Politecnico di Torino

GPU programming
01URVOV

Master's Degree in Computer Engineering

Image processing algorithms in video

A comparison between CPU and GPU

Candidate:
Diego Porto (s313169)

Referee:
Prof. Riccardo Cantoro
Josie Esteban Rodriguez Condia
Juan David Guerrero Balaguera

Abstract

Nowadays, image processing algorithms are increasing in importance for several reasons:

- Improving Image Quality: for example resizing
- Extracting Meaningful Information:
  - Edge Detection: Identifies boundaries and contours in images, helpful for object recognition and segmentation.
  - Binarization: Converts grayscale images into binary (black and white) images, simplifying analysis and feature extraction.
- Enabling Computer Vision Applications:
  - Object Detection: Locates and classifies objects within images, enabling applications like autonomous vehicles and surveillance systems.
  - Medical Image Analysis: Assists in diagnosing diseases by analyzing medical images like X-rays, CT scans, and MRIs.
  - Remote Sensing: Processes satellite and aerial images to monitor environmental changes, urban development, and natural disasters.

As technology advances and image quality improves, it becomes increasingly complicated for CPUs to be able to analyze large amount of data in near-real time. To overcome this problem, it becomes increasingly important to delegate these onerous tasks to GPUs. In fact, GPUs excel at parallel processing, making them significantly faster than CPUs for image processing tasks that involve numerous repetitive calculations across large datasets, such as filtering, convolution, and matrix operations.

The goal of this project is to apply at the same time different image processing algorithms to a video, and to compare the results obtained from the implementation on CPU and GPU (naive version and optimized version). By executing these algorithms on both platforms and measuring their performance across different videos, the projects aim to identify potential advantages and disadvantages of GPU-accelerated image processing.

The algorithms that will be analyzed are:

- Laplacian
- Gaussian Blur
- Bicubic interpolation
- Bilateral filtering

CHAPTER 1

Filters - mathematical aspects

1.1 Laplacian filter

The Laplacian Filter is a tool in image and video processing used for edge detection and sharpening. It is a second-order derivative filter that highlights regions of rapid intensity change in an image, those region indeed usually correspond to edges.

The Laplacian operator is defined as the sum of the second derivatives of the image intensity function $I(x, y)$:

$$\nabla^2 I(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{1.1}$$

In discrete terms, the Laplacian can be approximated using convolution with a kernel. The most common Laplacian kernels include:

- 3x3 kernel with positive center $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
- 3x3 kernel with negative center $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$
- 5x5 kernel $\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$

The idea behind the formulation of the kernel is that in all the cases the sum of values in the filter should be 0.

This method works fine on images for finding edges in both directions, but it works poorly when noise is found in the image. So, it is usally combined with the Guassian filter prior to the Laplacian filter. It is often termed Laplacian of Guassian (LoG) filter.

1.2 Gaussian filter

Gaussian Filter is a widely used technique in image and video processing to smooth and blur images by reducing noise and details. It is a type of convolution filter that applies a weighted average to each

pixel in an image, where the weights are determined by a Gaussian distribution.

The Gaussian Filter is defined by the Gaussian function, which is a bell-shaped curve given by:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1.2}$$

Where:

- (x, y) are the coordinates of a pixel relative to the center of the filter
- σ is the standard deviation of the Gaussain distribution, which controls the amount of smoothing. Larger σ results in a more blurred image.
- $G(x, y)$: the result is the weight assigned to the pixel (x, y).

The filter is implemented as a kernel that is convolved with the image. The size of the kernel is usally odd to ensure a well-defined center.

The Gaussian Filter in computer vision is often used for smoothing images before edge detection or feature extraction.

1.3 Bicubic interpolation

The Bicubic Interpolation is a high-quality resampling technique used in image and video processing to resize or scale images while preserving detail and smoothness. Unlike simpler interpolation methods like bilinear or nearest-neighbor, bicubic interpolation considers a neighborhood of pixels (16 pixels in a 4x4 grid) to compute the interpolated value, resulting in smoother and more visually appealing results.

Bicubic interpolation uses a cubic polynomial to approximate the intensity of pixels in a 4x4 grid. The pixels that are closer to the one that is to be estimated are given higher weights as compared to those that are further away. Therefore, the farthest pixels have the smallest amount of weight.

The general formula of bicubic interpolation is:

$$I(x,y) = \sum_{i=-1}^{2} \sum_{j=-1}^{2} I(x_i, y_j) \bullet W(x - x_i) \bullet W(y - y_j) \tag{1.3}$$

Where:

- $I(x_i, y_j)$: is the intensity value of the pixel at coordinates (x_i, y_j).
- $W(t)$: is the cubic weighting function

The cubic weighting function used in this project is the following one:

$$W(t) = \begin{cases} (a+2)|t|^3 - (a+3)|t|^2 + 1 & \text{if } |t| \leq 1 \\ a|t|^3 - 5a|t|^2 + 8a|t| - 4a & \text{if } 1 < |t| < 2 \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

Where:

- t is the distance from the interpolation point to the pixel
- a: is a parameter that controls the sharpness of the interpolation, the range is often between -0.5 and -0.75.

1.4 Bilateral filtering

The Bilateral Filter is a edge-preserving smoothing filter widely used in image and video processing. Unlike the Gaussian filter, which as previously said blur an image uniformly, the bilateral filter smooths an image while preserving edges by considering both the spatial distance and intensity difference between pixels. This makes it particularly useful for tasks like noise reduction, detail enhancement, and texture smoothing. The bilateral filter computes the weighted average of pixels in based on two factors:

- Spatial distance: how close the pixel is to the center pixel
- Intensity difference: how similar the pixel's intensity is to the center pixel's intensity.

The filtered value $I_{filtered}(x, y)$ at pixel (x, y) is given by the following formula:

$$I_{filtered}(x, y) = \frac{1}{W_{norm}} \sum_{i,j \in \Omega} I(i, j) \bullet G_{\sigma_s}(i - x, j - y) \bullet G_{\sigma_r}(I(i, j) - I(x, y))$$

Where:

- $I(i, j)$: is the intensity of the pixel at coordinates (i, j).
- Ω: is the neighborhood aroud the coordinates (x, y).
- G_{σ_s} : is the spatial Gaussian kernel which depends on the Euclidean distance between (i, j) and (x, y).
- G_{σ_r}: is the range Gaussian kernel which depends on the intensity difference between the intensity $I(i, j)$ and $I(x, y)$.
- W_{norm}: is the normalization factor, to ensure the weights sum to 1.

The two Gaussian kernels are defined as:

$$G_{sigma_s}(i - x, j - y) = e^{-\frac{(i-x)^2+(j-y)^2}{2\sigma_r^2}} \tag{1.5}$$

And

$$G_{sigma_r}(I(i, j) - I(x, y)) = e^{-\frac{I(i,j)-I(x,y)^2}{2\sigma_r^2}} \tag{1.6}$$

- σ_s: controls the spatial spread (larger σ_s means a larger neighborhood).
- σ_r: Controls the range spread (larger σ_r means more intensity variation is allowed).

CHAPTER 2

GPU implementation

The program is a GPU-accelerated video processing application with a GUI, written with Tkinter, that applies in real-time the four filters described before to a video uploaded by the user.

Different versions of the program were written:

- `cpu_main.py`: is the program used in the naive version that runs entirely on the host.
- `main_naive.py`: is the implementation of the program that runs on the GPU
- `main_profiler.py`: is a modified version of the program that runs without the GUI. This version was used to produce the results automatically via a bash script (`gpu_compute_all_videos.sh`).
- `main_optimized.py`: is a modified version of the original program in an attempt to improve performance over the naive version.
- `main_profiler_optimized.py`: is the optimized program that runs without GUI, similarly to `main_profiler.py`.

2.1 Naive version: `main_naive.py`

Tkinter was used to create a window with 5 panels: the uploaded video on the left and the four filtered outputs on the right.

Once the user selects the video to upload, this is divided in frames and each of them is passed as a parameter to the filters to be elaborated. Each filter extends the class VideoFrame and is responsible for processing each frame through a GPU kernel. The frames are transferred to the GPU using `(cuda.to_device())` The results of the kernels are the filtered images which are then copied back to the CPU to be displayed in the GUI (`d_output.copy_to_host()`).

During the execution of the program the elapsed time to elaborate each frame is computed and stored in logs files (one for each filter) using cuda events:

```
# Create CUDA events for timing
start_event = cuda.event()
end_event = cuda.event()

# Record the start event
start_event.record()

# Record the end event
```

```
    end_event.record()

    # Wait for the end event to complete
    end_event.synchronize()

    # Calculate the elapsed time in milliseconds
    elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)
```

In the case of bicubic interpolation filter, mean squared error (MSE) and peak signal-to-noise Ratio (PSNR) information are also recorded to quantify image quality after processing.

2.1.1 Laplacian filter

The GPu kernel that handles the computation of the Laplacian filter is `laplacianFilter` and it takes in input the pointers to the original frame, the result, and the size of the frame (width, height). Each GPU thread handles one pixel of the output image:

```
x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
```

The kernel used is a 3x3 with a negative center defined as follows:

```
    kernel = cuda.local.array((3, 3), dtype=np.float32)

    kernel[0, 0] = 0
    kernel[0, 1] = 1
    kernel[0, 2] = 0
    kernel[1, 0] = 1
    kernel[1, 1] = -4
    kernel[1, 2] = 1
    kernel[2, 0] = 0
    kernel[2, 1] = 1
    kernel[2, 2] = 0
```

The kernel is stored in the global memory and it is private to the current thread. The variable `sum` is used to store the value of the output pixel computed as the sum of the neighboring pixels after the multiplication with the 3x3 kernel.

```
for kx in range(-FILTER_WIDTH // 2, FILTER_WIDTH // 2 + 1):
    for ky in range(-FILTER_HEIGHT // 2, FILTER_HEIGHT // 2 + 1):
        fl : float = frame[x + kx][y + ky]
        sum += fl * kernel[kx + FILTER_WIDTH // 2][ky + FILTER_HEIGHT // 2]
result[x][y] = sum
```

2.1.2 Gaussian blur

To implement the Gaussian blur filter, two main CUDA kernels were used. One used to compute the Gaussian kernel on the device (`compute_gaussian_kernel(...)`) and the second one to compute the convolution between the image and the Gaussian kernel (`convolution(...)`.

```
compute_gaussian_kernel[blocks_per_grid, threads_per_block](d_kernel, sigma)
...
```

```
d_kernel = cuda.to_device(kernel)
...
# Perform convolution
convolution[blocks_per_grid, threads_per_block](frame, d_kernel, d_output)
...
```

In the first kernel, each GPU thread calculates one element of the kernel matrix using the Gaussian function (1.2).

In the second kernel, as for the previous filter 2.1.1, each thread processes one pixel of the output image. For each color channel of the image, the threads iterate over the kernel window and compute weighted sum as:

```
acc += oldimage[px, py, c] * kernel[x, y]
```

2.1.3 Bicubic interpolation

The scale factor used for the bicubic interpolation is defined in `main`, based on this value the size of the output frame is calculated and the memory on the device is allocated:

```
# Scale factor for the bicubic interpolation
scale = 2.0

# Calculate the dimensions of the scaled frame
scaled_height = int(frame.shape[0] * scale)
scaled_width = int(frame.shape[1] * scale)

# Allocate memory for the scaled frame on the GPU
d_output = cuda.device_array((scaled_height, scaled_width, frame.shape[2]), dtype=frame.dtype)
```

Inside the kernel the information about the scale and the coordinates on the output image are used to compute the coordinates on the original image as:

```
src_x = x / scale
src_y = y / scale

x1 = int(src_x)
y1 = int(src_y)
```

The offset, used to compute the weight, is calculated as the difference between the approximate integer value and the actual value of the division:

```
# Fractional offset (0  dx/dy < 1)
dx = src_x - x1
dy = src_y - y1
```

`dx` and `dy` are critical to determine how to blend pixels from the original image to compute the value of a pixel in the output image.

The function to compute the weight as seen in 1.4 is defined as follows:

```
def cubic_weight(t):
    """
    Cubic weight function: Calculates how much to "blend" nearby pixels using a smooth curve
    """
    a = -0.5    # This value determines the performance of the kernel and it lies between -0.5 and -0.7
```

```
    t = abs(t)
    if t <= 1:
        return (a + 2) * t**3 - (a + 3) * t**2 + 1
    elif t < 2:
        return a * t**3 - 5 * a * t**2 + 8 * a * t - 4 * a
    return 0
```

The weight computed with this function ensure that pixels closer to the exact source coordinate (`src_x`, `src_y`) contribute more for the final value of the output pixel, while distant pixels contribute less.

To access the pixels near the edges of the image the method `get_pixel_value(i,j,c)` is used, it replaces invalid coordinates with the nearest valid ones:

```
def get_pixel_value(i, j, c):
    if i < 0:
        i = 0
    elif i >= frame.shape[0]:
        i = frame.shape[0] - 1
    if j < 0:
        j = 0
    elif j >= frame.shape[1]:
        j = frame.shape[1] - 1
    return frame[i, j, c]
```

For each color channel, a 4x4 neighborhood is evaluated from the original image using cubic weights:

```
for c in range(3):
    value = 0.0
    for m in range(-1, 3):
        for n in range(-1, 3):
            weight = cubic_weight(m - dx) * cubic_weight(n - dy)
            value += get_pixel_value(x1 + m, y1 + n, c) * weight
    output[x, y, c] = min(max(int(value), 0), 255)
```

2.1.4 Bilateral filter

As in the other filters, each CUDA thread processes one pixel (`x`,`y`) in the input image. The parameters σ_s and σ_r are defined in the host code. σ_s is used in the CUDA kernel to determine the diameter of the neighborhood. Each thread iterate over the kernel window and compute the spatial and range component as defined respectively in 1.5 and 1.6:

```
# Spatial component: Gaussian weight based on Euclidean distance from the center:
spatial_dist = i**2 + j**2
G_s = math.exp(-spatial_dist / (2 * sigma_s**2))

# Range component: Gaussian weight based on intensity difference (RGB Euclidean distance)
intensity_dist = (current_r - center_r)**2 + \
                 (current_g - center_g)**2 + \
                 (current_b - center_b)**2
G_r = math.exp(-intensity_dist / (2 * sigma_r**2))
```

The weight is given by the combination of spatial and range components:

```
weight = G_s * G_r
```

As we can see, unlike the Gaussian blur, the bilateral filter avoids averaging across edges due to the G_r term, which penalizes pixels with very different intensities from the center pixel, even if they are spatially close. In fact, from the equation 1.6 it is clear that the larger the difference between the neighbor and center pixel intensities, the smaller G_r becomes and as a result the weight of the output pixel.

2.1.5 Profile

The execution profile of the naive version is the one in figure 2.1. It is clear that the kernels are executed sequentially and only the default stream is used. In order to improve the execution of the program, the bilateral filter has to be optimized, and multiple streams may be used to parallelize the execution.

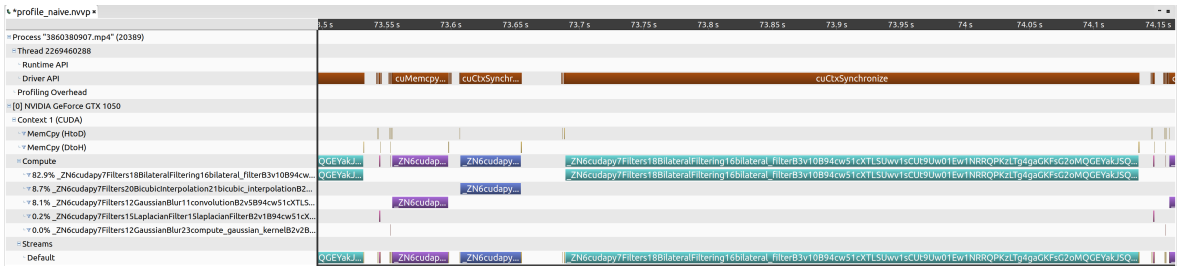

Figure 2.1: Naive execution profile

The results obtained from this version of the program are far from perfect but as shown in 3, the performances are far better than the version of the program running only on the CPU. See the appendix .2 for the CPU code.

2.2 Improved version

In this version, modified versions of bicubic interpolation and bilateral filtering were implemented. There are also some edits on the main code to parallelize the filters through CUDA streams. In order to reduce the memory allocation overhead, each filter uses a pre-allocated GPU buffers instead of allocating the memory at each frame as in the previous version. To do so, the code has been modified: when the `OriginalFrame` class is instantiated four pointers are initialized , which will later be allocated in memory within the `apply_filters` method only once for the entire execution of the program.

```
class OriginalFrame(VideoFrame.VideoFrame):
     def __init__(self, parent):
         super().__init__(parent)  # Call the parent class constructor
         ...
         self.d_output_filter_1 = None
         self.d_output_filter_2 = None
         self.d_output_filter_3 = None
         self.d_output_filter_4 = None
     ...

     def apply_filters(self, frame):
     if self.d_output_filter_1 is None:
```

```
            self.d_output_filter_1 = cuda.device_array_like(frame, stream_copy)
        if self.d_output_filter_2 is None:
            self.d_output_filter_2 = cuda.device_array_like(frame, stream_copy)
        if self.d_output_filter_3 is None:
            self.d_output_filter_3 = cuda.device_array_like(frame, stream_copy)
        if self.d_output_filter_4 is None:
            self.d_output_filter_4 = cuda.device_array(
                (frame.shape[0], frame.shape[1], 3), dtype=np.float32, stream=stream_copy)
```

Before calling the methods to update the frames in the GUI, the streams are synchronized to avoid showing partial or incorrect results:

```
# Update the frames
stream_filter_1.synchronize()
filter_1_frame.update(filtered_frame1)

stream_filter_2.synchronize()
filter_2_frame.update(filtered_frame2)

stream_filter_3.synchronize()
filter_3_frame.update(filtered_frame3)

stream_filter_4.synchronize()
filter_4_frame.update(filtered_frame4)
```

2.2.1 Bicubic interpolation

In this version, the code was modified to avoid redundant computations of the weights. In order to do this, two arrays are allocated in the local memory and used to store the weights:

```
# Use CUDA local arrays to store weights
wx = cuda.local.array(4, float32)
wy = cuda.local.array(4, float32)

for offset in range(4):
    wx[offset] = cubic_weight((offset - 1) - dx)
    wy[offset] = cubic_weight((offset - 1) - dy)
```

Furthermore, in order to avoid warp divergence, the pixel computation was modified through the removal of the if/else statements:

```
xs[i] = min(max(x1 + (i - 1), 0), max_x)
ys[i] = min(max(y1 + (i - 1), 0), max_y)
```

This approach eliminates the need for multiple boundary checks within the nested loops, thus reducing computational overhead.

2.2.2 Bilateral filter

The idea behind this version is to use the shared memory to cache sections of the image (tiles), in this way the number of accesses to the global memory is reduced, leading to better performances. The size of the tile is static and it is defined as a global variable inside the kernel:

```
TILE_SIZE = 16  # Size of the shared memory tile

@cuda.jit
def bilateral_filter(frame, result, sigma_s, sigma_r):
    # Shared memory for caching image tiles
    shared_frame = cuda.shared.array((TILE_SIZE, TILE_SIZE, 3),
                     dtype=np.float32)
```

During the computation, the neighborhood pixels are accessed from the shared memory rather than the global memory.

In this implementation it is important to use the command `cuda.syncthreads()` which ensures that all threads within a block have finished loading the data into the shared memory. This is needed to avoid performing operations on the wrong data

2.2.3 Profile

The execution profile of this new version of the program shows that kernels are properly handled by different streams, however, there is no overlapping of kernels. This is a sign that the program could be further optimized.

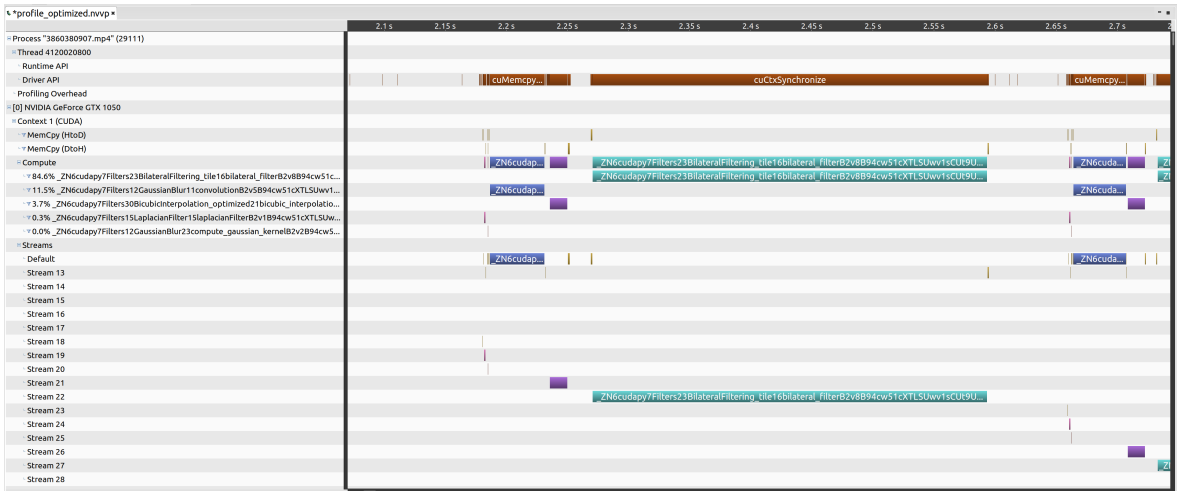

Figure 2.2: Streams execution profile

CHAPTER 3

Comparisons

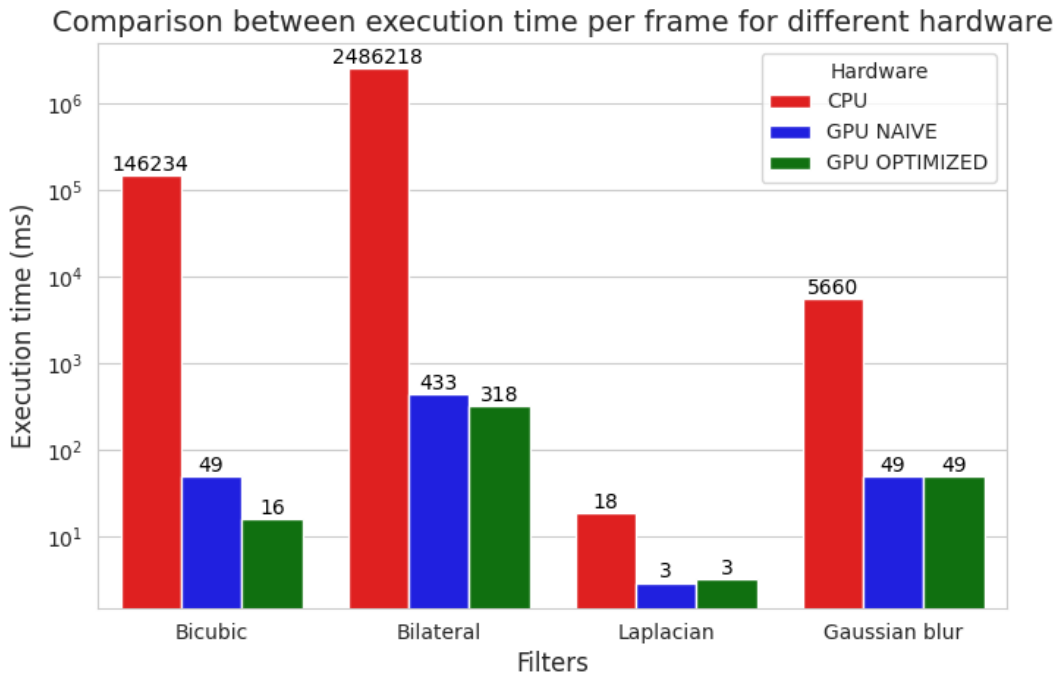

Figure 3.1: Results

In figure 3.1 are reported the results of the execution time for different versions of the program. As hypothesized, a clear difference in performance is evident between the CPU and GPU versions: both the naive and optimized GPU implementations show a great improvement over the CPU across all filters, highlighting the power of parallel processing for image manipulation.
Between the optimized version and the naive version the results are comparable, but in the case of computationally intensive filters like in Bicubic interpolation the 'GPU optimized' version the execution time is halved. In the Bilateral filter is shown as shared memory tiles reduce, in average, execution time by around 100ms going from 433ms of the naive implementation to 318ms of the optimized version.

CHAPTER 4

Appendices

.1 GUI

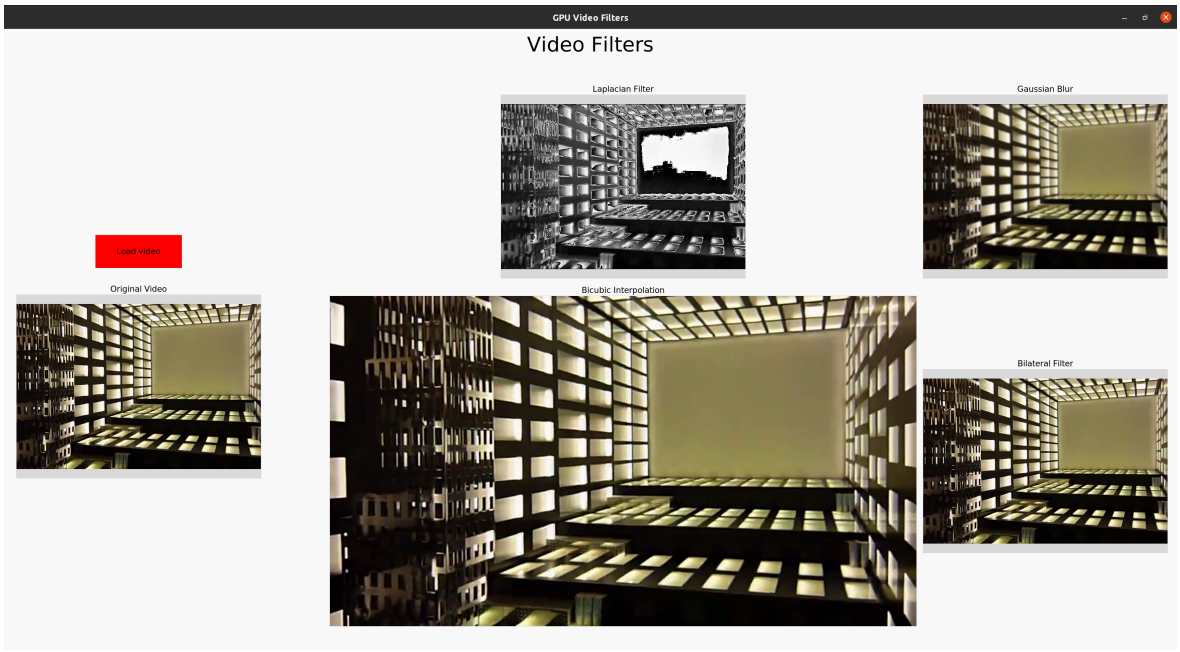

Figure 1: Graphical interface

.2 Code: CPU only implementation

.2.1 cpu_main.py

```
    from tkinter import *
from tkinter.filedialog import askopenfilename
from PIL import Image as Pil_image, ImageTk as Pil_imageTk, ImageFilter
import cv2
import numpy as np
```

```
import cpu.VideoFrame as VideoFrame
from skimage.filters import laplace
import math
import logging
import time
import cpu.BicubicInterpolation as BicubicInterpolation
import cpu.BilateralFiltering as BilateralFiltering
import cpu.GaussianFilter as GaussianFilter
import os # Used to get the video name
from datetime import datetime # Used in the logs to timestamp the execution

RESULTS_DIR = "results/cpu" # Directory to store the results
if not os.path.exists(RESULTS_DIR):
    os.makedirs(RESULTS_DIR)    # Create the results directory if it does not exist


# Configure logging for each filter
logging.basicConfig(level=logging.INFO)
filter1_logger = logging.getLogger("Filter1")
filter2_logger = logging.getLogger("Filter2")
filter3_logger = logging.getLogger("Filter3")
filter4_logger = logging.getLogger("Filter4")

def get_video_name(video_path):
    return os.path.splitext(os.path.basename(video_path))[0]


def measure_distortion(original_frame, filtered_frame):

    if (original_frame.shape[0] != filtered_frame.shape[0])
        or (original_frame.shape[1] != filtered_frame.shape[1]):
        # Resize the original frame to match the dimensions of the filtered frame
        original_frame = cv2.resize(original_frame,
                        (filtered_frame.shape[1], filtered_frame.shape[0]))


    # Convert frames to grayscale
    original_gray = cv2.cvtColor(original_frame, cv2.COLOR_BGR2GRAY)
    # Ensure filtered_frame is in 8-bit unsigned integer format and has 3 channels
    filtered_frame_8u = cv2.convertScaleAbs(filtered_frame)
    if len(filtered_frame_8u.shape) == 2 or filtered_frame_8u.shape[2] == 1:
        filtered_frame_8u = cv2.cvtColor(filtered_frame_8u, cv2.COLOR_GRAY2BGR)

    filtered_gray = cv2.cvtColor(filtered_frame_8u, cv2.COLOR_BGR2GRAY)

    # Compute the Mean Squared Error (MSE) between the original and filtered frames
    mse = np.mean((original_gray - filtered_gray) ** 2)

    # Compute the Peak Signal-to-Noise Ratio (PSNR)
```

```
    if mse == 0:
        psnr = float("inf")
    else:
        psnr = 20 * math.log10(255.0 / math.sqrt(mse))

    return mse, psnr


class OriginalFrame(VideoFrame.VideoFrame):
    def __init__(self, parent):
        super().__init__(parent)  # Call the parent class constructor
        self.screens = []
        self.vid = None

    def update(self):
        ret, frame = self.vid.read()
        if ret:
            self.photo = Pil_imageTk.PhotoImage(
                image=Pil_image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
            )
            # Position the image in the center with padding
            x = (self.canvas.winfo_width() - self.photo.width()) / 2
            y = (self.canvas.winfo_height() - self.photo.height()) / 2
            self.canvas.create_image(x, y, image=self.photo, anchor=NW)
            for screen in self.screens:
                screen.update(frame)
        else:
            self.reset_frame()  # Reset the frame when the video ends
            self.vid.release()
            return

        self.parent.after(10, self.update)

    def load_video(self, video):
        self.vid = cv2.VideoCapture(video)  # Open the video file
        self.update()

    def reset_frame(self):
        # Set a blank frame or a specific image to indicate the end of the video
        blank_frame = np.zeros((self.canvas.winfo_height(), self.canvas.winfo_width(), 3),
                               dtype=np.uint8)
        self.photo = Pil_imageTk.PhotoImage(image=Pil_image.fromarray(blank_frame))
        self.canvas.create_image(0, 0, image=self.photo, anchor=NW)


    def __del__(self):
        if self.vid.isOpened():
            self.vid.release()
```

```
class Filter1Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame):
        """
        Filter: Laplacian - CPU
        """
        start_time = time.time()
        # Convert the frame to grayscale
        # gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        filtered_frame = laplace(frame)
        filtered_frame = cv2.convertScaleAbs(filtered_frame)
        # self.photo = Pil_imageTk.PhotoImage(image=Pil_image.fromarray(filtered_frame))
        # self.canvas.create_image(0, 0, image=self.photo, anchor=NW)

        stop_time = time.time()
        elapsed_time_ms = (stop_time - start_time) * 1000

        # Log the results
        filter1_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")

        return filtered_frame


class Filter2Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame):
        """
        Filter: Gaussian Blur - CPU
        """
        start_time = time.time()
        filtered_frame = GaussianFilter.GaussianBlurImage(frame, 2.0)
        stop_time = time.time()
        mse, psnr = measure_distortion(frame, filtered_frame)
        elapsed_time_ms = (stop_time - start_time) * 1000
        filter2_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")
        return filtered_frame


class Filter3Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame):
        """
        Filter: Bicubic interpolation - CPU
        """
        # Scale factor for the bicubic interpolation
        scale = 2.0
        # Coefficient
        a = -1/2
        start_time = time.time()
        filtered_frame = BicubicInterpolation.bicubic(frame, scale, a)
        stop_time = time.time()
```

```
        elapsed_time_ms = (stop_time - start_time) * 1000
        # Convert frame to 8-bit unsigned integer format
        frame_8u = cv2.convertScaleAbs(filtered_frame)
        mse, psnr = measure_distortion(frame, filtered_frame)
        # Update the canvas size to fit the filtered frame
        self.update_canvas_size(filtered_frame)

        filter3_logger.info(
        f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}, MSE: {mse}, PSNR: {psnr}")

        return frame_8u


class Filter4Frame(VideoFrame.VideoFrame):

    def apply_filter(self, frame):
        '''
        Filter: Bilateral filtering - CPU
        '''
        sigma_s = 10.0
        sigma_r = 0.5
        start_time = time.time()
        filtered_frame = BilateralFiltering.bilateral_filter(frame, sigma_s, sigma_r)
        # Convert the filtered frame to uint8
        filtered_frame = cv2.convertScaleAbs(filtered_frame)
        stop_time = time.time()
        elapsed_time_ms = (stop_time - start_time) * 1000

        # Measure distortion
        mse, psnr = measure_distortion(frame, filtered_frame)

        # Log the results
        filter4_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")

        return filtered_frame

def button1_click(frame):
    video_source = askopenfilename(initialdir="./videos/KoNViD_1k_videos", title="Select a video file")
    if video_source:
        video_name = get_video_name(video_source)
        frame.load_video(video_source)
        logging.basicConfig(level=logging.INFO)

        filter1_handler = logging.FileHandler(
                f"{RESULTS_DIR}/{video_name}_filter_laplacian.log", "w")
        filter2_handler = logging.FileHandler(
                f"{RESULTS_DIR}/{video_name}_filter_gaussian_blur.log", "w")
        filter3_handler = logging.FileHandler(
                f"{RESULTS_DIR}/{video_name}_filter_bicubic_interpolation.log", "w")
```

```
        filter4_handler = logging.FileHandler(
                f"{RESULTS_DIR}/{video_name}_filter_bilateral.log", "w")

        filter1_logger.addHandler(filter1_handler)
        filter2_logger.addHandler(filter2_handler)
        filter3_logger.addHandler(filter3_handler)
        filter4_logger.addHandler(filter4_handler)



# Create the main window
root = Tk()
root.title("CPU Video Filters")
# root.geometry('1200x800')
root.geometry("3840x2160")  # Set the window size to 4K
root.configure(background="#F8F8F8")


text_label = Label(root, text="Video Filters", fg="black", bg="#F8F8F8")
text_label.pack(pady=(10, 10))
text_label.config(font=("Vandana", 25))

left_side = Frame(root, bg="#F8F8F8")
left_side.pack(side=LEFT, padx=20, pady=20)
left_side_top = Frame(left_side, bg="#F8F8F8")
left_side_top.pack(side=TOP, padx=20, pady=20)
left_side_bottom = Frame(left_side, bg="#F8F8F8")
left_side_bottom.pack(side=BOTTOM, padx=20, pady=20)
Label(left_side_bottom, text="Original Video", bg="#F8F8F8").pack()

right_side = Frame(root, bg="#F8F8F8")
right_side.pack(side=RIGHT, padx=20, pady=20)

right_top_left_frame = Frame(right_side, bg="#F8F8F8")
right_top_left_frame.grid(row=0, column=0, padx=10, pady=10)
Label(right_top_left_frame, text="Laplacian Filter", bg="#F8F8F8").pack()

right_top_right_frame = Frame(right_side, bg="#F8F8F8")
right_top_right_frame.grid(row=0, column=1, padx=10, pady=10)
Label(right_top_right_frame, text="Gaussian Blur", bg="#F8F8F8").pack()

right_bottom_left_frame = Frame(right_side, bg="#F8F8F8")
right_bottom_left_frame.grid(row=1, column=0, padx=10, pady=10)
Label(right_bottom_left_frame, text="Bicubic Interpolation", bg="#F8F8F8").pack()

right_bottom_right_frame = Frame(right_side, bg="#F8F8F8")
right_bottom_right_frame.grid(row=1, column=1, padx=10, pady=10)
Label(right_bottom_right_frame, text="Bilateral Filter", bg="#F8F8F8").pack()
```

```
# Add button for open file dialog
button1 = Button(
    left_side_top,
    text="Load video",
    command=lambda: button1_click(camera_frame),
    bg="red",
    width=15,
    height=3,
)
button1.pack(side=LEFT, padx=10, pady=10)

# Add live camera feed to the left side of the GUI
camera_frame = OriginalFrame(left_side_bottom)

filter_1_frame = Filter1Frame(right_top_left_frame)
filter_2_frame = Filter2Frame(right_top_right_frame)
filter_3_frame = Filter3Frame(right_bottom_left_frame)
filter_4_frame = Filter4Frame(right_bottom_right_frame)

# camera_frame.screens.append(filter_1_frame)
# camera_frame.screens.append(filter_2_frame)
# camera_frame.screens.append(filter_3_frame)
camera_frame.screens.append(filter_4_frame)
# camera_frame.update()


root.mainloop()
```

.2.2 BicubicInterpolation.py

```
from numba import cuda, float32, int32

@cuda.jit
def cubic_weight(t):
    """
    Cubic weight function: Calculates how much to "blend" nearby pixels using a smooth curve
    """
    a = -0.5
    t = abs(t)
    if t <= 1:
        return (a + 2) * t**3 - (a + 3) * t**2 + 1
    elif t < 2:
        return a * t**3 - 5 * a * t**2 + 8 * a * t - 4 * a
    return 0

@cuda.jit
def bicubic_interpolation(frame, output, scale):
    x, y = cuda.grid(2)
```

```
    if x >= output.shape[0] or y >= output.shape[1]:
        return

    src_x = x / scale
    src_y = y / scale
    x1 = int(src_x)
    y1 = int(src_y)
    dx = src_x - x1
    dy = src_y - y1

    # Use CUDA local arrays to store weights
    wx = cuda.local.array(4, float32)
    wy = cuda.local.array(4, float32)

    for offset in range(4):
        wx[offset] = cubic_weight((offset - 1) - dx)
        wy[offset] = cubic_weight((offset - 1) - dy)

    # Precompute coordinates using arrays
    xs = cuda.local.array(4, int32)
    ys = cuda.local.array(4, int32)
    max_x = frame.shape[0] - 1
    max_y = frame.shape[1] - 1

    for i in range(4):
        # Clamp to image boundaries with min/max instead of if conditions
        xs[i] = min(max(x1 + (i - 1), 0), max_x)
        ys[i] = min(max(y1 + (i - 1), 0), max_y)

    for c in range(3):
        val = 0.0
        for i in range(4):
            for j in range(4):
                val += frame[xs[i], ys[j], c] * wx[i] * wy[j]
        output[x, y, c] = int(max(0, min(val, 255)))
```

.2.3 BilateralFiltering.py

```
    import math
import numpy as np

def bilateral_filter(frame, sigma_s, sigma_r):
    frame = frame.astype(np.float32)    # To avoid overflow
    result = np.zeros_like(frame)
    height, width = frame.shape[0], frame.shape[1]
    kernel_size = int(2 * sigma_s + 1)
    half_kernel = kernel_size // 2

    for x in range(height):
```

```
        for y in range(width):
            total_weight = 0.0
            sum_r = 0.0
            sum_g = 0.0
            sum_b = 0.0

            center_r = frame[x, y, 0]
            center_g = frame[x, y, 1]
            center_b = frame[x, y, 2]

            for i in range(-half_kernel, half_kernel + 1):
                for j in range(-half_kernel, half_kernel + 1):
                    nx = x + i
                    ny = y + j

                    if 0 <= nx < height and 0 <= ny < width:
                        current_r = frame[nx, ny, 0]
                        current_g = frame[nx, ny, 1]
                        current_b = frame[nx, ny, 2]

                        spatial_dist = i**2 + j**2
                        G_s = math.exp(-spatial_dist / (2 * sigma_s**2))

                        intensity_dist = ((current_r - center_r)**2 +
                                          (current_g - center_g)**2 +
                                          (current_b - center_b)**2)
                        G_r = math.exp(-intensity_dist / (2 * sigma_r**2))

                        weight = G_s * G_r
                        total_weight += weight

                        sum_r += current_r * weight
                        sum_g += current_g * weight
                        sum_b += current_b * weight

            if total_weight > 0:
                result[x, y, 0] = sum_r / total_weight
                result[x, y, 1] = sum_g / total_weight
                result[x, y, 2] = sum_b / total_weight
            else:
                result[x, y, 0] = center_r
                result[x, y, 1] = center_g
                result[x, y, 2] = center_b

    return result
```

.2.4 GaussianFilter.py

```
    """
Source: https://medium.com/@akumar5/computer-vision-gaussian-filter-from-scratch-b485837b6e09
```

```
"""

import numpy as np

def convolution(oldimage, kernel):
    kernel_h = kernel.shape[0]
    kernel_w = kernel.shape[1]

    if(len(oldimage.shape) == 3):
        image_pad = np.pad(oldimage, pad_width=(
            (kernel_h // 2, kernel_h // 2),(kernel_w // 2,
            kernel_w // 2),(0,0)), mode='constant',
            constant_values=0).astype(np.float32)
    elif(len(oldimage.shape) == 2):
        image_pad = np.pad(oldimage, pad_width=(
            (kernel_h // 2, kernel_h // 2),(kernel_w // 2,
            kernel_w // 2)), mode='constant', constant_values=0).astype(np.float32)


    h = kernel_h // 2
    w = kernel_w // 2

    image_conv = np.zeros(image_pad.shape)

    for i in range(h, image_pad.shape[0]-h):
        for j in range(w, image_pad.shape[1]-w):
            x = image_pad[i-h:i-h+kernel_h, j-w:j-w+kernel_w]
            x = x.flatten()*kernel.flatten()
            image_conv[i][j] = x.sum()
    h_end = -h
    w_end = -w

    if(h == 0):
        return image_conv[h:,w:w_end]
    if(w == 0):
        return image_conv[h:h_end,w:]
    return image_conv[h:h_end,w:w_end]



def GaussianBlurImage(frame, sigma):
    filter_size = 2 * int(4 * sigma + 0.5) + 1
    gaussian_filter = np.zeros((filter_size, filter_size), np.float32)
    m = filter_size//2
    n = filter_size//2

    for x in range(-m, m+1):
        for y in range(-n, n+1):
            x1 = 2*np.pi*(sigma**2)
```

```
            x2 = np.exp(-(x**2 + y**2)/(2* sigma**2))
            gaussian_filter[x+m, y+n] = (1/x1)*x2

    im_filtered = np.zeros_like(frame, dtype=np.float32)
    for c in range(3):
        im_filtered[:, :, c] = convolution(frame[:, :, c], gaussian_filter)
    return (im_filtered.astype(np.uint8))
```

.2.5 VideoFrame.py

```
    from tkinter import *
from PIL import Image as Pil_image, ImageTk as Pil_imageTk
import cv2


class VideoFrame:
    def __init__(self, parent):
        self.parent = parent
        self.canvas = Canvas(self.parent, width=800, height=600)  # Set the canvas size
        self.canvas.pack()

    def update_canvas_size(self, frame):
        self.canvas.config(width=frame.shape[1], height=frame.shape[0])


    def update(self, frame):
        # Apply selected filter to the frame
        frame = self.apply_filter(frame)
        self.photo = Pil_imageTk.PhotoImage(
                image=Pil_image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)))
        # Position the image in the center with padding
        x = (self.canvas.winfo_width() - self.photo.width()) / 2
        y = (self.canvas.winfo_height() - self.photo.height()) / 2
        self.canvas.create_image(x, y, image=self.photo, anchor=NW)
```

.3 Code: GPU naive

.3.1 main_naive.py

```
    from tkinter import *
from tkinter.filedialog import askopenfilename
from PIL import Image as Pil_image, ImageTk as Pil_imageTk
import cv2
import numpy as np
from numba import cuda
import VideoFrame
from Filters.BicubicInterpolation import bicubic_interpolation
from Filters.BilateralFiltering import bilateral_filter
```

```
from Filters.GaussianBlur import compute_gaussian_kernel, convolution
from Filters.LaplacianFilter import laplacianFilter
import math
import logging
import os # Used to get the video name
from datetime import datetime # Used in the logs to timestamp the execution

RESULTS_DIR = "results/gpu" # Directory to store the results
if not os.path.exists(RESULTS_DIR):
    os.makedirs(RESULTS_DIR)    # Create the results directory if it does not exist


# Configure logging for each filter
logging.basicConfig(level=logging.INFO)
filter1_logger = logging.getLogger("Filter1")
filter2_logger = logging.getLogger("Filter2")
filter3_logger = logging.getLogger("Filter3")
filter4_logger = logging.getLogger("Filter4")

def get_video_name(video_path):
    return os.path.splitext(os.path.basename(video_path))[0]


def measure_distortion(original_frame, filtered_frame):

    if (original_frame.shape[0] != filtered_frame.shape[0]) or (original_frame.shape[1] != filtered_fra
        # Resize the original frame to match the dimensions of the filtered frame
        original_frame = cv2.resize(original_frame, (filtered_frame.shape[1], filtered_frame.shape[0]))


    # Convert frames to grayscale
    original_gray = cv2.cvtColor(original_frame, cv2.COLOR_BGR2GRAY)
    filtered_gray = cv2.cvtColor(filtered_frame, cv2.COLOR_BGR2GRAY)

    # Compute the Mean Squared Error (MSE) between the original and filtered frames
    mse = np.mean((original_gray - filtered_gray) ** 2)

    # Compute the Peak Signal-to-Noise Ratio (PSNR)
    if mse == 0:
        psnr = float("inf")
    else:
        psnr = 20 * math.log10(255.0 / math.sqrt(mse))

    return mse, psnr


class OriginalFrame(VideoFrame.VideoFrame):
    def __init__(self, parent):
        super().__init__(parent)  # Call the parent class constructor
```

```
        self.screens = []
        self.vid = None

    def update(self):
        ret, frame = self.vid.read()
        if ret:
            self.photo = Pil_imageTk.PhotoImage(
                image=Pil_image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
            )
            # Position the image in the center with padding
            x = (self.canvas.winfo_width() - self.photo.width()) / 2
            y = (self.canvas.winfo_height() - self.photo.height()) / 2
            self.canvas.create_image(x, y, image=self.photo, anchor=NW)
            for screen in self.screens:
                screen.update(frame)
        else:
            self.reset_frame()  # Reset the frame when the video ends
            self.vid.release()
            return

        self.parent.after(10, self.update)

    def load_video(self, video):
        self.vid = cv2.VideoCapture(video)  # Open the video file
        self.update()

    def reset_frame(self):
        # Set a blank frame or a specific image to indicate the end of the video
        blank_frame = np.zeros((self.canvas.winfo_height(), self.canvas.winfo_width(), 3), dtype=np.uin
        self.photo = Pil_imageTk.PhotoImage(image=Pil_image.fromarray(blank_frame))
        self.canvas.create_image(0, 0, image=self.photo, anchor=NW)


    def __del__(self):
        if self.vid.isOpened():
            self.vid.release()


class Filter1Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame, d_frame):
        """
        Filter: Laplacian Filter
        """
        # Convert the frame to grayscale
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Allocate the new frame on the GPU memory:
        d_frame_gray = cuda.to_device(np.ascontiguousarray(gray_frame))
        # d_frame = cuda.to_device(frame)
```

```
        d_output = cuda.device_array_like(frame)

        # Define the grid and block dimensions
        threads_per_block = (16, 16)

        # Compute the grid size to cover the whole frame:
        blocks_per_grid_x = int(
            np.ceil((frame.shape[0] + threads_per_block[0] - 1) / threads_per_block[0])
        )
        blocks_per_grid_y = int(
            np.ceil((frame.shape[1] + threads_per_block[1] - 1) / threads_per_block[1])
        )
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        # Create CUDA events for timing
        start_event = cuda.event()
        end_event = cuda.event()

        # Record the start event
        start_event.record()

        # Launch the kernel
        laplacianFilter[blocks_per_grid, threads_per_block](
            d_frame_gray, d_output, frame.shape[0], frame.shape[1]
        )

        cuda.synchronize()  # Wait for all threads to complete
        # Record the end event
        end_event.record()

        # Wait for the end event to complete
        end_event.synchronize()

        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        # Copy the result back to the host
        filtered_frame = d_output.copy_to_host()

        # Log the results
        filter1_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")

        return filtered_frame


class Filter2Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame, d_frame):
        """
        Filter: Gaussian Blur
```

```
"""

# Determine the filter size
sigma = 2.0
filter_size = 2 * int(4 * sigma + 0.5) + 1
kernel = np.zeros((filter_size, filter_size), dtype=np.float32)

# Allocate memory on the GPU
d_output = cuda.device_array_like(frame)
d_kernel = cuda.to_device(kernel)

# Define the grid and block dimensions
threads_per_block = (16, 16)
blocks_per_grid_x = (filter_size + threads_per_block[0] - 1) // threads_per_block[0]
blocks_per_grid_y = (filter_size + threads_per_block[1] - 1) // threads_per_block[1]
blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

# Create CUDA events for timing
start_event = cuda.event()
end_event = cuda.event()

# Record the start event
start_event.record()

# Compute Gaussian kernel on the device
compute_gaussian_kernel[blocks_per_grid, threads_per_block](d_kernel, sigma)

# Normalize the kernel (copy back to host, sum, normalize, send back)
kernel = d_kernel.copy_to_host()
kernel_sum = kernel.sum()
kernel /= kernel_sum
d_kernel = cuda.to_device(kernel)



# Configure convolution kernel launch parameters
# threads_per_block = (16, 16)
grid_x = int(np.ceil((frame.shape[0] + threads_per_block[0] - 1) / threads_per_block[0]))
grid_y = int(np.ceil((frame.shape[1] + threads_per_block[1] - 1) / threads_per_block[1]))
blocks_per_grid = (grid_x, grid_y)
# Perform convolution
convolution[blocks_per_grid, threads_per_block](frame, d_kernel, d_output)


# Record the end event
end_event.record()

# Wait for the end event to complete
end_event.synchronize()
```

```
        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        filtered_frame = d_output.copy_to_host()
        filter2_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")
        return filtered_frame


class Filter3Frame(VideoFrame.VideoFrame):

    def apply_filter(self, frame, d_frame):
        """
        Filter: Bicubic interpolation
        """
        # Scale factor for the bicubic interpolation
        scale = 2.0

        # Calculate the dimensions of the scaled frame
        scaled_height = int(frame.shape[0] * scale)
        scaled_width = int(frame.shape[1] * scale)

        # Allocate memory for the scaled frame on the GPU
        d_output = cuda.device_array((scaled_height, scaled_width, frame.shape[2]), dtype=frame.dtype)

        # Define the grid and block dimensions
        threads_per_block = (16, 16)

        blocks_per_grid_x = int(np.ceil((scaled_height + threads_per_block[0] - 1) / threads_per_block[
        blocks_per_grid_y = int(np.ceil((scaled_width + threads_per_block[1] - 1) / threads_per_block[1
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        # Create CUDA events for timing
        start_event = cuda.event()
        end_event = cuda.event()

        # Record the start event
        start_event.record()

        # Launch the kernel
        bicubic_interpolation[blocks_per_grid, threads_per_block](d_frame, d_output, scale)

        cuda.synchronize()  # wait for all threads to complete. The copy to the host performs an implic

        # Record the end event
        end_event.record()

        # Wait for the end event to complete
        end_event.synchronize()
```

```
        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        filtered_frame = d_output.copy_to_host()
        mse, psnr = measure_distortion(frame, filtered_frame)

        filter3_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}, MSE: {

        # Update the canvas size to fit the filtered frame
        self.update_canvas_size(filtered_frame)
        return filtered_frame


class Filter4Frame(VideoFrame.VideoFrame):

    def apply_filter(self, frame, d_frame):
        """
        Filter: Bilateral Filter
        """
        frame = frame.astype(np.float32)  # Convert the frame to float32 to avoid potential overflow
        d_frame = cuda.to_device(frame)
        d_result = cuda.device_array_like(d_frame)


        # Define the block and grid dimensions
        threads_per_block = (16, 16)
        blocks_per_grid_x = math.ceil(frame.shape[0] / threads_per_block[0])
        blocks_per_grid_y = math.ceil(frame.shape[1] / threads_per_block[1])
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        # Define sigma values
        sigma_s = 10.0
        sigma_r = 0.5

        # Create CUDA events for timing
        start_event = cuda.event()
        end_event = cuda.event()

        # Record the start event
        start_event.record()

        # Launch the kernel
        bilateral_filter[blocks_per_grid, threads_per_block](d_frame, d_result, sigma_s, sigma_r)

        cuda.synchronize()  # wait for all threads to complete. The copy to the host performs an implic

        # Record the end event
        end_event.record()
```

```
        # Wait for the end event to complete
        end_event.synchronize()

        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        # Copy the final result back to the host
        result = d_result.copy_to_host()

        # Convert the result to uint8
        filtered_frame = np.clip(result, 0, 255).astype(np.uint8)   # Clip may be not necessary

        # Log the results
        filter4_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")

        return filtered_frame

def button1_click(frame):
    video_source = askopenfilename(initialdir="./videos/KoNViD_1k_videos", title="Select a video file")
    if video_source:
        video_name = get_video_name(video_source)
        frame.load_video(video_source)
        logging.basicConfig(level=logging.INFO)

        filter1_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_laplacian.log", "w")
        filter2_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_gaussian_blur.log", "
        filter3_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_bicubic_interpolation
        filter4_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_bilateral.log", "w")

        filter1_logger.addHandler(filter1_handler)
        filter2_logger.addHandler(filter2_handler)
        filter3_logger.addHandler(filter3_handler)
        filter4_logger.addHandler(filter4_handler)



# Create the main window
root = Tk()
root.title("GPU Video Filters")
# root.geometry('1200x800')
root.geometry("3840x2160")  # Set the window size to 4K
root.configure(background="#F8F8F8")


text_label = Label(root, text="Video Filters", fg="black", bg="#F8F8F8")
text_label.pack(pady=(10, 10))
text_label.config(font=("Vandana", 25))
```

```
left_side = Frame(root, bg="#F8F8F8")
left_side.pack(side=LEFT, padx=20, pady=20)
left_side_top = Frame(left_side, bg="#F8F8F8")
left_side_top.pack(side=TOP, padx=20, pady=20)
left_side_bottom = Frame(left_side, bg="#F8F8F8")
left_side_bottom.pack(side=BOTTOM, padx=20, pady=20)
Label(left_side_bottom, text="Original Video", bg="#F8F8F8").pack()

right_side = Frame(root, bg="#F8F8F8")
right_side.pack(side=RIGHT, padx=20, pady=20)

right_top_left_frame = Frame(right_side, bg="#F8F8F8")
right_top_left_frame.grid(row=0, column=0, padx=10, pady=10)
Label(right_top_left_frame, text="Laplacian Filter", bg="#F8F8F8").pack()

right_top_right_frame = Frame(right_side, bg="#F8F8F8")
right_top_right_frame.grid(row=0, column=1, padx=10, pady=10)
Label(right_top_right_frame, text="Gaussian Blur", bg="#F8F8F8").pack()

right_bottom_left_frame = Frame(right_side, bg="#F8F8F8")
right_bottom_left_frame.grid(row=1, column=0, padx=10, pady=10)
Label(right_bottom_left_frame, text="Bicubic Interpolation", bg="#F8F8F8").pack()

right_bottom_right_frame = Frame(right_side, bg="#F8F8F8")
right_bottom_right_frame.grid(row=1, column=1, padx=10, pady=10)
Label(right_bottom_right_frame, text="Bilateral Filter", bg="#F8F8F8").pack()


# Add button for open file dialog
button1 = Button(
    left_side_top,
    text="Load video",
    command=lambda: button1_click(camera_frame),
    bg="red",
    width=15,
    height=3,
)
button1.pack(side=LEFT, padx=10, pady=10)

# Add live camera feed to the left side of the GUI
camera_frame = OriginalFrame(left_side_bottom)

filter_1_frame = Filter1Frame(right_top_left_frame)
filter_2_frame = Filter2Frame(right_top_right_frame)
filter_3_frame = Filter3Frame(right_bottom_left_frame)
filter_4_frame = Filter4Frame(right_bottom_right_frame)

camera_frame.screens.append(filter_1_frame)
camera_frame.screens.append(filter_2_frame)
```

```
camera_frame.screens.append(filter_3_frame)
camera_frame.screens.append(filter_4_frame)
# camera_frame.update()


root.mainloop()
```

.3.2 BicubicInterpolation.py

```
    from numba import cuda

@cuda.jit
def bicubic_interpolation(frame, output, scale):
    x, y = cuda.grid(2) # it uses the formula cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x for
    if x < output.shape[0] and y < output.shape[1]:

        src_x = x / scale   # x position of the original image
        src_y = y / scale   # y position of the original image

        x1 = int(src_x)
        y1 = int(src_y)

        dx = src_x - x1
        dy = src_y - y1

        def cubic_weight(t):
            """
            Cubic weight function: Calculates how much to "blend" nearby pixels using a smooth curve
            """
            a = -0.5    # This value determines the performance of the kernel and
                        # it lies between -0.5 and -0.75 for optimum performances
            t = abs(t)
            if t <= 1:
                return (a + 2) * t**3 - (a + 3) * t**2 + 1
            elif t < 2:
                return a * t**3 - 5 * a * t**2 + 8 * a * t - 4 * a
            return 0

        def get_pixel_value(i, j, c):
            if i < 0:
                i = 0
            elif i >= frame.shape[0]:
                i = frame.shape[0] - 1
            if j < 0:
                j = 0
            elif j >= frame.shape[1]:
                j = frame.shape[1] - 1
            return frame[i, j, c]
```

```
        for c in range(3):
            value = 0.0
            for m in range(-1, 3):
                for n in range(-1, 3):
                    weight = cubic_weight(m - dx) * cubic_weight(n - dy)
                    value += get_pixel_value(x1 + m, y1 + n, c) * weight
            output[x, y, c] = min(max(int(value), 0), 255)
```

.3.3 BilateralFiltering.py

```
    from numba import cuda
import math
import numpy as np

@cuda.jit
def bilateral_filter(frame, result, sigma_s, sigma_r):
    x, y = cuda.grid(2)
    if x >= frame.shape[0] or y >= frame.shape[1]:
        # Outside of image bounds
        return

    kernel_size = int(2 * sigma_s + 1)
    half_kernel = kernel_size // 2

    total_weight = 0.0
    sum_r = 0.0
    sum_g = 0.0
    sum_b = 0.0

    # Center pixel values
    center_r = frame[x, y, 0]
    center_g = frame[x, y, 1]
    center_b = frame[x, y, 2]

    # Iterate over kernel window
    for i in range(-half_kernel, half_kernel + 1):
        for j in range(-half_kernel, half_kernel + 1):
            nx = x + i
            ny = y + j

            # Boundary check
            if 0 <= nx < frame.shape[0] and 0 <= ny < frame.shape[1]:
                # Current pixel values
                current_r = frame[nx, ny, 0]
                current_g = frame[nx, ny, 1]
                current_b = frame[nx, ny, 2]

                # Spatial component
                spatial_dist = i**2 + j**2
```

```
                G_s = math.exp(-spatial_dist / (2 * sigma_s**2))

                # Range component
                intensity_dist = (current_r - center_r)**2 + \
                                 (current_g - center_g)**2 + \
                                 (current_b - center_b)**2
                G_r = math.exp(-intensity_dist / (2 * sigma_r**2))

                # Calculate weight
                weight = G_s * G_r
                total_weight += weight

                # Accumulate weighted values
                sum_r += current_r * weight
                sum_g += current_g * weight
                sum_b += current_b * weight

    # Normalize and write result
    if total_weight > 0:
        result[x, y, 0] = sum_r / total_weight
        result[x, y, 1] = sum_g / total_weight
        result[x, y, 2] = sum_b / total_weight
    else:
        result[x, y, 0] = frame[x, y, 0]
        result[x, y, 1] = frame[x, y, 1]
        result[x, y, 2] = frame[x, y, 2]
```

.3.4 GaussianBlur.py

```
import numpy as np
import math
from numba import cuda

@cuda.jit
def compute_gaussian_kernel(d_kernel, sigma):
    x, y = cuda.grid(2)
    if x >= d_kernel.shape[0] or y >= d_kernel.shape[1]:
        return
    center_x = d_kernel.shape[0] // 2
    center_y = d_kernel.shape[1] // 2
    dx = x - center_x
    dy = y - center_y
    scale = 1.0 / (2 * math.pi * sigma ** 2)
    exponent = -(dx**2 + dy**2) / (2 * sigma ** 2)
    d_kernel[x, y] = scale * math.exp(exponent)

@cuda.jit
def convolution(oldimage, kernel, output):
    i, j = cuda.grid(2)
    if i >= output.shape[0] or j >= output.shape[1]:
```

```
        # Return if the thread is out of bounds
        return

    kh = kernel.shape[0]
    kw = kernel.shape[1]
    h_radius = kh // 2
    w_radius = kw // 2


    # Apply the Gaussian blur to each color channel
    channels = oldimage.shape[2]
    for c in range(channels):
        acc = 0.0
        for x in range(kh):
            for y in range(kw):
                px = i + (x - h_radius)
                py = j + (y - w_radius)
                if 0 <= px < oldimage.shape[0] and 0 <= py < oldimage.shape[1]:
                    acc += oldimage[px, py, c] * kernel[x, y]
        output[i, j, c] = acc
```

.3.5 LaplacianFilter.py

```
import numpy as np
from numba import cuda

FILTER_WIDTH = 3
FILTER_HEIGHT = 3

@cuda.jit
def laplacianFilter(frame, result, width, height):
    x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # Define the kernel to use
    kernel = cuda.local.array((3, 3), dtype=np.float32)

    # Define the kernel values: Positive laplacian mask
    kernel[0, 0] = 0
    kernel[0, 1] = 1
    kernel[0, 2] = 0
    kernel[1, 0] = 1
    kernel[1, 1] = -4
    kernel[1, 2] = 1
    kernel[2, 0] = 0
    kernel[2, 1] = 1
    kernel[2, 2] = 0

    # Check if inside the limit of the frame:
```

```
    if((x >= FILTER_WIDTH // 2) and (x < (width - FILTER_WIDTH // 2)) and (y >= FILTER_HEIGHT // 2) and
        # Initialize the result
        sum : float = 0
        for kx in range(-FILTER_WIDTH // 2, FILTER_WIDTH // 2 + 1):
            for ky in range(-FILTER_HEIGHT // 2, FILTER_HEIGHT // 2 + 1):
                fl : float = frame[x + kx][y + ky]
                sum += fl * kernel[kx + FILTER_WIDTH // 2][ky + FILTER_HEIGHT // 2]

        result[x][y] = sum
```

.3.6 VideoFrame.py

```
from tkinter import *
from PIL import Image as Pil_image, ImageTk as Pil_imageTk
import cv2
from numba import cuda
import numpy as np


class VideoFrame:
    def __init__(self, parent):
        self.parent = parent
        self.canvas = Canvas(self.parent, width=800, height=600)  # Set the canvas size
        self.canvas.pack()

    def update_canvas_size(self, frame):
        self.canvas.config(width=frame.shape[1], height=frame.shape[0])


    def update(self, frame):
        d_frame = cuda.to_device(np.ascontiguousarray(frame))
        # Apply selected filter to the frame
        frame = self.apply_filter(frame, d_frame)
        self.photo = Pil_imageTk.PhotoImage(image=Pil_image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR
        # Position the image in the center with padding
        x = (self.canvas.winfo_width() - self.photo.width()) / 2
        y = (self.canvas.winfo_height() - self.photo.height()) / 2
        self.canvas.create_image(x, y, image=self.photo, anchor=NW)
```

.4 Code: GPU optimized

.4.1 main_optimized.py

```
    from tkinter import *
from tkinter.filedialog import askopenfilename
from PIL import Image as Pil_image, ImageTk as Pil_imageTk
```

```python
import cv2
import numpy as np
from numba import cuda
import VideoFrameOptimized as VideoFrame
from Filters.BicubicInterpolation_optimized import bicubic_interpolation
from Filters.BilateralFiltering_tile import bilateral_filter
from Filters.GaussianBlur import compute_gaussian_kernel, convolution
from Filters.LaplacianFilter import laplacianFilter
import math
import logging
import os # Used to get the video name
from datetime import datetime # Used in the logs to timestamp the execution

RESULTS_DIR = "results/gpu_optimized" # Directory to store the results
if not os.path.exists(RESULTS_DIR):
    os.makedirs(RESULTS_DIR)    # Create the results directory if it does not exist


# Configure logging for each filter
logging.basicConfig(level=logging.INFO)
filter1_logger = logging.getLogger("Filter1")
filter2_logger = logging.getLogger("Filter2")
filter3_logger = logging.getLogger("Filter3")
filter4_logger = logging.getLogger("Filter4")

def get_video_name(video_path):
    return os.path.splitext(os.path.basename(video_path))[0]


def measure_distortion(original_frame, filtered_frame):

    if (original_frame.shape[0] != filtered_frame.shape[0]) or (original_frame.shape[1] != filtered_fra
        # Resize the original frame to match the dimensions of the filtered frame
        original_frame = cv2.resize(original_frame, (filtered_frame.shape[1], filtered_frame.shape[0]))


    # Convert frames to grayscale
    original_gray = cv2.cvtColor(original_frame, cv2.COLOR_BGR2GRAY)
    filtered_gray = cv2.cvtColor(filtered_frame, cv2.COLOR_BGR2GRAY)

    # Compute the Mean Squared Error (MSE) between the original and filtered frames
    mse = np.mean((original_gray - filtered_gray) ** 2)

    # Compute the Peak Signal-to-Noise Ratio (PSNR)
    if mse == 0:
        psnr = float("inf")
    else:
        psnr = 20 * math.log10(255.0 / math.sqrt(mse))
```

```
    return mse, psnr


class OriginalFrame(VideoFrame.VideoFrame):
    def __init__(self, parent):
        super().__init__(parent)  # Call the parent class constructor
        self.screens = []
        self.vid = None

        self.d_output_filter_1 = None
        self.d_output_filter_2 = None
        self.d_output_filter_3 = None
        self.d_output_filter_4 = None

    def update(self):
        ret, frame = self.vid.read()
        if ret:
            self.photo = Pil_imageTk.PhotoImage(
                image=Pil_image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
            )
            # Position the image in the center with padding
            x = (self.canvas.winfo_width() - self.photo.width()) / 2
            y = (self.canvas.winfo_height() - self.photo.height()) / 2
            self.canvas.create_image(x, y, image=self.photo, anchor=NW)
            self.apply_filters(frame)
        else:
            self.reset_frame()  # Reset the frame when the video ends
            self.vid.release()
            return

        self.parent.after(10, self.update)

    def load_video(self, video):
        self.vid = cv2.VideoCapture(video)  # Open the video file
        self.update()

    def reset_frame(self):
        # Set a blank frame or a specific image to indicate the end of the video
        blank_frame = np.zeros(
            (self.canvas.winfo_height(), self.canvas.winfo_width(), 3), dtype=np.uint8)
        self.photo = Pil_imageTk.PhotoImage(image=Pil_image.fromarray(blank_frame))
        self.canvas.create_image(0, 0, image=self.photo, anchor=NW)

    def apply_filters(self, frame):
        # Create the CUDA streams to execute the filters in parallel
        stream_copy = cuda.stream()
        stream_filter_1 = cuda.stream()
        stream_filter_2 = cuda.stream()
        stream_filter_3 = cuda.stream()
```

```
        stream_filter_4 = cuda.stream()

        # Copy the data to the GPU (with the streams so asynchronous copies can be performed)
        d_frame = cuda.to_device(frame, stream_copy)

        if self.d_output_filter_1 is None:
            self.d_output_filter_1 = cuda.device_array_like(frame, stream_copy)
        if self.d_output_filter_2 is None:
            self.d_output_filter_2 = cuda.device_array_like(frame, stream_copy)
        if self.d_output_filter_3 is None:
            self.d_output_filter_3 = cuda.device_array_like(frame, stream_copy)
        if self.d_output_filter_4 is None:
            self.d_output_filter_4 = cuda.device_array(
                (frame.shape[0], frame.shape[1], 3), dtype=np.float32, stream=stream_copy)

        stream_copy.synchronize()

        # Apply the filters:
        filtered_frame1 = filter_1_frame.apply_filter(
            frame, d_frame, self.d_output_filter_1, stream_filter_1)
        filtered_frame2 = filter_2_frame.apply_filter(
            frame, d_frame, self.d_output_filter_2, stream_filter_2)
        filtered_frame3 = filter_3_frame.apply_filter(
            frame, d_frame, self.d_output_filter_3, stream_filter_3)
        filtered_frame4 = filter_4_frame.apply_filter(
            frame, d_frame, self.d_output_filter_4, stream_filter_4)


        # Update the frames
        stream_filter_1.synchronize()
        filter_1_frame.update(filtered_frame1)

        stream_filter_2.synchronize()
        filter_2_frame.update(filtered_frame2)

        stream_filter_3.synchronize()
        filter_3_frame.update(filtered_frame3)

        stream_filter_4.synchronize()
        filter_4_frame.update(filtered_frame4)


    def __del__(self):
        if self.vid.isOpened():
            self.vid.release()


class Filter1Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame, d_frame, d_output, stream):
```

```
"""
Filter: Laplacian Filter
"""
# Convert the frame to grayscale
gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# Allocate the new frame on the GPU memory:
d_frame_gray = cuda.to_device(np.ascontiguousarray(gray_frame))
# d_frame = cuda.to_device(frame)


# Define the grid and block dimensions
threads_per_block = (16, 16)

# Compute the grid size to cover the whole frame:
blocks_per_grid_x = int(
    np.ceil(
        (frame.shape[0] + threads_per_block[0] - 1) / threads_per_block[0]
        )
)
blocks_per_grid_y = int(
    np.ceil(
        (frame.shape[1] + threads_per_block[1] - 1) / threads_per_block[1]
        )
)
blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

# Create CUDA events for timing
start_event = cuda.event()
end_event = cuda.event()

# Record the start event
start_event.record()

# Launch the kernel
laplacianFilter[blocks_per_grid, threads_per_block, stream](
    d_frame_gray, d_output, frame.shape[0], frame.shape[1]
)

cuda.synchronize()  # Wait for all threads to complete
# Record the end event
end_event.record()

# Wait for the end event to complete
end_event.synchronize()

# Calculate the elapsed time in milliseconds
elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)
```

```
        # Copy the result back to the host
        filtered_frame = d_output.copy_to_host()

        # Log the results
        filter1_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")

        return filtered_frame


class Filter2Frame(VideoFrame.VideoFrame):
    def apply_filter(self, frame, d_frame, d_output, stream):
        """
        Filter: Gaussian Blur
        """

        # Determine the filter size
        sigma = 2.0
        filter_size = 2 * int(4 * sigma + 0.5) + 1
        kernel = np.zeros((filter_size, filter_size), dtype=np.float32)

        # Allocate memory on the GPU
        d_kernel = cuda.to_device(kernel)

        # Define the grid and block dimensions
        threads_per_block = (16, 16)
        blocks_per_grid_x = (filter_size + threads_per_block[0] - 1) // threads_per_block[0]
        blocks_per_grid_y = (filter_size + threads_per_block[1] - 1) // threads_per_block[1]
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        # Create CUDA events for timing
        start_event = cuda.event()
        end_event = cuda.event()

        # Record the start event
        start_event.record()

        # Compute Gaussian kernel on the device
        compute_gaussian_kernel[blocks_per_grid, threads_per_block, stream](d_kernel, sigma)

        # Normalize the kernel (copy back to host, sum, normalize, send back)
        kernel = d_kernel.copy_to_host()
        kernel_sum = kernel.sum()
        kernel /= kernel_sum
        d_kernel = cuda.to_device(kernel)



        # Configure convolution kernel launch parameters
        # threads_per_block = (16, 16)
```

```
        grid_x = int(np.ceil((frame.shape[0] + threads_per_block[0] - 1) / threads_per_block[0]))
        grid_y = int(np.ceil((frame.shape[1] + threads_per_block[1] - 1) / threads_per_block[1]))
        blocks_per_grid = (grid_x, grid_y)
        # Perform convolution
        convolution[blocks_per_grid, threads_per_block](frame, d_kernel, d_output)


        # Record the end event
        end_event.record()

        # Wait for the end event to complete
        end_event.synchronize()

        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        filtered_frame = d_output.copy_to_host()
        filter2_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")
        return filtered_frame


class Filter3Frame(VideoFrame.VideoFrame):

    def apply_filter(self, frame, d_frame, d_output, stream):
        """
        Filter: Bicubic interpolation
        """
        # Scale factor for the bicubic interpolation
        scale = 2.0

        # Calculate the dimensions of the scaled frame
        scaled_height = int(frame.shape[0] * scale)
        scaled_width = int(frame.shape[1] * scale)

        # Allocate memory for the scaled frame on the GPU
        d_output = cuda.device_array((scaled_height, scaled_width, frame.shape[2]), dtype=frame.dtype)

        # Define the grid and block dimensions
        threads_per_block = (16, 16)

        blocks_per_grid_x = int(np.ceil((scaled_height + threads_per_block[0] - 1) / threads_per_block[
        blocks_per_grid_y = int(np.ceil((scaled_width + threads_per_block[1] - 1) / threads_per_block[1
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        # Create CUDA events for timing
        start_event = cuda.event()
        end_event = cuda.event()

        # Record the start event
```

```
        start_event.record()

        # Launch the kernel
        bicubic_interpolation[blocks_per_grid, threads_per_block, stream](d_frame, d_output, scale)

        cuda.synchronize()  # wait for all threads to complete. The copy to the host performs an implic

        # Record the end event
        end_event.record()

        # Wait for the end event to complete
        end_event.synchronize()

        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        filtered_frame = d_output.copy_to_host()
        mse, psnr = measure_distortion(frame, filtered_frame)

        filter3_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}, MSE: {

        # Update the canvas size to fit the filtered frame
        self.update_canvas_size(filtered_frame)
        return filtered_frame


class Filter4Frame(VideoFrame.VideoFrame):

    def apply_filter(self, frame, d_frame, d_output, stream):
        """
        Filter: Bilateral Filter
        """
        frame = frame.astype(np.float32)
        d_frame = cuda.to_device(frame)

        # Define the block and grid dimensions
        TILE_SIZE = 16
        threads_per_block = (TILE_SIZE, TILE_SIZE)
        blocks_per_grid_x = math.ceil((frame.shape[0] + TILE_SIZE - 1)/ threads_per_block[0])
        blocks_per_grid_y = math.ceil((frame.shape[1] + TILE_SIZE - 1) / threads_per_block[1])
        blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

        # Define sigma values
        sigma_s = 10.0
        sigma_r = 0.5

        # Create CUDA events for timing
        start_event = cuda.event()
        end_event = cuda.event()
```

```
        # Record the start event
        start_event.record()

        # Launch the kernel
        bilateral_filter[blocks_per_grid, threads_per_block, stream](d_frame, d_output, sigma_s, sigma_

        cuda.synchronize()  # wait for all threads to complete. The copy to the host performs an implic

        # Record the end event
        end_event.record()

        # Wait for the end event to complete
        end_event.synchronize()

        # Calculate the elapsed time in milliseconds
        elapsed_time_ms = cuda.event_elapsed_time(start_event, end_event)

        # Copy the final result back to the host
        result = d_output.copy_to_host()

        # Convert the result to uint8
        filtered_frame = np.clip(result, 0, 255).astype(np.uint8)   # Clip may be not necessary

        # Log the results
        filter4_logger.info(f"Timestamp: {datetime.now()}, EXECUTION TIME ms: {elapsed_time_ms}")

        return filtered_frame

def button1_click(frame):
    video_source = askopenfilename(initialdir="./videos/KoNViD_1k_videos", title="Select a video file")
    if video_source:
        video_name = get_video_name(video_source)
        frame.load_video(video_source)
        logging.basicConfig(level=logging.INFO)

        filter1_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_laplacian.log", "w")
        filter2_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_gaussian_blur.log", "
        filter3_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_bicubic_interpolation
        filter4_handler = logging.FileHandler(f"{RESULTS_DIR}/{video_name}_filter_bilateral.log", "w")

        filter1_logger.addHandler(filter1_handler)
        filter2_logger.addHandler(filter2_handler)
        filter3_logger.addHandler(filter3_handler)
        filter4_logger.addHandler(filter4_handler)



# Create the main window
```

```
root = Tk()
root.title("GPU Video Filters")
# root.geometry('1200x800')
root.geometry("3840x2160")  # Set the window size to 4K
root.configure(background="#F8F8F8")


text_label = Label(root, text="Video Filters", fg="black", bg="#F8F8F8")
text_label.pack(pady=(10, 10))
text_label.config(font=("Vandana", 25))

left_side = Frame(root, bg="#F8F8F8")
left_side.pack(side=LEFT, padx=20, pady=20)
left_side_top = Frame(left_side, bg="#F8F8F8")
left_side_top.pack(side=TOP, padx=20, pady=20)
left_side_bottom = Frame(left_side, bg="#F8F8F8")
left_side_bottom.pack(side=BOTTOM, padx=20, pady=20)
Label(left_side_bottom, text="Original Video", bg="#F8F8F8").pack()

right_side = Frame(root, bg="#F8F8F8")
right_side.pack(side=RIGHT, padx=20, pady=20)

right_top_left_frame = Frame(right_side, bg="#F8F8F8")
right_top_left_frame.grid(row=0, column=0, padx=10, pady=10)
Label(right_top_left_frame, text="Laplacian Filter", bg="#F8F8F8").pack()

right_top_right_frame = Frame(right_side, bg="#F8F8F8")
right_top_right_frame.grid(row=0, column=1, padx=10, pady=10)
Label(right_top_right_frame, text="Gaussian Blur", bg="#F8F8F8").pack()

right_bottom_left_frame = Frame(right_side, bg="#F8F8F8")
right_bottom_left_frame.grid(row=1, column=0, padx=10, pady=10)
Label(right_bottom_left_frame, text="Bicubic Interpolation", bg="#F8F8F8").pack()

right_bottom_right_frame = Frame(right_side, bg="#F8F8F8")
right_bottom_right_frame.grid(row=1, column=1, padx=10, pady=10)
Label(right_bottom_right_frame, text="Bilateral Filter", bg="#F8F8F8").pack()


# Add button for open file dialog
button1 = Button(
    left_side_top,
    text="Load video",
    command=lambda: button1_click(camera_frame),
    bg="red",
    width=15,
    height=3,
)
button1.pack(side=LEFT, padx=10, pady=10)
```

```
# Add live camera feed to the left side of the GUI
camera_frame = OriginalFrame(left_side_bottom)

filter_1_frame = Filter1Frame(right_top_left_frame)
filter_2_frame = Filter2Frame(right_top_right_frame)
filter_3_frame = Filter3Frame(right_bottom_left_frame)
filter_4_frame = Filter4Frame(right_bottom_right_frame)

camera_frame.screens.append(filter_1_frame)
camera_frame.screens.append(filter_2_frame)
camera_frame.screens.append(filter_3_frame)
camera_frame.screens.append(filter_4_frame)
# camera_frame.update()


root.mainloop()
```

.4.2 BicubicInterpolation_optimized.py

```
from numba import cuda, float32, int32

@cuda.jit
def cubic_weight(t):
    """
    Cubic weight function: Calculates how much to "blend" nearby pixels using a smooth curve
    """
    a = -0.5
    t = abs(t)
    if t <= 1:
        return (a + 2) * t**3 - (a + 3) * t**2 + 1
    elif t < 2:
        return a * t**3 - 5 * a * t**2 + 8 * a * t - 4 * a
    return 0

@cuda.jit
def bicubic_interpolation(frame, output, scale):
    x, y = cuda.grid(2)

    if x >= output.shape[0] or y >= output.shape[1]:
        return

    src_x = x / scale
    src_y = y / scale
    x1 = int(src_x)
    y1 = int(src_y)
    dx = src_x - x1
    dy = src_y - y1
```

```
    # Use CUDA local arrays to store weights
    wx = cuda.local.array(4, float32)
    wy = cuda.local.array(4, float32)

    for offset in range(4):
        wx[offset] = cubic_weight((offset - 1) - dx)
        wy[offset] = cubic_weight((offset - 1) - dy)

    # Precompute coordinates using arrays
    xs = cuda.local.array(4, int32)
    ys = cuda.local.array(4, int32)
    max_x = frame.shape[0] - 1
    max_y = frame.sheape[1] - 1

    for i in range(4):
        # Clamp to image boundaries with min/max instead of if conditions
        xs[i] = min(max(x1 + (i - 1), 0), max_x)
        ys[i] = min(max(y1 + (i - 1), 0), max_y)

    for c in range(3):
        val = 0.0
        for i in range(4):
            for j in range(4):
                val += frame[xs[i], ys[j], c] * wx[i] * wy[j]
        output[x, y, c] = int(max(0, min(val, 255)))
```

.4.3 BilateralFiltering$_{t}ile.py$

```
from numba import cuda
import math
import numpy as np

TILE_SIZE = 16  # Size of the shared memory tile

@cuda.jit
def bilateral_filter(frame, result, sigma_s, sigma_r):
    # Shared memory for caching image tiles
    shared_frame = cuda.shared.array((TILE_SIZE, TILE_SIZE, 3), dtype=np.float32)

    # Thread and block indices
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y

    # Pixel coordinates in the output image
    x = bx * TILE_SIZE + tx
    y = by * TILE_SIZE + ty

    # Load the tile into shared memory
```

```
if x < frame.shape[0] and y < frame.shape[1]:
    for c in range(3):  # Load all color channels
        shared_frame[tx, ty, c] = frame[x, y, c]
else:
    for c in range(3):
        shared_frame[tx, ty, c] = 0.0  # Pad with zeros if out of bounds

# Wait for all threads to finish loading
cuda.syncthreads()

# Only threads within the image bounds compute the result
if x < frame.shape[0] and y < frame.shape[1]:
    # Initialize accumulators
    total_weight = 0.0
    sum_r = 0.0
    sum_g = 0.0
    sum_b = 0.0

    # Center pixel values
    center_r = shared_frame[tx, ty, 0]
    center_g = shared_frame[tx, ty, 1]
    center_b = shared_frame[tx, ty, 2]

    # Kernel radius
    kernel_size = int(2 * sigma_s + 1)
    half_kernel = kernel_size // 2

    # Iterate over the kernel window
    for i in range(-half_kernel, half_kernel + 1):
        for j in range(-half_kernel, half_kernel + 1):
            # Neighbor coordinates in shared memory
            nx = tx + i
            ny = ty + j

            # Check if the neighbor is within the shared memory tile
            if 0 <= nx < TILE_SIZE and 0 <= ny < TILE_SIZE:
                # Neighbor pixel values
                current_r = shared_frame[nx, ny, 0]
                current_g = shared_frame[nx, ny, 1]
                current_b = shared_frame[nx, ny, 2]

                # Spatial distance
                spatial_dist = i**2 + j**2
                G_s = math.exp(-spatial_dist / (2 * sigma_s**2))

                # Intensity distance
                intensity_dist = (current_r - center_r)**2 + \
                                 (current_g - center_g)**2 + \
                                 (current_b - center_b)**2
```

```
                    G_r = math.exp(-intensity_dist / (2 * sigma_r**2))

                    # Weight
                    weight = G_s * G_r
                    total_weight += weight

                    # Accumulate weighted values
                    sum_r += current_r * weight
                    sum_g += current_g * weight
                    sum_b += current_b * weight

        # Normalize and write result
        if total_weight > 0:
            result[x, y, 0] = sum_r / total_weight
            result[x, y, 1] = sum_g / total_weight
            result[x, y, 2] = sum_b / total_weight
        else:
            result[x, y, 0] = frame[x, y, 0]
            result[x, y, 1] = frame[x, y, 1]
            result[x, y, 2] = frame[x, y, 2]
```

.4.4 VideoFrameOptimized.py

```
    from tkinter import *
from PIL import Image as Pil_image, ImageTk as Pil_imageTk
import cv2
from numba import cuda
import numpy as np


class VideoFrame:
    def __init__(self, parent):
        self.parent = parent
        self.canvas = Canvas(self.parent, width=800, height=600)  # Set the canvas size
        self.canvas.pack()

    def update_canvas_size(self, frame):
        self.canvas.config(width=frame.shape[1], height=frame.shape[0])


    def update(self, frame):
        # Apply selected filter to the frame

        self.photo = Pil_imageTk.PhotoImage(image=Pil_image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR
        # Position the image in the center with padding
        x = (self.canvas.winfo_width() - self.photo.width()) / 2
        y = (self.canvas.winfo_height() - self.photo.height()) / 2
        self.canvas.create_image(x, y, image=self.photo, anchor=NW)
```

Bibliography

[1] Ashish Kumar *Computer Vision: Gaussian Filter from Scratch*. URL: https://medium.com/@akumar5/computer-vision-gaussian-filter-from-scratch-b485837b6e09

[2] geeksforgeeks *Bicubic Interpolation for Resizing Image*. URL: https://www.geeksforgeeks.org/python-opencv-bicubic-interpolation-for-resizing-image/

[3] *CUDA Python Reference*. URL: https://numba.pydata.org/numba-doc/dev/cuda-reference/index.html

[4] Raji Lini *Laplacian of Gaussian Filter (LoG) for Image Processing*. URL: https://medium.com/@rajilini/laplacian-of-gaussian-filter-log-for-image-processing-c2d1659d5d2

[5] Amanrao *Image Upscaling using Bicubic Interpolation*. URL: https://medium.com/@amanrao032/image-upscaling-using-bicubic-interpolation-ddb37295df0

Bibliography