Pixel Perfect Effects

EE 568 Final Project

Kalana Sahabandu

Table of Contents

1. Introduction

This project illustrates implementation of Pencil sketch filter and Oil filter algorithms which can be applied on an image specified by the end user. Overall system architecture of the system can be broken down to following components.

- Front-end components
- Back-end components

Where, front-end component was developed using tkinter which is a GUI library for python and consists of following modules:

1. Main Canvas GUI module
2. Pencil sketch GUI module
3. Oil paint GUI module

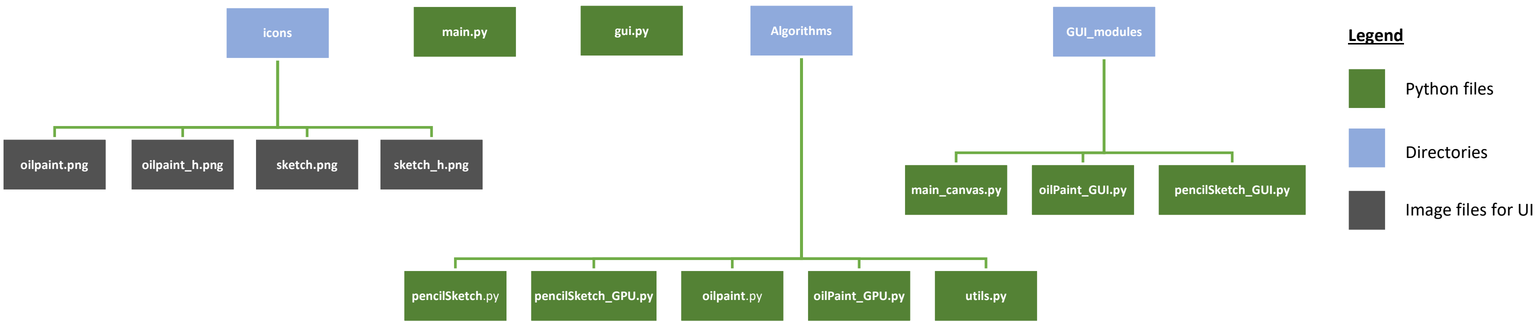
The sole purpose of these GUI modules is to represent results processed by the back-end modules to the user as well as setting different parameters such as kernel size, sigma values and threshold values for each back-end algorithm in a user-friendly manner.

On the other hand, the back-end component was developed using OpenCV to load and convert between color spaces, numpy for matrix initialization and matrix operations. In addition, to optimize these algorithms I have used numba which is a python library that helps to compile python code using a technique called Just In Time (JIT) compilation such that the code can utilize system GPU to parallelize matrix based calculations which will speed up processing in order of magnitude compared to running the code on CPU. The back-end component consists of following modules:

1. Pencil Sketch algorithm module – CPU based image processing
2. Pencil Sketch GPU algorithm module – Algorithm optimized for GPU processing
3. Oil Paint algorithm module – CPU based image processing
4. Oil Paint GPU algorithm module – Algorithm optimized for GPU processing
5. Utility functions – To encapsulate commonly used functions through out the application such as loading image data, displaying image data, color space conversions etc.

Pencil Sketch algorithm was developed to transform a given image to black and white pencil sketch like art, while oil paint algorithm was developed to transform a given image into a watercolor based image. In addition to Image processing techniques this project also demonstrates utilization of GPU to optimize image processing algorithms as well as parallel processing of matrix operations to speed up above mentioned algorithms.

In a nutshell, we could summarize over all architecture of the application as follows:



Note: In-depth information on each of the components and modules will be discussed in the implementation description section of this report.

2. <u>Instructions on how to run your program</u>

Important Note:

This python application is developed only to use on windows operating systems since it contains windows dependent libraries to extract window sizes and working area sizes.

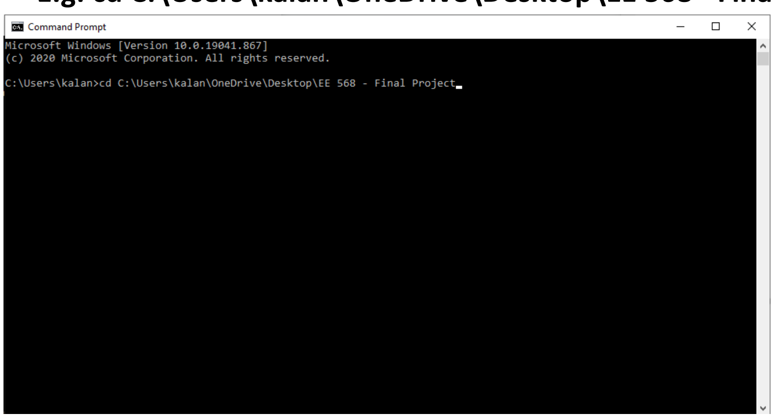
GPU optimization feature was tested and validated on a system with following specs and was able to achieve real time processing results. However, this GPU optimized code was not tested on AMD GPUs or Intel eGPUs

CPU: Intel i7- 7700k (4-cores and 8-threads) clocked at 5.0 GHz
RAM: 32 Gb DDR4 clocked at 3200 MHz
GPU: RTX 2080 with 8 Gb GDDR6 VRAM, clocked at 1800 MHz (base clock)
Tested screen Resolutions: 1080p, 1440p, 2160p and widescreen 1440p (5120x1440)

Also, the user interface is optimized to work on 1080p (1920 x 1080) and up monitors

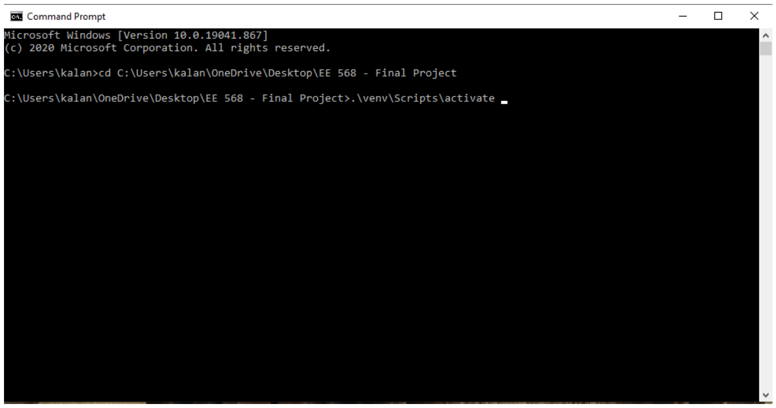
2.1 Running the application:

1. First open a command prompt (CMD) window
2. Then type following command “cd <PATH TO THE PROJECT DIRECTORY>”
    - **E.g: cd C:\Users\kalan\OneDrive\Desktop\EE 568 - Final Project**

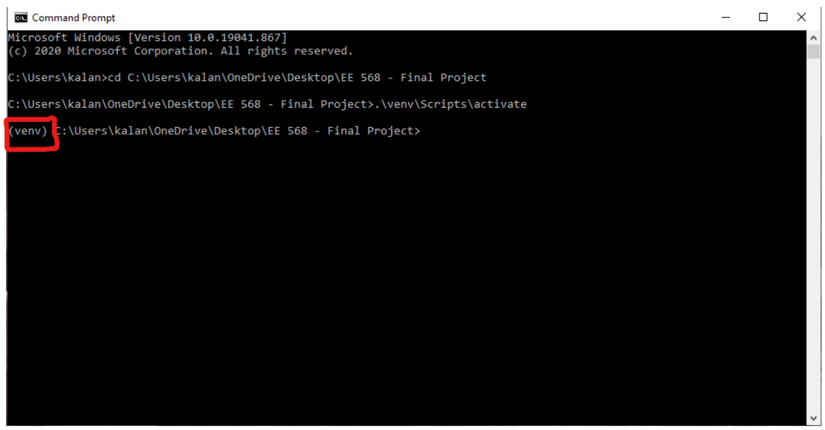


3. Once navigation to the project directory is completed. Type following command to activate python virtual environment (venv). This way you do not have to install dependencies to run the program. To activate venv type following command on your command prompt

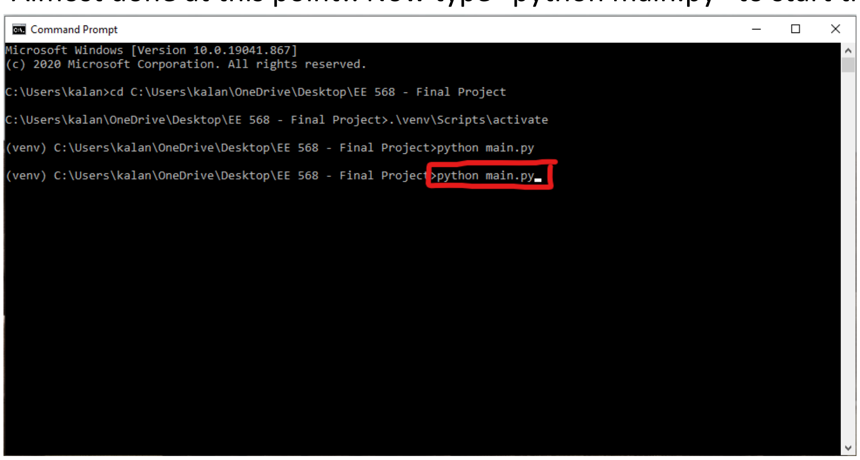
- **.\venv\Scripts\activate** (make sure to include the dot at the beginning)



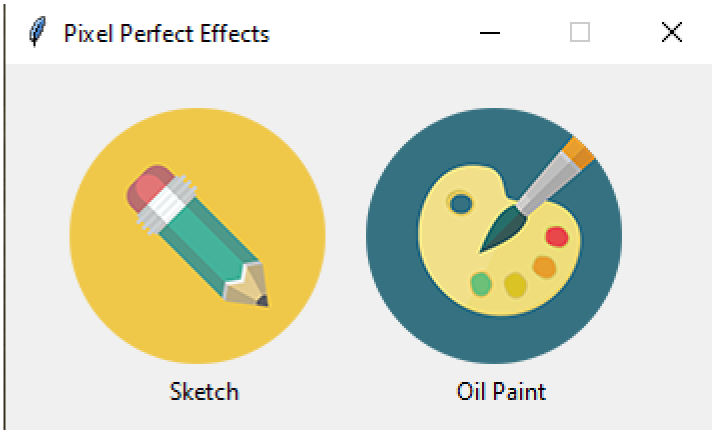
4. Once this command is executed, you should see "(venv)" at the beginning of the next command prompt input line, representing virtual environment has been activated.



5. Almost done at this point!. Now type "python main.py" to start the application!

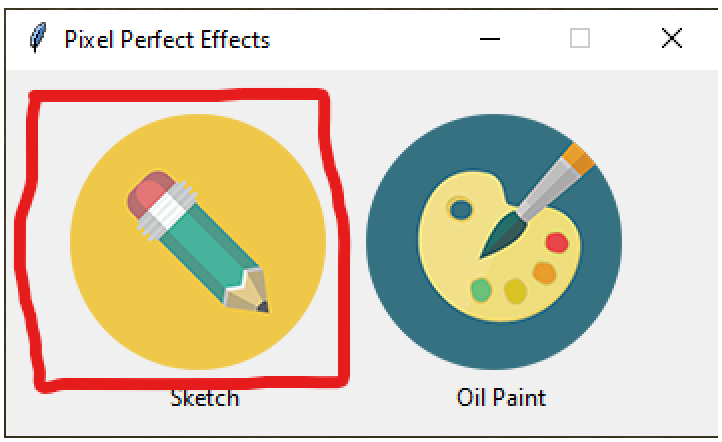


6. Once the application is started you should be greeted with following user interface

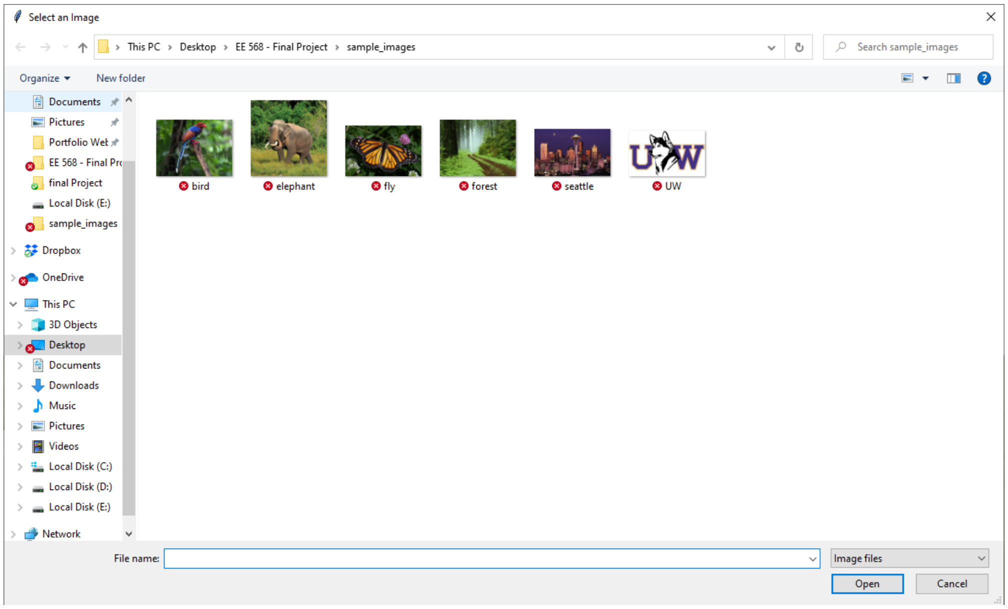


2.2 Applying Sketch effect

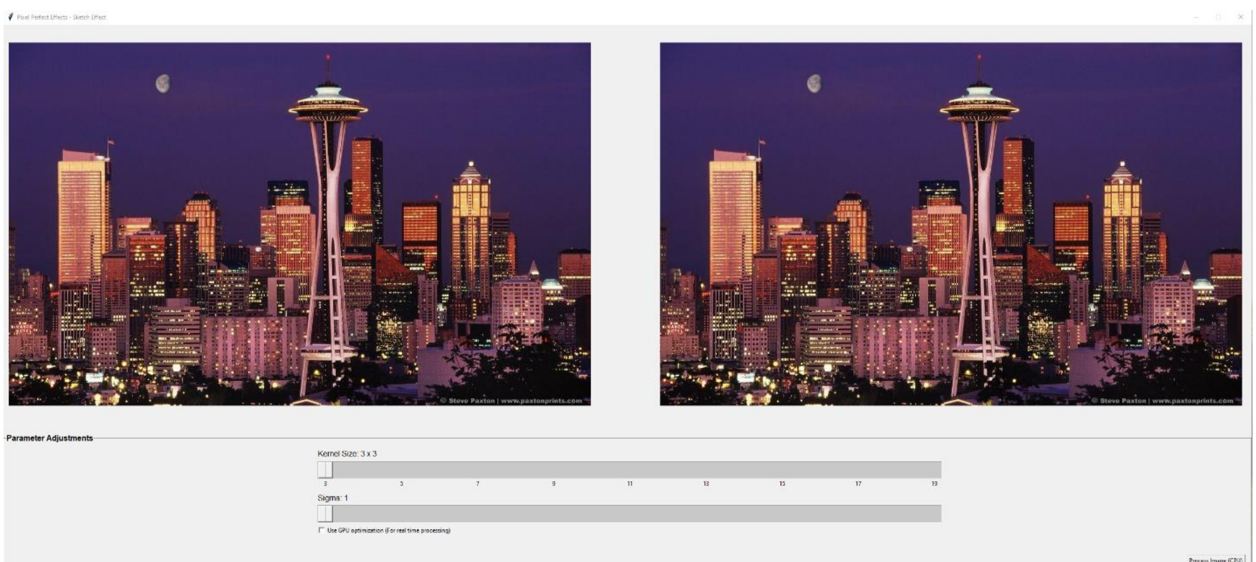
1. On the first interface window click on Sketch button



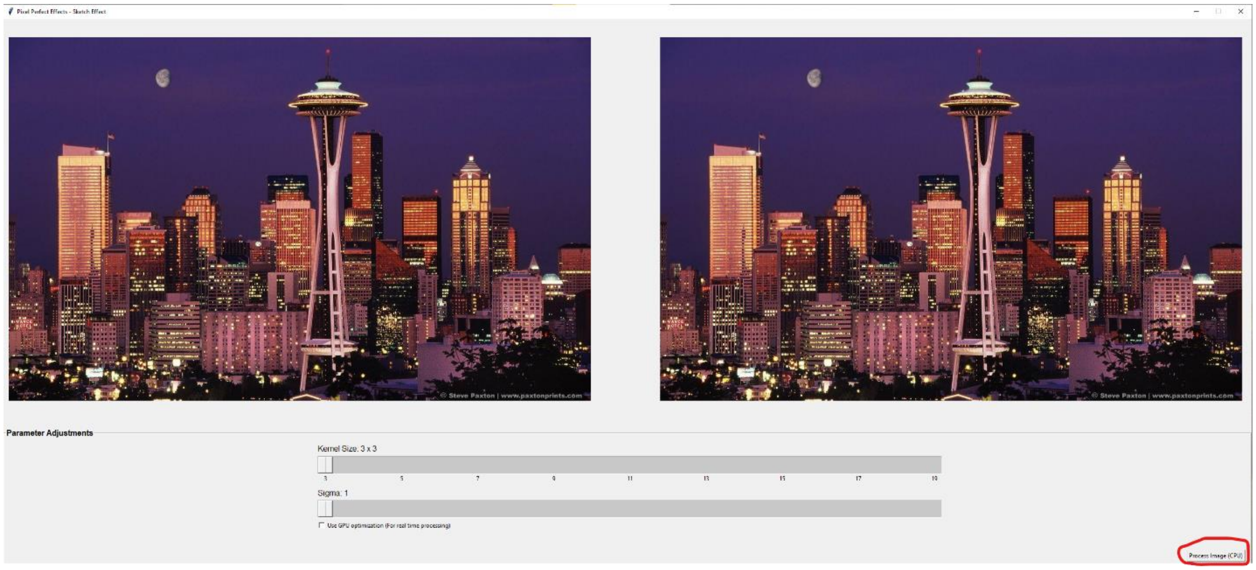
2. Once you clicked on the sketch pencil button, you will be greeted with a file selector pointing to "sample_images" directory within the project folder which contains six sample images that you could use to apply different effects (But feel free try any other image files you would like to apply effects on, the application is smart enough to filter out image file types supported by OpenCV). For the purpose of demonstration, I'm using "seattle.jpg"



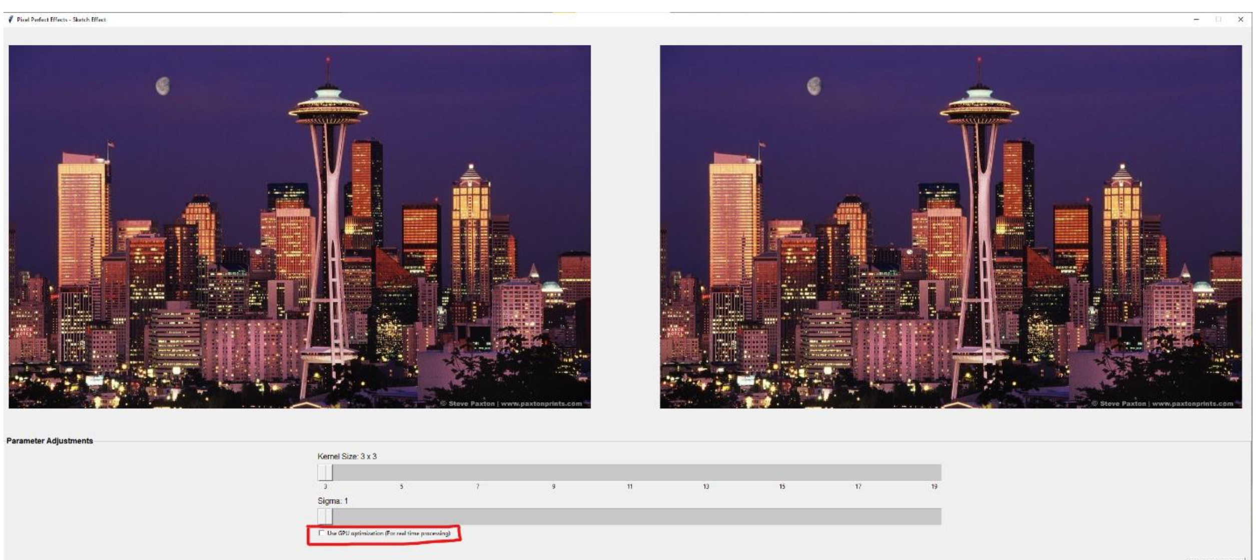
3. Upon selection of the file, you will be greeted by the main canvas window with two sliders. Where the user could change kernel size of the gaussian kernel used for the convolution as well as sigma value used to build the gaussian kernel.



4. This application supports processing the image using CPU and GPU. In order to use the CPU to process the image, first change the parameter sliders to you desired values and then click "Process Image (CPU)" button.

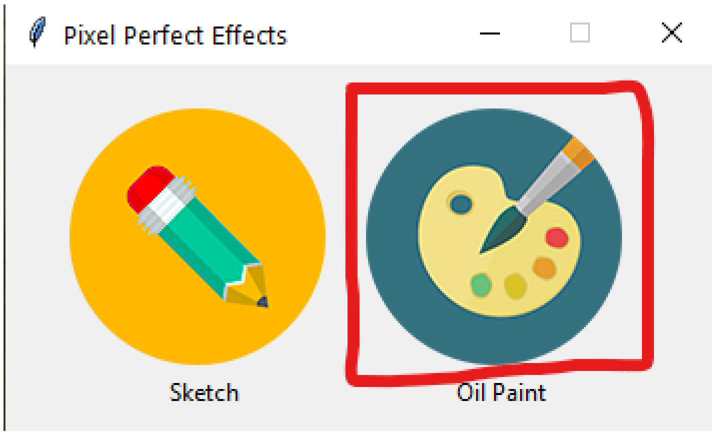


5. However, if you prefer real time image processing results, I recommend turning on "Use GPU optimization (For real time processing)" check box. This will allow numba interpreter to optimize the code and send matrix related computations to your GPU (CUDA cores if NVIDIA GPU, OpenCL if AMD GPU) and yield much faster results compared to processing with CPU. Once the checkbox is checked, you will be able to see changes to the resulting image in real time. Also, if you want to use the CPU again to process the image. Just uncheck the check box. This will allow you to use "Process Image (CPU)" button to process the image once all the sliders are set to your desired values.

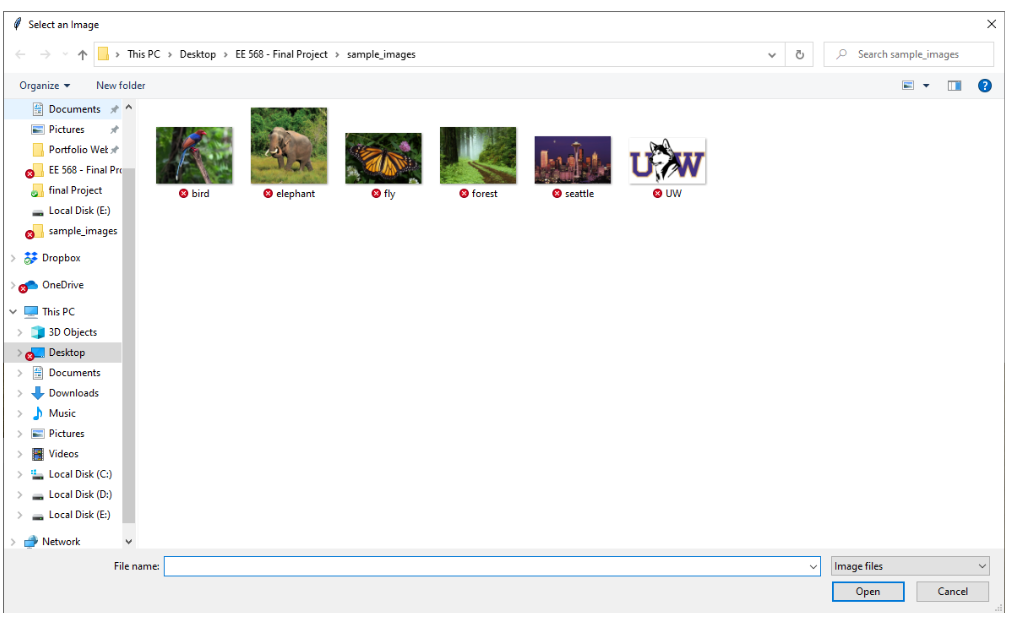


2.3 Applying Oil Paint effect

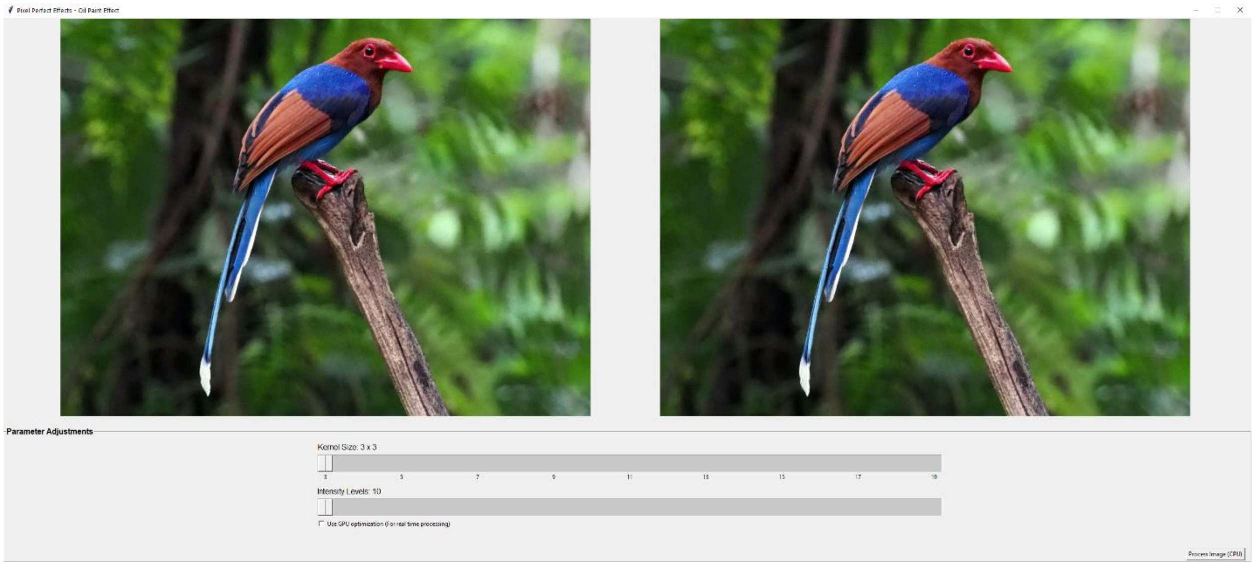
1. On the first interface window click on Oil Paint button



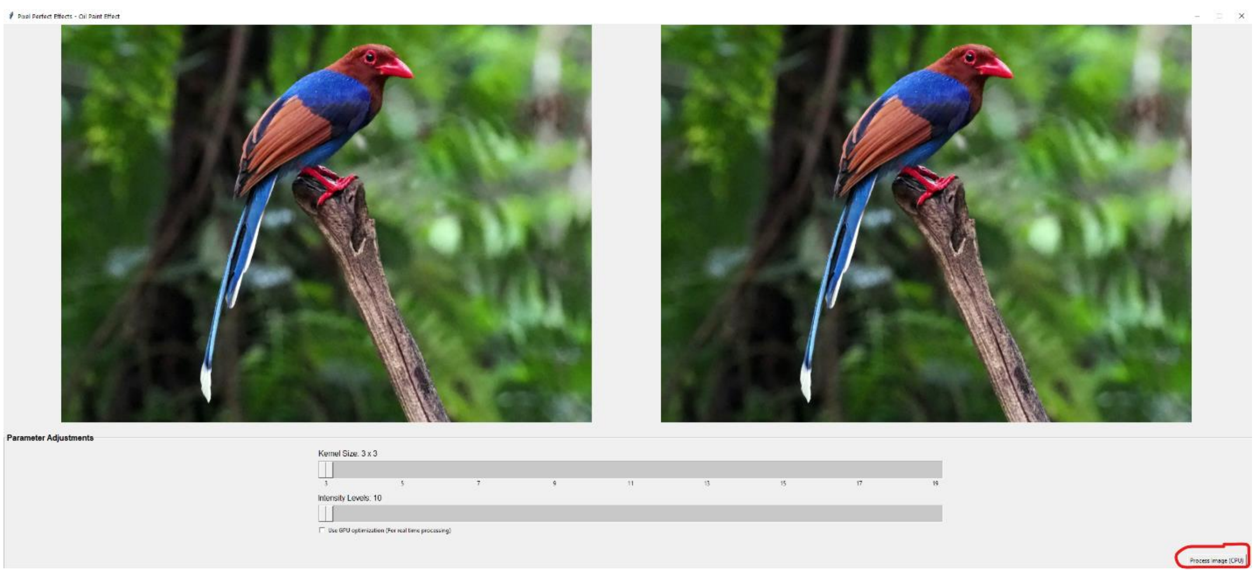
2. Once you clicked on the Oil Paint button, you will be greeted with a file selector pointing to "sample_images" directory within the project folder which contains six sample images that you could use to apply different effects (But feel free try any other image files you would like to apply effects on, the application is smart enough to filter out image file types supported by OpenCV). For the purpose of demonstration I'm using "bird.jpg"



3. Upon selection of the file, you will be greeted by the main canvas window with two sliders. Where the user could change kernel size of the kernel used for the convolution as well as intensity levels value.

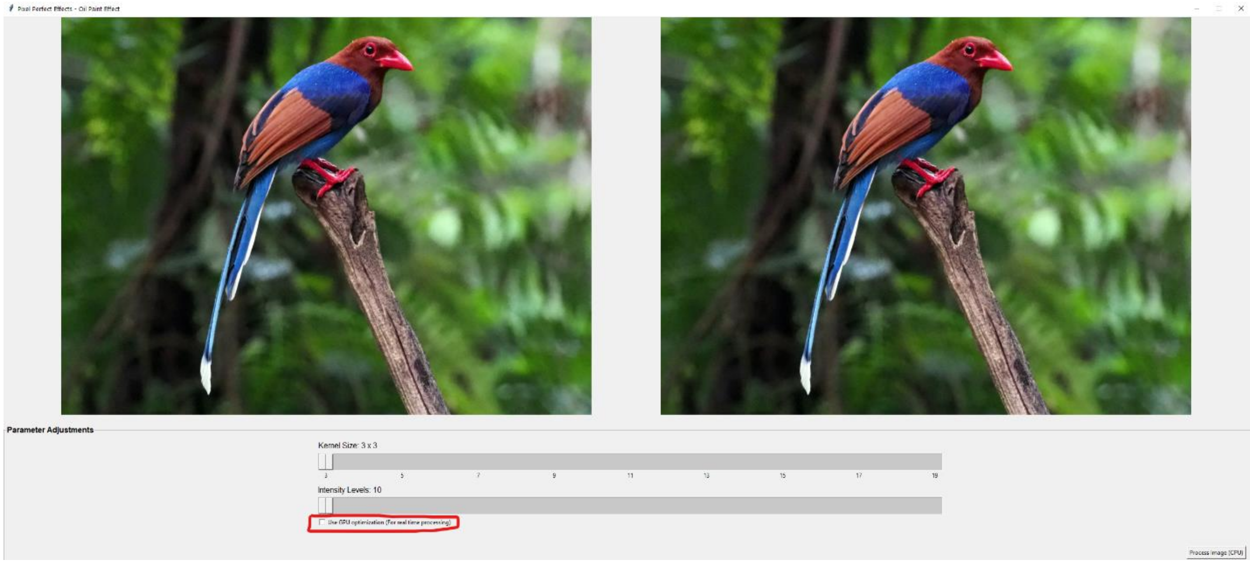


4. This application supports processing the image using CPU and GPU. In order to use the CPU to process the image first change the parameter sliders to you desired values and then click “Process Image (CPU)” button.



5. However, if you prefer real time image processing results, I recommend turning on “Use GPU optimization (For real time processing)” check box. This will allow numba interpreter to optimize the code and send matrix related computations to your GPU (CUDA cores if NVIDIA GPU, OpenCL if AMD GPU) and yield much faster results compared to processing with CPU. Once the checkbox is checked, you will be able to see

changes to the resulting image in real time. Also, if you want to use the CPU again to process the image. Just uncheck the check box. This will allow you to use "Process Image (CPU)" button to process the image once all the sliders are set to your desired values.



3. Implementation descriptions

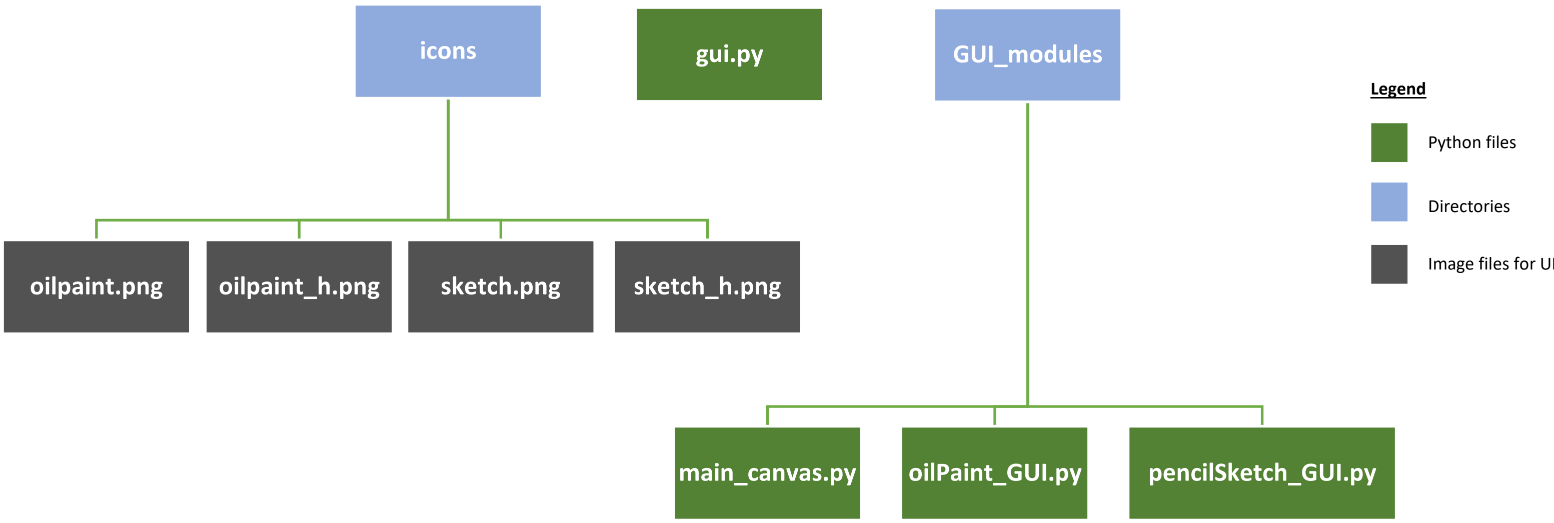
The entry point for this application is "main.py" where it creates an instance of GUI class (resides in gui.py) and run the "draw_main_window()" method within the GUI class that draws the main screen of the Pixel Perfect Effects application. We will discuss GUI implementation in-depth in the next sub-section

```python
from gui import *


def main():
    GUI_ = GUI()
    GUI_.draw_main_window()


if __name__ == "__main__":
    main()
```

Initializing and drawing the main window component

main.py file

3.1 GUI (Front-end) Implementation



As we discussed in the introduction, code for the GUI consists of 3 modules, one component (gui.py) and Icons directory which hold UI elements used for this project

3.1.1 Icons directory

Icons directory consists of 2 versions of the same icon. Where the [effectName].png consists of the original image that we see on the main window of the Pixel Perfect Effects which is visible when user is not hovering the mouse on top of it. While [effectName]_h.png images consist of brightened version of the original image that will draw on to the exact same position of the original image when the user hovers over the original image. This was done to give user a highlighting effect when hovering over buttons.

3.1.2 GUI component

This component is implemented under gui.py python file. The sole purpose of this module is to implement the first window of the Pixel Perfect Effects application. Where it calculates the size of the window relative to the screen size of the user as well as placing it above the windows task bar by calculating its position relative to the user screen. Also, this module implements button highlighting effects as well click events to each button on the UI. In addition, this module handles the creation of tkinter file chooser upon clicking on the effect button and initializing corresponding canvas screen along with the path to the image user has chosen.

```
from tkinter import PhotoImage, filedialog
from win32api import GetMonitorInfo, MonitorFromPoint
from GUI_modules.pencilSketch_GUI import *
from GUI_modules.oilPaint_GUI import *
import ctypes #to get screen resolution (This is exclusive to windows)

class GUI:
    def __init__(self):
        self.mainWindow = None
        self.main_window_width = 30 + 30 + (128 * 2) + (20 * 2)
        self.main_window_height = 128 + 15 + (20 * 2)
```

Calculating height and width required to draw buttons and labels within the main window

```
        self.screen_size_work = None
        self.sketch_icon_label = None
        self.sketch_text_label = None
        self.oilPaint_icon_label = None
        self.oilPaint_text_label = None


    def draw_main_window(self):
        self.screen_size_work = self.getWorking_area()
        self.mainWindow = tk.Tk()
        # turn off main window resize
        self.mainWindow.resizable(False,False)
        # set window name
        self.mainWindow.title("Pixel Perfect Effects")
        self.mainWindow.geometry(f"{(int)(self.main_window_width)}x{(int)(self.main_window_height)}+{(int)((self.screen_size_work[2]/2) - self.main_window_width/2)}+{(int)(self.screen_size_work[3] - self.main_window_height*1.2)}")
```

Drawing the main window with calculated size and width

```


        # Creating buttons to click
        # Sketch button label
        img_sketch = PhotoImage(file='icons\\sketch.png')
        self.sketch_icon_label = tk.Label(self.mainWindow, image=img_sketch, cursor="hand2")
        self.sketch_icon_label.place(x=30, y=20)
        self.sketch_text_label = tk.Label(self.mainWindow, text="Sketch", justify=tk.CENTER, cursor="hand2")
        self.sketch_text_label.place(x= (30 + 128)/2, y= (20 + 128 + 5))
        self.sketch_icon_label.bind("<Enter>", lambda event, arg="sketch", obj=self.sketch_icon_label: self.on_enter(event, arg, obj))
        self.sketch_icon_label.bind("<Leave>", lambda event, arg="sketch", obj=self.sketch_icon_label: self.on_leave(event, arg, obj))
        self.sketch_icon_label.bind("<Button-1>", lambda event, arg="sketch": self.on_click(event, arg))
        self.sketch_text_label.bind("<Enter>", lambda event, arg="sketch", obj=self.sketch_icon_label: self.on_enter(event, arg, obj))
        self.sketch_text_label.bind("<Leave>", lambda event, arg="sketch", obj=self.sketch_icon_label: self.on_leave(event, arg, obj))
```

Drawing Pencil Sketch button and label

```

        # oilpaint button label
        img_oil = PhotoImage(file='icons\\oilpaint.png')
        self.oilPaint_icon_label = tk.Label(self.mainWindow, image=img_oil, cursor="hand2")
        self.oilPaint_icon_label.place(x= 30 + 128 + 20, y=20)
        self.oilPaint_text_label = tk.Label(self.mainWindow, text="Oil Paint", justify=tk.CENTER, cursor="hand2")
        self.oilPaint_text_label.place(x= (30 + 128) + 128/2, y= (20 + 128 + 5))
        self.oilPaint_icon_label.bind("<Enter>", lambda event, arg="oilpaint", obj=self.oilPaint_icon_label: self.on_enter(event, arg, obj))
        self.oilPaint_icon_label.bind("<Leave>", lambda event, arg="oilpaint", obj=self.oilPaint_icon_label: self.on_leave(event, arg, obj))
        self.oilPaint_icon_label.bind("<Button-1>", lambda event, arg="oilpaint": self.on_click(event, arg))
        self.oilPaint_text_label.bind("<Enter>", lambda event, arg="oilpaint", obj=self.oilPaint_icon_label: self.on_enter(event, arg, obj))
        self.oilPaint_text_label.bind("<Leave>", lambda event, arg="oilpaint", obj=self.oilPaint_icon_label: self.on_leave(event, arg, obj))
```

Drawing Oil paint button and label

```


        self.mainWindow.mainloop()



    def getScreenSize(self):
        user32 = ctypes.windll.user32
```

Getting monitor resolution

```python
        screensize = user32.GetSystemMetrics(0), user32.GetSystemMetrics(1)
        return screensize

    def getWorking_area(self):
        monitor_info = GetMonitorInfo(MonitorFromPoint((0, 0)))
        return monitor_info.get("Work")

    def on_enter(self, event, icon_name, obj):
        img = PhotoImage(file=f'./icons/{icon_name}_h.png')
        obj.config(image= img)
        obj.image = img

    def on_leave(self, event, icon_name, obj):
        img_sketch = PhotoImage(file=f'./icons/{icon_name}.png')
        obj.config(image=img_sketch)
        obj.image = img_sketch

   def on_click(self, event, effectName):
    filename = filedialog.askopenfilename(initialdir="./sample_images", title="Select an
Image",filetypes=[("Image files",
                  "*.bmp; *.dib; *.jpeg; *.jpg; *.jp2; *.png; *.webp; *.pbm; "
                  "*.pgm; *.ppm; *.pxm; *.pnm; *.sr; *.ras; *.hdr; *.pic")])
    if filename: # Make sure file is selected
       if effectName.lower() == "sketch":
        pencilSketch_GUI(self.mainWindow,filename,self.screen_size_work[2], self.screen_size_work[3], "Sketch")
       elif effectName.lower() == "oilpaint":
        oilPaint_GUI(self.mainWindow,filename,self.screen_size_work[2], self.screen_size_work[3], "Oil Paint")
```

Getting working area resolution. Excluding task bar

Logic to change the button picture to highlighted picture on mouse entering the region

Logic to change the button picture to original picture on mouse leaving the region

Logic to handle button click in the UI and to pop up the file selector window

gui.py file

3.1.3 Main Canvas module

This is the module responsible for drawing the canvas window. This module uses screen resolution/size of the user screen to calculate the size of the window as well as sizes of the original image and resulting image canvases and drawing the images on the canvases. In addition, this module is responsible of creating a place holder for slider/parameter area. Such that we could create an object of this module within the Pencil Sketch and Oil Paint GUI module to extend its functionality. Canvas section of the GUI is written this way since it will allow me to add new effects to this application in future without re-writing the same code. Moreover, this module resides in “main_canvas.py” python file

3.1.4 Pencil sketch GUI module

This the module responsible for executing both of pencil sketch algorithms (CPU based and GPU based codes) and drawing the resulting image generated by the pencil sketch algorithm to the resulting canvas of the main canvas module. In addition to above task, this module draws Kernel Size and Sigma sliders within the parameter adjustments place holder area, as well as drawing “Process Image (CPU)” button along with “Use GPU optimization (For real time processing)” check box. Also, this module handles click event for the button, changing label values when moving the slider and executing the real time algorithm while moving the sliders when check box is checked. This module resides in “pencilSketch_GUI.py” python file.

```
from GUI_modules.main_canvas import *
from tkinter.font import BOLD, Font
from algorithms.oilpaint import *
from algorithms.oilPaint_GPU import *

class oilPaint_GUI:
    def __init__(self, root, imagePath, work_width, work_height, effectName):
        self.mainCanvas = canvas_GUI(root, imagePath, work_width, work_height, effectName)
        self.slider = None
        self.past = 3
        self.kernelLabel = None
        self.sigmaLabel = None
        self.sigmaSlider = None
        self.processButton = None
        self.GPU_OPTIMIZE = None
        self.IS_GPU_OPTIMIZED = False
        self.draw_GUI()
        # priming the code, this way numba can build code cache and during the next run things willmuch faster !
        oil_paint(np.array(self.mainCanvas.PIL_image), 3, 10)

    def draw_GUI(self):
        self.mainCanvas.draw_window()
        self.mainCanvas.root.update()

        adjust_height = self.mainCanvas.adjustment_area.winfo_height()
        adjust_width = self.mainCanvas.adjustment_area.winfo_width()

        # draw slider for kernel size
        self.slider = tk.Scale(self.mainCanvas.adjustment_area, length=adjust_width//2, width=adjust_height//8, orient=tk.HORIZONTAL, from_=3, to=19,  command=self.fix, tickinterval=2, showvalue=0)
        self.slider.place(x=adjust_width//4, y=adjust_height//8)
        self.kernelLabel = tk.Label(self.mainCanvas.adjustment_area, text="Kernel Size: 3 x 3", font=Font(self.mainCanvas.adjustment_area, size=12))
        self.kernelLabel.place(x=adjust_width // 4, y=adjust_height // 8 - 25)
        self.mainCanvas.root.update()

        # Sigma label
        self.sigmaLabel = tk.Label(self.mainCanvas.adjustment_area, text="Intensity Levels: 10", font=Font(self.mainCanvas.adjustment_area, size=12))
        self.sigmaLabel.place(x=adjust_width // 4, y=adjust_height // 8 + self.slider.winfo_height() + 5)
        self.mainCanvas.root.update()
        # Sigma label
        self.sigmaSlider = tk.Scale(self.mainCanvas.adjustment_area, length=adjust_width // 2, width=adjust_height // 8, orient=tk.HORIZONTAL, from_=10, to=255, showvalue=0, command=self.sigma_slide)
        self.sigmaSlider.place(x=adjust_width // 4, y=adjust_height // 8 + self.slider.winfo_height() + self.sigmaLabel.winfo_height() + 5)
        self.mainCanvas.root.update()

        # GPU OPTIMIZE Button
```

Drawing slider and label for kernel size

Drawing slider and label for Intensity Levels

```python
        self.GPU_OPTIMIZE = tk.Checkbutton(self.mainCanvas.adjustment_area, text="Use GPU optimization (For real time processing)", command=self.GPU_toggle)

        self.GPU_OPTIMIZE.place(x=adjust_width // 4,y=adjust_height // 8 + self.slider.winfo_height() + self.sigmaLabel.winfo_height() + self.sigmaSlider.winfo_height() + 5)
        self.GPU_OPTIMIZE.deselect()

        # Process button
        self.processButton = tk.Button(self.mainCanvas.adjustment_area, text="Process Image (CPU)", command=self.processImage)
        self.processButton.place(x=adjust_width, y=adjust_height)
        self.mainCanvas.root.update()
        self.processButton.place(x=adjust_width - self.processButton.winfo_width() - 15, y=adjust_height - self.processButton.winfo_height() - 25)


    def fix(self, n):
        n = int(n)
        if not n % 2:
            self.slider.set(n + 1 if n > self.past else n - 1)
            self.past = self.slider.get()
            self.kernelLabel['text'] = f"Kernel Size: {self.past} x {self.past}"
            self.real_time_process()


    def sigma_slide(self, n):
        self.sigmaLabel['text'] = f"Intensity Levels: {n}"
        self.real_time_process()

    def GPU_toggle(self):
        if(self.IS_GPU_OPTIMIZED):
            self.IS_GPU_OPTIMIZED = False
        else:
            self.IS_GPU_OPTIMIZED = True
        self.real_time_process()

    def processImage(self):
        if (not self.IS_GPU_OPTIMIZED):
            oilPaint_ = oilPaint(self.mainCanvas.ImagePath, np.array(self.mainCanvas.PIL_image))
            results = oilPaint_.perform_oilPaint((int)(self.slider.get()), (int)(self.sigmaSlider.get()))
            PIL_Image = Image.fromarray(results)
            photo_result = ImageTk.PhotoImage(PIL_Image)
            self.mainCanvas.render_result_image(photo_result)

    def real_time_process(self):
        if (self.IS_GPU_OPTIMIZED):
            results = oil_paint(np.array(self.mainCanvas.PIL_image), (int)(self.slider.get()), (int)(self.sigmaSlider.get()))
            PIL_Image = Image.fromarray(results)
            photo_result = ImageTk.PhotoImage(PIL_Image)
            self.mainCanvas.render_result_image(photo_result)
```

Drawing check box and label for GPU optimization

Drawing button for process image using CPU

Slider event for kernel size slider (increment by odd number and print the kernel size to label)

Slider event for sigma and print current sigma value to the label

Logic for GPU optimization check box

Process image using CPU logic

Process image using GPU logic

pencilSketch_GUI.py file

3.1.5 Oil Paint GUI module

Functionality of this module is very similar to the pencil sketch GUI module. This module is responsible for executing both of oil paint algorithms (CPU based and GPU based codes) and drawing the resulting image generated by the oil paint algorithm to the resulting canvas of the main canvas module. In addition to above task, this module draws Kernel Size and Intensity levels sliders within the parameter adjustments place holder area, as well as drawing “Process Image (CPU)” button along with “Use GPU optimization (For real time processing)” check box. Also, this module handles click event for the button, changing label values when moving the slider and executing the real time algorithm while moving the sliders when check box is checked. This module resides in “oilPaint_GUI.py” python file.

```
from GUI_modules.main_canvas import *
from tkinter.font import BOLD, Font
from algorithms.oilpaint import *
from algorithms.oilPaint_GPU import *

class oilPaint_GUI:
    def __init__(self, root, imagePath, work_width, work_height, effectName):
        self.mainCanvas = canvas_GUI(root, imagePath, work_width, work_height, effectName)
        self.slider = None
        self.past = 3
        self.kernelLabel = None
        self.sigmaLabel = None
        self.sigmaSlider = None
        self.processButton = None
        self.GPU_OPTIMIZE = None
        self.IS_GPU_OPTIMIZED = False
        self.draw_GUI()
        # priming the code, this way numba can build code cache and during the next run things willmuch faster !
        oil_paint(np.array(self.mainCanvas.PIL_image), 3, 10)

    def draw_GUI(self):
        self.mainCanvas.draw_window()
        self.mainCanvas.root.update()

        adjust_height = self.mainCanvas.adjustment_area.winfo_height()
        adjust_width = self.mainCanvas.adjustment_area.winfo_width()

        # draw slider for kernel size
        self.slider = tk.Scale(self.mainCanvas.adjustment_area, length=adjust_width//2, width=adjust_height//8, orient=tk.HORIZONTAL, from_=3, to=19,  command=self.fix, tickinterval=2, showvalue=0)
        self.slider.place(x=adjust_width//4, y=adjust_height//8)
        self.kernelLabel = tk.Label(self.mainCanvas.adjustment_area, text="Kernel Size: 3 x 3", font=Font(self.mainCanvas.adjustment_area, size=12))
        self.kernelLabel.place(x=adjust_width // 4, y=adjust_height // 8 - 25)
        self.mainCanvas.root.update()

        # Sigma label
        self.sigmaLabel = tk.Label(self.mainCanvas.adjustment_area, text="Intensity Levels: 10", font=Font(self.mainCanvas.adjustment_area, size=12))
        self.sigmaLabel.place(x=adjust_width // 4, y=adjust_height // 8 + self.slider.winfo_height() + 5)
        self.mainCanvas.root.update()
        # Sigma label
        self.sigmaSlider = tk.Scale(self.mainCanvas.adjustment_area, length=adjust_width // 2, width=adjust_height // 8, orient=tk.HORIZONTAL, from_=10, to=255, showvalue=0, command=self.sigma_slide)
        self.sigmaSlider.place(x=adjust_width // 4, y=adjust_height // 8 + self.slider.winfo_height() + self.sigmaLabel.winfo_height() + 5)
        self.mainCanvas.root.update()
```

Drawing slider and label for kernel size

Drawing slider and label for Intensity Levels

```python

        # GPU OPTIMIZE Button
        self.GPU_OPTIMIZE = tk.Checkbutton(self.mainCanvas.adjustment_area, text="Use GPU optimization (For real time processing)", command=self.GPU_toggle)

        self.GPU_OPTIMIZE.place(x=adjust_width // 4,y=adjust_height // 8 + self.slider.winfo_height() + self.sigmaLabel.winfo_height() + self.sigmaSlider.winfo_height() + 5)
        self.GPU_OPTIMIZE.deselect()
```

Drawing check box and label for GPU optimization

```python

        # Process button
        self.processButton = tk.Button(self.mainCanvas.adjustment_area, text="Process Image (CPU)", command=self.processImage)
        self.processButton.place(x=adjust_width, y=adjust_height)
        self.mainCanvas.root.update()
        self.processButton.place(x=adjust_width - self.processButton.winfo_width() - 15, y=adjust_height - self.processButton.winfo_height() - 25)

```

Drawing button for process image using CPU

```python
    def fix(self, n):
        n = int(n)
        if not n % 2:
            self.slider.set(n + 1 if n > self.past else n - 1)
            self.past = self.slider.get()
            self.kernelLabel['text'] = f"Kernel Size: {self.past} x {self.past}"
            self.real_time_process()

```

Slider event for kernel size slider (increment by odd number and print the kernel size to label)

```python
    def sigma_slide(self, n):
        self.sigmaLabel['text'] = f"Intensity Levels: {n}"
        self.real_time_process()

```

Slider event for Intensity levels and print current intensity to the label

```python
    def GPU_toggle(self):
        if(self.IS_GPU_OPTIMIZED):
            self.IS_GPU_OPTIMIZED = False
        else:
            self.IS_GPU_OPTIMIZED = True
        self.real_time_process()

```

Logic for GPU optimization check box

```python
    def processImage(self):
        if (not self.IS_GPU_OPTIMIZED):
            oilPaint_ = oilPaint(self.mainCanvas.ImagePath, np.array(self.mainCanvas.PIL_image))
            results = oilPaint_.perform_oilPaint((int)(self.slider.get()), (int)(self.sigmaSlider.get()))
            PIL_Image = Image.fromarray(results)
            photo_result = ImageTk.PhotoImage(PIL_Image)
            self.mainCanvas.render_result_image(photo_result)

```

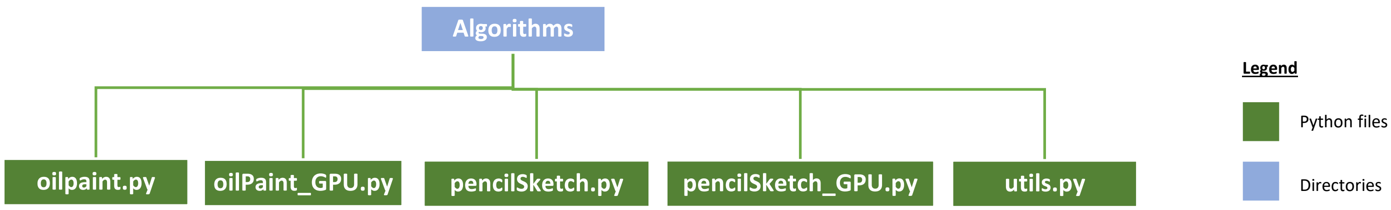
Process image using CPU logic

```python
    def real_time_process(self):
        if (self.IS_GPU_OPTIMIZED):
            results = oil_paint(np.array(self.mainCanvas.PIL_image), (int)(self.slider.get()), (int)(self.sigmaSlider.get()))
            PIL_Image = Image.fromarray(results)
            photo_result = ImageTk.PhotoImage(PIL_Image)
            self.mainCanvas.render_result_image(photo_result)
```

Process image using GPU logic

oilPaint_GUI.py file

3.2 Algorithm (Back-end) Implementation



3.2.1 Utils module

This module consists of utility functions that have been shared between pencil sketch and oil paint algorithms. This module implements following functions

- Loading an image using OpenCV imread function
- Displaying image using OpenCV imshow function (This function is used for debugging purposes when these algorithms were being developed)
- Converting BGR image into grayscale space using cvtColor function
- Creating an image negative given an grayscale (or one channel image) by subtracting each pixel by 255 (since images loaded into the program can represent up 255 intensity values)

This module is implemented under “utils.py” python file

```
import cv2

class utils:
    def __init__(self, imagePath):
        self.Imagepath = imagePath

    def load_image_data(self):
        return cv2.imread(self.Imagepath, cv2.IMREAD_COLOR)

    def display_Image(self, ImageName, ImageData):
        cv2.imshow(ImageName, ImageData)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    def convert_to_grayScale(self, ImageData):
        return cv2.cvtColor(ImageData, cv2.COLOR_BGR2GRAY)

    def image_negative(self, ImageData):
        return 255 - ImageData
```

Loading image using OpenCV

Displaying image using OpenCV

Converting BGR image to Grayscale

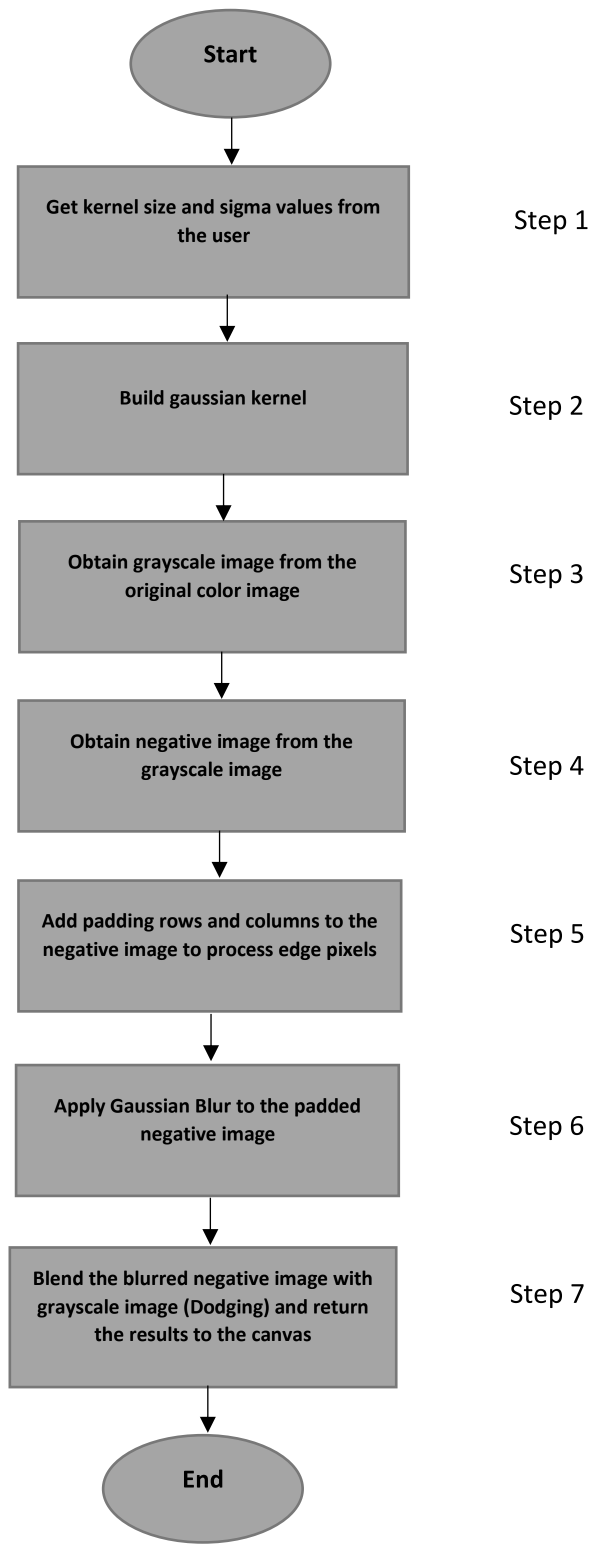
Convert grayscale image to its negative

utils.py file

3.2.2 Pencil sketch algorithm

As mentioned in the introduction section, code for the pencil sketch algorithm consists of two versions. First version is developed to run utilize CPU while the second version is developed to optimize using Numba JIT to run on GPU.

Firstly, let us discuss about the algorithm for the pencil sketch using following flow chart:



Step 1: Is accomplished by getting the user selected values through the sliders (for kernel size and sigma) through the GUI

Step 2: In order to achieve this step, I have used following equation to build a gaussian kernel from scratch given kernel size and sigma (standard deviation). However, this can be done using OpenCV built in functions. But to practice building gaussian kernels from scratch I wanted to implement this function myself.

$$G_\sigma = \frac{1}{2\pi\sigma^2}e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Step 3: In order to obtain grayscale image from a color image I have used OpenCV built in function. However, this can be easily done by stripping out 2 channels from each pixel of the image.

Step 4: Once grayscale image is obtained, I have subtracted current pixel intensity of every pixel in the image from 255 to obtain the negative of the image.

Step 5: In this algorithm, instead of skipping edge pixels it also processes edge pixels. This is achieved by adding padding columns and rows to the negative image. To calculate number of padding rows and columns we could use following formula:

 # padding rows and columns = floor (kernel size / 2)

In this case kernel size can be height of the kernel or the width of the kernel. Since in image processing we are using **nxn** kernels to apply 2D convolutions, where n is an odd number such that the kernel has a mid/center point.

Step 6: Once creation of the gaussian kernel is done, we can iterate through each and every pixel (excluding padded pixels) of the negative image and obtain image patches of the size of the kernel (e.g if kernel size is 3x3 we obtain 3x3 image patches where center pixel is the pixel we are currently processing). Once the image patch around the processing pixel is selected. We can multiply obtained image patch with the kernel and then get the sum of all the pixels values within the resulting matrix. Then we divide it by the sum of each and every value of the kernel. Which yields us the resulting value for the processing pixel. This can be written as follows:

Resulting value of the processing pixel = sum((selected image patch * Gaussian kernel)) / (sum of gaussian kernel elements)

Step 7: Once gaussian blur is applied we can blend the blurred negative image with the grayscale image we obtained from step 3. This technique is known as “Dodge” technique used in traditional photography. This is basically dividing the grayscale channel of an image pixel by the inverse mask of the pixel value. However, we must make sure that the resulting pixel value will still stay within 0 – 255 range as well as we are not dividing by zero.

3.2.3 Pencil sketch code (Utilizing CPU)

Pencil sketch code utilizing the CPU is implemented under “pencilSketch.py” python file

Obtaining Grayscale image

```python
import numpy as np
from algorithms.utils import *


class pencilSketch:
    def __init__(self, imageData, imagepath):
        self.originalImage = imageData
        self.utils_ = utils(imagepath)
        self.grayScaleImage = self.utils_.convert_to_grayScale(self.originalImage)
        self.negativeImage = self.utils_.image_negative(self.grayScaleImage)
```

Obtaining negative image

Generating Gaussian Kernel

```python
    def dnorm(self, x, mu, sd):
        return 1 / (np.sqrt(2 * np.pi) * sd) * np.e ** (-np.power((x - mu) / sd, 2) / 2)

    def gaussian_kernel(self, kernel_size, sigma=1):
        kernel_1D = np.linspace(-(kernel_size // 2), kernel_size // 2, kernel_size)
        for i in range(kernel_size):
            kernel_1D[i] = self.dnorm(kernel_1D[i], 0, sigma)
        kernel_2D = np.outer(kernel_1D.T, kernel_1D.T)

        kernel_2D *= 1.0 / kernel_2D.max()

        return kernel_2D

    '''''
    This function only accepts kernels with a mid point i.e 3x3, 5x5, 7x7, etc
    '''
    def gaussian_blur(self, kernel_size, sigma, imageData):
        kernel = self.gaussian_kernel(kernel_size, sigma)
```

Padding the negative image

```python
        # add padding to the image
        padding_pixels = kernel_size // 2
        image_padded = np.pad(imageData, ((padding_pixels, padding_pixels), (padding_pixels, padding_pixels)), 'constant')
        padded_h, padded_w = np.shape(image_padded)
```

Applying Gaussian blur

```python
        for row in range(padding_pixels, padded_h - padding_pixels):
            for col in range(padding_pixels, padded_w - padding_pixels):
                newPixel_val = np.sum(image_padded[row - padding_pixels:row + padding_pixels + 1, col - padding_pixels:col + padding_pixels + 1] * kernel)/ np.sum(kernel)
                if(newPixel_val > 255):
                    newPixel_val = 255
                image_padded[row][col] = newPixel_val

        #finally we need to remove the extra padding and return the image data
        return image_padded[padding_pixels:padded_h - padding_pixels, padding_pixels:padded_w - padding_pixels]

    def pencil_sketch(self, kernel_size, sigma):
```

```python
        gaussian_negative = self.gaussian_blur(kernel_size, sigma, self.negativeImage)
        return self.dodge(self.grayScaleImage, gaussian_negative)


    def dodge(self, grayscale, negative):
        height, width = np.shape(grayscale)
        result = np.zeros((height, width), np.uint8)

        for row in range(height):
            for col in range(width):
                if negative[row, col] == 255:
                    result[row, col] = 255
                else:
                    pix = (grayscale[row, col] << 8) / (255 - negative[row, col])
                    if pix > 255:
                        pix = 255
                    result[row, col] = pix

        return result
```

Applying Gaussian blur

Dodging grayscale and blurred negative images

Dodging grayscale and blurred negative images

pencilSketch.py file

3.2.4 Pencil sketch code (Utilizing GPU)

Pencil sketch code utilizing the GPU is implemented under “pencilSketch_GPU.py” python file

```python
import numpy as np
from algorithms.utils import *
from numba import jit, cuda

@jit(nopython=True)
def dnorm(x, mu, sd):
    return 1 / (np.sqrt(2 * np.pi) * sd) * np.e ** (-np.power((x - mu) / sd, 2) / 2)

@jit(nopython=True)
def gaussian_kernel(kernel_size, sigma=1):
    kernel_1D = np.linspace(-(kernel_size // 2), kernel_size // 2, kernel_size)
    for i in range(kernel_size):
        kernel_1D[i] = dnorm(kernel_1D[i], 0, sigma)
    kernel_2D = np.outer(kernel_1D.T, kernel_1D.T)

    kernel_2D *= 1.0 / kernel_2D.max()

    return kernel_2D

@jit(nopython=True)
def gaussian_blur(kernel_size, sigma, padding_pixels, image_padded):
    kernel = gaussian_kernel(kernel_size, sigma)
    padded_h, padded_w = np.shape(image_padded)

    for row in range(padding_pixels, padded_h - padding_pixels):
        for col in range(padding_pixels, padded_w - padding_pixels):
            newPixel_val = np.sum(image_padded[row - padding_pixels:row + padding_pixels + 1,
                                  col - padding_pixels:col + padding_pixels + 1] * kernel) / np.sum(kernel)
            if (newPixel_val > 255):
                newPixel_val = 255
            image_padded[row][col] = newPixel_val

    # finally we need to remove the extra padding and return the image data
    return image_padded[padding_pixels:padded_h - padding_pixels, padding_pixels:padded_w - padding_pixels]
```

Using Numba JIT instead of python interpreter to optimize the code to run on GPU

Generating Gaussian Kernel

Generating Gaussian Kernel

Applying Gaussian blur

```python

def pencil_sketch(imagePath, ImageData,kernel_size, sigma):
    utils_ = utils(imagePath)
    originalImage = ImageData
    grayScaleImage = utils_.convert_to_grayScale(originalImage)
    negativeImage = utils_.image_negative(grayScaleImage)
    # add padding to the image
    padding_pixels = kernel_size // 2
    image_padded = np.pad(negativeImage, ((padding_pixels, padding_pixels), (padding_pixels, padding_pixels)), 'constant')

    gaussian_negative = gaussian_blur(kernel_size, sigma, padding_pixels, image_padded)
    return dodge(grayScaleImage, gaussian_negative)

@jit(nopython=True)
def dodge(grayscale, negative):
    # determine the shape of the input image
    height, width = np.shape(grayscale)

    # prepare output argument with same size as image
    blend = np.zeros((height, width), np.uint8)

    for row in range(height):
        for col in range(width):
            # do for every pixel
            if negative[row, col] == 255:
                # avoid division by zero
                blend[row, col] = 255
            else:
                # shift image pixel value by 8 bits
                # divide by the inverse of the mask
                tmp = (grayscale[row, col] << 8) / (255 - negative[row, col])

                # make sure resulting value stays within bounds
                if tmp > 255:
                    tmp = 255
                blend[row, col] = tmp

    return blend
```

Obtaining Grayscale image

Obtaining negative image

Padding the negative image

Applying Gaussian blur

Dodging grayscale and blurred negative images

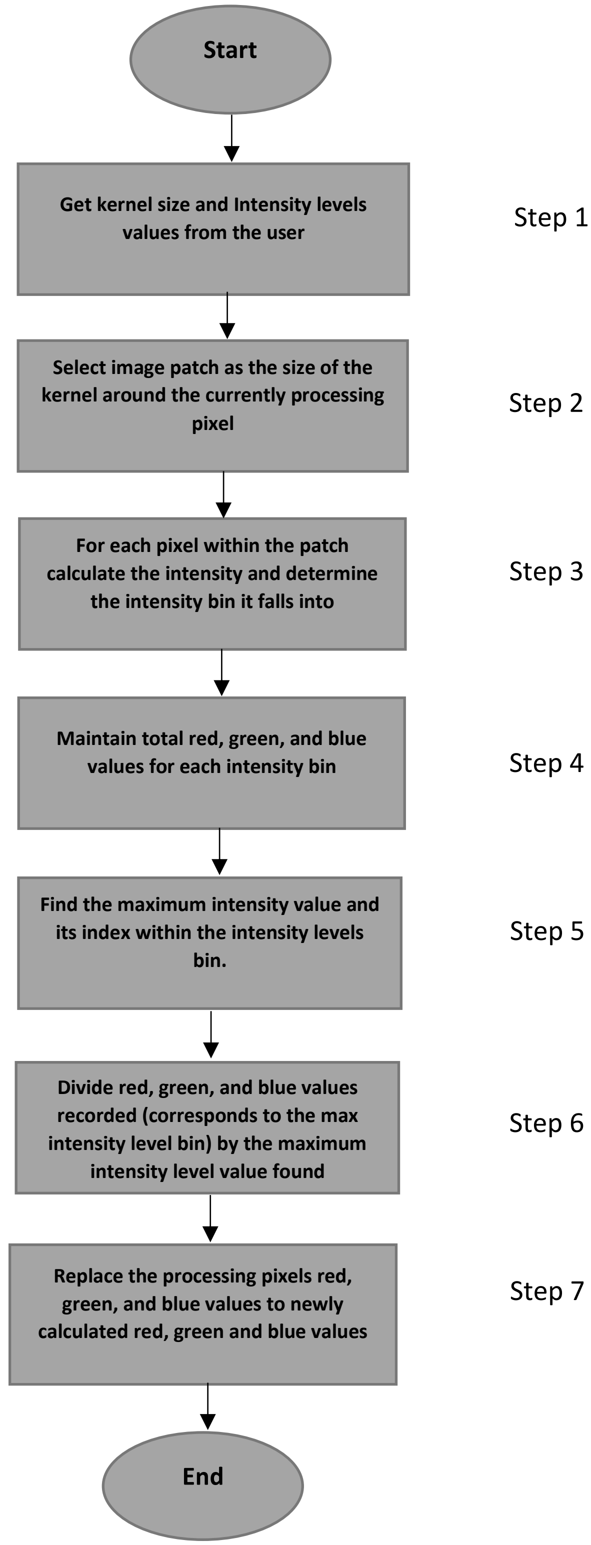
Dodging grayscale and blurred negative images

pencilSketch_GPU.py file

3.2.5 Oil paint algorithm

As mentioned in the introduction section, code for the oil paint algorithm consists of two versions. First version is developed to run utilize CPU while the second version is developed to optimize using Numba JIT to run on GPU.

Firstly, let us discuss about the algorithm for the oil paint using following flow chart:



Start
Get kernel size and Intensity levels values from the user
Step 1
Select image patch as the size of the kernel around the currently processing pixel
Step 2
For each pixel within the patch calculate the intensity and determine the intensity bin it falls into
Step 3
Maintain total red, green, and blue values for each intensity bin
Step 4
Find the maximum intensity value and its index within the intensity levels bin.
Step 5
Divide red, green, and blue values recorded (corresponds to the max intensity level bin) by the maximum intensity level value found
Step 6
Replace the processing pixels red, green, and blue values to newly calculated red, green and blue values
Step 7
End

Step 1: Is accomplished by getting the user selected values through the sliders (for kernel size and Intensity levels value) through the GUI.

Step 2: Iterate through each and every pixel of the negative image and obtain image patches of the size of the kernel (e.g if kernel size is 3x3 we obtain 3x3 image patches where center pixel is the pixel we are currently processing).

Step 3: For each pixel withing the selected patch, calculate its intensity. This can be achieved by using following formula:

Intensity of the current pixel = ((Red intensity + Green Intensity + Blue Intensity) * Intensity Levels) / 255

Once the intensity is calculated, store that in an intensity counter array where intensity value calculated is the index of the array element. When another pixel with the same intensity is found increment the count.

Step 4: Maintain 3 more array to store sum of red, green, and blue values of the pixel that falls into the same intensity level. Where, the index of these arrays is the intensity we have determined in the previous step. If another pixel with the same intensity is found add its red, green, and blue values to the corresponding index of the array.

Step 5: Once processing of each and every pixel within the patch is done. Find the maximum intensity value within the intensity bin (i.e. maximum count value found in the intensity bin) as well as its index (i.e. intensity value calculated) and store them in variables.

Step 6: Divide red, green, and blue values recorded (i.e. the value represented by the index of the maximum count value found in the intensity bin) by the maximum intensity value found in the previous step.

Step 7: Assign the current processing pixel in the image (not the image patch we are processing) with the newly calculated red, green and blue values.

3.2.6 Oil paint code (Utilizing CPU)

Oil paint code for the CPU consists of two versions. The first version, shown below is quite inefficient due to the use of four for loops. Which result in O(n^4) computation time. Since for loops in python are generally slower to execute compared to other languages such as C, C++, C# and Java. I have decided to use Sklearn to library extract image patches using "extract_patches_2d" function which is much efficient than using for loops. Using this method and numpy array reshaping and flattening methods I was able to lower the computational time to O(n^2). Oil paint code utilizing the CPU is implemented under "oilPaint.py" python file

```
from utils import *
import copy

class oilPaint:
    def __init__(self, imagePath):
        self.utils_ = utils(imagePath)
        self.originalImage = utils.load_image_data(self.utils_)

    '''''
    This function only accepts kernels with a mid point i.e 3x3, 5x5, 7x7, etc.
    '''
    def perform_oilPaint(self, kernelSize, intensityLevels):
        original_height, original_width, _ = np.shape(self.originalImage)
        # create an empty array to hold resulting data
        result = copy.deepcopy(self.originalImage)
        # add padding to the image
        padding_pixels = kernelSize // 2


        for row in range(padding_pixels, original_height - padding_pixels):
            for col in range(padding_pixels, original_width - padding_pixels):
                result[row][col] = self.calculate_new_pixel_value(self.originalImage[row - padding_pixels:row + padding_pixels + 1, col - padding_pixels:col
+ padding_pixels + 1], intensityLevels)


        return result

    def calculate_new_pixel_value(self, imagePatch, intensityLevels):
        patch_height, patch_width, _ = np.shape(imagePatch)
        intensityLevels_bin, averageB, averageG, averageR = np.zeros(256), np.zeros(256), np.zeros(256), np.zeros(256)

        for row in range(patch_height):
            for col in range(patch_width):
                curIntensity = (int)((((imagePatch[row][col][0] + imagePatch[row][col][1] + imagePatch[row][col][2])/3) * intensityLevels) / 255)
                print(curIntensity)
                intensityLevels_bin[curIntensity] += 1
                averageB[curIntensity] += imagePatch[row][col][0]
                averageG[curIntensity] += imagePatch[row][col][1]
                averageR[curIntensity] += imagePatch[row][col][2]

        # find the max intensity and its index
        maxIntensity = max(intensityLevels_bin)
        maxIndex =  intensityLevels_bin.argmax()


        # return new value of the pixel
        final_pixel = np.array([averageB[maxIndex]/maxIntensity, averageG[maxIndex]/maxIntensity, averageR[maxIndex]/maxIntensity])
        # clipping values
        final_pixel[final_pixel > 255] = 255

        return final_pixel
```

O(n^4) run time complexity. This makes algorithm perform slow!

Extracting image patches from the image

Numpy arrays to maintain intensity bins and red, green, and blue values for each intensity bin

Calculation of intensity for current pixel within the image patch

Storing intensity value count for the corresponding intensity value

Maintaining Red, green, and blue values for the calculated intensity value

Recording maximum intensity value count

Recording maximum intensity value

Assigning new intensity value for the processing pixel within the image

First version of the oil paint algorithm

```python
from algorithms.utils import *
import copy
from sklearn.feature_extraction.image import extract_patches_2d

class oilPaint:
    def __init__(self, imagePath, imageData):
        self.utils_ = utils(imagePath)
        self.originalImage = imageData

    '''''
    This function only accepts kernels with a mid point i.e 3x3, 5x5, 7x7, etc.
    '''
    def perform_oilPaint(self, kernelSize, intensityLevels):
        original_height, original_width, _ = np.shape(self.originalImage)
        # create an empty array to hold resulting data
        result = copy.deepcopy(self.originalImage)

        padding = kernelSize // 2

        #extract patches from the image
        patches = extract_patches_2d(self.originalImage, (kernelSize, kernelSize))

        intensity_values = []

        for currPatch in patches:
            intensity_values.append(self.calculate_new_pixel_value(np.array_split(currPatch.flatten(), (kernelSize * kernelSize)), intensityLevels))


        new_pixel_intensities = np.array(intensity_values).flatten().reshape(original_height - (padding * 2), original_width - (padding * 2), 3)

        result[padding: original_height - padding, padding: original_width - padding] = new_pixel_intensities
        return result

    def calculate_new_pixel_value(self, imagePatch, intensityLevels):
        intensityLevels_bin, averageB, averageG, averageR = np.zeros(256), np.zeros(256), np.zeros(256), np.zeros(256)

        for pxl in range(len(imagePatch)):
            curIntensity = (int)((((imagePatch[pxl][0] + imagePatch[pxl][1] + imagePatch[pxl][2]) / 3) * intensityLevels) / 255)
            intensityLevels_bin[curIntensity] += 1
            averageB[curIntensity] += imagePatch[pxl][0]
            averageG[curIntensity] += imagePatch[pxl][1]
            averageR[curIntensity] += imagePatch[pxl][2]

        # find the max intensity and its index
        maxIntensity = max(intensityLevels_bin)
        maxIndex = intensityLevels_bin.argmax()


        # return new value of the pixel
        final_pixel = np.array([averageB[maxIndex]/maxIntensity, averageG[maxIndex]/maxIntensity, averageR[maxIndex]/maxIntensity], dtype=np.uint8)
        # clipping values
        final_pixel[final_pixel > 255] = 255

        return final_pixel
```

O(n^2) run time complexity. Thanks to sklearn and numpy slicing, flattening, and reshaping techniques

Extracting image patches from the image

Processing each extracted image patch

Reshaping the 1D array back to a 3D array that match original image height and width

Replacing the original pixel values with the new pixel values since we do not process edge pixels in this algorithm

Numpy arrays to maintain intensity bins and red, green, and blue values for each intensity bin

Calculation of intensity for current pixel within the image patch

Maintaining Red, green, and blue values for the calculated intensity value

Recording maximum intensity value count

Recording maximum intensity value

Assigning new intensity value for the processing pixel within the image

oilPaint.py file (Improved version of the first implementation)

3.2.7 Oil paint code (Utilizing GPU)

Oil paint code utilizing the GPU is implemented under “oilPaint_GPU.py” python file

```python
from algorithms.utils import *
import copy
from numba import jit, cuda


'''''
This function only accepts kernels with a mid point i.e 3x3, 5x5, 7x7, etc.
'''
def oil_paint(ImageData, kernelSize, intensityLevels):
    # add padding to the image
    padding_pixels = kernelSize // 2
    padded_Image =  np.pad(ImageData, ((padding_pixels,padding_pixels), (padding_pixels,padding_pixels), (0, 0)), 'constant')
    return perform_oilPaint(padded_Image, padding_pixels, intensityLevels)


@jit(nopython=True)
def perform_oilPaint(originalImage, padding_pixels, intensityLevels):
    original_height, original_width, _ = np.shape(originalImage)
    # create an empty array to hold resulting data
    result = np.zeros((original_height, original_width,3), dtype=np.uint8)

    for row in range(padding_pixels, original_height - padding_pixels):
        for col in range(padding_pixels, original_width - padding_pixels):
            result[row][col] = calculate_new_pixel_value(originalImage[row - padding_pixels:row + padding_pixels + 1, col - padding_pixels:col + padding_pixels + 1], intensityLevels)

    # finally we remove the padding pixels and return
    return result[padding_pixels:original_height - padding_pixels, padding_pixels:original_width - padding_pixels, :]

@jit(nopython=True)
def calculate_new_pixel_value(imagePatch, intensityLevels):
    patch_height, patch_width, _ = np.shape(imagePatch)
    intensityLevels_bin, averageB, averageG, averageR = np.zeros(256), np.zeros(256), np.zeros(256), np.zeros(256)

    for row in range(patch_height):
        for col in range(patch_width):
            curIntensity = (int)((((imagePatch[row][col][0] + imagePatch[row][col][1] + imagePatch[row][col][2]) / 3) * intensityLevels) / 255)
            #print(curIntensity)
            intensityLevels_bin[curIntensity] += 1
            averageB[curIntensity] += imagePatch[row][col][0]
            averageG[curIntensity] += imagePatch[row][col][1]
            averageR[curIntensity] += imagePatch[row][col][2]

    # find the max intensity and its index
    maxIntensity = max(intensityLevels_bin)
    maxIndex = intensityLevels_bin.argmax()

    # return new value of the pixel
    final_pixel = np.array(
        [averageB[maxIndex] / maxIntensity, averageG[maxIndex] / maxIntensity, averageR[maxIndex] / maxIntensity])
    # clipping values
    final_pixel[final_pixel > 255] = 255

    return final_pixel
```

O(n^4) run time complexity. Since I could not find a way to make sklearn work with numba JIT. However, this code is order of magnitudes faster than above 2 CPU utilized versions. Due to these matrix computations are handled parallelly within the GPU

Using Numba JIT instead of python interpreter to optimize the code to run on GPU

Extracting image patches from the image

Numpy arrays to maintain intensity bins and red, green, and blue values for each intensity bin

Calculation of intensity for current pixel within the image patch

Maintaining Red, green, and blue values for the calculated intensity value

Recording maximum intensity value count

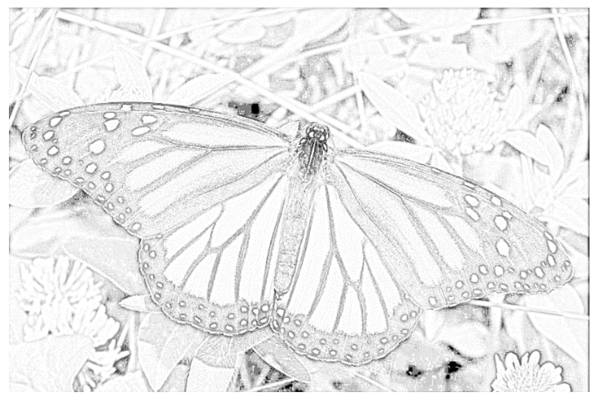
Recording maximum intensity value

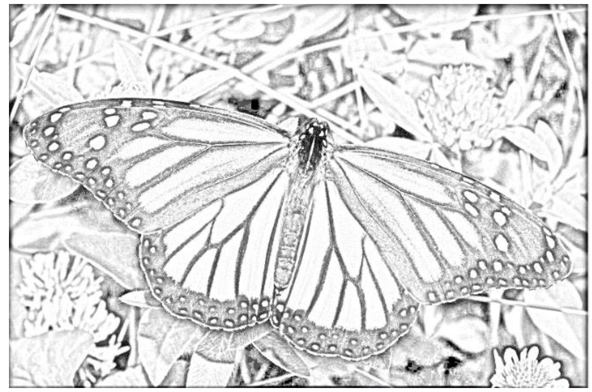
Assigning new intensity value for the processing pixel within the image

4. Results discussion and output images

4.1 Pencil Sketch algorithm

Original Image	Kernel Size	Sigma	Resulting Image
	3x3	1	
	3x3	50	

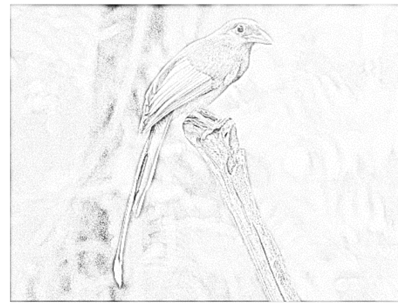
	3x3	100	
	9x9	1	
	9x9	50	

	9x9	100	
	19x19	1	
	19x19	50	
	19x19	100	

4.1.1 Kernel size vs output image

3 x 3 Kernel, sigma = 10

5 x 5 Kernel, sigma = 10



7 x 7 Kernel, sigma = 10

In order to discuss how kernel size of the gaussian kernel affects output image let us consider following two examples for both examples let us use following 5x5 grayscale negative image:

254	253	252	251	250
239	238	237	236	235
234	233	232	231	230
244	243	242	241	240
249	248	247	246	245

Grayscale

1	2	3	4	5
16	17	18	19	20
21	22	23	24	25
11	12	13	14	15
6	7	8	9	10

Grayscale negative

<u>Example 1:</u>

Kernel size = 3 x 3

Sigma = 1

Gaussian Kernel:

0.368	0.607	0.368
0.607	1	0.607
0.368	0.607	0.368

Let’s perform 2D convolution on center pixel (pixel with intensity value 23) of the 5x5 image using above Gaussian kernel

1	2	3	4	5
16	17	18	19	20
21	22	23	24	25
11	12	13	14	15
6	7	8	9	10

X

0.368	0.607	0.368
0.607	1	0.607
0.368	0.607	0.368

New pixel value of 23 = ((17 x 0.368) + (18 x 0.607) + (19 x 0.368) + (22 x 0.607) + (23 x 1) + (24 x 0.607) + (12 x 0.368) + (13 x 0.607) + (14 x 0.368)) / 4.9

= 18.889 **≈ 19**

Now let's perform dodging to find the new pixel value of the processing pixel (23 pixel of the grayscale negative image)

254	253	252	251	250
239	238	237	236	235
234	233	232	231	230
244	243	242	241	240
249	248	247	246	245

Grayscale

1	2	3	4	5
16	17	18	19	20
21	22	19	24	25
11	12	13	14	15
6	7	8	9	10

Grayscale negative
(After processing pixel 23)

New pixel value of 23 = (232 x 2^8) / (255 - 19)

= 251.661 **≈ 252**

Therefore, after dodging new pixel value is 252

<u>Example 2:</u>

Kernel size = 5 x 5

Sigma = 1

Gaussian Kernel:

0.018	0.082	0.135	0.082	0.018
0.082	0.368	0.607	0.368	0.082
0.135	0.607	1	0.607	0.135
0.082	0.368	0.607	0.368	0.082
0.018	0.082	0.135	0.082	0.018

Let's perform 2D convolution on center pixel (pixel with intensity value 23) of the 5x5 image using above Gaussian kernel

1	2	3	4	5
16	17	18	19	20
21	22	23	24	25
11	12	13	14	15
6	7	8	9	10

X

0.018	0.082	0.135	0.082	0.018
0.082	0.368	0.607	0.368	0.082
0.135	0.607	1	0.607	0.135
0.082	0.368	0.607	0.368	0.082
0.018	0.082	0.135	0.082	0.018

New pixel value of 23 = ((1 x 0.018) + (2 x 0.082) + (3 x 0.135) + (4 x 0.082) + (5 x 0.018)

+ (16 x 0.082) + (17 x 0.368) + (18 x 0.607) + (19 x 0.368) + (20 x 0.082) +

+ (21 x 0.135) + (22 x 0.607) + (23 x 1) + (24 x 0.607) + (25 x 0.135) +

+ (11 x 0.082) + (12 x 0.368) + (13 x 0.607) + (14 x 0.368) + (15 x 0.082) +

+ (6 x 0.018) + (7 x 0.082) + (8 x 0.135) + (9 x 0.082) + (10 x 0.018)) / 6.17

= 17.434 **≈ 18**

Now let's perform dodging to find the new pixel value of the processing pixel (23 pixel of the grayscale negative image)

254	253	252	251	250
239	238	237	236	235
234	233	232	231	230
244	243	242	241	240
249	248	247	246	245

Grayscale

1	2	3	4	5
16	17	18	19	20
21	22	18	24	25
11	12	13	14	15
6	7	8	9	10

Grayscale negative
(After processing pixel 23)

New pixel value of 23 = (232 x 2^8) / (255 - 18)

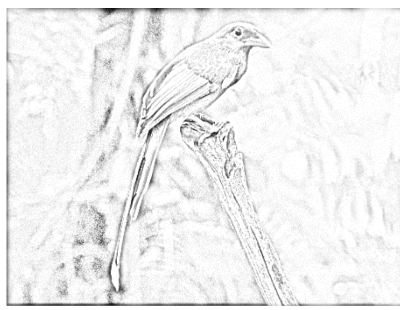
= 250.599 **≈ 251**

Therefore, after dodging new pixel value is 251

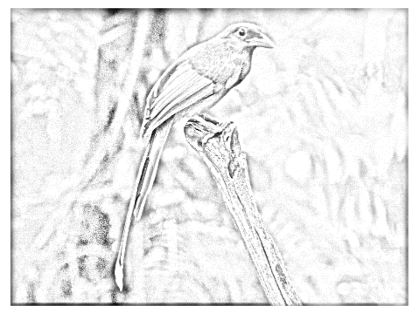
Now that we calculated our new pixel value for pixel with intensity 23, it is clear that when the kernel size increase the intensity value of the pixel decrease. Meaning the darkness of the pixel is increasing this is the reason when we increase the kernel size, we see the pencil sketch becomes much darker.

4.1.2 Sigma (Standard deviation) vs output image

19 x 19 Kernel, sigma = 1



19 x 19 Kernel, sigma = 4



19 x 19 Kernel, sigma = 10

In order to discuss how sigma of the gaussian kernel affects output image let us consider following two examples for both examples let us use following 5x5 grayscale negative image:

254	253	252	251	250
239	238	237	236	235
234	233	232	231	230
244	243	242	241	240
249	248	247	246	245

Grayscale

1	2	3	4	5
16	17	18	19	20
21	22	23	24	25
11	12	13	14	15
6	7	8	9	10

Grayscale negative

<u>Example 1:</u>

Kernel size = 3 x 3

Sigma = 1

Gaussian Kernel:

0.368	0.607	0.368
0.607	1	0.607
0.368	0.607	0.368

Let's perform 2D convolution on center pixel (pixel with intensity value 23) of the 5x5 image using above Gaussian kernel

1	2	3	4	5
16	17	18	19	20
21	22	23	24	25
11	12	13	14	15
6	7	8	9	10

X

0.368	0.607	0.368
0.607	1	0.607
0.368	0.607	0.368

New pixel value of 23 = ((17 x 0.368) + (18 x 0.607) + (19 x 0.368) + (22 x 0.607) + (23 x 1) + (24 x 0.607) + (12 x 0.368) + (13 x 0.607) + (14 x 0.368)) / 4.9

= 18.889 **≈ 19**

Now let's perform dodging to find the new pixel value of the processing pixel (23 pixel of the grayscale negative image)

254	253	252	251	250
239	238	237	236	235
234	233	232	231	230
244	243	242	241	240
249	248	247	246	245

Grayscale

1	2	3	4	5
16	17	18	19	20
21	22	19	24	25
11	12	13	14	15
6	7	8	9	10

Grayscale negative
(After processing pixel 23)

New pixel value of 23 = (232 x 2^8) / (255 - 19)

= 251.661 **≈ 252**

Therefore, after dodging new pixel value is 252

Example 2:

Kernel size = 3 x 3

Sigma = 100

Gaussian Kernel:

0.9999	0.99995	0.9999
0.99995	1	0.99995
0.9999	0.99995	0.9999

Let's perform 2D convolution on center pixel (pixel with intensity value 23) of the 5x5 image using above Gaussian kernel

1	2	3	4	5
16	17	18	19	20
21	22	23	24	25
11	12	13	14	15
6	7	8	9	10

X

0.9999	0.99995	0.9999
0.99995	1	0.99995
0.9999	0.99995	0.9999

New pixel value of 23 = ((17 x 0.9999) + (18 x 0.99995) + (19 x 0.9999) + (22 x 0.99995) +

(23 x 1) + (24 x 0.99995) + (12 x 0.9999) + (13 x 0.99995) +

(14 x 0.9999)) / 8.9994

= 18.00008 **≈ 18**

Now let's perform dodging to find the new pixel value of the processing pixel (23 pixel of the grayscale negative image)

254	253	252	251	250
239	238	237	236	235
234	233	232	231	230
244	243	242	241	240
249	248	247	246	245

Grayscale

1	2	3	4	5
16	17	18	19	20
21	22	18	24	25
11	12	13	14	15
6	7	8	9	10

Grayscale negative
(After processing pixel 23)

New pixel value of 23 = $(232 \times 2^8) / (255 - 18)$

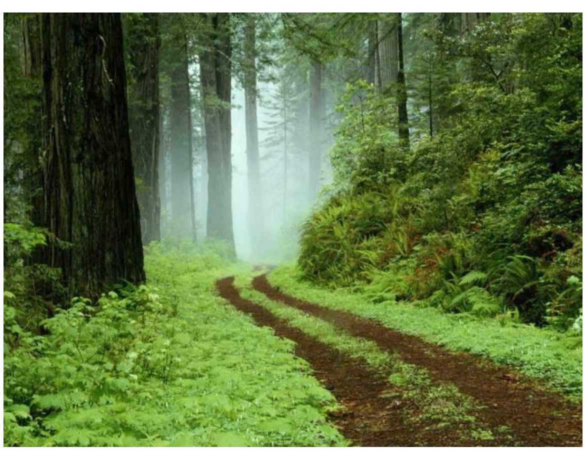
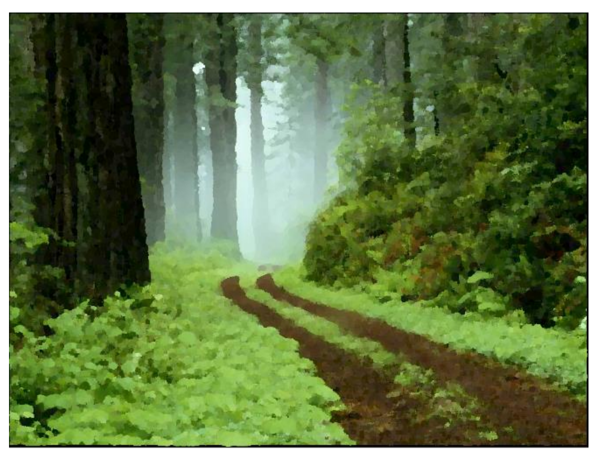
= 250.599 **≈ 251**

Therefore, after dodging new pixel value is 251

Now that we calculated our new pixel value for pixel with intensity 23, it is clear that when the sigma value increase the intensity value of the pixel decrease. Meaning the darkness of the pixel is increasing this is the reason when we increase the sigma, we see the pencil sketch becomes much darker.

4.2 Oil paint algorithm

Original Image	Kernel Size	Intensity Levels	Resulting Image
	3x3	10	
	3x3	100	

	3x3	200	
	9x9	10	
	9x9	100	

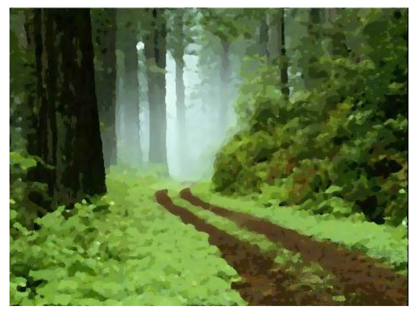
	9x9	200	
	19x19	10	
	19x19	100	

	19x19	200	

4.2.1 Kernel size vs output image

3 x 3 Kernel
Intensity Levels = 10

7 x 7 Kernel
Intensity Levels = 10



11 x 11 Kernel
Intensity Levels = 10

To explore the effect of change of kernel size, in the above experiment, I have set the intensity value to a constant (10 intensity values) and started changing kernel size from 3 x 3 to 5 x 5 and finally to 11 x 11. As we can see from the above images when we increase the kernel size the image becomes much pixelated. This is kind of like painting an image with larger paint brushes. Where the size of the paint brush used in 3 x 3 case is smaller than paint brush used in 5 x 5 case.

4.2.2 Intensity Levels vs output image

5 x 5 Kernel
Intensity Levels = 10

5 x 5 Kernel
Intensity Levels = 100

5 x 5 Kernel
Intensity Levels = 255

To explore the effect of change of intensity level values, in the above experiment, I have set the kernel size to a constant (5 x 5 kernel) and started changing intensity values from 10 to 100 and finally to 255. As we can see from the above images, when we increase the intensity level value the oil paint image gets more detailed/crispier. The reason behind this is when we increase the intensity value, number of colors a pixel can represents increase. For example, when intensity value is set 10 the maximum intensities of red, green and blue can represent is 10 respectively therefore, the number of colors a pixel can represent is smaller compared to a pixel with 100 intensity levels where this pixel can represent red, green and blue intensities of 100 respectively. This is the reason we see image get much crisper when we increase the intensity level value.

5. <u>Future work and lessons learned</u>

If I am being very honest with myself, the biggest lesson I learned during the development of this application is related to reading and understanding given specification. When I first received the specification for this project, I got very excited and just scanned through the document instead of thoroughly reading the document. Which ended up me misunderstanding the requirements. I thought this project requires us to build more than one effect and ended up developing 5 effects. However, after spending 2 weeks of sleep less nights for developing 5 algorithms I thought to myself this project cannot be this time consuming! So, I started reading the specs again and found out I just need to develop one effect. However, at this point I was done building all the algorithms except for the GUI for each of them. However, to save time when writing this document I decided to only include 2 effects that I found pretty hard to implement. But the version with 5 effects can be found under following GitHub repository:
https://github.com/kalanaS95/Pixel-Perfect-Effects

In addition, I have learned more about optimizing algorithms, using GPU power to process matrix related calculations to speed up the processing time.

For future work, I'm planning to extend this application to add more effects. Since I really enjoyed developing this application. At the moment, status of the full version of this application (stored in GitHub) as follows:

Effect	Algorithm status	GUI status	GPU optimization status
Pencil Sketch	Completed	Completed	Completed
Pop art	Completed	Completed	Still in the works
Oil paint	Completed	Completed	Completed
Cartoon	Still in the works	Still in the works	Still in the works
Solarize	Completed	Completed	Still in the works
Morphography	Completed	Still in the works	Still in the works

So, please feel free to take a look at the project and let me know what you think and where I can improve! The process to run this version is exactly similar to the process I described in this report.

In addition, I found using GPU for Image processing very interesting topic that I could explore in future. Therefore, I want to improve the GPU optimization code, since due to the time crunch, I had to implement each effect as a script instead of a classes. Also, I would like to do more research parallelizing the CPU version of each effect, such that it can utilize multiple cores if the system supports multi-core processing (Especially for oil paint effect, it takes lot of time to process larger images when using the CPU)

To conclude, I believe this project really helped me to understand concepts of 2D convolution, building kernels from scratch, building GUIs using python TKinter library as well as using numba JIT to optimize the code.

6. References

- Beyeler, M. (2016, January 13). How to create a beautiful pencil sketch effect with OpenCV and Python. Retrieved Winter, 2021, from https://www.askaswiss.com/2016/01/how-to-create-pencil-sketch-opencv-python.html
- Kumar, A. (2019, March 19). Computer vision: Gaussian filter from scratch. Retrieved Winter, 2021, from https://medium.com/@akumar5/computer-vision-gaussian-filter-from-scratch-b485837b6e09

- T. (2019, October 9). Programmer help. Retrieved Winter, 2021, from https://programmer.help/blogs/special-effects-of-oil-painting-based-on-opencv-using-python.html
- Esterhuizen, D. (2013, June 30). C# how to: Oil painting and cartoon filter. Retrieved Winter, 2021, from https://softwarebydefault.com/2013/06/29/oil-painting-cartoon-filter/
- G_, S. (2012, October 21). Oil paint effect: Implementation of oil painting effect on an image. Retrieved Winter, 2021, from https://www.codeproject.com/Articles/471994/OilPaintEffect
- Unknown. (2011, September 11). Oil painting algorithm. Retrieved Winter, 2021, from http://supercomputingblog.com/graphics/oil-painting-algorithm/
- Leero11Leero11 19522 silver badges1414 bronze badges, Bryan OakleyBryan Oakley 301k3333 gold badges428428 silver badges571571 bronze badges, Louis DurandLouis Durand 17799 bronze badges, BPLBPL 9, & Dblclikdblclik 40622 silver badges88 bronze badges. (1965, July 01). Python/Tkinter - Identify object on click. Retrieved Winter, 2021, from https://stackoverflow.com/questions/38982313/python-tkinter-identify-object-on-click
- Unknown. (n.d.). Writing cuda-python. Retrieved Winter, 2021, from https://numba.pydata.org/numba-doc/0.13/CUDAJit.html
- Grover, P. (2019, November 30). Speed up your algorithms part 2- numba. Retrieved Winter, 2021, from https://towardsdatascience.com/speed-up-your-algorithms-part-2-numba-293e554c5cc1