Computer Vision algorithms in CUDA

Angad Kambli, 19114041
Ayush Gupta, 19114017
Devanshu Chaudhari, 19114027
Shubham Bansode, 19114019

Abstract

This project aims to implement some common Computer Vision and Image Processing algorithms in C++ harnessing CUDA's power to parallelize the work and give efficient results. We have used OpenCV for accessing and displaying images and video inputs. Project management was done using Visual Studio. The GitHub repository can be accessed here.

Contents

1 Problem Statement

Computer Vision:
Implement basic computer vision algorithms using **CUDA**:

- **Edge Detection** : Canny Edge detection, Sobel Filter
- **Blurs** : Mean Blur, Gaussian Blur, Bokeh Blur with three shapes
- **Noise** : Addition and reduction
- **Separable filters** for mean blur and Gaussian Blur
- **Image sharpening**

2 Individual Contributions

While all the team members were involved in the brainstorming for deciding the project's flow and implementation, there was a clear division of labour with regards to the domains a certain person would be responsible for. Following are the contributions of various members :

1. **Angad Kambli, 19114041 :** Initialized the Visual Studio project. Involved in research and implementation of Sobel Edge detection algorithm, Mean Blur algorithm, Canny Edge Detection, Noise Addition and Bokeh blur. Involved in all the discussions for algorithm implementations and bug fixing.
2. **Ayush Gupta, 19114017 :** Implemented naive version for various algorithms as a reference for CUDA implementation. Involved in research and implementation of algorithms like Noise addition, Noise Reduction , Gaussian Blur and Canny algorithm for edge detection. Involved in discussion for improvisation of algorithms and minor bug fixes.
3. **Devanshu Chaudhari, 19114027 :** Involved in Implementation of Bokeh Blur and separable Gaussian blur. Involved in Research and implementation of Noise addition. Invloved in almost all discussions of implementation of algorithms.
4. **Shubham Bansode, 19114019 :** Involved in implementation of Bokeh Blur , Gaussian blur, error solving during Sobel filter implementation, created dynamic mask images for Bokeh.

3 Evaluation Parameters

Time was noted two times, before calling the function for the required filter and after all the required function(s) for that filter is/are called. Then the time taken for applying the filter is calculated by subtracting the former from later. Such time noted in miliseconds proves to be a good yardstick to measure the relative performance of each algorithm. For final results, read Results and Discussion.

4 Methodology

4.1 Sobel Edge Detection filter

The Sobel edge detection filter is a very basic and fast method for detecting edges in a photo. It is also called as the gradient filter since what it does is a rough approximation of calculating a derivative of the image. There are two filters that are convolved over the image which calculate the derivative in horizontal and vertical direction. They are basically the same filters rotated by 90 degrees.Gx is :

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

and Gy is :

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Gx and Gy are the two filters which are convolved over the image and the square root of the sum of their squares is the final result.

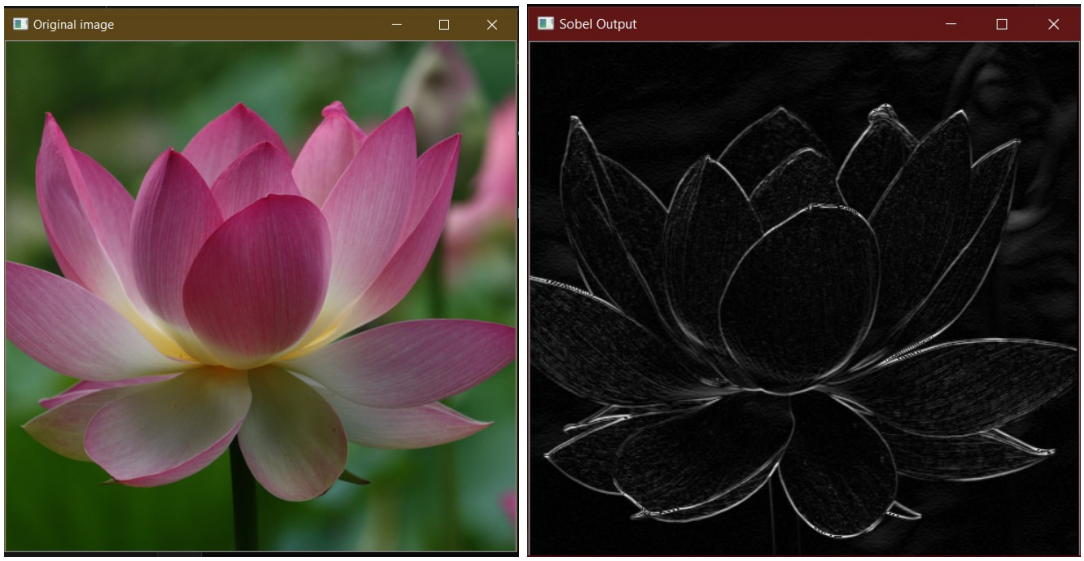

Figure 1: Sobel input and output

As seen in figure 1, the sobel operator is successful in highlighting the edges. However, the edges are not sharp and also varying in intensity. The edge data can be made more precise and useful with the more sophisticated canny edge detection algorithm as discussed further

4.2 Canny Edge Detection

Canny Edge Detection algorithm is a more advanced and better edge detection algorithm as compared to Sobel Edge Detection.Unlike Sobel operator which produces thick edges with gradual transition of pixel values, Canny algorithm produces a sharp, single pixel edges with a strong contrast. Instead of just convolving a filter with the image, a lot more number of steps are involved.

- **Conversion to Gray Scale** : In this step, the image is converted to a single channel gray scale image, pixel values ranging from 0 to 255 (8 bit) from the RGB channel. This was done since the channel information is irrelevant in edge detection and could had increased the run time unnecessarily.
- **Apply Gaussian Blur** : Since Canny operation is sensitive to noise, this step reduces some amount of abrupt pixels by smoothening the image.
- **Gradient Calculation** : Gradient at a particular pixel is calculated to gain the information about the direction of edge at that point. This is accomplished using the two Sobel filters Gx and Gy . Amplitude and direction is given by

$$\text{Amplitude} = \sqrt{Gx^2 + Gy^2}$$

$$\theta = tan^{-1}(\frac{Gy}{Gx})$$

- **Non Maximum Suppression** This is the main step of Canny Edge Detection. This steps helps to convert a thick low contrast edge to a single pixel edge. We have used Non Maximum Suppression without interpolation, for which we use 8 neighboring pixels along with the current pixel. First the 3*3 pixels are divided into 8 parts according to the angle that they make with the central pixel. Based on the value of θ . If the adjacent pixel values along the gradient are smaller than the current value, the pixel is enhanced otherwise suppressed.

- **Double Thresholding** Here two thresholds, a lower one and a higher ones are set. Any pixel greater then the higher threshold is set to 255(highest value), any pixel value lesser then the lower threshold is set to 0 and the rest are left unchanged. This enhances strong edges and suppresses weak edges.
- **Edge Tracking** Some of the genuine weak edges were removed by the previous step. In this step all the weak edges are restored which were connected to the strong edges. This step is similar to DFS algorithm used in graph traversal.
- **Clean up** All the weak edges are removed.
  The effect of Canny operator can be seen in figure 2. As we can see the edges are much more sharper and bright.

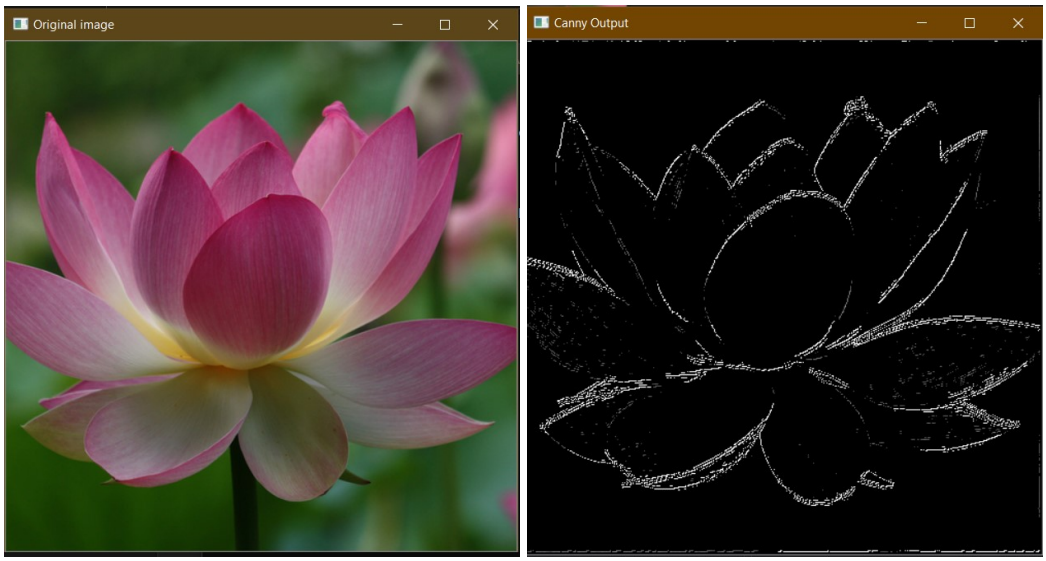

Figure 2: Canny input and output

4.3 Mean Blur

The Mean Blur is the most basic blur that exists and as the name suggests, it takes an average of all pixels around it. With bigger dimensions of the area considered while blurring the image, the blur gets stronger. This can get out of hand rather quickly in terms of execution time when we do this sequentially one pixel at a time. However, with CUDA this can be done relatively fast. Still for each pixel, the complexity remain $O(N^2)$ where N is the dimension chosen for blurring. This can be further reduced to O(N) with the use of seperable filters as discussed in Separable filters. Output of mean blur can be seen in figure 3.

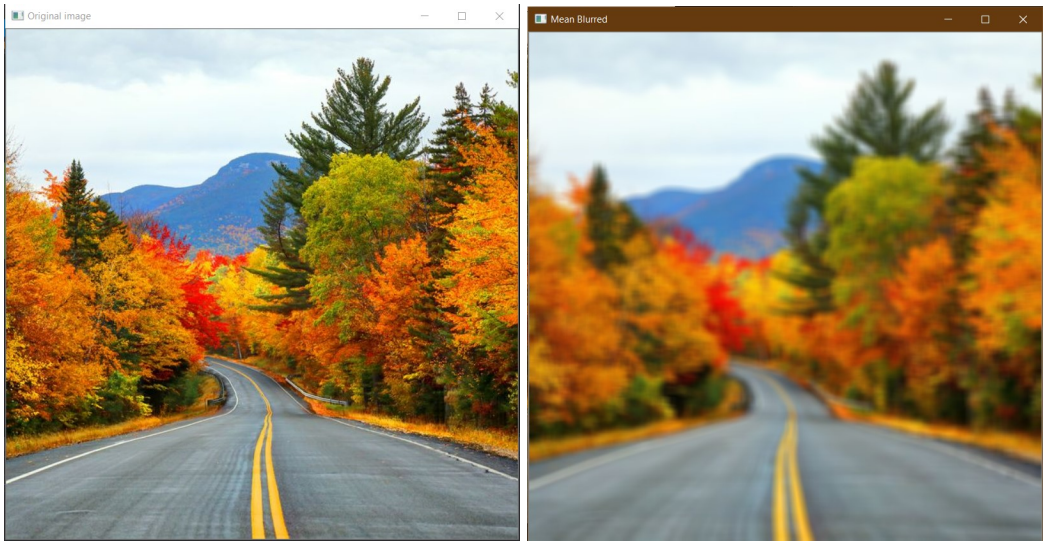

Figure 3: Mean Blur input and output

4.4 Gaussian Blur

Gaussian function (in Fig 4.4.1) is used to attain the Gaussian blur by smoothing an image. Gaussian blur is achieved by convolving the input image with a predefined Gaussian Kernel. It is considered to be a low pass filter and also reduces negligible details in an image.

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

Figure 4: The Gaussian function

We have implemented the 9*9 Gaussian blur using a box blur by approximation. Gaussian blur is therefore achieved by central limit theorem. In naïve implementation with a 9 * 9 kernel the resultant complexity of the program will be O(n^2), but due to Gaussian having separable properties instead of applying 2D convolution kernel we can apply a 1D convolution on row and column separately. This method will bring down the complexity to O(n) which significantly faster.
As we can see in 5, the blur is much more softer and feels more natural than mean blur. The last kind of blur is Bokeh Blur as discussed next.

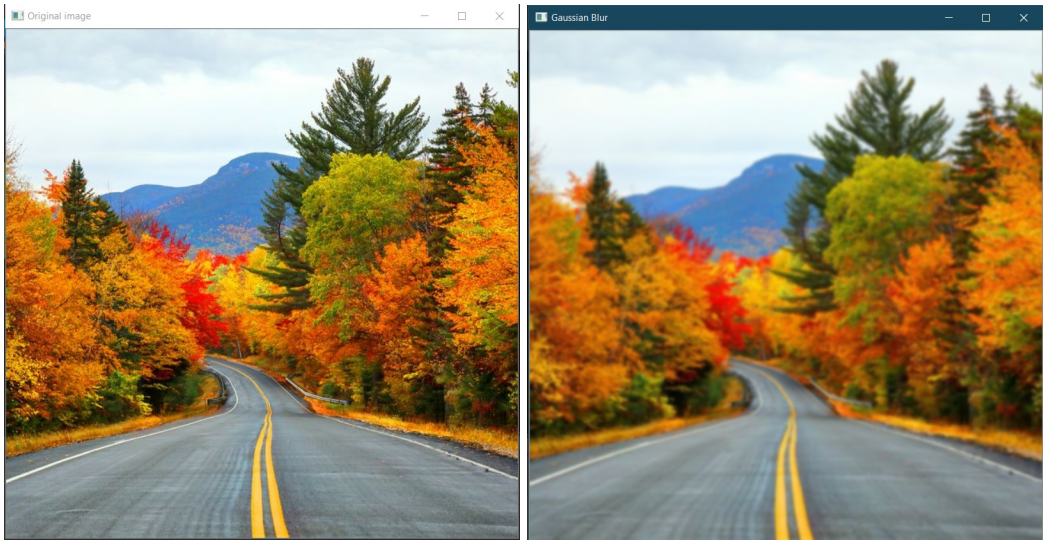

Figure 5: Gaussian blur input and output

4.5 Bokeh Blur

Bokeh Blur is an out of focus effect given by a camera. This effect is replicated in image processing using another image of any shapes. The convolution of both the images gives the bokeh blur effect. The shape of the bright point in the output image will be same as that of the image with which the original image was convolved. The time complexity would be O(a*b*c*d), where the dimension of two images are a*b and c*d. But with the implementation of CUDA, time complexity of each kernel used is constant. A pixel of the image near bright point source i.e. A pixel with neighbouring pixels having high RGB value after convolution becomes bright. The neighbouring area of pixel is decided by the mask image. So the Bokeh Blur actually acts like a convolution where the mask is a very big filter. To parallelize this, we have indexed the input image through the number of grids in CUDA and the number of threads in each block are h/4 * w/4 where h and w are height and width of mask image. Thus in each thread, we calculate 4 pixels. We are doing this to mantain simplicity of code of as the maximum threads can be 1024. So, by this method, the mask image can be maximum 128*128 size. For more details on this, read about Bokeh Blur's novelty.

4.6 Image Sharpening

Kernel based image sharpening is one of the most simple and effective image sharpening techniques. The operation essentially involves the convolution of a filter with the input image. Filter used is :

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Since the filter values sum to 0, no normalization was required. This operation is equivalent to subtracting the mean blurred image from a double intensity input image. The sharpening effect of the kernel can be understood from the nature of the kernel applied. If the intensity difference between adjacent pixels is not much, nothing happens since the pixels value gets added 9 times but also gets subtracted 8 times. If the intensity difference is significant the pixel value gets boosted.

Figure 6: Image Sharpening input and output

4.7 Noise Addition

We implemented Salt and Pepper noise addition technique for noise addition. In this technique we randomly assign some pixels value of 255 (Salt) and some pixels a value of 0(Pepper). The amount of noise added can be controlled. For this we generate a random number between 0 and x . if the number is 0 the pixel if assigned 0 and if number is x/2 255 is assigned.

$$Probability\ of\ salt\ noise = \frac{1}{x} \tag{1}$$

$$Probability\ of\ pepper\ noise = \frac{1}{x} \tag{2}$$

$$Probability\ of\ total\ noise = \frac{2}{x} \tag{3}$$

Figure 7 shows an example of this.

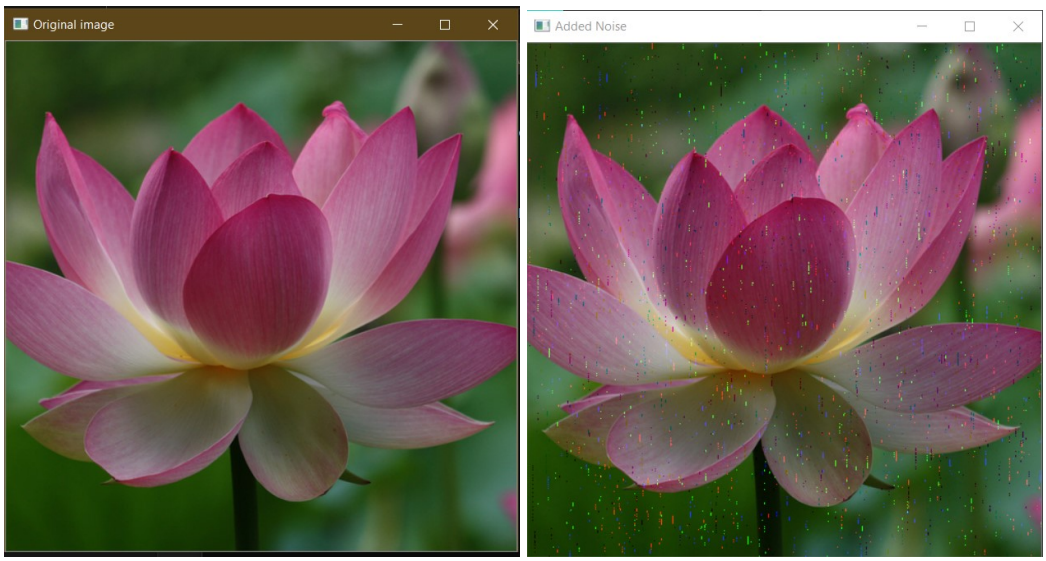

Figure 7: LHS : Input image — RHS : added noise

4.8 Noise Reduction

For noise reduction we first apply a Mean Blur filter for smoothening the image and to remove abrupt pixels. This also guarantees that there is less discontinuity in the image. Now we roughly get a blurred image of the original noise free image. Now we need to sharpen to get back our noise free image. This can easily be done by applying an image sharpening filter yielding an approximate noise free version on the image as seen in figure 8.

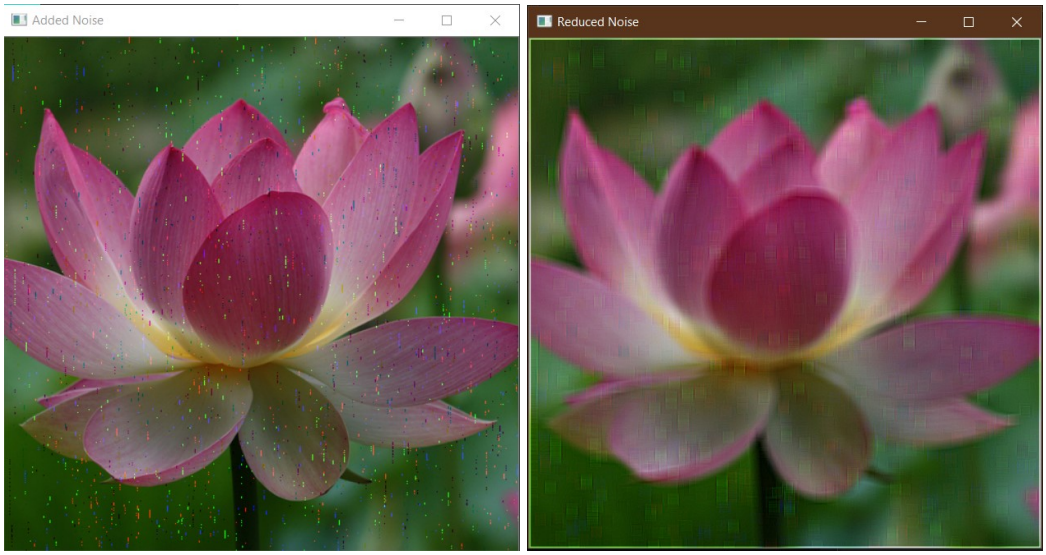

Figure 8: LHS : Noisy image — RHS : reduced noise

5 Novelty of Work done

5.1 Parallelization of Canny edge detection

Unlike Sobel Edge detection algorithm, Canny operator was more complex and involved far too many steps. If not for parallelization, the delay could have compounded. Among the different processes, we faced a challenge in implementing a parallelized version of Edge tracking, an algorithm for detecting weak edges that we removed in the previous steps. Since Edge tracking is essentially just a Depth First Search algorithm with all the strong edges as source vertices, parallelizing was difficult. The difficulty was primarily because of the fact that all the pixels were interdependent, a property not very suitable for parallelization. We solved this problem by parallelly updating each pixel value based on it's neighboring pixels.Initially we store the pixels values which were weak

edges that were removed in Double thresholding. Here we made a trade off between time and memory (we chose time, of course). With only a few iterations we get a good edge enhancement.

5.2 Bokeh Blur

The Bokeh Blur emulates how a photo would look if the camera was out of focus. Its difference with mean blur can be seen in figure 9 where the image on the right just looks like a low quality image but the image on the left looks like something that was taken from a camera when it was out of focus.

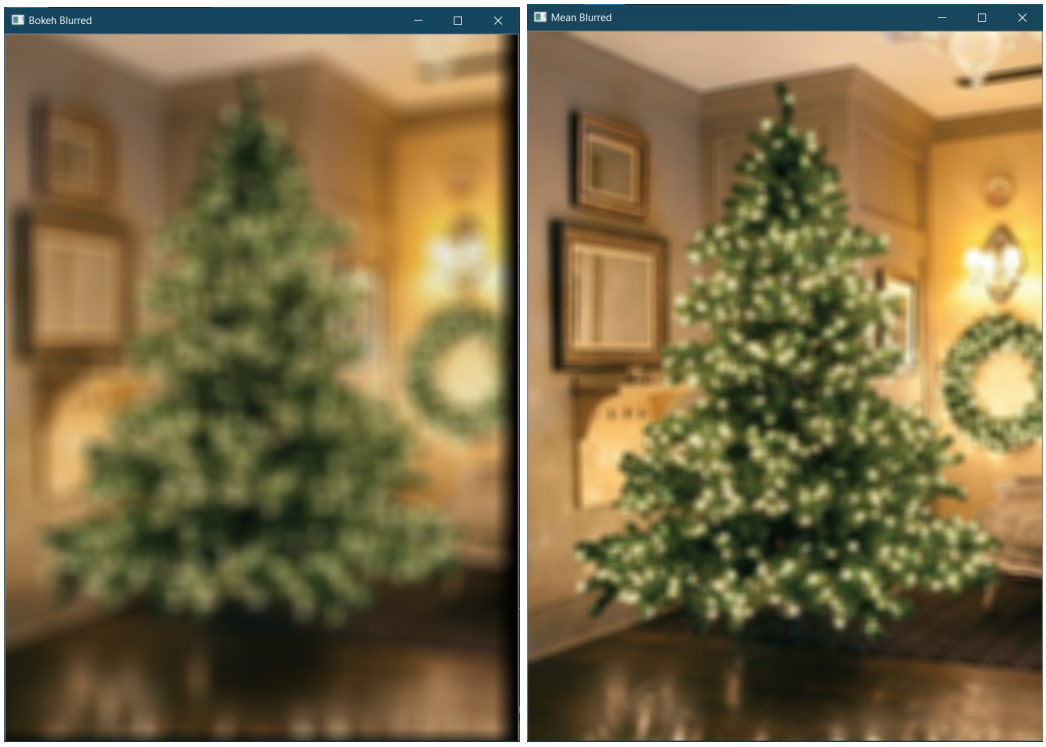

Figure 9: Bokeh Blur and Mean Blur

We have also implemented three different shapes and dynamic size that the user can choose while running the program. The different shapes are shown in figures 10, 11 and 12. The effect when different sizes are used can be seen in figures 13, 14 and 15.

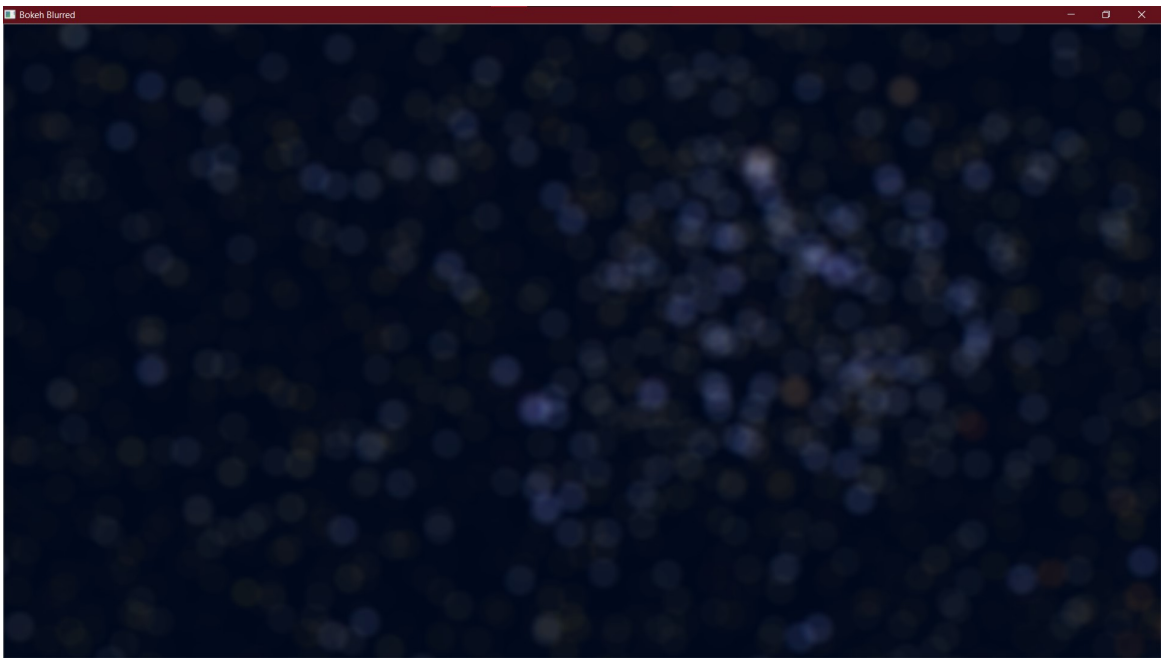

Figure 10: Bokeh Blur with circle mask

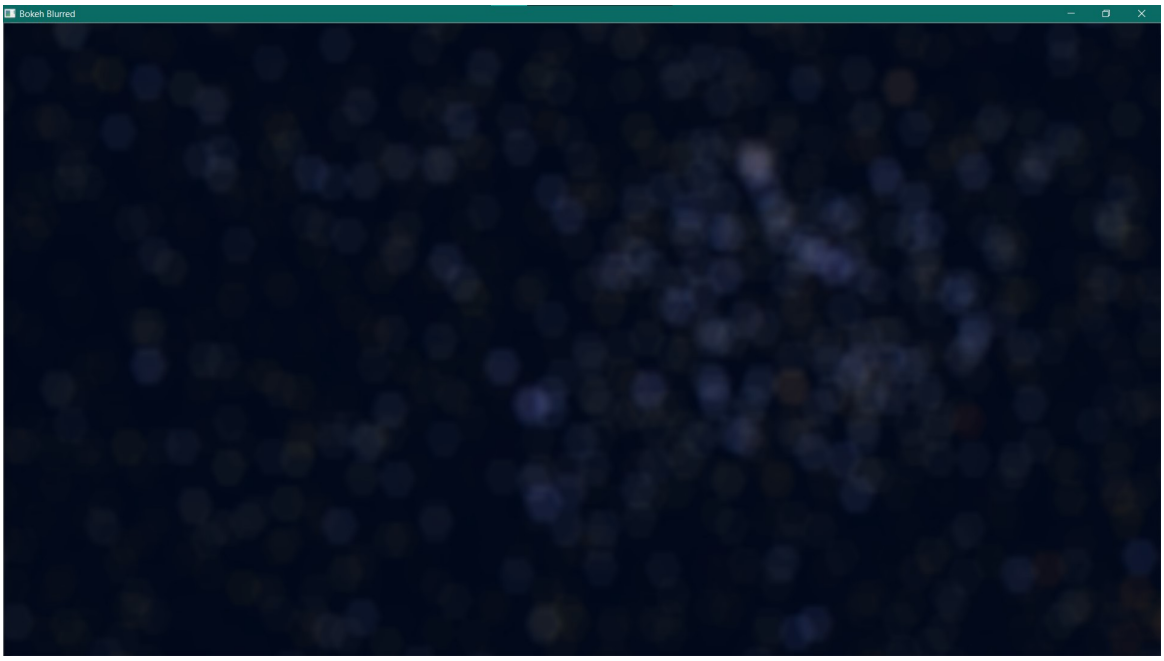

Figure 11: Bokeh Blur with hexagonal mask

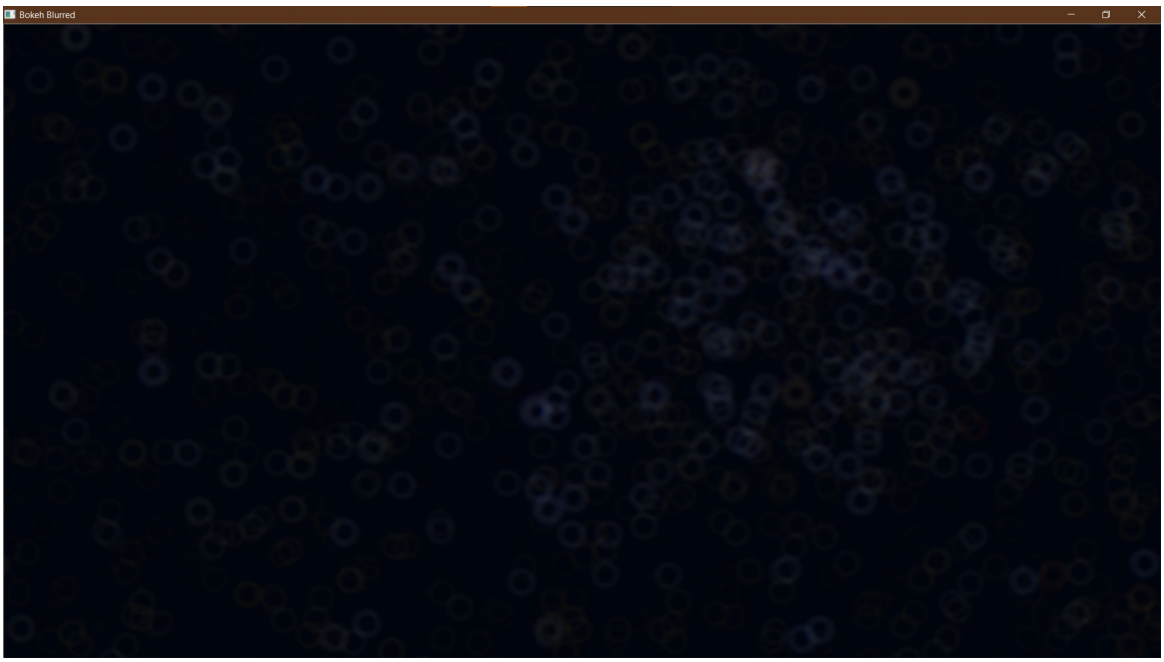

Figure 12: Bokeh Blur with ring mask

Figure 13: Bokeh Blur with circle mask of radius 16

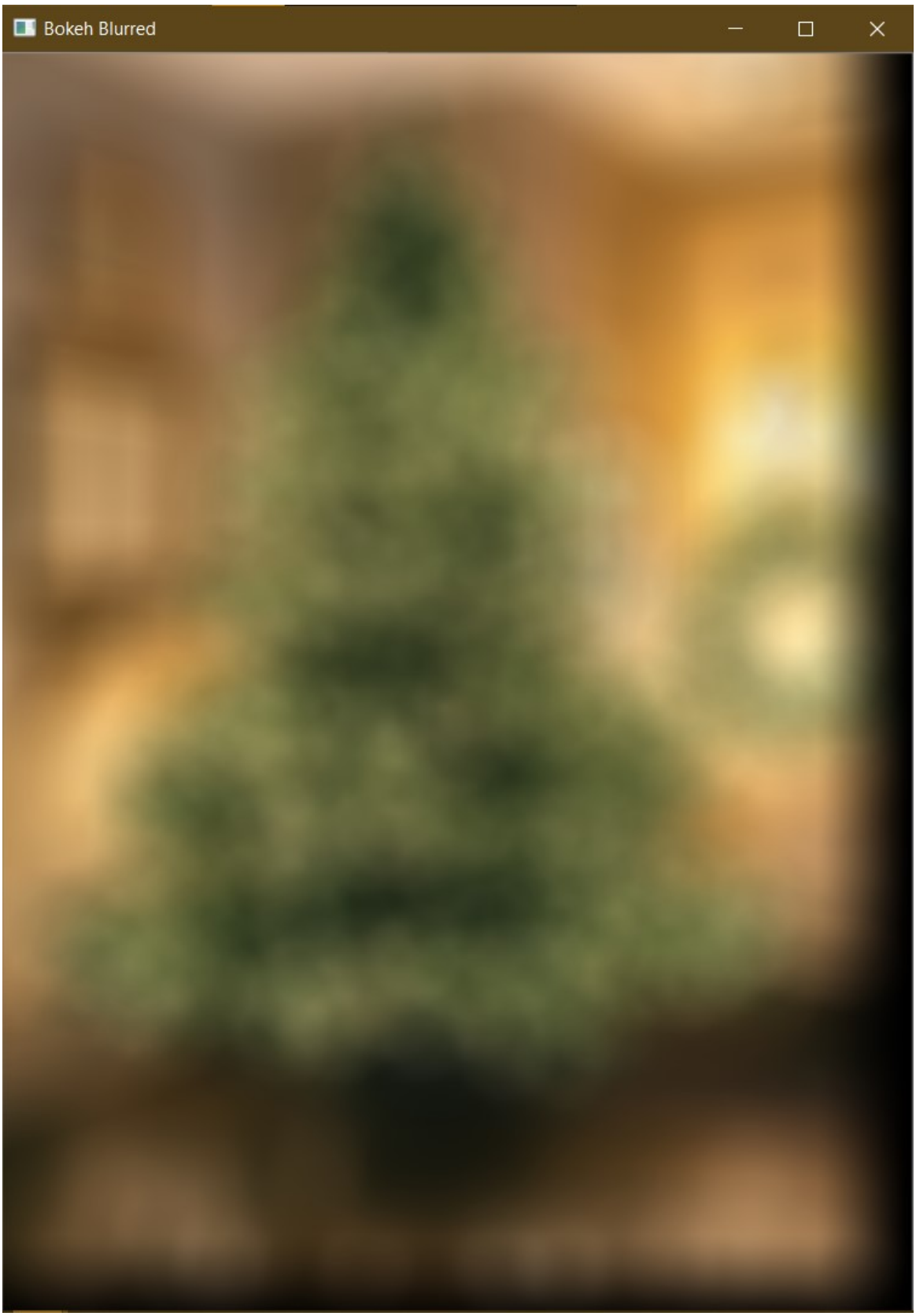

Figure 14: Bokeh Blur with circle mask of radius 32

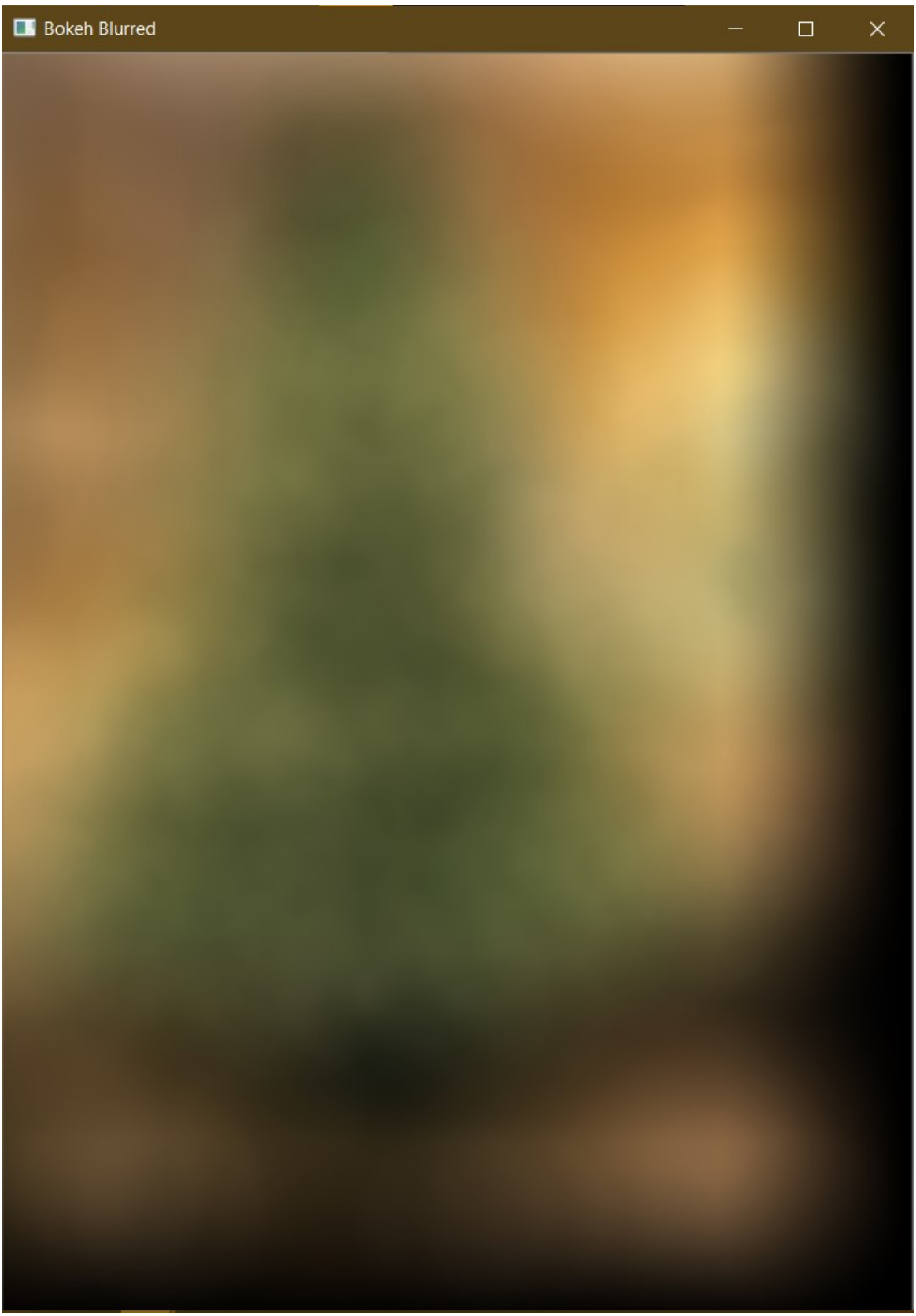

Figure 15: Bokeh Blur with circle mask of radius 64

5.3 Separable filters

Separable convolution is the idea of separating 1 convolution into 2 passes, one horizontal and one vertical. For example, in Gaussian Blur we can use the property of separable filter, where in instead of using a kernel

of size 9x9, we can divide the kernel into 1x9 and 9x1. This will result in a smaller number of calculations and less complexity. After separating the kernel into two we convolve our image with the 1x9 kernel and then by 9x1 kernel we will get the same output. This method is not relevant for every kernel as we cannot divide every kernel into two different kernels. We have used this in Gaussian blur and Mean Blur since the Gaussian function is compatible with this method and Mean Blur is just averaging the pixels and shall give the same output irrespective of the way how the pixel values are added. For Gaussian Blur, the seperate filters are :

$$\begin{bmatrix} 22.9 \\ 59.77 \\ 60.598 \\ 24.1732 \\ 38.2928 \\ 24.1732 \\ 60.598 \\ 59.77 \\ 22.9 \end{bmatrix} \qquad \begin{bmatrix} 22.9 & 59.77 & 60.598 & 24.1732 & 38.2928 & 24.1732 & 60.598 & 59.77 & 22.9 \end{bmatrix}$$

The cumulative result of the horizontal and vertical passes is same as that which we would get if we had carried out the whole convolution. Thus, the complexity is reduced with no negative effect on the output.

6 Results and Discussion

6.1 Results (Time taken in milliseconds for each operation)

1. **Sobel :** 223 ms
2. **Noise Addition :** 196 ms
3. **Noise Reduction :** 279 ms
4. **Gaussian Blur :** 306 ms
5. **Sharpening :** 208 ms
6. **Canny edge detection :** 331 ms
7. **Mean Blur :** 233 ms
8. **Bokeh Blur :**
    - For a circle of radius 20 : 351 ms
    - For a ring of radius 20 and thickness 5 : 393 ms
    - For a hexagon of side 20 : 341 ms

6.2 Analysis and discussion of the results

As we can see, the fastest execution is of the noise addition algorithm, this is because in noise addition, we just seed the random state once and then every pixel has to just do one lookup to decide on its value.
The slowest execution is of the three bokeh blurs closely followed by Canny edge detection. Bokeh Blur takes time owing to the huge masks that act as convolution filters. Canny edge detection takes time since it iterates upon the image a number of times for the edge tracking step. It is to be noted that all these algorithms in their parallelized form surely give better results than their sequential form.

7 Conclusions

With the advent of ML and AI and its widespread use in nearly every consumer applications, there is a great need for quick ML results. Computer Vision forms a major part of such of implementations. A lot of algorithms implemented as a part of this project form the building blocks for a number of common ML/DL Models. Parallelizing this models with CUDA highly speed up the process in comparison to sequentially processing a pixel at a time. This was felt a lot while implementing the bokeh filter as it involves convolving two images where one is used as a mask. CUDA implementations for other such filters can also be implemented with relative ease with the help of the code from this project. Thus parallelization proves to be a important tool when it comes to data related workloads.

8 References/Bibiography

Canny edge related links:

1. http://justin-liang.com/tutorials/canny/

Canny edge related links:

1. http://justin-liang.com/tutorials/canny/
2. https://github.com/JustinLiang/ComputerVisionProjects/blob/master/CannyEdgeDetector/CannyEdgeDetec
   m

Gaussian Blur related links :

1. https://www.cs.auckland.ac.nz/courses/compsci373s1c/PatricesLectures/Image%20Filtering.pdf
2. http://dev.theomader.com/gaussian-kernel-calculator/
3. https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html

Bokeh Blur related links :

1. https://github.com/mikepound/convolve
2. https://www.scratchapixel.com/lessons/digital-imaging/simple-image-manipulations/bookeh-effect
3. https://www.scratchapixel.com/code.php?id=51&origin=/lessons/digital-imaging/simple-image-manipula
4. https://py2py.com/bokeh-effect-in-image-using-python/

Setup related links :

1. https://www.youtube.com/watch?v=IzU4AVcMFys&feature=youtu.be
2. https://www.pyimagesearch.com/2016/05/23/opencv-with-tkinter/
3. https://youtu.be/1XTqE7LFQjI
4. https://stackoverflow.com/questions/31711093/the-program-cant-start-because-opencv-world300-dll-is
5. https://medium.com/@subwaymatch/adding-opencv-4-2-0-to-visual-studio-2019-project-in-windows-usin
   ~:text=To%20use%20OpenCV%20with%20Visual,%2B%2B%20during%20Visual%20Studio%20install.

Noise Reduction links:

1. https://courses.cs.washington.edu/courses/cse576/11sp/notes/Basics2_white.pdf

Noise Addition links:

1. https://courses.cs.washington.edu/courses/cse576/11sp/notes/Basics2_white.pdf
2. https://medium.com/image-vision/noise-in-digital-image-processing-55357c9fab71#:~:text=
   Salt%20and%20Pepper%20Noise%3A%20Salt,original%20data%20values%20%5B5%5D.

Mean blur links:

1. https://courses.cs.washington.edu/courses/cse576/11sp/notes/Basics2_white.pdf