APPLICATION OF PARALLEL ALGORITHM APPROACH FOR PERFORMANCE OPTIMIZATION OF OIL PAINT IMAGE FILTER ALGORITHM

Siddhartha Mukherjee

Samsung R&D Institute, India - Bangalore

ABSTRACT

This paper gives a detailed study on the performance of image filter algorithm with various parameters applied on an image of RGB model. There are various popular image filters, which consumes large amount of computing resources for processing. Oil paint image filter is one of the very interesting filters, which is very performance hungry. Current research tries to find improvement in oil paint image filter algorithm by using parallel pattern library. With increasing kernel-size, the processing time of oil paint image filter algorithm increases exponentially. I have also observed in various blogs and forums, the questions for faster oil paint have been asked repeatedly.

KEYWORDS

Image Processing, Image Filters, Linear Image Filters, Colour Image Processing, Spatial Image Filter, Oil Paint algorithm, Parallel Pattern Library.

1. INTRODUCTION

This document provides an analytical study on the performance of Oil Paint Image Filter Algorithm. There are various popular linear image filters are available. One of the very popular and interesting image filters is Oil Paint image effect. This algorithm, being heavy in terms of processing it is investigated in this study.

There related studies are detailed in the Section 7.

2. BACKGROUND

Modern days, human beings have hands full of digital companions, e.g. Digital Camera, Smart Phones, Tablets and laptops so on. Most of these are having built-in camera, which are widely used compared to traditional cameras. The usage of these has started a new stream of applications, which include various categories e.g. image editing, image enhancement, camera extension application and so on. Almost all of these applications include applying different kinds of image filters.

Image filters are of different kinds, with respect their nature of processing or mathematical model. The execution time of any image filter is a very important aspect, which specifies whether the filter is acceptable for an application with respect to its acceptable execution performance in a given environment. E.g. The exaction time of any image filter, clearly specifies, whether the filter can be used for pre-processing or post-processing. Oil Paint is one of the very popular linear image filters, which is very heavy in terms of execution time. The due course of this paper portrays the investigation of the performance of oil paint image filter.

3. INVESTIGATION METHOD

A simple Win32 windows application is developed for simulation to analyse different types of image filters. The purpose of this windows application is to accept different JPG image files as an input, and apply different kinds of image filters on to it. In the process of applying image filters the application keeps log of processing time. The overall operation of the application is explained here.

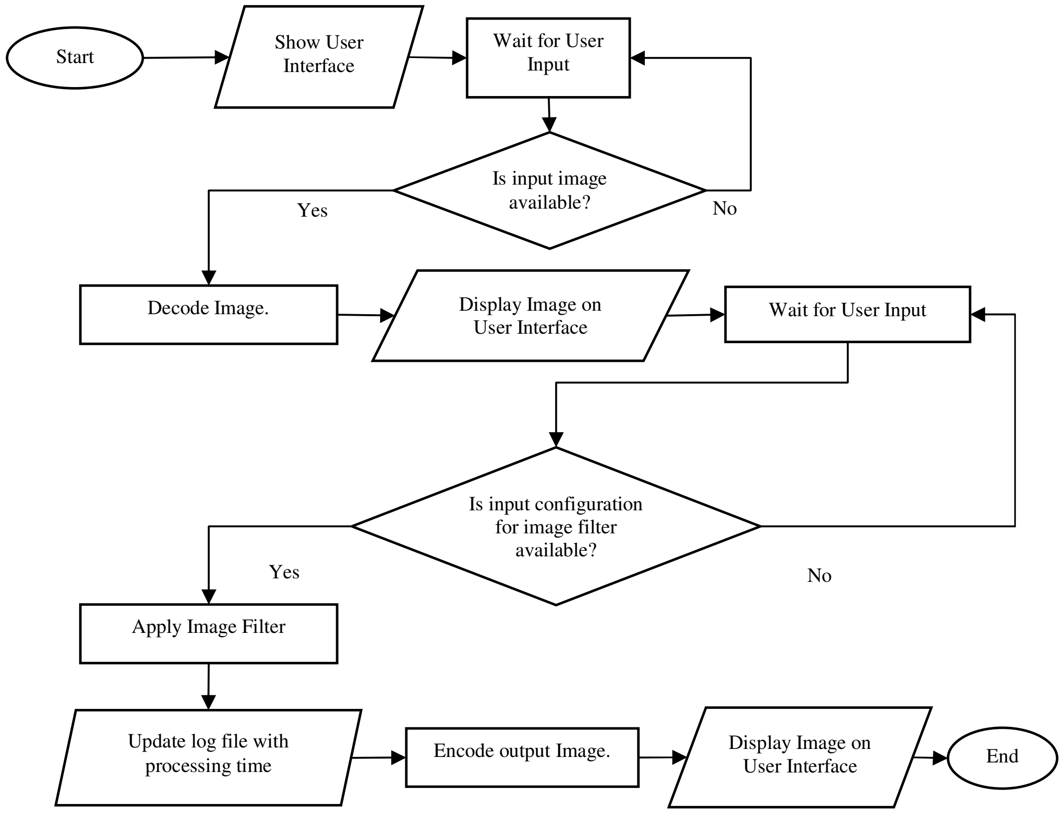
3.1. Operational Overview

The application is realised with two primary inputs: input jpg image files and configuration of image filter parameters. The application is designed with three major components: user interface (UI), jpeg image encoder-decoder and image filter algorithm.

The UI is developed using Microsoft's Win32 APIs. For this sub-module I have written approximately 300 LOC in C.

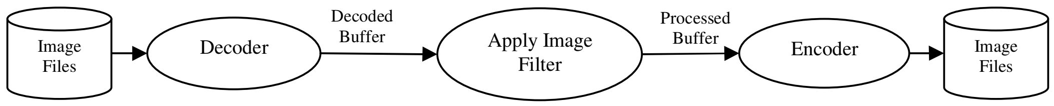
The image encoder and decoder component is designed with WIC (Windows Imaging Component), provided by Microsoft and developed with approximately 100 LOC in C.

The following flowchart diagram shows the operational overview of the test environment. During this process of testing, the processing time is logged and analysed for the study. The Oil paint image filter algorithm is developed using the approach as mentioned in reference [3]. This algorithm and its subsequent improvements are in developed with C language, in approximated 300 LOC.


Start
Show User Interface
Wait for User Input
Is input image available?
Yes
No
Decode Image.
Display Image on User Interface
Wait for User Input
Is input configuration for image filter available?
Yes
No
Apply Image Filter
Update log file with processing time
Encode output Image.
Display Image on User Interface
End

3.2. Implementation Overview

Considering the above workflow diagram, main focus of the current study is done with the application of image filter (marked as "Apply Image Filter" operation). Other operations are considered to be well known and do not affect the study. The code snippet below will provide the clear view of implementation. The user interface can be designed in various ways; even this experiment can be performed without a GUI also. That is why the main operational point of interests can be realized with the following way.



Decoder

The interface for decoding is exposed as shown here.

```
/* ***********************************************************************************
 * Function Name : Decode
 * Description: The function decodes an image file and gets the decoded buffer.
 *
 * **********************************************************************************/
HRESULT Decode(LPCWSTR imageFilename, PUINT pWidth, PUINT pHeight, PBYTE* ppDecodedBuffer, PUINT pStride,
PUINT pBufferSize, WICPixelFormatGUID*        pWicPixelFormatGUID);
```

The implementation of the decoder interface is provided here.

```
HRESULT Decode(LPCWSTR imageFilename, PUINT pWidth, PUINT pHeight, PBYTE* ppDecodedBuffer, PUINT pStride,
PUINT pBufferSize, WICPixelFormatGUID* pWicPixelFormatGUID)
{
         HRESULT                             hr = S_OK;
         UINT                                frameCount = 0;
         IWICImagingFactory                  *pFactory = NULL;
         IWICBitmapDecoder                   *pBitmapJpgDecoder = NULL;
         IWICBitmapFrameDecode               *pBitmapFrameDecode = NULL;

         do
         {
                  /* Create Imaging Factory  */
                  BREAK_IF_FAILED(CoCreateInstance( CLSID_WICImagingFactory, NULL, CLSCTX_INPROC_SERVER,
                                   IID_IWICImagingFactory, (LPVOID*)&pFactory))

                  /*  Create Imaging Decoder for JPG File */
                  BREAK_IF_FAILED(pFactory->CreateDecoderFromFilename(imageFilename, NULL, GENERIC_READ,
                                                  WICDecodeMetadataCacheOnDemand, &pBitmapJpgDecoder))

                  /* Get decoded frame & its related information from Imaging Decoder for JPG File */
                  BREAK_IF_FAILED(pBitmapJpgDecoder->GetFrameCount(&frameCount))

                  BREAK_IF_FAILED(pBitmapJpgDecoder->GetFrame(0, &pBitmapFrameDecode))

                  /* Get Width and Height of the Frame */
                  BREAK_IF_FAILED(pBitmapFrameDecode->GetSize(pWidth, pHeight))

                  /* Get Pixel format and accordingly allocate memory for decoded frame */
                  BREAK_IF_FAILED(pBitmapFrameDecode->GetPixelFormat(pWicPixelFormatGUID))

                  *ppDecodedBuffer = allocateBuffer(pWicPixelFormatGUID, *pWidth, *pHeight,
                                                  pBufferSize, pStride))
                  if(*ppDecodedBuffer == NULL) break;

                  /* Get decoded frame  */
                  BREAK_IF_FAILED(pBitmapFrameDecode->CopyPixels(NULL, *pStride,
                                                           *pBufferSize, *ppDecodedBuffer))

         }while(false);

         if(NULL != pBitmapFrameDecode)      pBitmapFrameDecode->Release();
         if(NULL != pBitmapJpgDecoder)       pBitmapJpgDecoder->Release();
         if(NULL != pFactory)                pFactory->Release();

         return hr;
}
```

Encoder

The interface for encoding is exposed as shown here.

```
/* ********************************************************************************
 * Function Name : Encode
 *
 * Description: The function encodes a decoded buffer into an image file.
 *
 * ********************************************************************************/
HRESULT Encode(LPCWSTR outFilename, UINT imageWidth, UINT imageHeight, PBYTE pDecodedBuffer, UINT cbStride,
UINT cbBbufferSize, WICPixelFormatGUID*        pWicPixelFormatGUID);
```

The implementation of the encoder interface is provided here.

```
HRESULT Encode(LPCWSTR outFilename, UINT imageWidth, UINT imageHeight, PBYTE pDecodedBuffer, UINT cbStride,
UINT
                   cbBbufferSize, WICPixelFormatGUID*    pWicPixelFormatGUID)
{
         HRESULT                              hr = S_OK;
         UINT                                 frameCount = 0;
         IWICImagingFactory                   *pFactory = NULL;
         IWICBitmapEncoder                    *pBitmapJpgEncoder = NULL;
         IWICBitmapFrameEncode                *pBitmapFrameEncode = NULL;
         IWICStream                           *pJpgFileStream = NULL;

         do
         {
                  /* Create Imaging Factory  */
                  BREAK_IF_FAILED(CoCreateInstance(CLSID_WICImagingFactory, NULL, CLSCTX_INPROC_SERVER,
                                                   IID_IWICImagingFactory, (LPVOID*)&pFactory))

                  /* Create & Initialize Stream for an output JPG file */
                  BREAK_IF_FAILED(pFactory->CreateStream(&pJpgFileStream))

                  BREAK_IF_FAILED(pJpgFileStream->InitializeFromFilename(outFilename, GENERIC_WRITE))

                  /* Create & Initialize Imaging Encoder  */
                  BREAK_IF_FAILED(pFactory->CreateEncoder(GUID_ContainerFormatJpeg,
                                                  &GUID_VendorMicrosoft,
                                                           &pBitmapJpgEncoder))

                  /* Initialize a JPG Encoder */
                  BREAK_IF_FAILED(pBitmapJpgEncoder->Initialize(pJpgFileStream, WICBitmapEncoderNoCache))

                  /* Create & initialize a JPG Encoded frame */
                  BREAK_IF_FAILED(pBitmapJpgEncoder->CreateNewFrame(&pBitmapFrameEncode, NULL))
                  BREAK_IF_FAILED(pBitmapFrameEncode->Initialize(NULL))

                  /* Update the pixel information */
                  BREAK_IF_FAILED(pBitmapFrameEncode->SetPixelFormat(pWicPixelFormatGUID))
                  BREAK_IF_FAILED(pBitmapFrameEncode->SetSize(imageWidth, imageHeight))
                  BREAK_IF_FAILED(pBitmapFrameEncode->WritePixels(imageHeight, cbStride,
                                                         cbBbufferSize, pDecodedBuffer))

                  BREAK_IF_FAILED(pBitmapFrameEncode->Commit())
                  BREAK_IF_FAILED(pBitmapJpgEncoder->Commit())


         }while(false);

         if(NULL != pJpgFileStream)           pJpgFileStream->Release();
         if(NULL != pBitmapFrameEncode)       pBitmapFrameEncode->Release();
         if(NULL != pBitmapJpgEncoder)        pBitmapJpgEncoder->Release();
         if(NULL != pFactory)                 pFactory->Release();

         return hr;
}
```

```
/* ********************************************************************************
 * Utility Macros
 * ********************************************************************************/
#define BREAK_IF_FAILED(X)  hr = X; \
                            if(FAILED(hr)) { break; } \
```

Application of Image Filter

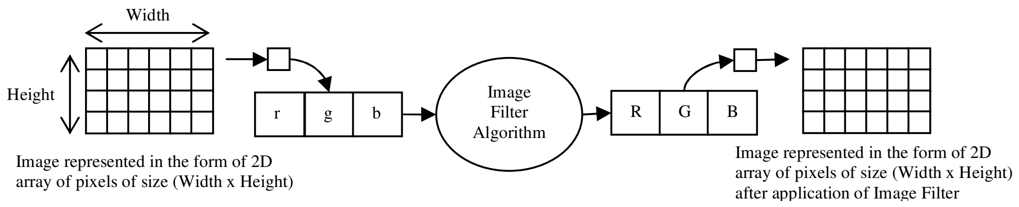
The image processing algorithm is the subject of study in current experiment. Details of the algorithms are explained later. Following code snippet used to capture performances for any simple filter (e.g. oil paint).

```
HRESULT ApplyOilPaintOnFile (LPCWSTR inImageFile, LPCWSTR outImageFile)
{

         HRESULT                     hr = S_OK;
         PBYTE                       pDecodedBuffer = NULL;
         PBYTE                       pOutputBuffer = NULL;
         UINT                        decodedBufferLen = 0;
         UINT                        inImageWidth = 0;
         UINT                        inImageHeight = 0;
         UINT                        cbStride = 0;
         WICPixelFormatGUID          wicPixelFormatGUID;
         DWORD                       dTimeStart = 0;
         DWORD                       dTimeDecode = 0;
         DWORD                       dTimeProcess = 0;
         DWORD                       dTimeEncode = 0;
         char                        sMessage[256] = {0};

         do
         {
                  /* --------- Decode. --------- */
                  dTimeStart = GetTickCount();

                  BREAK_IF_FAILED(Decode( inImageFile, &inImageWidth, &inImageHeight, &pDecodedBuffer,
                                          &cbStride, &decodedBufferLen, &wicPixelFormatGUID))

                  dTimeDecode = GetTickCount() - dTimeStart;

                  /* Allocate Memory for output. */
                  pOutputBuffer = (PBYTE)calloc(sizeof(BYTE), decodedBufferLen);
                  if(NULL == pOutputBuffer)
                           break;

                  /* ------------  Process Image Filter ------------ */
                  dTimeStart = GetTickCount();

                  BREAK_IF_FAILED(ApplyOilPaintOnBuffer(pDecodedBuffer,
                                                       inImageWidth, inImageHeight, pOutputBuffer))

                  dTimeProcess = GetTickCount() - dTimeStart;

                  /* --------- Encode ---------  */
                  dTimeStart = GetTickCount();

                  BREAK_IF_FAILED(Encode( outImageFile, inImageWidth, inImageHeight, pOutputBuffer,
                                          cbStride, decodedBufferLen, &wicPixelFormatGUID))

                  dTimeEncode = GetTickCount() - dTimeStart;

                  sprintf(sMessage,
                  "Grey Scale : Width=%d, Height=%d, Time(Decode)=%lu Time(Process)=%lu
         Time(Encode)=%lu\r\n",
                   inImageWidth, inImageHeight, dTimeDecode, dTimeProcess, dTimeEncode);

                  Log(sMessage);

         }while(false);

         if(NULL != pDecodedBuffer)  free(pDecodedBuffer);
         if(NULL != pOutputBuffer)   free(pOutputBuffer);

         return hr;
}
```

For time measurement standard windows API GetTickCount is used. This API retrieves the number of milliseconds that have elapsed since the system was started.

4. Oil Paint Image Filter in RGB Colour Model

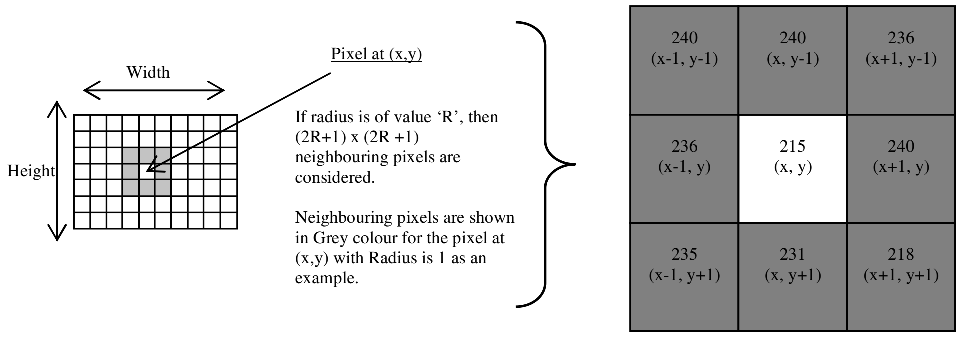
During this study, the input images are considered to be in RGB model. In this model, an image consists of two dimensional arrays of pixels. Each pixel of a 2D array contains data of red, green and blue colour channel respectively.


Width
Height
r
g
b
Image Filter Algorithm
R
G
B
Image represented in the form of 2D array of pixels of size (Width x Height)
Image represented in the form of 2D array of pixels of size (Width x Height) after application of Image Filter

The Image Filters are basically algorithm for changing the values of Red, Green and Blue component of a pixel to a certain value. Depending upon the amount of access to neighbouring pixels in spatial domain, the performance of image filters is affected.

Histogram based algorithm for Oil Paint

This approach is mentioned in the reference [3]. For pixel at position (x, y), find the most frequently occurring intensity value in its neighbourhood. And set it as the new colour value at position (x, y).


Pixel at (x,y)
Width
Height
If radius is of value 'R', then (2R+1) x (2R +1) neighbouring pixels are considered.
Neighbouring pixels are shown in Grey colour for the pixel at (x,y) with Radius is 1 as an example.
240 (x-1, y-1)
240 (x, y-1)
236 (x+1, y-1)
236 (x-1, y)
215 (x, y)
240 (x+1, y)
235 (x-1, y+1)
231 (x, y+1)
218 (x+1, y+1)

1) The right side provides the larger and clear picture of the neighbouring pixels or Radius 1, with respect to pixel at (x, y). The intensities of the respective pixels are also provided (as an example).
2) The pixels at (x-1, y-1), (x, y-1), (x+1, y) have the maximum occurring intensity i.e. 240.
3) The each colour channel of the pixel at (x, y) is set with an average of each colour channel of 3 pixels [(x-1, y-1), (x, y-1), (x+1, y)].

The intensity (I) of a pixel is calculated by equation I = (R + G + B) / 3. Here R, G & B are Red, Green and Blue component of a pixel.

The interface for the oil paint algorithm is exposed as follows.

```
/* *********************************************************************************
 * Function Name : ApplyOilPaintOnBuffer
 * Description: Apply oil paint effect on decoded buffer.
 *
 * *********************************************************************************/
HRESULT ApplyOilPaintOnBuffer(PBYTE pInBuffer, UINT width, UINT height, const UINT intensity_level, const int
radius, PBYTE pOutBuffer);
```

Just mentioned oil paint algorithm is implemented as follows.

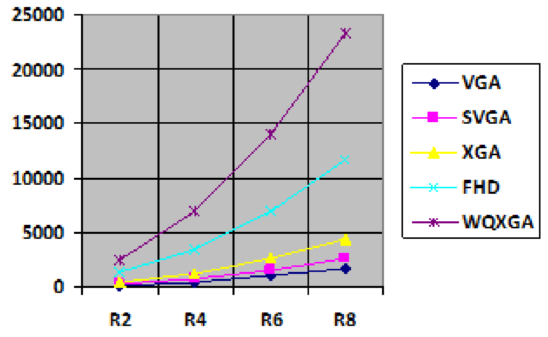
```
HRESULT ApplyOilPaintOnBuffer(PBYTE pInBuffer, UINT width, UINT height, const UINT intensity_level,
const int radius, PBYTE pOutBuffer)
{
         int                                     index = 0;
         int                                     intensity_count[255] = {0};
         int                                     sumR[255] = {0};
         int                                     sumG[255] = {0};
         int                                     sumB[255] = {0};
         int                                     current_intensity = 0;
         int                                     row,col, x,y;
         BYTE                                    r,g,b;
         int                                     curMax = 0;
         int                                     maxIndex = 0;

         if(NULL == pInBuffer || NULL == pOutBuffer)
                  return E_FAIL;

         for(col = radius; col < (height - radius); col++) {
                  for(row = radius; row < (width - radius); row++) {
                           memset(&intensity_count[0], 0, ARRAYSIZE(intensity_count));
                           memset(&sumR[0], 0, ARRAYSIZE(sumR));
                           memset(&sumG[0], 0, ARRAYSIZE(sumG));
                           memset(&sumB[0], 0, ARRAYSIZE(sumB));

                           /* Calculate the highest intensity Neighbouring Pixels. */
                           for(y = -radius; y <= radius; y++) {
                                    for(x = -radius; x <= radius; x++) {
                                      index = ((col + y) * width * 3) + ((row + x) * 3);

                                      r = pInBuffer[index + 0];
                                      g = pInBuffer[index + 1];
                                      b = pInBuffer[index + 2];

                                      current_intensity = ((r + g + b) * intensity_level/3.0)/255;
                                      intensity_count[current_intensity]++;

                                      sumR[current_intensity] += r;
                                      sumG[current_intensity] += g;
                                      sumB[current_intensity] += b;
                                    }
                           }

                           index = (col * width * 3) + (row * 3);

               /* The highest intensity neighbouring pixels are averaged out to get the exact color. */
               maxIndex = 0;
               curMax = intensity_count[maxIndex];

               for( int i = 0; i < intensity_level; i++ )   {
                  if( intensity_count[i] > curMax )   {
                      curMax = intensity_count[i];
                      maxIndex = i;
                  }
               }

            if(curMax > 0) {
                 pOutBuffer[index + 0] = sumR[maxIndex]/curMax;
                 pOutBuffer[index + 1] = sumG[maxIndex]/curMax;
                 pOutBuffer[index + 2] = sumB[maxIndex]/curMax;
            }
        }
    }

         return S_OK;
}
```

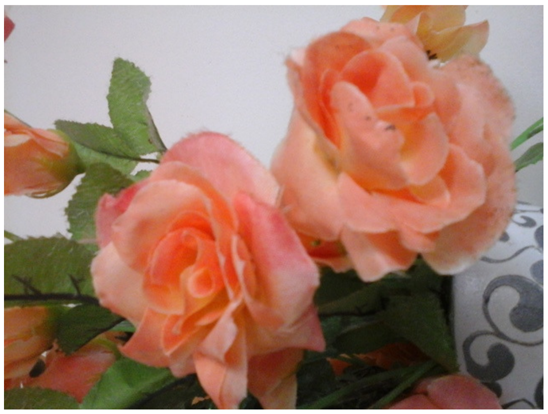
Experimental Results

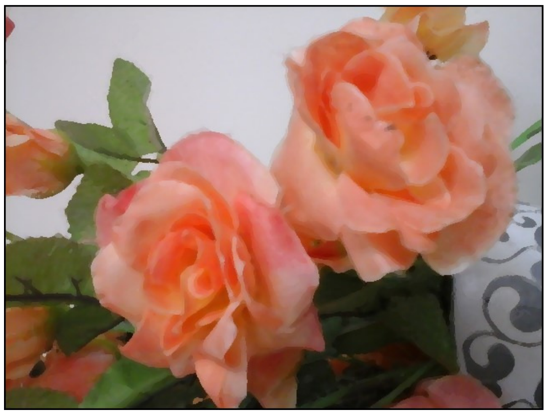
The experimental is conducted with images of different size and application of oil paint with different radius. The following data shows the time of execution with different parameters.

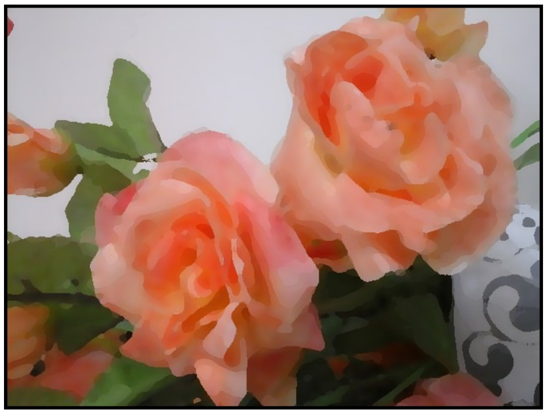
Size	Radius	Time
VGA(640x480)	2	218
VGA(640x480)	4	531
VGA(640x480)	6	1046
VGA(640x480)	8	1685
SVGA(800x600)	2	297
SVGA(800x600)	4	826
SVGA(800x600)	6	1606
SVGA(800x600)	8	2652
XGA(1024x768)	2	499
XGA(1024x768)	4	1326
XGA(1024x768)	6	2621
XGA(1024x768)	8	4383
FHD(1920x1080)	2	1466
FHD(1920x1080)	4	3526
FHD(1920x1080)	6	7020
FHD(1920x1080)	8	11716
WQXGA(2560x1600)	2	2559
WQXGA(2560x1600)	4	6973
WQXGA(2560x1600)	6	14008
WQXGA(2560x1600)	8	23229

System	Details
Processor	Intel® Core™ i7-3630QM CPU @ 2.40 GHz, 2.40 GHz
Operating System	Windows 7 Enterprise, 64 bit Operating System.
RAM	8.00GB

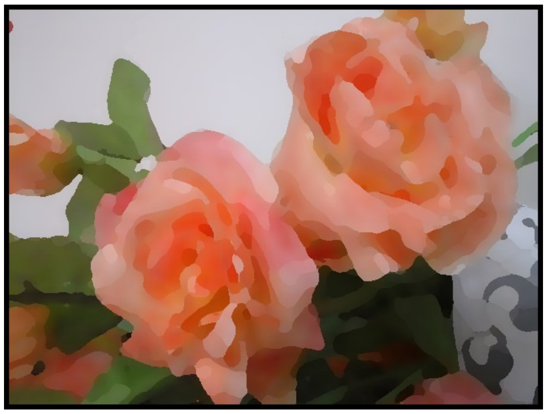



Input image of VGA (640x480) resolution. The image is resized for the purpose of documentation only.

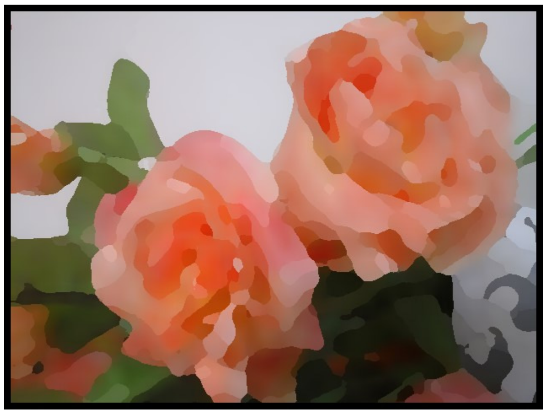

After application of oil paint algorithm with radius of **2**. Output image resolution is VGA 640x480. The image is resized for the purpose of documentation only.



After application of oil paint algorithm with radius of **4**. Output image resolution is VGA 640x480. The image is resized for the purpose of documentation only.



After application of oil paint algorithm with radius of **6**. Output image resolution is VGA 640x480. The image is resized for the purpose of documentation only.



After application of oil paint algorithm with radius of **8**. Output image resolution is VGA 640x480. The image is resized for the purpose of documentation only.

In due course of our investigation, I have observed that the performance of oil paint image filter increases in greater degree with increasing width, height and radius (i.e. usage of neighbouring pixel).

More importantly, I have observed most of the high resolution images are captured by more powerful camera. For these kinds of higher resolution photos, kernel size needs to be increased to generate Oil Paint effect of an acceptable quality

.

5. Oil Paint Image Filter by Parallel Algorithm Approach

I tried to improve the oil paint algorithm by Parallel Algorithm approach. I have used Microsoft Parallel Patterns Library for this purpose.

Parallel Patterns Library

The Parallel Patterns Library (PPL) is a Microsoft library designed for use by native C++ developers. PPL provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications. The PPL builds on the scheduling and resource management components of the Concurrency Runtime. It raises the level of abstraction between your application code and the underlying threading mechanism by providing generic, type-safe algorithms and containers that act on data in parallel.

The PPL provides the following features:

1) Task Parallelism: a mechanism to execute several work items (tasks) in parallel

2) Parallel algorithms: generic algorithms that act on collections of data in parallel

3) Parallel containers and objects: generic container types that provide safe concurrent access to their elements

Following code snippet will provide clear picture of the implementation using Microsoft PPL.

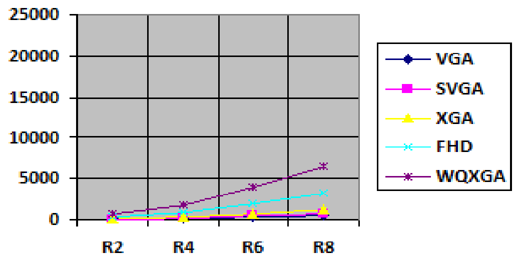
```
HRESULT ApplyOilPaintOnBuffer(PBYTE pInBuffer, UINT width, UINT height, const UINT intensity_level, const int
radius, PBYTE pOutBuffer)
{
        int tStart = radius;
        int tEnd =(height - radius);

        if(NULL == pInBuffer || NULL == pOutBuffer)
                        return E_FAIL;

        parallel_for(tStart, tEnd, [&pInBuffer, &width, &height, &intensity_level, &radius, &pOutBuffer]
        (int col){
                int                     index = 0;
                int                     intensity_count[255] = {0};
                int                     sumR[255] = {0};
                int                     sumG[255] = {0};
                int                     sumB[255] = {0};
                int                     current_intensity = 0;
                int                     row,x,y;
                BYTE                    r,g,b;
                int                     curMax = 0;
                int                     maxIndex = 0;

                for(row = radius; row < (width - radius); row++)
                {
                   /* This portion of the code remains same, as mentioned above */
                }

        });

        return S_OK;
}
```

Experimental Results

The experiment is conducted with same set of images, used for the experiment, mentioned in the section above. Better execution performance is achieved with same quality, achieved in the previous stage of experiment.

Size	Radius	Time
VGA(640x480)	2	94
VGA(640x480)	4	156
VGA(640x480)	6	281
VGA(640x480)	8	483
SVGA(800x600)	2	78
SVGA(800x600)	4	234
SVGA(800x600)	6	452
SVGA(800x600)	8	734
XGA(1024x768)	2	140
XGA(1024x768)	4	375
XGA(1024x768)	6	733
XGA(1024x768)	8	1248
FHD(1920x1080)	2	343
FHD(1920x1080)	4	967
FHD(1920x1080)	6	1935
FHD(1920x1080)	8	3261
WQXGA(2560x1600)	2	686
WQXGA(2560x1600)	4	1872
WQXGA(2560x1600)	6	3915
WQXGA(2560x1600)	8	6490

System	Details
Processor	Intel® Core™ i7-3630QM CPU @ 2.40 GHz, 2.40 GHz
Operating System	Windows 7 Enterprise, 64 bit Operating System.
RAM	8.00GB



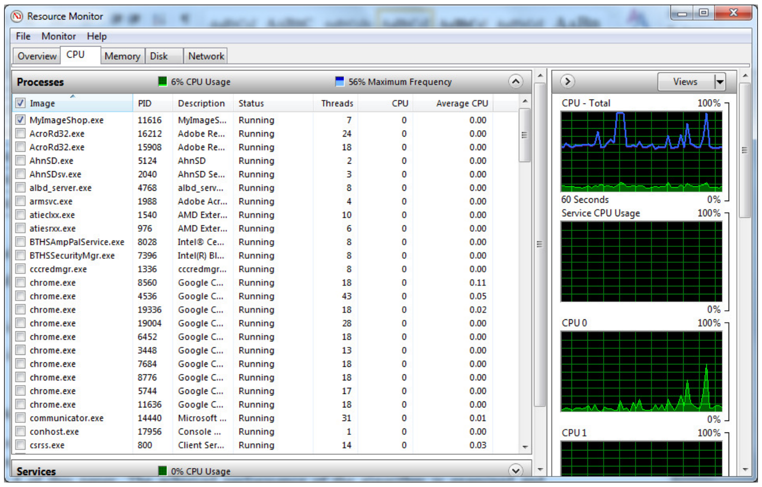
6. COMPARATIVE ANALYSIS OF BOTH APPROACHES

The improvement of the performance in terms of percentage is deduced as [100 * (T1 – T2)/ t1], where T1 is time required for processing by 1st approach and T2 is the time required for processing time by latest approach.

Size	Radius	T1	T2	Improvement (%)
VGA(640x480)	2	218	94	56.88073394
VGA(640x480)	4	531	156	70.62146893
VGA(640x480)	6	1046	281	73.13575526
VGA(640x480)	8	1685	483	71.33531157
SVGA(800x600)	2	297	78	73.73737374
SVGA(800x600)	4	826	234	71.67070218
SVGA(800x600)	6	1606	452	71.85554172
SVGA(800x600)	8	2652	734	72.32277526
XGA(1024x768)	2	499	140	71.94388778
XGA(1024x768)	4	1326	375	71.71945701
XGA(1024x768)	6	2621	733	72.03357497
XGA(1024x768)	8	4383	1248	71.52635181
FHD(1920x1080)	2	1466	343	76.60300136
FHD(1920x1080)	4	3526	967	72.57515598
FHD(1920x1080)	6	7020	1935	72.43589744
FHD(1920x1080)	8	11716	3261	72.16626835
WQXGA(2560x1600)	2	2559	686	73.19265338
WQXGA(2560x1600)	4	6973	1872	73.15359243
WQXGA(2560x1600)	6	14008	3915	72.05168475
WQXGA(2560x1600)	8	23229	6490	72.06078609

CPU & Thread Usage:

I have used 'Microsoft Resource Monitor' tool to analyse the CPU and Thread utilization of the system.



Processes 🟩 6% CPU Usage 🟦 56% Maximum Frequency

Image	PID	Description	Status	Threads	CPU	Average CPU
✓ MyImageShop.exe	11616	MyImageS...	Running	7	0	0.00
AcroRd32.exe	16212	Adobe Re...	Running	24	0	0.00
AcroRd32.exe	15908	Adobe Re...	Running	18	0	0.00
AhnSD.exe	5124	AhnSD	Running	2	0	0.00
AhnSDsv.exe	2040	AhnSD Se...	Running	3	0	0.00
albd_server.exe	4768	albd_serv...	Running	8	0	0.00
armsvc.exe	1988	Adobe Acr...	Running	4	0	0.00
atieclxx.exe	1540	AMD Exter...	Running	10	0	0.00
atiesrxx.exe	976	AMD Exter...	Running	6	0	0.00
BTHSAmpPalService.exe	8028	Intel® Ce...	Running	8	0	0.00
BTHSSecurityMgr.exe	7396	Intel(R) Bl...	Running	8	0	0.00
cccredmgr.exe	1336	cccredmgr...	Running	8	0	0.00
chrome.exe	8560	Google C...	Running	18	0	0.11
chrome.exe	4536	Google C...	Running	43	0	0.05
chrome.exe	19336	Google C...	Running	18	0	0.02
chrome.exe	19004	Google C...	Running	28	0	0.00
chrome.exe	6452	Google C...	Running	18	0	0.00
chrome.exe	3448	Google C...	Running	13	0	0.00
chrome.exe	7684	Google C...	Running	18	0	0.00
chrome.exe	8776	Google C...	Running	18	0	0.00
chrome.exe	5744	Google C...	Running	17	0	0.00
chrome.exe	11636	Google C...	Running	18	0	0.00
communicator.exe	14440	Microsoft ...	Running	31	0	0.01
conhost.exe	17956	Console ...	Running	1	0	0.00
csrss.exe	800	Client Ser...	Running	14	0	0.03

Services 🟩 0% CPU Usage

The application process is run and killed several times to find the average usage. This data is very likely to vary between various systems.

1) When PPL isn't used, minimum approximate CPU and Thread usage captured as 0.08 & 2

2) When PPL isn't used, maximum approximate CPU and Thread usage captured as 12.0 & 20

3) When PPL is used, minimum approximate CPU and Thread usage captured as 0.08 & 8

4) When PPL is used, maximum approximate CPU and Thread usage captured as 98 & 36

7. References Studied

From reference [3] the histogram based oil paint image filter algorithm has been studied. The algorithm (mentioned in reference [3], section 'Oil-paint Effect') is implemented, as explained in the section 4 of this paper. The achieved performance of the algorithm is examined and captured in the section 4 (sub-section: Experimental Result) here. The result shows high growth of the processing time with respect to kernel-size. Reference [4] is another reference, where algorithm similar reference [3] is proposed for implementation. The reference [1] and [2] are used for way of analysis and follow the broadened scope in this arena of image processing. Reference [5] also proposes algorithm which are similar in nature with reference [3]. So we can clearly depict algorithms similar to reference [3] and [5], will face similar performance problem.

8. Conclusions

As in section 4 & 7, obtained result depicts huge growth in processing time with respect to the increase in kernel size. The current paper conducts study on improving execution time of oil paint image filter algorithm using the Microsoft technology.

As shown in section 6, I conclude Microsoft Parallel Pattern library yielded 71.6% (average) performance improvement for Oil Paint Algorithm in given environment.

Applicability:

There are various similar image filter algorithm, where processing depends on neighbouring pixels. The image filters, face similar performance issues, as oil paint. The approach mentioned in this paper can be applied for similar issues.

In future, more well-known or new techniques in conjunction with the current idea can be used for betterment. Not only in image processing in other dimensions of signal processing as well similar approach can be tried.

Limitations and Areas of improvement:

The library for parallel execution depends on multi-core processing architecture. The result may differ on different processing architecture. More importantly, parallel pattern library may not be available for various operating systems (mainly, embedded devices). In those areas, the current approach may not be effective. Other programming techniques can be incorporated in those areas.

ACKNOWLEDGEMENTS

I thank my organization to provide me the opportunity for conducting this research!

REFERENCES

[1] Dr.G.Padmavathi, Dr.P.Subashini, Mr.M.Muthu Kumar and Suresh Kumar Thakur (2009) "Performance analysis of Non Linear Filtering Algorithms for underwater images", (IJCSIS) International Journal of Computer Science and Information Security. Vol.6, No. 2, 2009

[2] Aaron Hertzmann (1998) "Painterly rendering with curved brush strokes of multiple sizes", Proceedings of the 25th annual conference on Computer graphics and interactive techniques. Pages 453-460

[3] Feng Xiao (2000) "Oil-paint Effect". Spring 2000/EE368 Course Project.

[4] Oil Paint Algorithm [http://supercomputingblog.com/graphics/oil-painting-algorithm/]

[5] P. S. Grover, Priti Sehgal (2004) "A Proposed Glass-Painting Filter". University of Delhi /icvgip/2004 / proceedings /cg1.1_109

[6] Parallel Patterns Library (PPL) [http://en.wikipedia.org/wiki/Parallel_Patterns_Library]

Authors

Siddhartha Mukherjee is a B.Tech (Computer Science and Engineering) from *RCC Institute of Information Technology, Kolkata.* Siddhartha is currently working as a Technical Manager in Samsung R&D Institute, India- Bangalore. Siddhartha has almost 10 years of working experience in software development. He has previously worked with Wipro Technologies. He has been contributing for various technical papers & innovations at different forums in Wipro and Samsung. His main area of work is mobile application developments.