAt(0),
(byte)this.textBox01.Text.ToCharArray().ElementAt(0),

(byte)this.textBox02.Text.ToCharArray().ElementAt(0),

(byte)this.textBox03.Text.ToCharArray().ElementAt(0),

(byte)this.textBox04.Text.ToCharArray().ElementAt(0)};

                myKernel[1] = new byte[5]{
(byte)this.textBox10.Text.ToCharArray().ElementAt(0),

(byte)this.textBox11.Text.ToCharArray().ElementAt(0),

(byte)this.textBox12.Text.ToCharArray().ElementAt(0),

(byte)this.textBox13.Text.ToCharArray().ElementAt(0),

(byte)this.textBox14.Text.ToCharArray().ElementAt(0)};

                myKernel[2] = new byte[5]{
(byte)this.textBox20.Text.ToCharArray().ElementAt(0),

(byte)this.textBox21.Text.ToCharArray().ElementAt(0),

(byte)this.textBox22.Text.ToCharArray().ElementAt(0),

(byte)this.textBox23.Text.ToCharArray().ElementAt(0),

(byte)this.textBox24.Text.ToCharArray().ElementAt(0)};

                myKernel[3] = new byte[5]{
(byte)this.textBox30.Text.ToCharArray().ElementAt(0),

(byte)this.textBox31.Text.ToCharArray().ElementAt(0),

(byte)this.textBox32.Text.ToCharArray().ElementAt(0),

(byte)this.textBox33.Text.ToCharArray().ElementAt(0),

(byte)this.textBox34.Text.ToCharArray().ElementAt(0)};

                myKernel[4] = new byte[5]{
(byte)this.textBox40.Text.ToCharArray().ElementAt(0),

(byte)this.textBox41.Text.ToCharArray().ElementAt(0),

(byte)this.textBox42.Text.ToCharArray().ElementAt(0),

(byte)this.textBox43.Text.ToCharArray().ElementAt(0),

(byte)this.textBox44.Text.ToCharArray().ElementAt(0)};
            }
            else
            {
                myKernel = new byte[7][];
                myKernel[0] = new byte[7]{
(byte)this.textBox00.Text.ToCharArray().ElementAt(0),
(byte)this.textBox01.Text.ToCharArray().ElementAt(0),

(byte)this.textBox02.Text.ToCharArray().ElementAt(0),

(byte)this.textBox03.Text.ToCharArray().ElementAt(0),

(byte)this.textBox04.Text.ToCharArray().ElementAt(0),

(byte)this.textBox05.Text.ToCharArray().ElementAt(0),

(byte)this.textBox06.Text.ToCharArray().ElementAt(0)};

                myKernel[1] = new byte[7]{
(byte)this.textBox10.Text.ToCharArray().ElementAt(0),

(byte)this.textBox11.Text.ToCharArray().ElementAt(0),

(byte)this.textBox12.Text.ToCharArray().ElementAt(0),

(byte)this.textBox13.Text.ToCharArray().ElementAt(0),

(byte)this.textBox14.Text.ToCharArray().ElementAt(0),

(byte)this.textBox15.Text.ToCharArray().ElementAt(0),

(byte)this.textBox16.Text.ToCharArray().ElementAt(0)};

                myKernel[2] = new byte[7]{
(byte)this.textBox20.Text.ToCharArray().ElementAt(0),

(byte)this.textBox21.Text.ToCharArray().ElementAt(0),

(byte)this.textBox22.Text.ToCharArray().ElementAt(0),

(byte)this.textBox23.Text.ToCharArray().ElementAt(0),

(byte)this.textBox24.Text.ToCharArray().ElementAt(0),

(byte)this.textBox25.Text.ToCharArray().ElementAt(0),

(byte)this.textBox26.Text.ToCharArray().ElementAt(0)};

                myKernel[3] = new byte[7]{
(byte)this.textBox30.Text.ToCharArray().ElementAt(0),

(byte)this.textBox31.Text.ToCharArray().ElementAt(0),

(byte)this.textBox32.Text.ToCharArray().ElementAt(0),

(byte)this.textBox33.Text.ToCharArray().ElementAt(0),

(byte)this.textBox34.Text.ToCharArray().ElementAt(0),

(byte)this.textBox35.Text.ToCharArray().ElementAt(0),

(byte)this.textBox36.Text.ToCharArray().ElementAt(0)};
```

```
                myKernel[4] = new byte[7]{
(byte)this.textBox40.Text.ToCharArray().ElementAt(0),

(byte)this.textBox41.Text.ToCharArray().ElementAt(0),

(byte)this.textBox42.Text.ToCharArray().ElementAt(0),

(byte)this.textBox43.Text.ToCharArray().ElementAt(0),

(byte)this.textBox44.Text.ToCharArray().ElementAt(0),

(byte)this.textBox45.Text.ToCharArray().ElementAt(0),

(byte)this.textBox46.Text.ToCharArray().ElementAt(0)};

                myKernel[5] = new byte[7]{
(byte)this.textBox50.Text.ToCharArray().ElementAt(0),

(byte)this.textBox51.Text.ToCharArray().ElementAt(0),

(byte)this.textBox52.Text.ToCharArray().ElementAt(0),

(byte)this.textBox53.Text.ToCharArray().ElementAt(0),

(byte)this.textBox54.Text.ToCharArray().ElementAt(0),

(byte)this.textBox55.Text.ToCharArray().ElementAt(0),

(byte)this.textBox56.Text.ToCharArray().ElementAt(0)};

                myKernel[6] = new byte[7]{
(byte)this.textBox60.Text.ToCharArray().ElementAt(0),

(byte)this.textBox61.Text.ToCharArray().ElementAt(0),

(byte)this.textBox62.Text.ToCharArray().ElementAt(0),

(byte)this.textBox63.Text.ToCharArray().ElementAt(0),

(byte)this.textBox64.Text.ToCharArray().ElementAt(0),

(byte)this.textBox65.Text.ToCharArray().ElementAt(0),

(byte)this.textBox66.Text.ToCharArray().ElementAt(0)};
            }
            return myKernel;
        }
        private void erosionButton_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = imageList.Last();
                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                myMorphus.AddBmpImage(myBitmap);
                string kernel = this.comboBox1.SelectedItem.ToString();

                byte[][] myKernelValues = this.getKernelValues(kernel);

                myMorphus.setKernelReference(myKernelValues);

                Bitmap myErosionImage = 
myMorphus.applyErosionToGrayScaleImage(grayScaleList.Last(), kernel);

                this.pictureBox2.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox2.Image = myErosionImage;
            }
            catch(Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original 
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }

        }

        private void dilationButton_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = imageList.Last();

                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                myMorphus.AddBmpImage(myBitmap);
                Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);

                string kernel = this.comboBox1.SelectedItem.ToString();

                byte[][] myKernelValues = this.getKernelValues(kernel);

                myMorphus.setKernelReference(myKernelValues);

                Bitmap myDilationImage = 
myMorphus.applyDilationToGrayScaleImage(grayScaleList.Last(), kernel);
                this.pictureBox3.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox3.Image = myDilationImage;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original 
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }
        }

        private void openImage_Click(object sender, EventArgs e)
        {
            OpenFileDialog openFileDialog1 = new OpenFileDialog();


            openFileDialog1.FilterIndex = 2;
            openFileDialog1.RestoreDirectory = true;

            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                try
                {
                    System.IO.FileInfo fInfo = new
System.IO.FileInfo(openFileDialog1.FileName);

                    string strFileName = fInfo.Name;

                    string strFilePath = fInfo.DirectoryName;
                    this.imageSourceDirectory = strFilePath;

                    string fullPath    = fInfo.DirectoryName + "\\" + fInfo.Name;

                    Bitmap myBitmap = (Bitmap)Bitmap.FromFile(@fullPath);

                    imageList.Add(myBitmap);

                    MorphologicalOperations myMorphus = new
MorphologicalOperations();

                    myMorphus.AddBmpImage(myBitmap);
                    Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);
                    grayScaleList.Add(grayScale);

                    this.pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;

                    this.pictureBox1.Image = myBitmap;
                    this.pictureBox2.Image = null;
                    this.pictureBox3.Image = null;
                    this.pictureBox4.Image = null;
                    this.pictureBox5.Image = null;
                    this.pictureBox6.Image = null;
                    this.pictureBox7.Image = null;
                    this.pictureBox8.Image = null;
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Error: Could not read file from disk.
Original error: " + ex.Message,
                                        "Exception Message",
                                        MessageBoxButtons.OK,
                                        MessageBoxIcon.Exclamation);
                }
            }
        }
        private void Open_Click(object sender, EventArgs e)
        {
            try
            {

            Bitmap myBitmap = imageList.Last();

            MorphologicalOperations myMorphus = new MorphologicalOperations();

            this.progressBar1.Value = 1;
            myMorphus.setProgressBarReference(this.progressBar1);

            myMorphus.AddBmpImage(myBitmap);
            Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);

            string kernel = this.comboBox1.SelectedItem.ToString();

            byte[][] myKernelValues = this.getKernelValues(kernel);

            myMorphus.setKernelReference(myKernelValues);
            Bitmap erodedImage =
myMorphus.applyErosionToGrayScaleImage(grayScaleList.Last(), kernel);

            this.progressBar1.Value = 1;

            Bitmap dilateErodedImage =
myMorphus.applyDilationToGrayScaleImage(erodedImage, kernel);

            this.pictureBox4.SizeMode = PictureBoxSizeMode.StretchImage;

            this.pictureBox4.Image = dilateErodedImage;
            }
            catch(Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original
error: " + ex.Message,
                                           "Exception Message",
                                           MessageBoxButtons.OK,
                                           MessageBoxIcon.Exclamation);
            }
        }

        private void Close_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = imageList.Last();

                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                myMorphus.AddBmpImage(myBitmap);
                Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);

                string kernel = this.comboBox1.SelectedItem.ToString();
                byte[][] myKernelValues = this.getKernelValues(kernel);

                myMorphus.setKernelReference(myKernelValues);
                Bitmap dilateImage = 
myMorphus.applyDilationToGrayScaleImage(grayScaleList.Last(), kernel);

                this.progressBar1.Value = 1;

                Bitmap erodedDilatedImage = 
myMorphus.applyErosionToGrayScaleImage(dilateImage, kernel);

                this.pictureBox5.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox5.Image = erodedDilatedImage;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original 
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }
        }

        private void Median_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = imageList.Last();

                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                myMorphus.AddBmpImage(myBitmap);
                Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);

                string kernel = this.comboBox1.SelectedItem.ToString();

                byte[][] myKernelValues = this.getKernelValues(kernel);

                myMorphus.setKernelReference(myKernelValues);
                Bitmap medianImage = 
myMorphus.applyMedianToGrayScaleImage(grayScaleList.Last(), kernel);
                this.pictureBox6.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox6.Image = medianImage;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original 
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }
        }

        private void Mean_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = imageList.Last();

                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                myMorphus.AddBmpImage(myBitmap);
                Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);

                string kernel = this.comboBox1.SelectedItem.ToString();

                byte[][] myKernelValues = this.getKernelValues(kernel);

                myMorphus.setKernelReference(myKernelValues);
                Bitmap meanImage =
myMorphus.applyMeanToGrayScaleImage(grayScaleList.Last(), kernel);

                this.pictureBox7.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox7.Image = meanImage;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }
        }

        private void textBoxToggleValue(TextBox myTextBox)
        {
            string text = myTextBox.Text;
            if (text.Equals("1"))
            {
                myTextBox.Text = "0";
                myTextBox.BackColor = Color.Red;
            }
            else
            {
                myTextBox.Text = "1";
                myTextBox.BackColor = Color.Silver;
            }

        }

        private void textBoxKerenl_Click(object sender, EventArgs e)
        {
            TextBox eventTextBox = (TextBox)sender;
            this.textBoxToggleValue(eventTextBox);
        }

        private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            this.pictureBox2.Image = null;
            this.pictureBox3.Image = null;
            this.pictureBox4.Image = null;
            this.pictureBox5.Image = null;
            this.pictureBox6.Image = null;
            this.pictureBox7.Image = null;

            for (int i = 0; i < this.kerenlTexBoxes.Count; i++)
            {
                this.kerenlTexBoxes.ElementAt(i).Enabled = false;
                this.kerenlTexBoxes.ElementAt(i).Visible = false;
                this.kerenlTexBoxes.ElementAt(i).Text = "1";
                this.kerenlTexBoxes.ElementAt(i).BackColor = Color.Silver;
            }

            string kernel = this.comboBox1.SelectedItem.ToString();
            if (kernel.Equals("3x3"))
            {
                this.textBox00.Enabled = true;
                this.textBox01.Enabled = true;
                this.textBox02.Enabled = true;

                this.textBox10.Enabled = true;
                this.textBox11.Enabled = false;
                this.textBox12.Enabled = true;

                this.textBox20.Enabled = true;
                this.textBox21.Enabled = true;
                this.textBox22.Enabled = true;
                this.textBox00.Visible = true;
                this.textBox01.Visible = true;
                this.textBox02.Visible = true;

                this.textBox10.Visible = true;
                this.textBox11.Visible = true;
                this.textBox12.Visible = true;

                this.textBox20.Visible = true;
                this.textBox21.Visible = true;
                this.textBox22.Visible = true;
            }
            else if (kernel.Equals("5x5"))
            {
                this.textBox00.Enabled = true;
                this.textBox01.Enabled = true;
                this.textBox02.Enabled = true;
                this.textBox03.Enabled = true;
                this.textBox04.Enabled = true;

                this.textBox10.Enabled = true;
                this.textBox11.Enabled = true;
                this.textBox12.Enabled = true;
                this.textBox13.Enabled = true;
                this.textBox14.Enabled = true;
                this.textBox20.Enabled = true;
                this.textBox21.Enabled = true;
                this.textBox22.Enabled = false;
                this.textBox23.Enabled = true;
                this.textBox24.Enabled = true;

                this.textBox30.Enabled = true;
                this.textBox31.Enabled = true;
                this.textBox32.Enabled = true;
                this.textBox33.Enabled = true;
                this.textBox34.Enabled = true;

                this.textBox40.Enabled = true;
                this.textBox41.Enabled = true;
                this.textBox42.Enabled = true;
                this.textBox43.Enabled = true;
                this.textBox44.Enabled = true;

                this.textBox00.Visible = true;
                this.textBox01.Visible = true;
                this.textBox02.Visible = true;
                this.textBox03.Visible = true;
                this.textBox04.Visible = true;

                this.textBox10.Visible = true;
                this.textBox11.Visible = true;
                this.textBox12.Visible = true;
                this.textBox13.Visible = true;
                this.textBox14.Visible = true;

                this.textBox20.Visible = true;
                this.textBox21.Visible = true;
                this.textBox22.Visible = true;
                this.textBox23.Visible = true;
                this.textBox24.Visible = true;

                this.textBox30.Visible = true;
                this.textBox31.Visible = true;
                this.textBox32.Visible = true;
                this.textBox33.Visible = true;
                this.textBox34.Visible = true;

                this.textBox40.Visible = true;
                this.textBox41.Visible = true;
                this.textBox42.Visible = true;
                this.textBox43.Visible = true;
                this.textBox44.Visible = true;
            }
            else
            {
                for (int i = 0; i < this.kerenlTexBoxes.Count; i++)
                {
                    this.kerenlTexBoxes.ElementAt(i).Enabled = true;
                    this.kerenlTexBoxes.ElementAt(i).Visible = true;
                }
                this.textBox33.Enabled = false;
            }
```

```
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void button2_Click(object sender, EventArgs e)
        {
            OpenFileDialog openFileDialog8 = new OpenFileDialog();

            openFileDialog8.InitialDirectory = this.imageSourceDirectory;
            openFileDialog8.Filter = "(*.bmp)|*.bmp|(*.jpg)|*.jpg|DICOM Files
(*.dcm;*.dic)|*.dcm;*.dic|All Files (*.*)|*.*";
            openFileDialog8.FilterIndex = 2;
            openFileDialog8.RestoreDirectory = true;

            if (openFileDialog8.ShowDialog() == DialogResult.OK)
            {
                try
                {
                    System.IO.FileInfo fInfo = new
System.IO.FileInfo(openFileDialog8.FileName);

                    string strFileName = fInfo.Name;

                    string strFilePath = fInfo.DirectoryName;
                    this.imageSourceDirectory = strFilePath;

                    string fullPath = fInfo.DirectoryName + "\\" + fInfo.Name;

                    Bitmap myBitmap1 = (Bitmap)Bitmap.FromFile(@fullPath);


                    int width = myBitmap1.Width;
                    int height = myBitmap1.Height;

                    Color p;

                    for (int y = 0; y < height; y++)
                    {
                        for (int x = 0; x < width; x++)
                        {
                            p = myBitmap1.GetPixel(x, y);

                            int a = p.A;
                            int r = p.R;
                            int g = p.G;
                            int b = p.B;

                            int avg = (r + g + b) / 3;

                       myBitmap1.SetPixel(x, y, Color.FromArgb(a, avg, avg, avg));
                        }
                    }

                    pictureBox8.Image = myBitmap1;
                    imageList.Add(myBitmap1);

                    MorphologicalOperations myMorphus = new
MorphologicalOperations();

                    myMorphus.AddBmpImage(myBitmap1);
                    Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap1);
                    grayScaleList.Add(grayScale);

                    this.pictureBox8.SizeMode = PictureBoxSizeMode.StretchImage;

                    this.pictureBox8.Image = myBitmap1;
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Error: Could not read file from disk.
Original error: " + ex.Message,
                                        "Exception Message",
                                        MessageBoxButtons.OK,
                                        MessageBoxIcon.Exclamation);
                }


            }
        }

        private void pictureBox1_Click(object sender, EventArgs e)
        {

        }


        private void button3_Click(object sender, EventArgs e)
        {
            try
            {
                Bitmap myBitmap = imageList.Last();

                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                myMorphus.AddBmpImage(myBitmap);
                Bitmap grayScale = myMorphus.ConvertBmpToGrayscale(myBitmap);
                string kernel = this.comboBox1.SelectedItem.ToString();

                byte[][] myKernelValues = this.getKernelValues(kernel);

                myMorphus.setKernelReference(myKernelValues);
                Bitmap meanImage =
myMorphus.applyMeanToGrayScaleImage(grayScaleList.Last(), kernel);

                this.progressBar1.Value = 1;

                Bitmap dilateImage =
myMorphus.applyDilationToGrayScaleImage(meanImage, kernel);
```

```
                this.progressBar1.Value = 1;

                Bitmap erodedDilatedImage =
myMorphus.applyErosionToGrayScaleImage(dilateImage, kernel);

                this.progressBar1.Value = 1;



                this.pictureBox9.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox9.Image = erodedDilatedImage;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }
        }

        private void pictureBox7_Click(object sender, EventArgs e)
        {

        }

        private void pictureBox9_Click(object sender, EventArgs e)
        {

        }

        private void AddGaussianNoise_Click(object sender, EventArgs e)
        {
            try
            {

                Bitmap myBitmap = imageList.Last();

                MorphologicalOperations myMorphus = new MorphologicalOperations();

                this.progressBar1.Value = 1;
                myMorphus.setProgressBarReference(this.progressBar1);

                Bitmap gaussianImage =
myMorphus.AddGaussianNoiseToGrayScale(grayScaleList.Last(), 0,
this.GaussianVarianceValue);

                grayScaleList.Add(gaussianImage);
                this.pictureBox7.SizeMode = PictureBoxSizeMode.StretchImage;

                this.pictureBox1.Image = gaussianImage;
            }
            catch (Exception ex)
            {
                MessageBox.Show("Error: Could not read file from disk. Original
error: " + ex.Message,
                                            "Exception Message",
                                            MessageBoxButtons.OK,
                                            MessageBoxIcon.Exclamation);
            }
        }


        private void trackBar1_Scroll_1(object sender, EventArgs e)
        {
            VarianceValue.Text = trackBar1.Value.ToString();
            this.GaussianVarianceValue = trackBar1.Value;
        }

        private void pictureBox3_Click(object sender, EventArgs e)
        {

        }

    }
}
```

APPENDIX B: SOURCE CODE LISTING FOR CUDA IMPLEMENTATION

```
#include <GL/freeglut.h>
#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <helper_functions.h>
#include <helper_cuda.h>
#include <time.h>

const char *filterMode[] =
{
    "No Filtering",
/* For inquiries please contact k.m.a.q@hotmail.com */
       "Morphological Erosion 3x3",
       "Morphological Erosion 5x5",
       "Morphological Erosion 7x7",
       "Morphological Dilation 3x3",
       "Morphological Dilation 5x5",
       "Morphological Dilation 7x7",
       "Morphological Close 3x3",
       "Morphological Close 5x5",
       "Morphological Close 7x7",
       "Morphological_Open 3x3",
       "Morphological_Open 5x5",
       "Morphological_Open 7x7",
       "Mean Operation 3x3",
       "Mean Operation 5x5",
       "Mean Operation 7x7",
       "Median operation 3x3",
       "Median operation 5x5",
       "Median operation 7x7",

    NULL
};


void cleanup(void);
void initializeData(char *file) ;

#define REFRESH_DELAY     1000
static int wWidth   = 512;
static int wHeight  = 512;
static int imWidth  = 0;
static int imHeight = 0;

int fpsCount = 0;
int fpsLimit = 8;
unsigned int frameCount = 0;
StopWatchInterface *timer = NULL;
```

```
unsigned int g_Bpp;

bool g_bQAReadback = false;
// Display Data
static GLuint pbo_buffer = 0;
struct cudaGraphicsResource *cuda_pbo_resource;

static GLuint texid = 0;
unsigned char *pixels = NULL;

enum DisplayMode g_DisplayMode;

int *pArgc   = NULL;
char **pArgv = NULL;

#define OFFSET(i) ((char *)NULL + (i))
#define MAX(a,b) ((a > b) ? a : b
void display(void)
{
       sdkResetTimer(&timer);

    sdkStartTimer(&timer);
    Pixel *data = NULL;

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_pbo_resource, 0));
    size_t num_bytes;
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&data,
&num_bytes,
                                                         cuda_pbo_resource));

       morphologicalFilter(data, imWidth, imHeight, g_DisplayMode);
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0));
    glClear(GL_COLOR_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texid);
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, imWidth, imHeight,
                    GL_LUMINANCE, GL_UNSIGNED_BYTE, OFFSET(0));
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
    glDisable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);
       glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glBegin(GL_QUADS);

    glVertex2f(0, 0);
    glTexCoord2f(0, 0);
    glVertex2f(0, 1);
    glTexCoord2f(1, 0);
    glVertex2f(1, 1);
    glTexCoord2f(1, 1);
    glVertex2f(1, 0);
    glTexCoord2f(0, 1);
    glEnd();
    glBindTexture(GL_TEXTURE_2D, 0);
    glutSwapBuffers();
```

```
}
void timerEvent(int value)
{
    if(glutGetWindow())
    {
        glutPostRedisplay();
        glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
    }

}

void keyboard(unsigned char key, int /*x*/, int /*y*/)
{

    char temp[256];
    switch (key)
    {
        case 'q':
        case 'Q':
            printf("Shutting down...\n");
            glutDestroyWindow(glutGetWindow());
            return;
            break;
        case 't':
        case 'T':
            g_DisplayMode = DISPLAY_IMAGE;
            sprintf(temp, "CUDA Erosin Detection (%s)",
filterMode[g_DisplayMode]);
            glutSetWindowTitle(temp);
                    break;
              case 'a':
              case 'A':
                    g_DisplayMode = MORPHOLOGICAL_EROSION;
                    sprintf(temp, "CUDA Morphologial Erosion 3x3(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;

              case 'b':
              case 'B':
                    g_DisplayMode = MORPHOLOGICAL_EROSION5x5;
                    sprintf(temp, "CUDA Morphologial Erosion 5x5 (%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
              case 'c':
              case 'C':
                    g_DisplayMode = MORPHOLOGICAL_EROSION7x7;
                    sprintf(temp, "CUDA Morphologial Erosion 7x7 (%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
              case 'd':
              case 'D':
                    g_DisplayMode = MORPHOLOGICAL_DILATION;
                    sprintf(temp, "CUDA Morphological Dilation (%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'e':
            case 'E':
                    g_DisplayMode = MORPHOLOGICAL_DILATION5x5;
                    sprintf(temp, "CUDA Morphological Dilation 5x5(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'f':
            case 'F':
                    g_DisplayMode = MORPHOLOGICAL_DILATION7x7;
                    sprintf(temp, "CUDA Morphological Dilation 7x7(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'g':
            case 'G':
                    g_DisplayMode = MORPHOLOGICAL_CLOSE;
                    sprintf(temp, "CUDA Morphological Close (%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'h':
            case 'H':
                    g_DisplayMode = MORPHOLOGICAL_CLOSE5x5;
                    sprintf(temp, "CUDA Morphological Close 5x5(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'i':
            case 'I':
                    g_DisplayMode = MORPHOLOGICAL_CLOSE7x7;
                    sprintf(temp, "CUDA Morphological Close 7x7(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'j':
            case 'J':
                    g_DisplayMode = MORPHOLOGICAL_OPEN;
                    sprintf(temp, "CUDA Morphological Open (%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'k':
            case 'K':
                    g_DisplayMode = MORPHOLOGICAL_OPEN5x5;
                    sprintf(temp, "CUDA Morphological Open 5x5(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'l':
            case 'L':
                    g_DisplayMode = MORPHOLOGICAL_OPEN7x7;
                    sprintf(temp, "CUDA Morphological Open 7x7(%s)",
filterMode[g_DisplayMode]);
                    glutSetWindowTitle(temp);
                    break;
            case 'm':
```

```
		case 'M':
			g_DisplayMode = MEAN_OPERATION;
			sprintf(temp, "CUDA Mean Operation 3x3(%s)",
filterMode[g_DisplayMode]);
			glutSetWindowTitle(temp);
			break;
		case 'n':
		case 'N':
			g_DisplayMode = MEAN_OPERATION5x5;
			sprintf(temp, "CUDA Mean Operation 5x5(%s)",
filterMode[g_DisplayMode]);
			glutSetWindowTitle(temp);
			break;
		case 'o':
		case 'O':
			g_DisplayMode = MEAN_OPERATION7x7;
			sprintf(temp, "CUDA Mean Operation 7x7(%s)",
filterMode[g_DisplayMode]);
			glutSetWindowTitle(temp);
			break;
		case 'p':
		case 'P':
			g_DisplayMode = MEDAIN_OPERATION;
			sprintf(temp, "CUDA medain Operation (%s)",
filterMode[g_DisplayMode]);
			glutSetWindowTitle(temp);
			break;
		case 'r':
		case 'R':
			g_DisplayMode = MEDAIN_OPERATION5x5;
			sprintf(temp, "CUDA medain Operation 5x5(%s)",
filterMode[g_DisplayMode]);
			glutSetWindowTitle(temp);
			break;
		case 's':
		case 'S':
			g_DisplayMode = MEDAIN_OPERATION7x7;
			sprintf(temp, "CUDA medain Operation 7x7(%s)",
filterMode[g_DisplayMode]);
			glutSetWindowTitle(temp);
			break;


        default:
            break;
    }
}


void reshape(int x, int y)
{
    glViewport(0, 0, x, y);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 1, 0, 1, 0, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
void cleanup(void)
{
    cudaGraphicsUnregisterResource(cuda_pbo_resource);

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
    glDeleteBuffers(1, &pbo_buffer);
    glDeleteTextures(1, &texid);
    deleteTexture();

       checkCudaErrors(cudaDeviceSynchronize());
       sdkStopTimer(&timer);
       printf("Processing time: %f (ms)\n", sdkGetTimerValue(&timer));
       printf("%.2f Mpixels/sec\n",
              (imWidth *imHeight / (sdkGetTimerValue(&timer) / 1000.0f)) / 1e6);

       sdkDeleteTimer(&timer);

    cudaDeviceReset();

}



void initializeData(char *file)
{
    GLint bsize;
    unsigned int w, h;
    size_t file_length= strlen(file);
    if (!strcmp(&file[file_length-3], "pgm"))
    {
        if (sdkLoadPGM<unsigned char>(file, &pixels, &w, &h) != true)
        {
            printf("Failed to load PGM image file: %s\n", file);
            exit(EXIT_FAILURE);
        }

        g_Bpp = 1;
    }
       else if (!strcmp(&file[file_length - 3], "bmp"))
       {
              int i;
              FILE* f = fopen(file, "rb");
              unsigned char info[54];
              fread(info, sizeof(unsigned char), 54, f);
              int width = *(int*)&info[18];
              int height = *(int*)&info[22];

              w = width;
              h = height;

              int size = 3 * width * height;
              unsigned char* data = new unsigned char[size];
              pixels = new unsigned char[w*h];
              fread(data, sizeof(unsigned char), size, f);
              fclose(f);

              int cnt = 0;
            for (int j = h-1; j >=0; j--)
            {
                   for (i = 0; i < w*3; i += 3)
                   {
                          pixels[cnt++] = data[j*w*3 + i];
      }
            }
            g_Bpp = 1;
      }
    else if (!strcmp(&file[file_length-3], "ppm"))
    {
        if (sdkLoadPPM4(file, &pixels, &w, &h) != true)
        {
            printf("Failed to load PPM image file: %s\n", file);
            exit(EXIT_FAILURE);
        }

        g_Bpp = 4;
    }
    else
    {

        cudaDeviceReset();
        exit(EXIT_FAILURE);
    }


    imWidth = (int)w;
    imHeight = (int)h;
    setupTexture(imWidth, imHeight, pixels, g_Bpp);

    memset(pixels, 0x0, g_Bpp * sizeof(Pixel) * imWidth * imHeight);

    if (!g_bQAReadback)
    {

        glGenBuffers(1, &pbo_buffer);
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
        glBufferData(GL_PIXEL_UNPACK_BUFFER,
                     g_Bpp * sizeof(Pixel) * imWidth * imHeight,
                     pixels, GL_STREAM_DRAW);

        glGetBufferParameteriv(GL_PIXEL_UNPACK_BUFFER, GL_BUFFER_SIZE, &bsize);

        if ((GLuint)bsize != (g_Bpp * sizeof(Pixel) * imWidth * imHeight))
        {
            printf("Buffer object (%d) has incorrect size (%d).\n",
(unsigned)pbo_buffer, (unsigned)bsize);

            cudaDeviceReset();
            exit(EXIT_FAILURE);
        }

        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);


        checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_pbo_resource,
pbo_buffer, cudaGraphicsMapFlagsWriteDiscard));
```

```
        glGenTextures(1, &texid);
        glBindTexture(GL_TEXTURE_2D, texid);
        glTexImage2D(GL_TEXTURE_2D, 0, ((g_Bpp==1) ? GL_LUMINANCE : GL_BGRA),
                     imWidth, imHeight,  0, GL_LUMINANCE, GL_UNSIGNED_BYTE, NULL);
        glBindTexture(GL_TEXTURE_2D, 0);

        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glPixelStorei(GL_PACK_ALIGNMENT, 1);
    }
}

void initGL(int *argc, char **argv)
{
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(wWidth, wHeight);
    glutCreateWindow("CUDA Morphological Operations");

    glewInit();
}

int main(int argc, char **argv)
{
    pArgc = &argc;
    pArgv = argv;


       printf("%s Starting...\n\n", "CUDA Morphological");
    initGL(&argc, argv);

    sdkCreateTimer(&timer);
    sdkResetTimer(&timer);

    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutReshapeFunc(reshape);

    loadDefaultImage(argv[0]);

       printf("\nQ: QUIT\n");
    printf("\nT: display Image (no filtering)\n");
       printf("A: Morphological Erosion 3x3 (Using Texture)\n");
       printf("B: Morphological Erosion 5x5 (Using Texture)\n");

       printf("C: Morphological Erosion 7x7 (Using Texture)\n");

       printf("D: Morphological Dilation 3x3 (Using Texture)\n");

       printf("E: Morphological Dilation 5x5 (Using Texture)\n");

       printf("F: Morphological Dilation 7x7 (Using Texture)\n");

       printf("G: Morphological Close 3x3 (Using Texture)\n");

       printf("H: Morphological Close 5x5 (Using Texture)\n");

       printf("I: Morphological Close 7x7 (Using Texture)\n");
```

```
        printf("J: Morphological Open 3x3 (Using Texture)\n");

        printf("K: Morphological Open 5x5 (Using Texture)\n");

        printf("L: Morphological Open 7x7 (Using Texture)\n");

        printf("M: Mean 3x3 (Using Texture)\n");

        printf("N: Mean 5x5 (Using Texture)\n");

        printf("O: Mean 7x7 (Using Texture)\n");

        printf("P: Median 3x3 (Using Texture)\n");

        printf("R: Median 5x5 (Using Texture)\n");

        printf("S: Median 7x7 (Using Texture)\n");

    fflush(stdout);

    glutCloseFunc(cleanup);

    glutTimerFunc(REFRESH_DELAY, timerEvent,0);
    glutMainLoop();


}
```

```
texture<unsigned char, 2> tex;
texture<unsigned char, 2> texTem;
extern __shared__ unsigned char LocalBlock[];
static cudaArray *array = NULL;

static cudaArray *arrayTemp = NULL;

#ifdef FIXED_BLOCKWIDTH

#endif




#define checkCudaErrors(err)           __checkCudaErrors (err, __FILE__, __LINE__)

inline void __checkCudaErrors(cudaError err, const char *file, const int line)
{
    if (cudaSuccess != err)

    {
        fprintf(stderr, "%s(%i) : CUDA Runtime API error %d: %s.\n",
                file, line, (int)err, cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
}


__device__ unsigned char
ComputeErosion3x3(unsigned char ul,
unsigned char um,
unsigned char ur,
unsigned char ml,
unsigned char mm,
unsigned char mr,
unsigned char ll,
unsigned char lm,
unsigned char lr
)
{
       unsigned char min = ul;
       if (um < min) min = um;
       if (ur < min) min = ur;

       if (ml < min) min = ml;
       if (mm < min) min = mm;
       if (mr < min) min = mr;
       if (ll < min) min = ll;
       if (lm < min) min = lm;
       if (lr < min) min = lr;

       return min;
}
```

```
__device__ unsigned char
ComputeErosion5x5(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44
)
{
       unsigned char min = pix00;
       if (pix01 < min) min = pix01;
       if (pix02 < min) min = pix02;
       if (pix03 < min) min = pix03;
       if (pix04 < min) min = pix04;

       if (pix10 < min) min = pix10;
       if (pix11 < min) min = pix11;
       if (pix12 < min) min = pix12;
       if (pix13 < min) min = pix13;
       if (pix14 < min) min = pix14;

       if (pix20 < min) min = pix20;
       if (pix21 < min) min = pix21;
       if (pix22 < min) min = pix22;
       if (pix23 < min) min = pix23;
       if (pix24 < min) min = pix24;

       if (pix30 < min) min = pix30;
       if (pix31 < min) min = pix31;
       if (pix32 < min) min = pix32;
       if (pix33 < min) min = pix33;
       if (pix34 < min) min = pix34;

       if (pix40 < min) min = pix40;
       if (pix41 < min) min = pix41;
       if (pix42 < min) min = pix42;
       if (pix43 < min) min = pix43;
       if (pix44 < min) min = pix44;

       return min;
}

__device__ unsigned char
ComputeErosion7x7(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04, unsigned char pix05, unsigned char
pix06,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14, unsigned char pix15, unsigned char pix16,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24, unsigned char pix25, unsigned char pix26,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34, unsigned char pix35, unsigned char pix36,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44, unsigned char pix45, unsigned char pix46,
unsigned char pix50, unsigned char pix51, unsigned char pix52, unsigned char
pix53, unsigned char pix54, unsigned char pix55, unsigned char pix56,
unsigned char pix60, unsigned char pix61, unsigned char pix62, unsigned char
pix63, unsigned char pix64, unsigned char pix65, unsigned char pix66
)
{
        unsigned char min = pix00;
        if (pix01 < min) min = pix01;
        if (pix02 < min) min = pix02;
        if (pix03 < min) min = pix03;
        if (pix04 < min) min = pix04;
        if (pix05 < min) min = pix05;
        if (pix06 < min) min = pix06;

        if (pix10 < min) min = pix10;
        if (pix11 < min) min = pix11;
        if (pix12 < min) min = pix12;
        if (pix13 < min) min = pix13;
        if (pix14 < min) min = pix14;
        if (pix15 < min) min = pix15;
        if (pix16 < min) min = pix16;

        if (pix20 < min) min = pix20;
        if (pix21 < min) min = pix21;
        if (pix22 < min) min = pix22;
        if (pix23 < min) min = pix23;
        if (pix24 < min) min = pix24;
        if (pix25 < min) min = pix25;
        if (pix26 < min) min = pix26;

        if (pix30 < min) min = pix30;
        if (pix31 < min) min = pix31;
        if (pix32 < min) min = pix32;
        if (pix33 < min) min = pix33;
        if (pix34 < min) min = pix34;
        if (pix35 < min) min = pix35;
        if (pix36 < min) min = pix36;

        if (pix40 < min) min = pix40;
        if (pix41 < min) min = pix41;
        if (pix42 < min) min = pix42;
        if (pix43 < min) min = pix43;
        if (pix44 < min) min = pix44;
        if (pix45 < min) min = pix45;
        if (pix46 < min) min = pix46;

        if (pix50 < min) min = pix50;
        if (pix51 < min) min = pix51;
        if (pix52 < min) min = pix52;
        if (pix53 < min) min = pix53;
        if (pix54 < min) min = pix54;
        if (pix55 < min) min = pix55;
        if (pix56 < min) min = pix56;

        if (pix60 < min) min = pix60;
        if (pix61 < min) min = pix61;
        if (pix62 < min) min = pix62;
        if (pix63 < min) min = pix63;
        if (pix64 < min) min = pix64;
        if (pix65 < min) min = pix65;
        if (pix66 < min) min = pix66;

        return min;
}

__device__ unsigned char
ComputeDilation3x3(unsigned char ul,
unsigned char um,
unsigned char ur,
unsigned char ml,
unsigned char mm,
unsigned char mr,
unsigned char ll,
unsigned char lm,
unsigned char lr
)
{
        unsigned char max = ul;
        if (um > max) max = um;
        if (ur > max) max = ur;

        if (ml > max) max = ml;
        if (mm > max) max = mm;
        if (mr > max) max = mr;
        if (ll > max) max = ll;
        if (lm > max) max = lm;
        if (lr > max) max = lr;

        return max;
}


__device__ unsigned char
ComputeDilation5x5(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44
)
{
        unsigned char max = pix00;
        if (pix01 > max) max = pix01;
        if (pix02 > max) max = pix02;
        if (pix03 > max) max = pix03;
        if (pix04 > max) max = pix04;

        if (pix10 > max) max = pix10;
        if (pix11 > max) max = pix11;
        if (pix12 > max) max = pix12;
        if (pix13 > max) max = pix13;
        if (pix14 > max) max = pix14;
        if (pix20 > max) max = pix20;
        if (pix21 > max) max = pix21;
        if (pix22 > max) max = pix22;
        if (pix23 > max) max = pix23;
        if (pix24 > max) max = pix24;

        if (pix30 > max) max = pix30;
        if (pix31 > max) max = pix31;
        if (pix32 > max) max = pix32;
        if (pix33 > max) max = pix33;
        if (pix34 > max) max = pix34;

        if (pix40 > max) max = pix40;
        if (pix41 > max) max = pix41;
        if (pix42 > max) max = pix42;
        if (pix43 > max) max = pix43;
        if (pix44 > max) max = pix44;

        return max;
}

__device__ unsigned char
ComputeDilation7x7(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04, unsigned char pix05, unsigned char
pix06,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14, unsigned char pix15, unsigned char pix16,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24, unsigned char pix25, unsigned char pix26,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34, unsigned char pix35, unsigned char pix36,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44, unsigned char pix45, unsigned char pix46,
unsigned char pix50, unsigned char pix51, unsigned char pix52, unsigned char
pix53, unsigned char pix54, unsigned char pix55, unsigned char pix56,
unsigned char pix60, unsigned char pix61, unsigned char pix62, unsigned char
pix63, unsigned char pix64, unsigned char pix65, unsigned char pix66
)
{
        unsigned char max = pix00;
        if (pix01 > max) max = pix01;
        if (pix02 > max) max = pix02;
        if (pix03 > max) max = pix03;
        if (pix04 > max) max = pix04;
        if (pix05 > max) max = pix05;
        if (pix06 > max) max = pix06;

        if (pix10 > max) max = pix10;
        if (pix11 > max) max = pix11;
        if (pix12 > max) max = pix12;
        if (pix13 > max) max = pix13;
        if (pix14 > max) max = pix14;
        if (pix15 > max) max = pix15;
        if (pix16 > max) max = pix16;

        if (pix20 > max) max = pix20;
        if (pix21 > max) max = pix21;
        if (pix22 > max) max = pix22;
        if (pix23 > max) max = pix23;
        if (pix24 > max) max = pix24;
        if (pix25 > max) max = pix25;
        if (pix26 > max) max = pix26;

        if (pix30 > max) max = pix30;
        if (pix31 > max) max = pix31;
        if (pix32 > max) max = pix32;
        if (pix33 > max) max = pix33;
        if (pix34 > max) max = pix34;
        if (pix35 > max) max = pix35;
        if (pix36 > max) max = pix36;

        if (pix40 > max) max = pix40;
        if (pix41 > max) max = pix41;
        if (pix42 > max) max = pix42;
        if (pix43 > max) max = pix43;
        if (pix44 > max) max = pix44;
        if (pix45 > max) max = pix45;
        if (pix46 > max) max = pix46;

        if (pix50 > max) max = pix50;
        if (pix51 > max) max = pix51;
        if (pix52 > max) max = pix52;
        if (pix53 > max) max = pix53;
        if (pix54 > max) max = pix54;
        if (pix55 > max) max = pix55;
        if (pix56 > max) max = pix56;

        if (pix60 > max) max = pix60;
        if (pix61 > max) max = pix61;
        if (pix62 > max) max = pix62;
        if (pix63 > max) max = pix63;
        if (pix64 > max) max = pix64;
        if (pix65 > max) max = pix65;
        if (pix66 > max) max = pix66;

        return max;
}

__device__ unsigned char
ComputeMean3x3(unsigned char ul,
unsigned char um,
unsigned char ur,
unsigned char ml,
unsigned char mm,
unsigned char mr,
unsigned char ll,
unsigned char lm,
unsigned char lr
)
{
        short sum = ul + um + ur + ml + mm + mr + ll + lm + lr;

        unsigned char mean = (sum / 9.0);

        return mean;
```

```
}

__device__ unsigned char
ComputeMean5x5(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44
)
{
       short sum = pix00 + pix01 + pix02 + pix03 + pix04 +
              pix10 + pix11 + pix12 + pix13 + pix14 +
              pix20 + pix21 + pix22 + pix23 + pix24 +
              pix30 + pix31 + pix32 + pix33 + pix34 +
              pix40 + pix41 + pix42 + pix43 + pix44;

       unsigned char mean = (sum / 25.0);

       return mean;
}



__device__ unsigned char
ComputeMean7x7(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04, unsigned char pix05, unsigned char
pix06,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14, unsigned char pix15, unsigned char pix16,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24, unsigned char pix25, unsigned char pix26,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34, unsigned char pix35, unsigned char pix36,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44, unsigned char pix45, unsigned char pix46,
unsigned char pix50, unsigned char pix51, unsigned char pix52, unsigned char
pix53, unsigned char pix54, unsigned char pix55, unsigned char pix56,
unsigned char pix60, unsigned char pix61, unsigned char pix62, unsigned char
pix63, unsigned char pix64, unsigned char pix65, unsigned char pix66
)
{

       unsigned short sum = (pix00 + pix01 + pix02 + pix03 + pix04 + pix05 +
pix06);

       sum = sum + (pix10 + pix11 + pix12 + pix13 + pix14 + pix15 + pix16);
       sum = sum + (pix20 + pix21 + pix22 + pix23 + pix24 + pix25 + pix26);
       sum = sum + (pix30 + pix31 + pix32 + pix33 + pix34 + pix35 + pix36);
       sum = sum + (pix40 + pix41 + pix42 + pix43 + pix44 + pix45 + pix46);
       sum = sum + (pix50 + pix51 + pix52 + pix53 + pix54 + pix55 + pix56);
       sum = sum + (pix60 + pix61 + pix62 + pix63 + pix64 + pix65 + pix66);

       unsigned char mean = (sum / 49.0);
```

```
        return mean;
}


__device__ unsigned char
ComputeMedian3x3(unsigned char ul,
unsigned char um,
unsigned char ur,
unsigned char ml,
unsigned char mm,
unsigned char mr,
unsigned char ll,
unsigned char lm,
unsigned char lr
)
{
        unsigned char medainArray[9] = { ul, um, ur, ml, mm, mr, ll, lm, lr };

        unsigned char iMin, temp;
        int i, iFirst;

        for (iFirst = 0; iFirst < 8; iFirst++)
        {
                iMin = iFirst;
                for (i = iFirst; i < 9; i++)
                {
                        if (medainArray[i] < medainArray[iMin])
                        {
                                iMin = i;

                        }
                }

                temp = medainArray[iMin];
                medainArray[iMin] = medainArray[iFirst];
                medainArray[iFirst] = temp;

        }

        return medainArray[4];
}


__device__ unsigned char
ComputeMedian5x5(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44
)
{
	unsigned char medainArray[25] = { pix00, pix01, pix02, pix03, pix04, pix10,
pix11, pix12, pix13, pix14, pix20, pix21, pix22, pix23, pix24, pix30, pix31,
pix32, pix33, pix34, pix40, pix41, pix42, pix43, pix44 };

	unsigned char iMin, temp;
	int i, iFirst;

	for (iFirst = 0; iFirst < 24; iFirst++)
	{
		iMin = iFirst;
		for (i = iFirst; i < 25; i++)
		{
			if (medainArray[i] < medainArray[iMin])
			{
				iMin = i;

			}
		}

		temp = medainArray[iMin];
		medainArray[iMin] = medainArray[iFirst];
		medainArray[iFirst] = temp;

	}

	return medainArray[12];
}


__device__ unsigned char
ComputeMedian7x7(unsigned char pix00, unsigned char pix01, unsigned char pix02,
unsigned char pix03, unsigned char pix04, unsigned char pix05, unsigned char
pix06,
unsigned char pix10, unsigned char pix11, unsigned char pix12, unsigned char
pix13, unsigned char pix14, unsigned char pix15, unsigned char pix16,
unsigned char pix20, unsigned char pix21, unsigned char pix22, unsigned char
pix23, unsigned char pix24, unsigned char pix25, unsigned char pix26,
unsigned char pix30, unsigned char pix31, unsigned char pix32, unsigned char
pix33, unsigned char pix34, unsigned char pix35, unsigned char pix36,
unsigned char pix40, unsigned char pix41, unsigned char pix42, unsigned char
pix43, unsigned char pix44, unsigned char pix45, unsigned char pix46,
unsigned char pix50, unsigned char pix51, unsigned char pix52, unsigned char
pix53, unsigned char pix54, unsigned char pix55, unsigned char pix56,
unsigned char pix60, unsigned char pix61, unsigned char pix62, unsigned char
pix63, unsigned char pix64, unsigned char pix65, unsigned char pix66
)
{
	unsigned char medainArray[49] = { pix00, pix01, pix02, pix03, pix04, pix05,
pix06,
		pix10, pix11, pix12, pix13, pix14, pix15, pix16,
		pix20, pix21, pix22, pix23, pix24, pix25, pix26,
		pix30, pix31, pix32, pix33, pix34, pix35, pix36,
		pix40, pix41, pix42, pix43, pix44, pix45, pix46,
		pix50, pix51, pix52, pix53, pix54, pix55, pix56,
		pix60, pix61, pix62, pix63, pix64, pix65, pix66};

	unsigned char iMin, temp;
	int i, iFirst;
```

```
        for (iFirst = 0; iFirst < 48; iFirst++)
        {
                iMin = iFirst;
                for (i = iFirst; i < 49; i++)
                {
                        if (medainArray[i] < medainArray[iMin])
                        {
                                iMin = i;

                        }
                }

                temp = medainArray[iMin];
                medainArray[iMin] = medainArray[iFirst];
                medainArray[iFirst] = temp;

        }

        return medainArray[24];
}


__global__ void
{
    unsigned char *pImage =
        (unsigned char *)(((char *) pImageOriginal)+blockIdx.x*Pitch);

    for (int i = threadIdx.x; i < w; i += blockDim.x)    {

        pImage[i] = min(max((tex2D(tex, (float) i, (float) blockIdx.x) * 1.f),
0.f), 255.f);
    }
}

__global__ void
{
        unsigned char *pImage =
                (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);


        for (int i = threadIdx.x; i < w; i += blockDim.x)
        {
                unsigned char pix00 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
                unsigned char pix01 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
                unsigned char pix02 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
                unsigned char pix10 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
                unsigned char pix11 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
0);
                unsigned char pix12 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
0);
                unsigned char pix20 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
1);
```

```
		unsigned char pix21 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
1);
		unsigned char pix22 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
1);
		pImage[i] = ComputeErosion3x3(pix00, pix01, pix02,
			pix10, pix11, pix12,
			pix20, pix21, pix22);
	}
}


__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
2);
		unsigned char pix01 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
2);
		unsigned char pix02 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
2);
		unsigned char pix03 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
2);
		unsigned char pix04 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
2);

		unsigned char pix10 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
1);
		unsigned char pix11 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
		unsigned char pix12 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
		unsigned char pix13 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
		unsigned char pix14 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
1);

		unsigned char pix20 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
0);
		unsigned char pix21 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
		unsigned char pix22 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
0);
		unsigned char pix23 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
0);
		unsigned char pix24 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
0);

		unsigned char pix30 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
1);
		unsigned char pix31 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
1);
		unsigned char pix32 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
1);
```

```
                unsigned char pix33 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
1);
                unsigned char pix34 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
1);

                unsigned char pix40 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
2);
                unsigned char pix41 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
2);
                unsigned char pix42 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
2);
                unsigned char pix43 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
2);
                unsigned char pix44 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
2);

                pImage[i] = ComputeErosion5x5(pix00, pix01, pix02, pix03, pix04,
                        pix10, pix11, pix12, pix13, pix14,
                        pix20, pix21, pix22, pix23, pix24,
                        pix30, pix31, pix32, pix33, pix34,
                        pix40, pix41, pix42, pix43, pix44);
        }
}

__global__ void
{
        unsigned char *pImage =
                (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

        for (int i = threadIdx.x; i < w; i += blockDim.x)
        {
                unsigned char pix00 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
3);
                unsigned char pix01 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
3);
                unsigned char pix02 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
3);
                unsigned char pix03 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
3);
                unsigned char pix04 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
3);
                unsigned char pix05 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
3);
                unsigned char pix06 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
3);

                unsigned char pix10 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
2);
                unsigned char pix11 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
2);
                unsigned char pix12 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
2);
                unsigned char pix13 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
2);
                unsigned char pix14 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
2);
                unsigned char pix15 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
2);
```

```
                unsigned char pix16 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
2);

                unsigned char pix20 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
1);
                unsigned char pix21 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
1);
                unsigned char pix22 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
                unsigned char pix23 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
                unsigned char pix24 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
                unsigned char pix25 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
1);
                unsigned char pix26 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
1);

                unsigned char pix30 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
0);
                unsigned char pix31 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
0);
                unsigned char pix32 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
                unsigned char pix33 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
0);
                unsigned char pix34 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
0);
                unsigned char pix35 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
0);
                unsigned char pix36 = tex2D(tex, (float)i + 3, (float)blockIdx.x +
0);

                unsigned char pix40 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
1);
                unsigned char pix41 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
1);
                unsigned char pix42 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
1);
                unsigned char pix43 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
1);
                unsigned char pix44 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
1);
                unsigned char pix45 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
1);
                unsigned char pix46 = tex2D(tex, (float)i + 3, (float)blockIdx.x +
1);

                unsigned char pix50 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
2);
                unsigned char pix51 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
2);
                unsigned char pix52 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
2);
                unsigned char pix53 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
2);
                unsigned char pix54 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
2);
```

```
		unsigned char pix55 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
2);
		unsigned char pix56 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
2);

		unsigned char pix60 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
3);
		unsigned char pix61 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
3);
		unsigned char pix62 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
3);
		unsigned char pix63 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
3);
		unsigned char pix64 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
3);
		unsigned char pix65 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
3);
		unsigned char pix66 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
3);

		pImage[i] = ComputeErosion7x7(pix00, pix01, pix02, pix03, pix04, 
pix05, pix06,
			pix10, pix11, pix12, pix13, pix14, pix15, pix16,
			pix20, pix21, pix22, pix23, pix24, pix25, pix26,
			pix30, pix31, pix32, pix33, pix34, pix35, pix36,
			pix40, pix41, pix42, pix43, pix44, pix45, pix46,
			pix50, pix51, pix52, pix53, pix54, pix55, pix56,
			pix60, pix61, pix62, pix63, pix64, pix65, pix66
			);
	}
}

__global__ void
{
	unsigned char *pImage = 
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
1);
		unsigned char pix01 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
1);
		unsigned char pix02 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
1);
		unsigned char pix10 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
0);
		unsigned char pix11 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
0);
		unsigned char pix12 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
0);
		unsigned char pix20 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
1);
		unsigned char pix21 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
1);
		unsigned char pix22 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
1);
		pImage[i] = ComputeDilation3x3(pix00, pix01, pix02,
                    pix10, pix11, pix12,
                    pix20, pix21, pix22);
        }
}


__global__ void
{
        unsigned char *pImage =
                (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

        for (int i = threadIdx.x; i < w; i += blockDim.x)
        {
                unsigned char pix00 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
2);
                unsigned char pix01 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
2);
                unsigned char pix02 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
2);
                unsigned char pix03 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
2);
                unsigned char pix04 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
2);

                unsigned char pix10 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
1);
                unsigned char pix11 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
                unsigned char pix12 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
                unsigned char pix13 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
                unsigned char pix14 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
1);

                unsigned char pix20 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
0);
                unsigned char pix21 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
                unsigned char pix22 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
0);
                unsigned char pix23 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
0);
                unsigned char pix24 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
0);

                unsigned char pix30 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
1);
                unsigned char pix31 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
1);
                unsigned char pix32 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
1);
                unsigned char pix33 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
1);
                unsigned char pix34 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
1);
```

```
		unsigned char pix40 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
2);
		unsigned char pix41 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
2);
		unsigned char pix42 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
2);
		unsigned char pix43 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
2);
		unsigned char pix44 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
2);

		pImage[i] = ComputeDilation5x5(pix00, pix01, pix02, pix03, pix04,
			pix10, pix11, pix12, pix13, pix14,
			pix20, pix21, pix22, pix23, pix24,
			pix30, pix31, pix32, pix33, pix34,
			pix40, pix41, pix42, pix43, pix44);
	}
}

__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 3, (float)blockIdx.x - 
3);
		unsigned char pix01 = tex2D(tex, (float)i - 2, (float)blockIdx.x - 
3);
		unsigned char pix02 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
3);
		unsigned char pix03 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
3);
		unsigned char pix04 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
3);
		unsigned char pix05 = tex2D(tex, (float)i + 2, (float)blockIdx.x - 
3);
		unsigned char pix06 = tex2D(tex, (float)i + 3, (float)blockIdx.x - 
3);

		unsigned char pix10 = tex2D(tex, (float)i - 3, (float)blockIdx.x - 
2);
		unsigned char pix11 = tex2D(tex, (float)i - 2, (float)blockIdx.x - 
2);
		unsigned char pix12 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
2);
		unsigned char pix13 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
2);
		unsigned char pix14 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
2);
		unsigned char pix15 = tex2D(tex, (float)i + 2, (float)blockIdx.x - 
2);
		unsigned char pix16 = tex2D(tex, (float)i + 3, (float)blockIdx.x - 
2);

		unsigned char pix20 = tex2D(tex, (float)i - 3, (float)blockIdx.x - 
1);
```

```
		unsigned char pix21 = tex2D(tex, (float)i - 2, (float)blockIdx.x - 
1);
		unsigned char pix22 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
1);
		unsigned char pix23 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
1);
		unsigned char pix24 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
1);
		unsigned char pix25 = tex2D(tex, (float)i + 2, (float)blockIdx.x - 
1);
		unsigned char pix26 = tex2D(tex, (float)i + 3, (float)blockIdx.x - 
1);

		unsigned char pix30 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
0);
		unsigned char pix31 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
0);
		unsigned char pix32 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
0);
		unsigned char pix33 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
0);
		unsigned char pix34 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
0);
		unsigned char pix35 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
0);
		unsigned char pix36 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
0);

		unsigned char pix40 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
1);
		unsigned char pix41 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
1);
		unsigned char pix42 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
1);
		unsigned char pix43 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
1);
		unsigned char pix44 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
1);
		unsigned char pix45 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
1);
		unsigned char pix46 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
1);

		unsigned char pix50 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
2);
		unsigned char pix51 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
2);
		unsigned char pix52 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
2);
		unsigned char pix53 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
2);
		unsigned char pix54 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
2);
		unsigned char pix55 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
2);
		unsigned char pix56 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
2);
```

```
		unsigned char pix60 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
3);
		unsigned char pix61 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
3);
		unsigned char pix62 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
3);
		unsigned char pix63 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
3);
		unsigned char pix64 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
3);
		unsigned char pix65 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
3);
		unsigned char pix66 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
3);

		pImage[i] = ComputeDilation7x7(pix00, pix01, pix02, pix03, pix04, 
pix05, pix06,
			pix10, pix11, pix12, pix13, pix14, pix15, pix16,
			pix20, pix21, pix22, pix23, pix24, pix25, pix26,
			pix30, pix31, pix32, pix33, pix34, pix35, pix36,
			pix40, pix41, pix42, pix43, pix44, pix45, pix46,
			pix50, pix51, pix52, pix53, pix54, pix55, pix56,
			pix60, pix61, pix62, pix63, pix64, pix65, pix66
			);
	}
}

__global__ void
{
	unsigned char *pImage = 
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(texTem, (float)i - 1, (float)blockIdx.x 
- 1);
		unsigned char pix01 = tex2D(texTem, (float)i + 0, (float)blockIdx.x 
- 1);
		unsigned char pix02 = tex2D(texTem, (float)i + 1, (float)blockIdx.x 
- 1);
		unsigned char pix10 = tex2D(texTem, (float)i - 1, (float)blockIdx.x 
+ 0);
		unsigned char pix11 = tex2D(texTem, (float)i + 0, (float)blockIdx.x 
+ 0);
		unsigned char pix12 = tex2D(texTem, (float)i + 1, (float)blockIdx.x 
+ 0);
		unsigned char pix20 = tex2D(texTem, (float)i - 1, (float)blockIdx.x 
+ 1);
		unsigned char pix21 = tex2D(texTem, (float)i + 0, (float)blockIdx.x 
+ 1);
		unsigned char pix22 = tex2D(texTem, (float)i + 1, (float)blockIdx.x 
+ 1);

		pImage[i] = ComputeErosion3x3(pix00, pix01, pix02, 
			pix10, pix11, pix12,
			pix20, pix21, pix22);
	}
}
```

```
__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 2);
		unsigned char pix01 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 2);
		unsigned char pix02 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 2);
		unsigned char pix03 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 2);
		unsigned char pix04 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 2);

		unsigned char pix10 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 1);
		unsigned char pix11 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 1);
		unsigned char pix12 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 1);
		unsigned char pix13 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 1);
		unsigned char pix14 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 1);

		unsigned char pix20 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 0);
		unsigned char pix21 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 0);
		unsigned char pix22 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 0);
		unsigned char pix23 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 0);
		unsigned char pix24 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 0);

		unsigned char pix30 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 1);
		unsigned char pix31 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 1);
		unsigned char pix32 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 1);
		unsigned char pix33 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 1);
		unsigned char pix34 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 1);

		unsigned char pix40 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 2);
		unsigned char pix41 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 2);
		unsigned char pix42 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 2);
```

```
              unsigned char pix43 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 2);
              unsigned char pix44 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 2);

              pImage[i] = ComputeErosion5x5(pix00, pix01, pix02, pix03, pix04,
                     pix10, pix11, pix12, pix13, pix14,
                     pix20, pix21, pix22, pix23, pix24,
                     pix30, pix31, pix32, pix33, pix34,
                     pix40, pix41, pix42, pix43, pix44);
       }
}

__global__ void
{
       unsigned char *pImage =
              (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

       for (int i = threadIdx.x; i < w; i += blockDim.x)
       {
              unsigned char pix00 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
- 3);
              unsigned char pix01 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 3);
              unsigned char pix02 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 3);
              unsigned char pix03 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 3);
              unsigned char pix04 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 3);
              unsigned char pix05 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 3);
              unsigned char pix06 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
- 3);

              unsigned char pix10 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
- 2);
              unsigned char pix11 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 2);
              unsigned char pix12 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 2);
              unsigned char pix13 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 2);
              unsigned char pix14 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 2);
              unsigned char pix15 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 2);
              unsigned char pix16 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
- 2);

              unsigned char pix20 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
- 1);
              unsigned char pix21 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 1);
              unsigned char pix22 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 1);
              unsigned char pix23 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 1);
```

```
			unsigned char pix24 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 1);
			unsigned char pix25 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 1);
			unsigned char pix26 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
- 1);

			unsigned char pix30 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 0);
			unsigned char pix31 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 0);
			unsigned char pix32 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 0);
			unsigned char pix33 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 0);
			unsigned char pix34 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 0);
			unsigned char pix35 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 0);
			unsigned char pix36 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 0);

			unsigned char pix40 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 1);
			unsigned char pix41 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 1);
			unsigned char pix42 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 1);
			unsigned char pix43 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 1);
			unsigned char pix44 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 1);
			unsigned char pix45 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 1);
			unsigned char pix46 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 1);

			unsigned char pix50 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 2);
			unsigned char pix51 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 2);
			unsigned char pix52 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 2);
			unsigned char pix53 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 2);
			unsigned char pix54 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 2);
			unsigned char pix55 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 2);
			unsigned char pix56 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 2);

			unsigned char pix60 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 3);
			unsigned char pix61 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 3);
			unsigned char pix62 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 3);
```

```
              unsigned char pix63 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 3);
              unsigned char pix64 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 3);
              unsigned char pix65 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 3);
              unsigned char pix66 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 3);

              pImage[i] = ComputeErosion7x7(pix00, pix01, pix02, pix03, pix04,
pix05, pix06,
                     pix10, pix11, pix12, pix13, pix14, pix15, pix16,
                     pix20, pix21, pix22, pix23, pix24, pix25, pix26,
                     pix30, pix31, pix32, pix33, pix34, pix35, pix36,
                     pix40, pix41, pix42, pix43, pix44, pix45, pix46,
                     pix50, pix51, pix52, pix53, pix54, pix55, pix56,
                     pix60, pix61, pix62, pix63, pix64, pix65, pix66
                     );
       }
}

__global__ void
{
       unsigned char *pImage =
              (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

       for (int i = threadIdx.x; i < w; i += blockDim.x)
       {
              unsigned char pix00 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 1);
              unsigned char pix01 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 1);
              unsigned char pix02 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 1);
              unsigned char pix10 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 0);
              unsigned char pix11 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 0);
              unsigned char pix12 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 0);
              unsigned char pix20 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 1);
              unsigned char pix21 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 1);
              unsigned char pix22 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 1);

              pImage[i] = ComputeDilation3x3(pix00, pix01, pix02,
                     pix10, pix11, pix12,
                     pix20, pix21, pix22);
       }
}


__global__ void
{
       unsigned char *pImage =
              (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);
```

```
	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 2);
		unsigned char pix01 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 2);
		unsigned char pix02 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 2);
		unsigned char pix03 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 2);
		unsigned char pix04 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 2);

		unsigned char pix10 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 1);
		unsigned char pix11 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 1);
		unsigned char pix12 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 1);
		unsigned char pix13 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 1);
		unsigned char pix14 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 1);

		unsigned char pix20 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 0);
		unsigned char pix21 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 0);
		unsigned char pix22 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 0);
		unsigned char pix23 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 0);
		unsigned char pix24 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 0);

		unsigned char pix30 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 1);
		unsigned char pix31 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 1);
		unsigned char pix32 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 1);
		unsigned char pix33 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 1);
		unsigned char pix34 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 1);

		unsigned char pix40 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 2);
		unsigned char pix41 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 2);
		unsigned char pix42 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 2);
		unsigned char pix43 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 2);
		unsigned char pix44 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 2);
```

```
			pImage[i] = ComputeDilation5x5(pix00, pix01, pix02, pix03, pix04,
				pix10, pix11, pix12, pix13, pix14,
				pix20, pix21, pix22, pix23, pix24,
				pix30, pix31, pix32, pix33, pix34,
				pix40, pix41, pix42, pix43, pix44);
		}
}


__global__ void
{
		unsigned char *pImage =
			(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

		for (int i = threadIdx.x; i < w; i += blockDim.x)
		{
			unsigned char pix00 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
- 3);
			unsigned char pix01 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 3);
			unsigned char pix02 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 3);
			unsigned char pix03 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 3);
			unsigned char pix04 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 3);
			unsigned char pix05 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 3);
			unsigned char pix06 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
- 3);

			unsigned char pix10 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
- 2);
			unsigned char pix11 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 2);
			unsigned char pix12 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 2);
			unsigned char pix13 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 2);
			unsigned char pix14 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 2);
			unsigned char pix15 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 2);
			unsigned char pix16 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
- 2);

			unsigned char pix20 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
- 1);
			unsigned char pix21 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
- 1);
			unsigned char pix22 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
- 1);
			unsigned char pix23 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
- 1);
			unsigned char pix24 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
- 1);
			unsigned char pix25 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
- 1);
```

```
              unsigned char pix26 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
- 1);

              unsigned char pix30 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 0);
              unsigned char pix31 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 0);
              unsigned char pix32 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 0);
              unsigned char pix33 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 0);
              unsigned char pix34 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 0);
              unsigned char pix35 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 0);
              unsigned char pix36 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 0);

              unsigned char pix40 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 1);
              unsigned char pix41 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 1);
              unsigned char pix42 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 1);
              unsigned char pix43 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 1);
              unsigned char pix44 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 1);
              unsigned char pix45 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 1);
              unsigned char pix46 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 1);

              unsigned char pix50 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 2);
              unsigned char pix51 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 2);
              unsigned char pix52 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 2);
              unsigned char pix53 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 2);
              unsigned char pix54 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 2);
              unsigned char pix55 = tex2D(texTem, (float)i + 2, (float)blockIdx.x
+ 2);
              unsigned char pix56 = tex2D(texTem, (float)i + 3, (float)blockIdx.x
+ 2);

              unsigned char pix60 = tex2D(texTem, (float)i - 3, (float)blockIdx.x
+ 3);
              unsigned char pix61 = tex2D(texTem, (float)i - 2, (float)blockIdx.x
+ 3);
              unsigned char pix62 = tex2D(texTem, (float)i - 1, (float)blockIdx.x
+ 3);
              unsigned char pix63 = tex2D(texTem, (float)i + 0, (float)blockIdx.x
+ 3);
              unsigned char pix64 = tex2D(texTem, (float)i + 1, (float)blockIdx.x
+ 3);
```

```
		unsigned char pix65 = tex2D(texTem, (float)i + 2, (float)blockIdx.x 
+ 3);
		unsigned char pix66 = tex2D(texTem, (float)i + 3, (float)blockIdx.x 
+ 3);

		pImage[i] = ComputeDilation7x7(pix00, pix01, pix02, pix03, pix04, 
pix05, pix06,
			pix10, pix11, pix12, pix13, pix14, pix15, pix16,
			pix20, pix21, pix22, pix23, pix24, pix25, pix26,
			pix30, pix31, pix32, pix33, pix34, pix35, pix36,
			pix40, pix41, pix42, pix43, pix44, pix45, pix46,
			pix50, pix51, pix52, pix53, pix54, pix55, pix56,
			pix60, pix61, pix62, pix63, pix64, pix65, pix66
			);
	}
}

__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
1);
		unsigned char pix01 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
1);
		unsigned char pix02 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
1);
		unsigned char pix10 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
0);
		unsigned char pix11 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
0);
		unsigned char pix12 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
0);
		unsigned char pix20 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
1);
		unsigned char pix21 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
1);
		unsigned char pix22 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
1);
		pImage[i] = ComputeMean3x3(pix00, pix01, pix02,
			pix10, pix11, pix12,
			pix20, pix21, pix22);
	}
}

__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 2, (float)blockIdx.x - 
2);
```

```
                unsigned char pix01 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
2);
                unsigned char pix02 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
2);
                unsigned char pix03 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
2);
                unsigned char pix04 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
2);

                unsigned char pix10 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
1);
                unsigned char pix11 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
                unsigned char pix12 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
                unsigned char pix13 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
                unsigned char pix14 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
1);

                unsigned char pix20 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
0);
                unsigned char pix21 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
                unsigned char pix22 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
0);
                unsigned char pix23 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
0);
                unsigned char pix24 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
0);

                unsigned char pix30 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
1);
                unsigned char pix31 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
1);
                unsigned char pix32 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
1);
                unsigned char pix33 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
1);
                unsigned char pix34 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
1);

                unsigned char pix40 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
2);
                unsigned char pix41 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
2);
                unsigned char pix42 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
2);
                unsigned char pix43 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
2);
                unsigned char pix44 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
2);

                pImage[i] = ComputeMean5x5(pix00, pix01, pix02, pix03, pix04,
                        pix10, pix11, pix12, pix13, pix14,
                        pix20, pix21, pix22, pix23, pix24,
                        pix30, pix31, pix32, pix33, pix34,
                        pix40, pix41, pix42, pix43, pix44);
```

```
       }
}


__global__ void
{
       unsigned char *pImage =
              (unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

       for (int i = threadIdx.x; i < w; i += blockDim.x)
       {
              unsigned char pix00 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
3);
              unsigned char pix01 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
3);
              unsigned char pix02 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
3);
              unsigned char pix03 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
3);
              unsigned char pix04 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
3);
              unsigned char pix05 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
3);
              unsigned char pix06 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
3);

              unsigned char pix10 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
2);
              unsigned char pix11 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
2);
              unsigned char pix12 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
2);
              unsigned char pix13 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
2);
              unsigned char pix14 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
2);
              unsigned char pix15 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
2);
              unsigned char pix16 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
2);

              unsigned char pix20 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
1);
              unsigned char pix21 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
1);
              unsigned char pix22 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
              unsigned char pix23 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
              unsigned char pix24 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
              unsigned char pix25 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
1);
              unsigned char pix26 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
1);

              unsigned char pix30 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
0);
```

```
                unsigned char pix31 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
0);
                unsigned char pix32 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
                unsigned char pix33 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
0);
                unsigned char pix34 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
0);
                unsigned char pix35 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
0);
                unsigned char pix36 = tex2D(tex, (float)i + 3, (float)blockIdx.x +
0);

                unsigned char pix40 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
1);
                unsigned char pix41 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
1);
                unsigned char pix42 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
1);
                unsigned char pix43 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
1);
                unsigned char pix44 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
1);
                unsigned char pix45 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
1);
                unsigned char pix46 = tex2D(tex, (float)i + 3, (float)blockIdx.x +
1);

                unsigned char pix50 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
2);
                unsigned char pix51 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
2);
                unsigned char pix52 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
2);
                unsigned char pix53 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
2);
                unsigned char pix54 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
2);
                unsigned char pix55 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
2);
                unsigned char pix56 = tex2D(tex, (float)i + 3, (float)blockIdx.x +
2);

                unsigned char pix60 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
3);
                unsigned char pix61 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
3);
                unsigned char pix62 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
3);
                unsigned char pix63 = tex2D(tex, (float)i + 0, (float)blockIdx.x +
3);
                unsigned char pix64 = tex2D(tex, (float)i + 1, (float)blockIdx.x +
3);
                unsigned char pix65 = tex2D(tex, (float)i + 2, (float)blockIdx.x +
3);
                unsigned char pix66 = tex2D(tex, (float)i + 3, (float)blockIdx.x +
3);
```

```
		pImage[i] = ComputeMean7x7(pix00, pix01, pix02, pix03, pix04, pix05, 
pix06,
			pix10, pix11, pix12, pix13, pix14, pix15, pix16,
			pix20, pix21, pix22, pix23, pix24, pix25, pix26,
			pix30, pix31, pix32, pix33, pix34, pix35, pix36,
			pix40, pix41, pix42, pix43, pix44, pix45, pix46,
			pix50, pix51, pix52, pix53, pix54, pix55, pix56,
			pix60, pix61, pix62, pix63, pix64, pix65, pix66
			);
	}
}

__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
1);
		unsigned char pix01 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
1);
		unsigned char pix02 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
1);
		unsigned char pix10 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
0);
		unsigned char pix11 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
0);
		unsigned char pix12 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
0);
		unsigned char pix20 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
1);
		unsigned char pix21 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
1);
		unsigned char pix22 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
1);
		pImage[i] = ComputeMedian3x3(pix00, pix01, pix02,
			pix10, pix11, pix12,
			pix20, pix21, pix22);
	}
}


__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 2, (float)blockIdx.x - 
2);
		unsigned char pix01 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
2);
		unsigned char pix02 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
2);
```

```
                unsigned char pix03 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
2);
                unsigned char pix04 = tex2D(tex, (float)i + 2, (float)blockIdx.x - 
2);

                unsigned char pix10 = tex2D(tex, (float)i - 2, (float)blockIdx.x - 
1);
                unsigned char pix11 = tex2D(tex, (float)i - 1, (float)blockIdx.x - 
1);
                unsigned char pix12 = tex2D(tex, (float)i + 0, (float)blockIdx.x - 
1);
                unsigned char pix13 = tex2D(tex, (float)i + 1, (float)blockIdx.x - 
1);
                unsigned char pix14 = tex2D(tex, (float)i + 2, (float)blockIdx.x - 
1);

                unsigned char pix20 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
0);
                unsigned char pix21 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
0);
                unsigned char pix22 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
0);
                unsigned char pix23 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
0);
                unsigned char pix24 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
0);

                unsigned char pix30 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
1);
                unsigned char pix31 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
1);
                unsigned char pix32 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
1);
                unsigned char pix33 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
1);
                unsigned char pix34 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
1);

                unsigned char pix40 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
2);
                unsigned char pix41 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
2);
                unsigned char pix42 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
2);
                unsigned char pix43 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
2);
                unsigned char pix44 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
2);

                pImage[i] = ComputeMedian5x5(pix00, pix01, pix02, pix03, pix04,
                        pix10, pix11, pix12, pix13, pix14,
                        pix20, pix21, pix22, pix23, pix24,
                        pix30, pix31, pix32, pix33, pix34,
                        pix40, pix41, pix42, pix43, pix44);

        }
}
```

```
__global__ void
{
	unsigned char *pImage =
		(unsigned char *)(((char *)pImageOriginal) + blockIdx.x*Pitch);

	for (int i = threadIdx.x; i < w; i += blockDim.x)
	{
		unsigned char pix00 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
3);
		unsigned char pix01 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
3);
		unsigned char pix02 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
3);
		unsigned char pix03 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
3);
		unsigned char pix04 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
3);
		unsigned char pix05 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
3);
		unsigned char pix06 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
3);

		unsigned char pix10 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
2);
		unsigned char pix11 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
2);
		unsigned char pix12 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
2);
		unsigned char pix13 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
2);
		unsigned char pix14 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
2);
		unsigned char pix15 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
2);
		unsigned char pix16 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
2);

		unsigned char pix20 = tex2D(tex, (float)i - 3, (float)blockIdx.x -
1);
		unsigned char pix21 = tex2D(tex, (float)i - 2, (float)blockIdx.x -
1);
		unsigned char pix22 = tex2D(tex, (float)i - 1, (float)blockIdx.x -
1);
		unsigned char pix23 = tex2D(tex, (float)i + 0, (float)blockIdx.x -
1);
		unsigned char pix24 = tex2D(tex, (float)i + 1, (float)blockIdx.x -
1);
		unsigned char pix25 = tex2D(tex, (float)i + 2, (float)blockIdx.x -
1);
		unsigned char pix26 = tex2D(tex, (float)i + 3, (float)blockIdx.x -
1);

		unsigned char pix30 = tex2D(tex, (float)i - 3, (float)blockIdx.x +
0);
		unsigned char pix31 = tex2D(tex, (float)i - 2, (float)blockIdx.x +
0);
		unsigned char pix32 = tex2D(tex, (float)i - 1, (float)blockIdx.x +
0);
```

```
		unsigned char pix33 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
0);
		unsigned char pix34 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
0);
		unsigned char pix35 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
0);
		unsigned char pix36 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
0);

		unsigned char pix40 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
1);
		unsigned char pix41 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
1);
		unsigned char pix42 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
1);
		unsigned char pix43 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
1);
		unsigned char pix44 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
1);
		unsigned char pix45 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
1);
		unsigned char pix46 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
1);

		unsigned char pix50 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
2);
		unsigned char pix51 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
2);
		unsigned char pix52 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
2);
		unsigned char pix53 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
2);
		unsigned char pix54 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
2);
		unsigned char pix55 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
2);
		unsigned char pix56 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
2);

		unsigned char pix60 = tex2D(tex, (float)i - 3, (float)blockIdx.x + 
3);
		unsigned char pix61 = tex2D(tex, (float)i - 2, (float)blockIdx.x + 
3);
		unsigned char pix62 = tex2D(tex, (float)i - 1, (float)blockIdx.x + 
3);
		unsigned char pix63 = tex2D(tex, (float)i + 0, (float)blockIdx.x + 
3);
		unsigned char pix64 = tex2D(tex, (float)i + 1, (float)blockIdx.x + 
3);
		unsigned char pix65 = tex2D(tex, (float)i + 2, (float)blockIdx.x + 
3);
		unsigned char pix66 = tex2D(tex, (float)i + 3, (float)blockIdx.x + 
3);


		pImage[i] = ComputeMedian7x7(pix00, pix01, pix02, pix03, pix04, 
pix05, pix06,
			pix10, pix11, pix12, pix13, pix14, pix15, pix16,
                    pix20, pix21, pix22, pix23, pix24, pix25, pix26,
                    pix30, pix31, pix32, pix33, pix34, pix35, pix36,
                    pix40, pix41, pix42, pix43, pix44, pix45, pix46,
                    pix50, pix51, pix52, pix53, pix54, pix55, pix56,
                    pix60, pix61, pix62, pix63, pix64, pix65, pix66
                    );
        }
}


extern "C" void setupTexture(int iw, int ih, Pixel *data, int Bpp)
{
    cudaChannelFormatDesc desc;

    if (Bpp == 1)
    {
        desc = cudaCreateChannelDesc<unsigned char>();
    }
    else
    {
        desc = cudaCreateChannelDesc<uchar4>();
    }

    checkCudaErrors(cudaMallocArray(&array, &desc, iw, ih));
    checkCudaErrors(cudaMemcpyToArray(array, 0, 0, data, Bpp*sizeof(Pixel)*iw*ih,
cudaMemcpyHostToDevice));

       checkCudaErrors(cudaMallocArray(&arrayTemp, &desc, iw, ih));
       checkCudaErrors(cudaMemcpyToArray(arrayTemp, 0, 0, data,
Bpp*sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice));
}


extern "C" void deleteTexture(void)
{
    checkCudaErrors(cudaFreeArray(array));
       checkCudaErrors(cudaFreeArray(arrayTemp));
}


extern "C" void morphologicalFilter(Pixel *odata, int iw, int ih, enum DisplayMode
mode)
{
    checkCudaErrors(cudaBindTextureToArray(tex, array));
       checkCudaErrors(cudaBindTextureToArray(texTem, arrayTemp));


       unsigned char* hmatrix = (unsigned char*)malloc(sizeof(unsigned
char)*iw*ih);

    switch (mode)
    {
        case DISPLAY_IMAGE:


            CopyImage<<<ih, 384>>>(odata, iw, iw, ih);

                    cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);
```

```
              break;
                case MORPHOLOGICAL_EROSION:
                       ErosionTex <<<ih, 384 >>>(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_EROSION5x5:
                       ErosionTex5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_EROSION7x7:
                       ErosionTex7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_DILATION:
                       DilationTex << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_DILATION5x5:
                       DilationTex5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_DILATION7x7:
                       DilationTex7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_CLOSE:
                       DilationTex << <ih, 384 >> >(odata, iw, iw, ih);
                       cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);
                       cudaMemcpyToArray(arrayTemp, 0, 0, hmatrix, 1 *
sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice);
                       ErosionTexClose << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_CLOSE5x5:
                       DilationTex5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                       cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);
                       cudaMemcpyToArray(arrayTemp, 0, 0, hmatrix, 1 *
sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice);
                       ErosionTexClose5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_CLOSE7x7:
                       DilationTex7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                       cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);
                       cudaMemcpyToArray(arrayTemp, 0, 0, hmatrix, 1 *
sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice);
                       ErosionTexClose7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_OPEN:
                       ErosionTex << <ih, 384 >> >(odata, iw, iw, ih);
                       cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);
                       cudaMemcpyToArray(arrayTemp, 0, 0, hmatrix, 1 *
sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice);
                       DilationTexOpen << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_OPEN5x5:
                       ErosionTex5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                       cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);

                       cudaMemcpyToArray(arrayTemp, 0, 0, hmatrix, 1 *
sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice);
                       DilationTexOpen5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                       break;
                case MORPHOLOGICAL_OPEN7x7:
                       ErosionTex7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                       cudaMemcpy(hmatrix, odata, iw*ih, cudaMemcpyDeviceToHost);
```

```
                    cudaMemcpyToArray(arrayTemp, 0, 0, hmatrix, 1 *
sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice);
                    DilationTexOpen7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                    break;
             case MEAN_OPERATION:
                    MeanTex << <ih, 384 >> >(odata, iw, iw, ih);
                    break;
             case MEAN_OPERATION5x5:
                    MeanTex5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                    break;
             case MEAN_OPERATION7x7:
                    MeanTex7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                    break;
             case MEDAIN_OPERATION:
                    MedianTex << <ih, 384 >> >(odata, iw, iw, ih);
                    break;
             case MEDAIN_OPERATION5x5:
                    MedianTex5x5 << <ih, 384 >> >(odata, iw, iw, ih);
                    break;
             case MEDAIN_OPERATION7x7:
                    MedianTex7x7 << <ih, 384 >> >(odata, iw, iw, ih);
                    break;

             default:
                    break;
    }

    checkCudaErrors(cudaUnbindTexture(tex));
       checkCudaErrors(cudaUnbindTexture(texTem));


       free(hmatrix);
}
```

```
#ifndef __MORPHOLOGICAL_KERNELS_H_
#define __MORPHOLOGICAL_KERNELS_H_

typedef unsigned char Pixel;


enum DisplayMode
{
	DISPLAY_IMAGE = 0,
	MORPHOLOGICAL_EROSION,
	MORPHOLOGICAL_EROSION5x5,
	MORPHOLOGICAL_EROSION7x7,
	MORPHOLOGICAL_DILATION,
	MORPHOLOGICAL_DILATION5x5,
	MORPHOLOGICAL_DILATION7x7,
	MORPHOLOGICAL_CLOSE,
	MORPHOLOGICAL_CLOSE5x5,
	MORPHOLOGICAL_CLOSE7x7,
	MORPHOLOGICAL_OPEN,
	MORPHOLOGICAL_OPEN5x5,
	MORPHOLOGICAL_OPEN7x7,
	MEAN_OPERATION,
	MEAN_OPERATION5x5,
	MEAN_OPERATION7x7,
	MEDAIN_OPERATION,
	MEDAIN_OPERATION5x5,
	MEDAIN_OPERATION7x7
};

extern enum DisplayMode g_DisplayMode;

extern "C" void morphologicalFilter(Pixel *odata, int iw, int ih, enum DisplayMode
mode);
extern "C" void setupTexture(int iw, int ih, Pixel *data, int Bpp);
extern "C" void deleteTexture(void);
extern "C" void initFilter(void);


#endif
```

BIBLIOGRAPHY

[1] T. Maintz, T., *Digital and medical image processing,* 2005.

[2] R. Gonzalez, *Digital image processing.* (3rd ed.). New Jersey. Pearson Education Ltd, 2013.

[3] S.W. Smith, *The scientist and engineer's guide to digital signal processing.* (2nd ed.). California, USA: California Technical Publishing, 1999.

[4] R. Rangayyan, *Biomedical image analysis.* Boca Raton, Florida: CRC Press, 2005.

[5] *MATLAB*: The language of technical computing, http://www.mathworks.com/help/matlab/, Accessed Feb 13, 2015.

[6] N. Efford, *Image processing: A practical introduction using Java.* Boston: Addison-Wesley Longman Publishing Co, 2000.

[7] C. Solomon, and T. Breckon, *Fundamentals of digital image processing.* Chichester, UK: John Wiley & Sons Ltd, 2011.

[8] *Morphology fundamentals: Dilation and erosion*, http://www.mathworks.com/help/images/morphology-fundamentals-dilation-and-erosion.html, Accessed Feb 13, 2015.

[9] *Morphological operators*, http://www.coe.utah.edu/~cs4640/slides/Lecture11.pdf, Accessed Feb 01, 2016.

[10] *Erosion*, http://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm, Accessed Feb 01, 2015.

[11] L. Vincent, "Grayscale area openings and closings, their efficient implementation and applications," *Proc. EURASIP workshop on mathematical morphology and its applications to signal processing, Barcelona, Spain,* 22-27, 1993.

[12] C. Klifa, *Image enhancement - Spatial domain*, http://depts.washington.edu/bicg/documents/BE244_Image_Enhancement_Spatial_Filtering.pdf, Accessed Feb 30, 2015.

[13] A. Gregerson, *Implementing fast MRI gridding on GPUs via CUDA.* Madison: University of Wisconsin, http://www.nvidia.ca/docs/IO/47905/ECE757_Project_Report_Gregerson.pdf, Accessed Jan 08, 2014.

[14] A. Georgantzoglou, J. da Silva & R. Jena, "Image processing with MATLAB and GPU. In K. Bennett (Ed.)," *MATLAB applications for the practical engineer*. doi: 10.5772/57070, 2014.

[15] A. Das, *Process time comparison between GPU and CPU.* Hamburger Sternwarte: Universität Hamburg, 2011.

[16] *Nvidia*, http://www.nvidia.com, Accessed Feb 04, 2014.

[17] D. Kirk & W. Hwu, *Programming massively parallel processors: A hands-on approach.* (1st ed.). Burlington, MA: Morgan Kaufmann Ltd, 2010.

[18] *Linear systems*, http://www.intelligence.tuc.gr/~petrakis/courses/computervision/filtering.pdf, Accessed March 19, 2015.

[19] Example of non-linear filtering: Median filtering, http://www.unit.eu/cours/videocommunication/Non-linear_filtering.pdf, Accessed Feb 19, 2015.

[20] D. Phillips, *Image Processing in C.* (2nd ed.). Lawrence, Kansas: R & D Publications, 2000.

[21] What are the mean and median filters?, http://www.markschulze.net/java/meanmed.html, Accessed Feb 19, 2015.

[22] Image processing techniques, http://www.ncsa.illinois.edu/People/kindr/phd/PART1.PDF, Accessed March 19, 2015.

[23] GPU accelerated computing with C and C++, https://developer.nvidia.com/how-to-cuda-c-cpp, Accessed Jan 03, 2016.

[24] T. Benazir, B. & Imran, "Removal of high and low density impulse noise from digital images using non linear filter," Conference of signal processing image processing and pattern recognition. doi: 10.1109/ICSIPR.2013.6498000, 2013.

[25] Z. Bao, X. You, C. Xing & Q. He, *Image denoising by using non-tensor product wavelets filter banks.* Wuhan, China: Hubei University, 2007.

[26] L. State & C. Sararu, *New approaches in image compression and noise removal.* Bucharest, Romania: Bucharest Academy of Economic Studies, 2009.

[27] C. Marwa, B. Haythem, S. Fatma & A. Mohamed, "Image processing application on graphics processors," *International Journal of Image Processing, 8(3), 66-72*, 2014.

[28] A. Gregerson, *Implementing fast MRI gridding on GPUs via CUDA*. Madison: University of Wisconsin.

[29] I. Jeong, M. Hong, K. Hahn, J. Choi C. and Kim, *Performance Study of Satellite Image Processing on Graphics Processors Unit Using CUDA.* Department of Applied Information Technology: Kookmin University, 2012.

[30] Binna, T., and Hofmann, M. (2011). Performance Measurements of Algorithms in Image Processing.

[31] A. Remenyi, S. Szenasi, T. Bandi, Z. Vamossy, G. Valcz, P. Bogdanov, S. Sergyan and M. Kozlovszky, *Parallel Biomedical Image Processing with GPGPUs in Cancer Research*, 2011.

[32] C. Marwa, B. Haythem, S. Fatma A. and Mohamed, *Image Processing Application on Graphics processors,* 2014.

[33] H. Jang, A. Park, K. and Jung, *Neural Network Implementation using CUDA and OpenMP,* 2008.

[34] *GeForce GTX 860M,* http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-860m, Accessed Jan 08, 2014.