Car plate OCR & Star Detection and Counting in Astronomical Images: Theory and Algorithms

Rashin Rahnamoun

HW2 Vision Research Q1 & Q2

1 Q1 :OCR for Car License Plate Recognition

This section details the methodology for performing Optical Character Recognition (OCR) on car license plates using image processing techniques and the EasyOCR library. The primary steps involve loading the image, isolating the license plate, converting it to black-and-white for enhanced readability, and then using OCR to extract text.

1.1 Steps in License Plate Recognition

The license plate recognition algorithm consists of the following steps:

1. **Image Loading**: Load the image, $\mathbf{I}$, containing the car license plate:

$$\mathbf{I} = \text{cv2.imread(path)}$$

   where $\mathbf{I}$ is a 3D array representing the color image in RGB format.

2. **Region of Interest (ROI) Definition**: Define the bounding box for the license plate region based on known or estimated coordinates (x, y, w, h), where:

$$\text{ROI} = \mathbf{I}[y : y + h, x : x + w]$$

   Here, ROI is the rectangular section of $\mathbf{I}$ containing the license plate.

3. **Grayscale Conversion**: Convert the license plate region ROI from RGB to grayscale, resulting in a single-channel grayscale image, $\mathbf{G}$:

$$\mathbf{G} = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

   where R, G, and B are the color channels of each pixel in ROI.

4. **Black-and-White Conversion**: Apply Otsu's thresholding to convert the grayscale image, $\mathbf{G}$, to a binary black-and-white image, $\mathbf{B}$, enhancing text readability. Otsu's method determines an optimal threshold T that minimizes within-class variance:

$$T = \arg\min_{\tau} \left(q_1(\tau) \cdot \sigma_1^2(\tau) + q_2(\tau) \cdot \sigma_2^2(\tau) \right)$$

   where τ is the threshold, $q_1(\tau)$ and $q_2(\tau)$ are the probabilities of pixel intensities below and above τ, and $\sigma_1^2(\tau)$ and $\sigma_2^2(\tau)$ are the variances of these pixel intensities. The binary image $\mathbf{B}$ is then defined as:

$$\mathbf{B}(x, y) = \begin{cases} 0 & \text{if } \mathbf{G}(x, y) < T \\ 255 & \text{if } \mathbf{G}(x, y) \geq T \end{cases}$$

5. **OCR Application**: Use EasyOCR to extract text from the binary image $\mathbf{B}$, restricting the recognition to alphanumeric characters:

$$\text{Text} = \text{EasyOCR}(\mathbf{B}, \text{allowlist} = \{\text{A-Z, 0-9}\})$$

   where Text represents the characters detected in the license plate region.

1.2 Algorithm: OCR for Car License Plate

Algorithm 1 OCR for Car License Plate Recognition

```
Load the image from the specified path.
if Image is loaded successfully then
    Define ROI coordinates for the license plate (x, y, w, h)
    if ROI coordinates are within image dimensions then
        Extract the license plate region from the image.
        Convert the license plate region to grayscale.
        Apply Otsu's thresholding to create a black-and-white image.
        Initialize the EasyOCR reader for English characters.
        Perform OCR on the black-and-white image to extract text, limiting characters to alphanumeric.
        Display the cropped license plate and the black-and-white image.
        Print the extracted text.
    else
        Output an error: "The specified ROI is out of bounds."
    end if
else
    Output an error: "Unable to load image from path."
end if
```

1.3 Mathematical Representation of Otsu's Thresholding

Otsu's thresholding is applied to convert the grayscale license plate image into a binary black-and-white image, maximizing the separation between background and foreground pixel intensity distributions. Mathematically, it selects a threshold T that minimizes the within-class variance σ_w^2 defined by:

$$\sigma_w^2(T) = q_1(T) \cdot \sigma_1^2(T) + q_2(T) \cdot \sigma_2^2(T)$$

where: - $q_1(T)$ and $q_2(T)$ are the probabilities of the two classes separated by T. - $\sigma_1^2(T)$ and $\sigma_2^2(T)$ are the variances of these two classes.

1.4 Results and Visualization

The algorithm produces two key outputs:

- The cropped license plate region.
- The black-and-white version of the license plate region.

The final OCR result displays the extracted text from the license plate, enabling further processing or data extraction.

2 Q2 :Star Detection and Counting

This document outlines a theoretical approach for detecting and counting stars in an astronomical image using pixel data parsing, grayscale conversion, and blob detection through the Laplacian of Gaussian (LoG) method.

3 Pixel Data Loading and Image Formation

Given pixel data with dimensions h (height), w (width), and c (channels), the image array is formed as:

$$\text{ImageArray} = \text{reshape}(\mathbf{P}, [h, w, c])$$

where $\mathbf{P} = \{(R_i, G_i, B_i) \mid i = 1, 2, \ldots, N\}$ represents pixel data with $N = h \times w$.

(a) Cropped License Plate and Results

3.1 Algorithm: Loading and Reshaping Pixel Data

Algorithm 2 Loading and Reshaping Pixel Data

1: $\mathbf{P} \leftarrow$ parse(file)
2: $h, w, c \leftarrow$ file_header
3: ImageArray $\leftarrow$ reshape($\mathbf{P}, [h, w, c]$)
4: **return** ImageArray

4 Saving the Image

The image is saved in JPEG format with maximum quality and PNG format with no compression:

$$\text{JPEG} = \text{save}(\text{ImageArray}, \text{quality} = 100)$$

$$\text{PNG} = \text{save}(\text{ImageArray}, \text{compression} = 0)$$

5 Grayscale Conversion

Each pixel (R, G, B) in the color image is transformed to grayscale intensity G using:

$$G = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$$

resulting in a single-channel grayscale image.

5.1 Algorithm: Grayscale Conversion

Algorithm 3 Convert Image to Grayscale

for $(R, G, B) \in$ ImageArray **do**
  $G \leftarrow 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B$
end for
return Grayscale image array

6 Star Detection using Laplacian of Gaussian (LoG)

6.1 Laplacian of Gaussian (LoG)

The Laplacian of Gaussian (LoG) function detects bright, circular regions in the grayscale image, following:

$$\text{LoG}(x, y, \sigma) = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \left(\exp\left(-\frac{x^2 + y^2}{2\sigma^2} \right) \right)$$

where σ varies to detect blobs of different scales.

Blobs are detected at multiple scales by:

$$\text{Blob}(x, y) \text{ if } |\text{LoG}(x, y, \sigma)| > T$$

where T is a threshold for detecting high-contrast areas.

6.2 Algorithm: Laplacian of Gaussian Star Detection

Algorithm 4 Star Detection using LoG

$\sigma \in \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$
Blobs $= \{(x, y, r) \mid |\text{LoG}(x, y, \sigma)| > T\}$
for $(x, y, r) \in$ Blobs **do**
  if $r > 2$ **then**
    discard (x, y, r)
  end if
end for
return $\{(x, y, r)\}$ for valid star coordinates

7 Counting Detected Stars

The total count of stars C_{stars} is the cardinality of valid blobs:

$$C_{\text{stars}} = |\text{Blobs}|$$

7.1 Algorithm: Counting Stars

Algorithm 5 Count Detected Stars

$C_{\text{stars}} \leftarrow 0$
for $(x, y, r) \in$ Blobs **do**
  $C_{\text{stars}} \leftarrow C_{\text{stars}} + 1$
end for
return C_{stars}

8 Results and Visualization

Each detected star (x, y, r) is displayed on the image:

$$\text{DrawCircle}(x, y, r)$$

8.1 Algorithm: Visualization of Detected Stars

Algorithm 6 Visualize Detected Stars

for $(x, y, r) \in$ Blobs **do**
  DrawCircle(x, y, r)
end for
Display the image with circles around detected stars.

9 Enhanced Star Detection with Near-Black Filtering

In astronomical images, artifacts or noise may appear in regions near black, which can interfere with accurate star detection. To mitigate this, I apply a near-black filtering threshold to ignore regions with pixel intensities close to zero, helping focus detection only on relevant, visible stars.

9.1 Algorithm: Star Detection with Near-Black Filtering

Let T_{black} denote a threshold to identify pixels close to black, with $T_{\text{black}} \in [0, 1]$. For each detected blob, if its grayscale intensity $G(x, y)$ at coordinates (x, y) is below T_{black}, it is excluded from detection. This process is formalized as follows:

1. Define the near-black threshold T_{black}, with a recommended value $T_{\text{black}} = 0.1$. 2. For each blob (candidate star) at coordinates (x, y), verify that:

$$G(x, y) > T_{\text{black}}$$

If this condition holds, the blob is counted as a valid star.

Algorithm 7 Enhanced Star Detection with Near-Black Filtering

Define near-black threshold $T_{\text{black}} \leftarrow 0.1$
Detect blobs in the grayscale image using Laplacian of Gaussian (LoG)
Initialize star count $C_{\text{stars}} \leftarrow 0$
for each blob $(x, y, r) \in$ Blobs **do**
  if $G(x, y) > T_{\text{black}}$ **then**
    Increment star count $C_{\text{stars}} \leftarrow C_{\text{stars}} + 1$
    Draw circle around (x, y) with radius r for visualization
  end if
end for
return C_{stars} as the count of detected stars

The updated count of detected stars, C_{stars}, thus excludes blobs detected near black regions, ensuring a more accurate count of visible stars:

$$C_{\text{stars}} = |\{(x, y, r) \mid \text{Blob}(x, y, r) \text{ and } G(x, y) > T_{\text{black}}\}|$$

9.2 Mathematical Formulation of Enhanced Detection

Given a grayscale intensity function $G(x, y)$, the enhanced detection function $f(x, y)$ can be represented as:

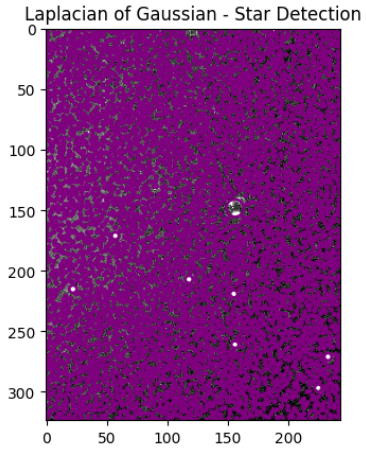
$$f(x, y) = \begin{cases} 1 & \text{if } |\text{LoG}(x, y, \sigma)| > T \text{ and } G(x, y) > T_{\text{black}} \\ 0 & \text{otherwise} \end{cases}$$

where $f(x, y) = 1$ indicates a detected star.

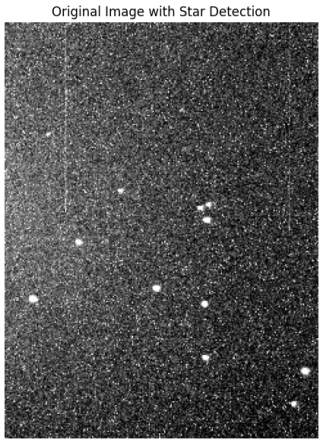
This enhanced approach filters out regions near black, allowing for a refined star count C_{stars} and improved accuracy in star detection.

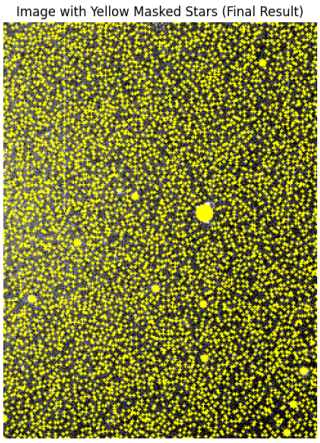
10 Figures: Original and Filtered Images

In this section, we present visual comparisons of the original image, the Gaussian-filtered image, and the enhanced Gaussian-filtered image with near-black filtering applied.



(a) Gaussian Filtered Image





(b) Original and Enhanced Gaussian Filtered Image

Figure 2: Comparison of Original, Gaussian-filtered, and Enhanced Gaussian-filtered images.

Big Stars Counting

The goal of this algorithm is to count the number of larger stars in a grayscale image and determine their locations. It specifically aims to exclude small, similar stars that may be detected as noise. The algorithm accomplishes this by isolating contours that meet a certain size threshold and calculating the centroids of each identified star.

Exclusion of Small Stars

To achieve accurate counting, the algorithm applies filtering techniques to ignore smaller stars that could be mistaken for noise, ensuring that only significant bright spots are counted.

Mathematical Formulation

1. Image Loading and Grayscale Conversion

Let I denote the input image, which contains bright spots representing stars against a dark background. The image is loaded in grayscale, resulting in a matrix I_g where each pixel $I_g(i, j)$ represents the brightness at location (i, j).

2. Binary Thresholding

To isolate the stars from the background, we apply a binary threshold T to the grayscale image. This thresholding operation sets pixel values above $T = 200$ to 255 (white) and those below T to 0 (black), producing a binary image I_b:

$$I_b(i,j) = \begin{cases} 255, & \text{if } I_g(i,j) \geq T, \\ 0, & \text{if } I_g(i,j) < T. \end{cases}$$

This results in a high-contrast image where the larger stars are represented as distinct white regions.

3. Contour Detection

Contours in the binary image I_b are then detected, which represent the boundaries of each bright region. Let $C = \{c_1, c_2, \ldots, c_n\}$ denote the set of detected contours, where each c_i represents a closed curve around a bright spot (potential star) in the image.

4. Bilateral Filter

The bilateral filter is a non-linear, edge-preserving, and noise-reducing smoothing filter that applies Gaussian weighting to both spatial proximity and intensity similarity. It is particularly useful for reducing noise in images while preserving edge details. The bilateral filter is defined mathematically as follows:

$$I_{\text{filtered}}(x) = \frac{1}{W(x)} \sum_{y \in \Omega} I(y) \cdot \exp\left(-\frac{\|x-y\|^2}{2\sigma_{\text{space}}^2}\right) \cdot \exp\left(-\frac{|I(x)-I(y)|^2}{2\sigma_{\text{color}}^2}\right) \tag{1}$$

where:

- $I(x)$ is the original intensity of the pixel at position x.
- $I_{\text{filtered}}(x)$ is the intensity of the pixel at position x after applying the bilateral filter.
- $W(x)$ is a normalization factor to ensure the weights sum up to 1:

$$W(x) = \sum_{y \in \Omega} \exp\left(-\frac{\|x-y\|^2}{2\sigma_{\text{space}}^2}\right) \cdot \exp\left(-\frac{|I(x)-I(y)|^2}{2\sigma_{\text{color}}^2}\right) \tag{2}$$

- Ω denotes the neighborhood of pixel x that the filter considers.
- σ_{space} controls the spatial closeness weight, making pixels closer to x have a stronger influence.
- σ_{color} controls the color similarity weight, making pixels with similar intensities to $I(x)$ have a stronger influence.

In the context of the code, the parameters used for the bilateral filter are:

- $d = 9$: The diameter of the pixel neighborhood.
- $\sigma_{\text{color}} = 50$: The color standard deviation.
- $\sigma_{\text{space}} = 75$: The spatial standard deviation.

This bilateral filtering approach helps to reduce image noise while maintaining sharp edges, which is particularly advantageous for the contour detection tasks in the star detection algorithm.

5. Filtering Contours by Area

To focus on larger stars, each contour c_i is analyzed for its area, $A(c_i)$, defined by:

$$A(c_i) = \sum_{(x,y) \in c_i} 1.$$

Only contours with area $A(c_i) > A_{\min}$ are considered stars, where $A_{\min} = 6$ is a threshold set to filter out small, irrelevant contours.

6. Centroid Calculation

For each contour c_i that meets the area threshold, the centroid (x_i, y_i) is calculated using image moments. The spatial moments M_{pq} for a contour c_i are defined as:

$$M_{pq} = \sum_{(x,y)\in c_i} x^p y^q.$$

The centroid coordinates (x_i, y_i) of each star are then given by:

$$x_i = \frac{M_{10}}{M_{00}}, \quad y_i = \frac{M_{01}}{M_{00}},$$

where M_{00} is the area of c_i, and M_{10} and M_{01} are the first-order moments.

7. Counting Stars and Output

The total number of stars N is the count of contours meeting the area threshold:

$$N = \sum_{i=1}^{n} \mathbf{1}(A(c_i) > A_{\min}),$$

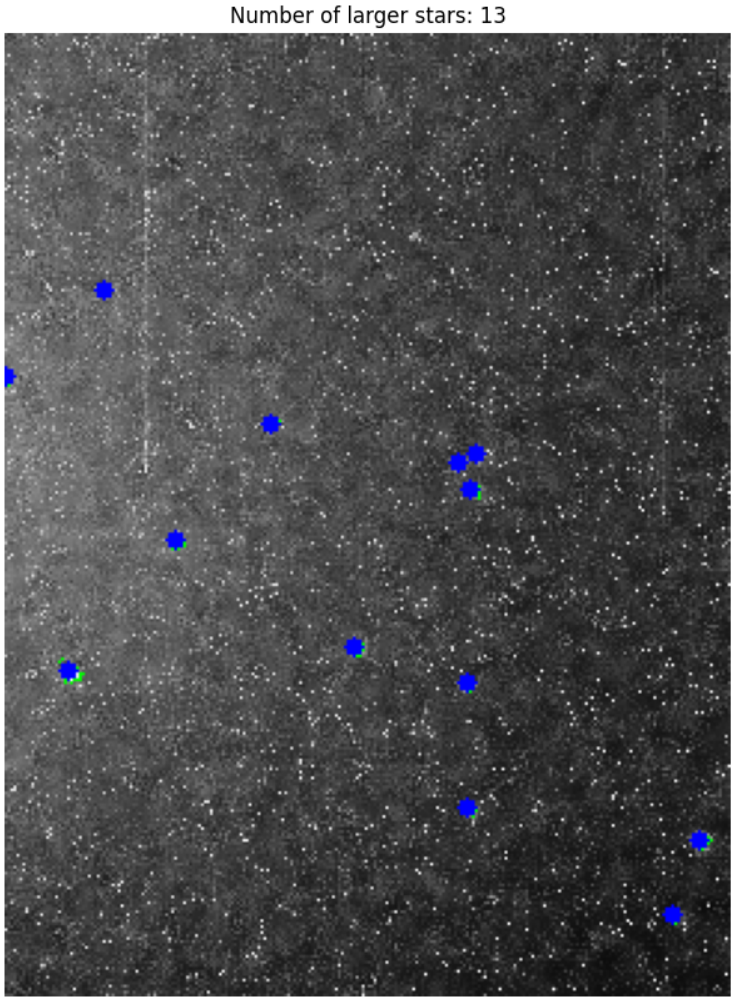
where $\mathbf{1}(\cdot)$ is the indicator function. The centroids (x_i, y_i) of each detected star are stored for location reporting.

Star Count	**Coordinates of Detected Stars**
13	(224, 296), (233, 271), (155, 260), (155, 218), (21, 214), (117, 206), (57, 170), (156, 153), (152, 144), (158, 141), (89, 131), (0, 115), (33, 86)

Table 1: Summary of Larger Detected Stars and Their Coordinates

11 Conclusion

This document presents a theoretical approach for detecting and counting stars using data parsing, grayscale conversion, and the Laplacian of Gaussian (LoG) method. These methods effectively locate and count star-like structures within an image, outputting both the locations and total count. Star 4989: (94, 215) Star 4990: (100, 153) Star 4991: (236, 196) Number of detected stars: 4991. Big stars counting is provided in last section. For more information and code, visit the following Google Colab notebook: Colab Notebook Link.



(a) Big stars masking

Non-Local Means Filter Analysis and Implementation

Rashin Rahnamoun

HW2 Vision Research Q3

1 Understanding the Non-Local Means Filter

1.1 Working Principle and Formula

The Non-Local Means (NLM) filter, introduced by Buades et al., operates on the principle that similar pixels in an image should contribute more to the denoising process. Unlike local filters that only consider spatially adjacent pixels, NLM searches for similar patterns across the entire image (or a large search window).

The filtered value for a pixel x is computed as:

$$\hat{I}(x) = \frac{\sum_{y \in S_x} w(x,y) I(y)}{\sum_{y \in S_x} w(x,y)} \tag{1}$$

where:

- $I(y)$ is the intensity of pixel y
- S_x is the search window centered at pixel x
- $w(x,y)$ is the weight function measuring similarity between pixels x and y

The weight function is defined as:

$$w(x,y) = \exp\left(-\frac{\|N_x - N_y\|_2^2}{h^2}\right) \tag{2}$$

where:

- N_x and N_y are neighborhoods (patches) centered at pixels x and y
- h is the filtering parameter controlling decay of the exponential function
- $\|\cdot\|_2^2$ denotes the Euclidean distance

1.2 Comparison with Averaging Filter

The key differences between NLM and the traditional averaging filter are:

1. **Non-locality**: While averaging filters only consider local neighborhoods, NLM searches for similar patterns across a larger region.
2. **Patch-based comparison**: NLM compares entire patches around pixels rather than just individual pixel values.
3. **Adaptive weighting**: The weights in NLM are computed based on patch similarity, making it more selective in averaging compared to uniform weights in averaging filters.
4. **Edge preservation**: Due to patch comparison and adaptive weighting, NLM preserves edges and textures better than averaging filters.

2 Advantages and Disadvantages

2.1 Advantages

1. **Superior denoising quality**: By considering similar patterns across the image, NLM achieves better noise reduction while preserving important details.
2. **Edge preservation**: The patch-based comparison helps maintain sharp edges and fine textures.
3. **Robustness**: Works well with different types of noise and image content.
4. **No blur effect**: Unlike traditional filters, NLM doesn't introduce significant blurring artifacts.

2.2 Disadvantages

1. **Computational complexity**: The original algorithm has high computational cost of $O(N^2(2S+1)^2(2W+1)^2)$ where:
   - N is the image size
   - S is the search window radius
   - W is the patch radius
2. **Parameter sensitivity**: The filter's performance depends on proper selection of parameters (patch size, search window size, and h).
3. **Memory requirements**: Processing large images requires significant memory resources.

3 GPU Implementation Analysis

Based on the provided research, the GPU implementation involves several optimization strategies:

3.1 Key Optimization Strategies

1. **Computational Complexity Reduction**: The implementation uses Condat's algorithm, which reduces complexity through:
   - Exploiting weight symmetry: $w(x, x+dx) = w(y, y-dx)$
     This relationship allows the algorithm to reduce redundant calculations, thus enhancing performance.
   - Using separable convolutions for patch difference computation.
     This technique simplifies the convolution operation into two one-dimensional convolutions, reducing computational overhead.
2. **Memory Access Optimization**:
   - Coalescing memory access using `cudaMallocPitch()` and `cudaMemcpy2D()`.
     These functions ensure that memory allocations are optimized for coalesced access patterns, improving data throughput.

- Using 2D textures for unaligned reads.
  Textures allow cached memory access which can be faster for spatially localized data, mitigating the penalty of unaligned memory accesses.
- Optimizing write operations for better memory performance.
  Adjusting how data is written back to memory can reduce the number of memory transactions required, enhancing performance.

3.2 Naive Implementation

The naive implementation consists of a straightforward implementation of Condat's algorithm. The host code controls the iteration through the search region while GPU kernel functions are invoked for displaced image subtraction, separable convolution, addition of weighted pixel contributions, and finally division of the contributions by the total summed weights. The pseudocode for this approach is described below:

Algorithm 1 Naive Implementation of Condat's Algorithm

1: Initialize $\hat{I}(x), \hat{I}_{sym} = 0, C, C_{sym} = 0$
2: **for** all dx in halved search region **do**
3: $u \leftarrow$ displaced image subtraction kernel(I, dx)
4: $v \leftarrow$ separable convolution kernel(u)
5: $(I, \hat{I}, C, C_{sym}) \leftarrow$ add weighted pixel contributions kernel(I, v, dx)
6: **end for**
7: $\hat{I}(x) \leftarrow$ weight normalization kernel$(\hat{I}(x), \hat{I}_{sym}, C(x), C_{sym})$

For the separable convolution, the CUDA Toolkit sample code has used . The other kernels are straightforward implementations of their CPU counterparts. The only addition is that, since a thread with linearized index x updates $\hat{I}(x)$ and $C(x)$ as well as $\hat{I}(x + dx)$ and $C(x + dx)$, it introduced another pair of accumulation and summed weight images $\hat{I}_{sym}$ and C_{sym} to store the symmetric contributions and thus eliminate concurrency overwrite issues between threads.

3.3 Implementation Algorithm

The overall algorithm for the GPU-optimized Non-Local Means (NLM) filter is summarized as follows:

Algorithm 2 GPU-Optimized NLM

Initialize output images $\hat{I}(.)$ and $C(.) = 0$
for dx in halved search region **do**
  Compute $u(x) = (I(x) - I(x + dx))^2$
  Compute $v(x) = u(x) * g$ using separable convolution
  for all pixels x **do**
    $w(x + dx) = \exp(-v(x)/h^2)$
    $\hat{I}(x) += w(x + dx)I(x + dx)$
    $\hat{I}(x + dx) += w(x + dx)I(x)$
    $C(x) += w(x + dx)$
    $C(x + dx) += w(x + dx)$
  end for
end for
$\hat{I}(x) = \hat{I}(x)/C(x)$ for all pixels x

3.4 Performance Optimizations

The performance optimizations in our GPU implementation involve several strategies:

- **Coalescing Memory Access**: To improve memory access speed, we utilized the functions `cudaMallocPitch()` and `cudaMemcpy2D()` for better memory alignment. This ensures that global memory accesses are coalesced, improving throughput and reducing latency.
- **Using 2D Textures**: The use of textures allows for cached memory access patterns, which are beneficial when reading unaligned pixel data, thereby increasing the effective memory bandwidth and speeding up computation.
- **Optimizing Write Operations**: By changing the strategy for writing contributions to pixel values, we ensured that all writes could be coalesced, which reduced the overhead associated with managing additional symmetric images.

4 Performance Analysis

The performance of the different implementations of the Non-Local Means (NLM) algorithm was evaluated on both CPU and GPU platforms. The results are summarized in Tables 1 and 2.

4.1 CPU Implementation

As shown in Table 1, the Naive Implementation on the CPU takes the longest time at 10.41 seconds. This is expected due to the straightforward approach that does not utilize any optimization techniques.

In contrast, the Symmetric Implementation significantly reduces the computation time to 2.59 seconds, demonstrating that optimizing the processing order can enhance performance. The most efficient CPU approach is the Condat Implementation, which completes in just 0.12 seconds. This dramatic improvement indicates that the Condat method leverages advanced strategies to reduce the computational complexity, making it the preferred choice for CPU-based processing.

Table 1: CPU Implementation Times

Implementation	**Time (seconds)**
Naive Implementation	10.41
Symmetric Implementation	2.59
Condat Implementation	0.12

4.2 GPU Implementation

Table 2 highlights the performance of various GPU implementations. The Naive Implementation still appears relatively efficient at 1.07 seconds; however, the time is considerably lower than the CPU's Naive Implementation. This showcases the GPU's inherent advantage in parallel processing capabilities.

The Coalesced Implementation further optimizes the processing time to 0.63 seconds, while the Texture Memory Implementation achieves 0.49 seconds. Both these approaches illustrate how efficient memory access patterns can significantly enhance performance on GPU architectures.

Finally, the Write Coalesced Implementation shows the best performance among all implementations at 0.47 seconds. This indicates that optimizing memory writes is crucial for achieving the highest efficiency in GPU-based computations.

Table 2: GPU Implementation Times

Implementation	**Time (seconds)**
Naive Implementation	1.07
Coalesced Implementation	0.63
Texture Memory Implementation	0.49
Write Coalesced Implementation	0.47

In summary, the results reveal that GPU implementations consistently outperform their CPU counterparts across all scenarios. However, it is important to note that the performance differences are influenced by the specific input images used, as highlighted in related literature. Despite these variations, the overall trends indicate that advanced techniques such as coalescing and utilizing texture memory are essential for maximizing performance on GPUs. The analysis suggests that for computationally intensive tasks like the NLM algorithm, leveraging GPU resources and optimized algorithms can yield substantial gains in processing speed

5 Image Denoising using PSNR and SSIM

This report evaluates the quality of denoised images using Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) across various noise levels (σ) in Gaussian noise. We analyze PSNR and SSIM metrics mathematically and present results for each image. Additionally, graphical plots provide visual representation of the results.

6 Image Quality Metrics

6.1 Peak Signal-to-Noise Ratio (PSNR)

PSNR is a widely used metric for assessing the quality of image reconstructions. It is defined based on the Mean Squared Error (MSE) between a

reference image f and a test image g:

$$\text{PSNR}(f, g) = 10 \cdot \log_{10} \left(\frac{255^2}{\text{MSE}(f, g)} \right) \tag{3}$$

where 255 is the maximum possible pixel value (for 8-bit images) and the MSE is calculated as:

$$\text{MSE}(f, g) = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} (f_{ij} - g_{ij})^2 \tag{4}$$

Higher PSNR values indicate lower error and higher similarity to the reference image.

6.2 Structural Similarity Index (SSIM)

The SSIM metric compares luminance, contrast, and structure between two images, yielding values in $[0, 1]$ where higher values indicate greater similarity. The SSIM is given by:

$$\text{SSIM}(f, g) = \frac{(2\mu_f \mu_g + C_1)(2\sigma_{fg} + C_2)}{(\mu_f^2 + \mu_g^2 + C_1)(\sigma_f^2 + \sigma_g^2 + C_2)} \tag{5}$$

where:

- μ_f and μ_g are the mean intensities of f and g,
- σ_f and σ_g are the standard deviations of f and g,
- σ_{fg} is the covariance between f and g,
- C_1 and C_2 are constants to stabilize the division.

7 Results and Analysis

For each test image, results are presented across different noise levels. For each image, we include a table of PSNR and SSIM metrics, a plot of these metrics across noise levels, and the processed image results in a PNG file.

7.1 Results for *bird.jpg*

Table 3: Quality Metrics for *bird.jpg*

Noise Level (σ)	**Noisy Image**		**Denoised Image**	
	PSNR	**SSIM**	**PSNR**	**SSIM**
0.05	29.53	0.583	34.04	0.941
0.10	23.58	0.323	29.72	0.873
0.15	20.29	0.217	25.13	0.609
0.20	18.06	0.158	19.23	0.193

Figure 1: PSNR and SSIM values for *bird.jpg* across noise levels.

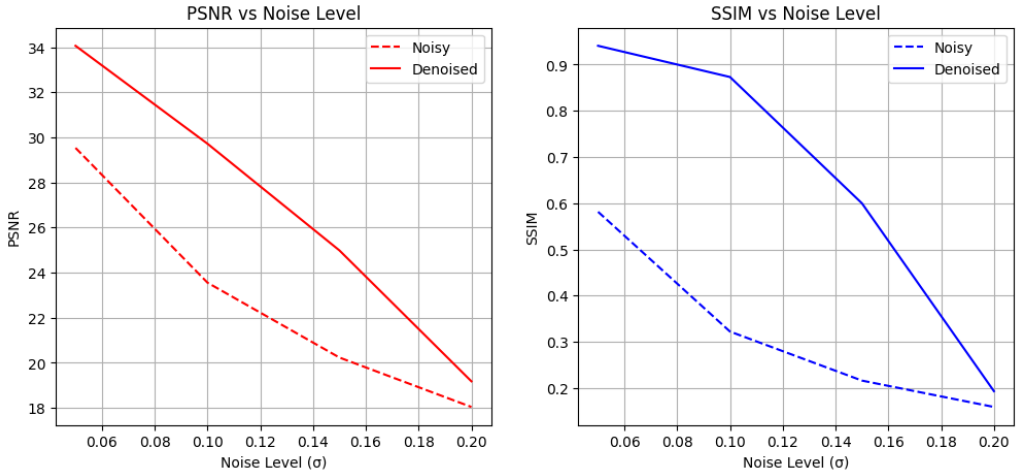


Figure 2: PSNR and SSIM values for *bird.jpg* across noise levels.

7.2 Results for *vegetables.jpg*

Table 4: Quality Metrics for *vegetables.jpg*

Noise Level (σ)	**Noisy Image**		**Denoised Image**	
	PSNR	**SSIM**	**PSNR**	**SSIM**
0.05	29.65	0.649	33.49	0.889
0.10	23.96	0.383	24.76	0.462
0.15	20.80	0.256	20.99	0.279
0.20	18.64	0.187	18.70	0.197

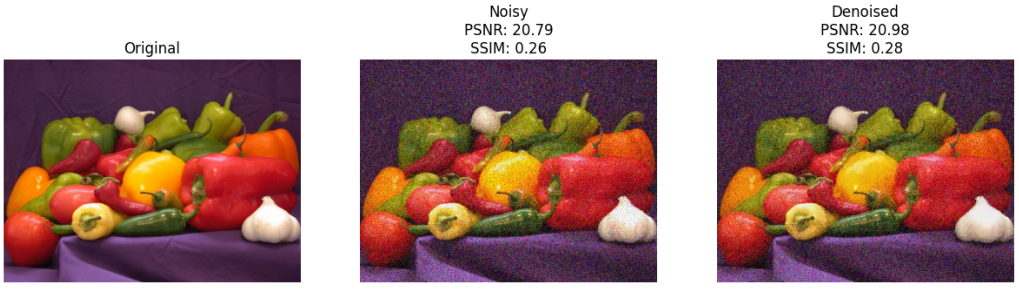


Figure 3: PSNR and SSIM values for *vegetables.jpg* across noise levels.

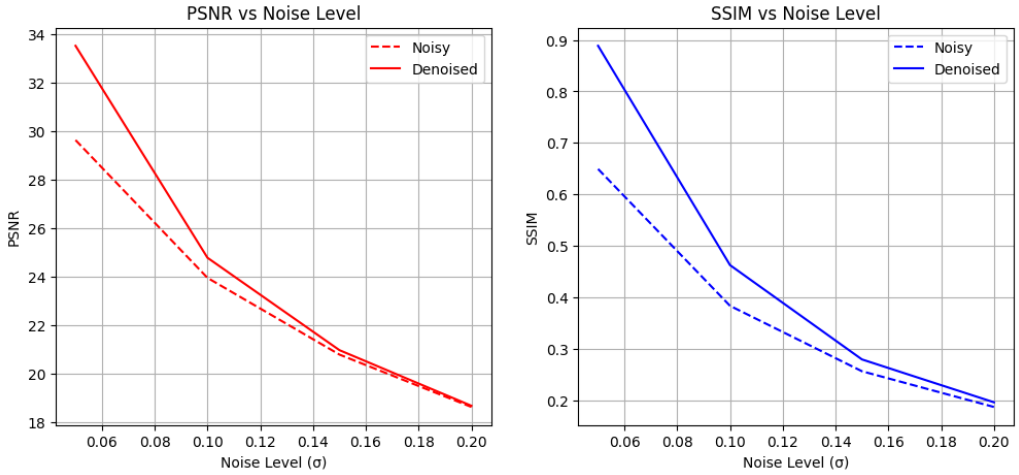


Figure 4: PSNR and SSIM values for *vegetables.jpg* across noise levels.

7.3 Results for *woman.jpg*

Table 5: Quality Metrics for *woman.jpg*

Noise Level (σ)	**Noisy Image**		**Denoised Image**	
	PSNR	**SSIM**	**PSNR**	**SSIM**
0.05	29.59	0.777	31.34	0.899
0.10	23.75	0.560	25.93	0.761
0.15	20.47	0.435	21.10	0.488
0.20	18.17	0.348	18.25	0.354

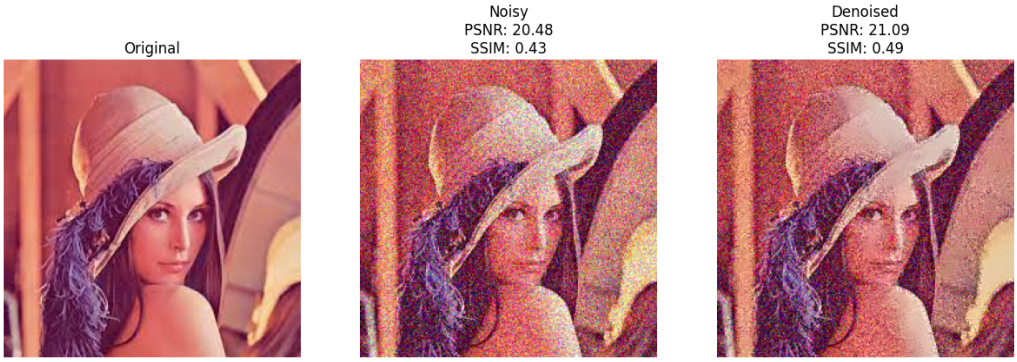


Figure 5: PSNR and SSIM values for *woman.jpg* across noise levels.

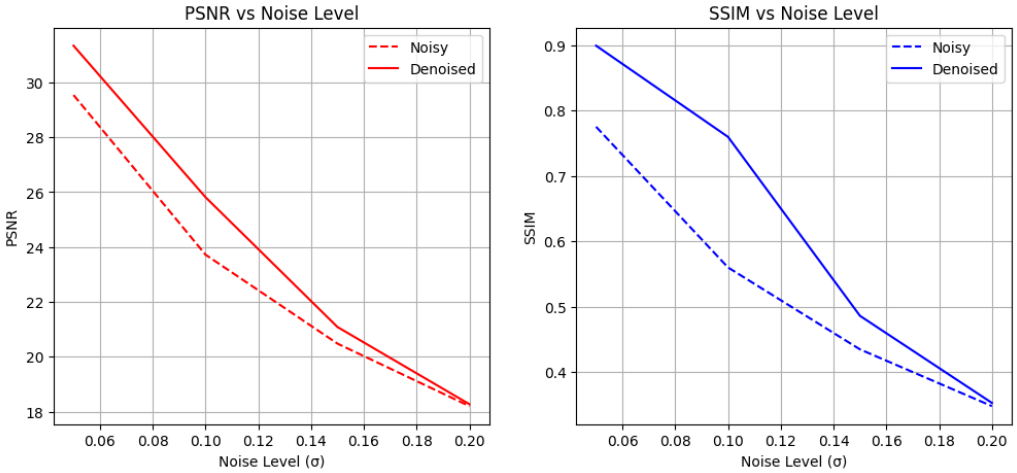


Figure 6: PSNR and SSIM values for *woman.jpg* across noise levels.

8 GPU Results

8.1 GPU Results for *bird.jpg*

Table 6: Quality Metrics for *bird.jpg*

Noise Level (σ)	**Noisy Image**		**Denoised Image**	
	PSNR	**SSIM**	**PSNR**	**SSIM**
0.05	22.72	0.56	29.54	0.58
0.10	20.74	0.31	23.58	0.32
0.15	18.67	0.21	20.22	0.22
0.20	17.03	0.15	18.05	0.16

8.2 GPU Results for *vegetables.jpg*

Table 7: Quality Metrics for *vegetables.jpg*

Noise Level (σ)	**Noisy Image**		**Denoised Image**	
	PSNR	**SSIM**	**PSNR**	**SSIM**
0.05	21.91	0.62	29.65	0.65
0.10	20.46	0.37	23.94	0.38
0.15	18.97	0.25	20.82	0.26
0.20	17.58	0.18	18.62	0.19

8.3 GPU Results for *woman.jpg*

Table 8: Quality Metrics for *woman.jpg*

Noise Level (σ)	**Noisy Image**		**Denoised Image**	
	PSNR	**SSIM**	**PSNR**	**SSIM**
0.05	23.16	0.77	29.57	0.78
0.10	20.94	0.55	23.72	0.56
0.15	18.89	0.43	20.42	0.43
0.20	17.20	0.34	18.26	0.35

9 Discussion

The PSNR and SSIM values exhibit different trends as noise level (σ) increases. Denoised images consistently show higher PSNR and SSIM values than noisy images. Notably:

- **PSNR**: A significant increase in PSNR for denoised images compared to noisy ones is observed, especially at lower noise levels. This suggests that the denoising algorithm effectively reduces noise.
- **SSIM**: SSIM improvements after denoising are particularly notable at lower noise levels ($\sigma = 0.05$ and $\sigma = 0.10$), indicating structural similarity restoration.

While the GPU implementation of the denoising algorithm demonstrates significantly faster processing times, it falls short in terms of image quality. The results indicate that the denoising effectiveness, as measured by PSNR and SSIM, is inferior to that of the cv2.fastNlMeansDenoising function. This suggests that despite the speed advantage offered by the GPU, the quality of the denoised images may not meet the standards required for applications where preserving image integrity is crucial.

10 Conclusion & Practical Results

The Non-Local Means filter provides excellent denoising results while preserving image details, though at a high computational cost. The GPU im-

plementation successfully addresses the computational burden, making the algorithm practical for real-world applications. The achieved speedup factors align well with those reported in the literature, demonstrating the effectiveness of the optimization strategies employed. For more information and code, visit the following Google Colab notebook: Colab Notebook Link.