Ain Shams University
Faculty of Engineering

University of East London

CSE354: Distributed Computing

Project Title:

Distributed Image Processing System using Cloud Computing

Phase 1: Project Planning and Design

Group(46) :

Eslam Mostafa Mohamed Haider	*20P4797*
Belal Mahmoud Amer Mahmoud	*19P2374*
Abdelrahman Mostafa	*20P6060*
Zeyad Essam Elsayed	*20P5728*

Submitted to:

Prof. Ayman M. Bahaa-Eldin

Contents

Project proposal

In this part of the report, we aim to define the project scope, and objectives of the project.

1. Introduction

This project aims to develop a distributed image processing system using cloud computing technologies. The system will be implemented in Python, leveraging cloud-based virtual machines for distributed computing. The application will use OpenCL or MPI for parallel processing of image data.

The system aims to harness the power of cloud-based virtual machines to distribute image processing tasks seamlessly across a network of computational nodes. By utilizing the computational resources of the cloud, the system endeavors to accelerate image processing workflows, enhance scalability, and ensure fault tolerance. This report provides an in-depth analysis of the system's architecture, features, implementation details, and potential applications.

2.Project objectives

The primary objectives of this project are to:

***1- Scalability**: Design a system that can handle a large number of image processing requests efficiently by scaling resources dynamically.*

***2-Performance**: Ensure quick processing times for uploaded images by distributing the workload across multiple computing nodes.*

***3-Reliability**: Create a reliable system that can handle failures gracefully and ensure high availability of services.*

***4-Cost-effectiveness:** Optimize resource usage to minimize operational costs while meeting performance requirements.*

***5-Security:** Implement robust security measures to protect user data and prevent unauthorized access to the system.*

***6-Flexibility:** Design the system to accommodate future changes and additions in image processing algorithms or features.*

***7-Resource Utilization**: Optimize resource utilization to minimize idle resources and maximize cost efficiency.*

***8-Interoperability**: Ensure compatibility with different image formats and processing libraries to support diverse use cases.*

3. Project Scope

The scope of the project involves designing and implementing a distributed image processing system leveraging cloud computing technologies. This system will allow users to upload images to a cloud-based platform, where they will be processed concurrently using multiple computing nodes. The processed images will then be available for download or further analysis.

***1-Integration:** Determine how the system will integrate with existing applications or systems, if applicable.*

***2-Geographical Considerations:** Define whether the system needs to be deployed globally or in specific regions, considering data sovereignty and latency requirements.*

***3-Regulatory Compliance:** Ensure the system complies with relevant data protection regulations (e.g., GDPR, HIPAA) and industry standards.*

***4-User Feedback:** Incorporate mechanisms for collecting user feedback to continuously improve the system.*

4.Project Requirements

1-Cloud Infrastructure: Select a cloud service provider (e.g., AWS, Azure, Google Cloud) and set up the necessary infrastructure for hosting the image processing system.
Image Upload and Storage: Develop functionality for users to upload images to the cloud storage.

2-Image Processing: Implement distributed image processing algorithms that can be parallelized across multiple computing nodes.

3-Monitoring and Logging: Set up monitoring and logging tools to track system performance, resource usage, and errors.

4-Fault Tolerance: Implement mechanisms for fault tolerance and automatic recovery to ensure the system remains operational even in the face of failures.

5-Security Measures: Implement encryption for data in transit and at rest, role-based access control, and other security measures to protect the system from unauthorized access and data breaches.

6-Scalability: Design the system to scale horizontally to handle increased load as the number of image processing requests grows.

7-API and User Interface: Develop an intuitive user interface and API for users to interact with the system, upload images, and retrieve processed results.

8-Resource Orchestration: Implement a resource orchestration mechanism (e.g., Kubernetes) for efficient management of computing resources.

9-Load Balancing: Design load balancing mechanisms to evenly distribute processing tasks among available nodes.

10-Data Persistence: Choose appropriate data storage solutions (e.g., object storage, databases) for storing uploaded images and processed results.

11-Version Control: Implement version control for image processing algorithms and configurations to track changes and facilitate rollback if needed.

12-Cost Monitoring: Set up tools for monitoring and analyzing cloud usage to identify cost-saving opportunities and optimize resource allocation.

13-Data Backup and Recovery: Establish backup and recovery procedures to protect against data loss and ensure data integrity.

14-Performance Benchmarking: Define performance metrics and conduct benchmarking tests to evaluate the system's efficiency and identify areas for optimization.

15-Task Distribution: Design a mechanism for distributing image processing tasks efficiently among available computing nodes.

5.Project Deliverables

1-System Architecture Diagram: A high-level overview of the distributed image processing system, showing components, interactions, and data flow.

2-Technical Specifications: Detailed specifications for each component of the system, including APIs, data formats, and communication protocols.

3-Prototype: A functional prototype demonstrating key features of the system, such as image upload, processing, and retrieval.

4-Documentation: Comprehensive documentation covering installation, configuration, usage, and troubleshooting of the system.

5-Testing Plan: A plan for testing the system to ensure it meets performance, reliability, and security requirements.

6-Deployment Plan: Guidelines for deploying the system in a production environment, including scaling strategies and best practices for maintenance and monitoring.

7-Training Materials: Develop training materials and documentation for users and administrators to onboard and operate the system effectively.

8-Community Engagement: Foster a community around the system by providing forums, documentation, and support channels for users and developers.

9-Continuous Improvement Plan: Outline strategies for continuously improving the system based on feedback, performance metrics, and emerging technologies.

10-Comprehensive Testing: Conduct thorough testing, including unit tests, integration tests, and end-to-end tests, to ensure the reliability and functionality of the system under various scenarios.

11-Scalability Testing: Perform scalability testing to validate the system's ability to handle increased load and scale resources accordingly.

By defining these elements clearly in the planning and design phase, you'll have a solid foundation for developing your Distributed Image Processing System using Cloud Computing.

#Design document

In this part of the report, we aim to specify system's functionalities, architecture, components, and Design the system architecture and select the appropriate technologies.
And Create a detailed project plan with tasks, responsibilities, timelines ,and User Stories

6. System Architecture Design

This system utilizes a 3-tier architecture to achieve distributed image processing using cloud computing. Here's a breakdown of the tiers:

Tier 1: Presentation Tier

- **Components:** user interface (UI)
- **Technology:** Python GUI (**Tkinter**)
- **Reason:**
- **Functionality:**
  - Provides a user interface for uploading images.
  - Allows users to select desired image processing algorithms.
  - Displays progress of the processing task.
  - Enables downloading of processed images.

Tier 2: Business Logic Tier

• Components:

• Processing Manager (Python, Cloud SDK):

- **Job Coordinator:** Receives user requests and uploaded images.
- **Algorithm Selector:** Interacts with the Algorithm Library to select the chosen processing algorithm.
- **Task Distributor:** Splits images into chunks and distributes tasks to the Worker Supervisor.
- **Result Aggregator:** Collects results from worker nodes and assembles the final processed image.
- **User Interface Coordinator:** Handles user interactions like progress updates and download requests.

• Worker Supervisor (Python, Cloud SDK, OpenCL):

- **Virtual Machine Manager:** Provisions and manages virtual machines in the cloud.
- **Task Scheduler:** Schedules image processing tasks on available worker nodes.
- **Worker Health Monitor:** Monitors worker node health and handles failures by reporting back to the Job Coordinator for task reassignment.

- **Technology:** Python with cloud APIs and libraries (Cloud SDK, OpenCL)
- **Reason:** Cloud providers offer APIs and libraries (like Cloud SDK) specifically designed to interact with their services. These tools simplify tasks like provisioning virtual machines, interacting with storage services, and managing resources within the cloud environment.

OpenCL leverages GPUs for efficient image processing tasks, while MPI facilitates communication and task distribution across multiple virtual machines. Choosing between them depends on the type of processing and hardware resources available in the cloud environment.

- ☐ **Functionality:**
  - Processing Manager:
    - Receives user requests and uploaded images.
    - Splits the image into smaller chunks for parallel processing.
    - Interacts with the worker manager to distribute tasks to virtual machines.
    - Collects results from worker nodes and assembles the final processed image.
    - Handles user interactions like progress updates and download requests.
  - Worker Manager:
    - Provisions and manages virtual machines in the cloud.
    - Sends image processing tasks (chunks) to worker nodes.
    - Monitors worker node health and handles failures by reassigning tasks.

Tier 3: Data Tier

- **Components:** Cloud storage service (Cloud Storage)
- **Technology:** Cloud provider's storage service APIs
- **Reason:** Cloud storage services offered by major providers (e.g., Cloud Storage, S3 buckets) provide scalable and reliable storage for user images, temporary processing data, and final processed images. They are cost-effective and integrate seamlessly with cloud compute services.
- ☐ **Functionality:**
  - Stores uploaded user images.
  - Stores temporary processing data on worker nodes (if needed).
  - Stores the final processed image after completion.

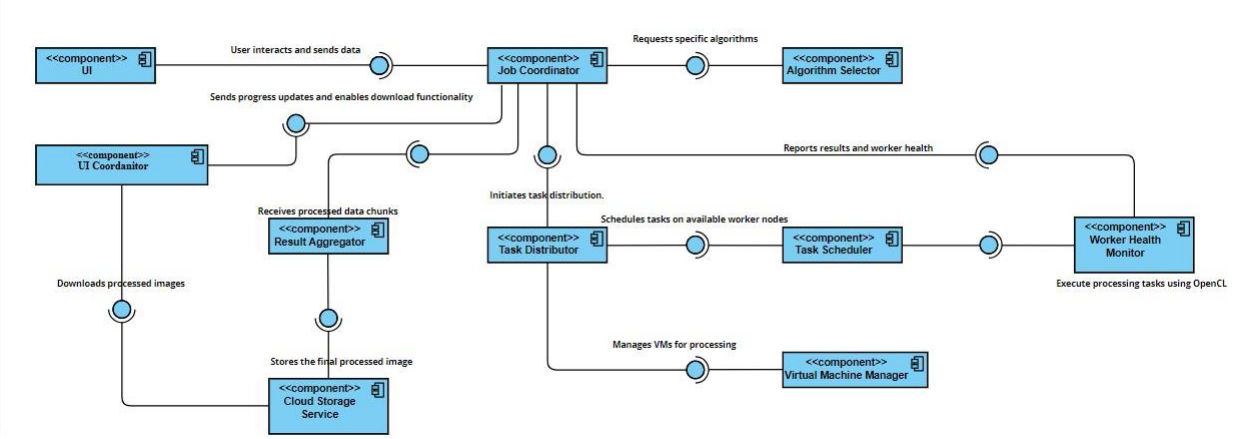
Communication Flow:

1. User interacts with the UI, uploading an image and selecting processing options.
2. Presentation tier sends the image data and user request to the processing manager.
3. Processing manager splits the image and interacts with the worker manager.
4. Worker manager provisions virtual machines (if needed) and distributes image chunks for processing.
5. Worker nodes perform the assigned image processing tasks using OpenCL or MPI for parallelization.
6. Processed image chunks are sent back to the processing manager.
7. Processing manager assembles the final image and stores it in the cloud storage.
8. Processing manager updates the UI with the processing status and allows downloading the final image.

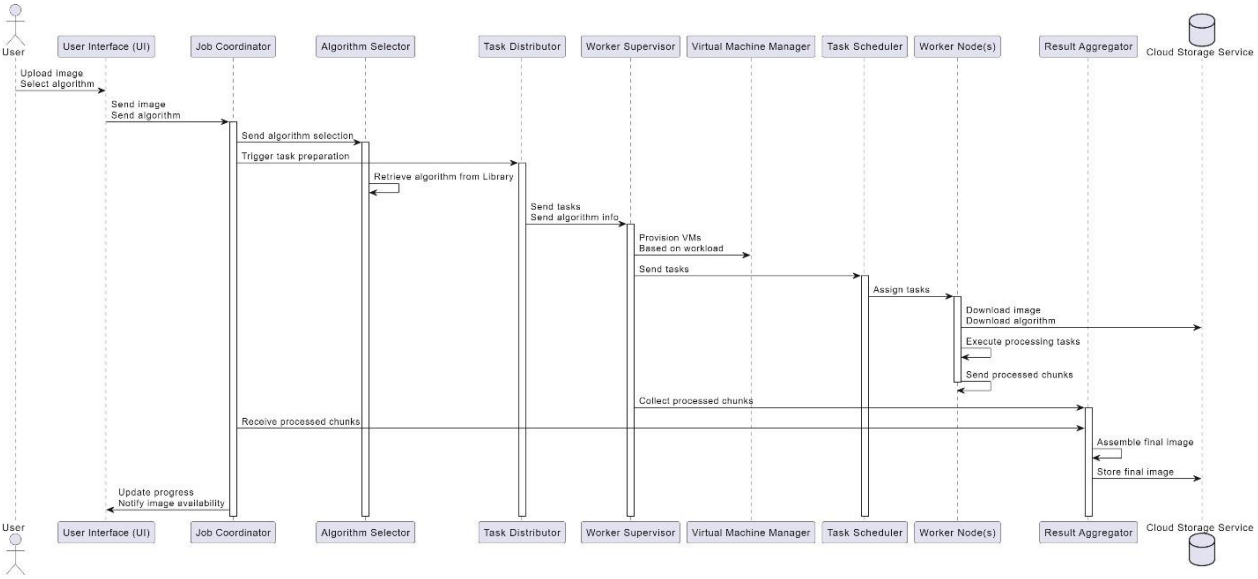
Benefits of this Architecture:

- **Scalability:** Easy to add more virtual machines for increased processing power.
- **Fault Tolerance:** System can recover from failures by reassigning tasks.
- **Distributed Processing:** Efficiently utilizes cloud resources for parallel image processing.
- **Clear Separation of Concerns:** Each tier handles specific functionalities.

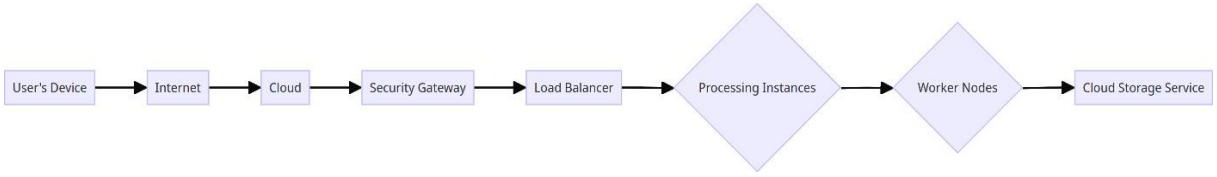
Component Diagram: -



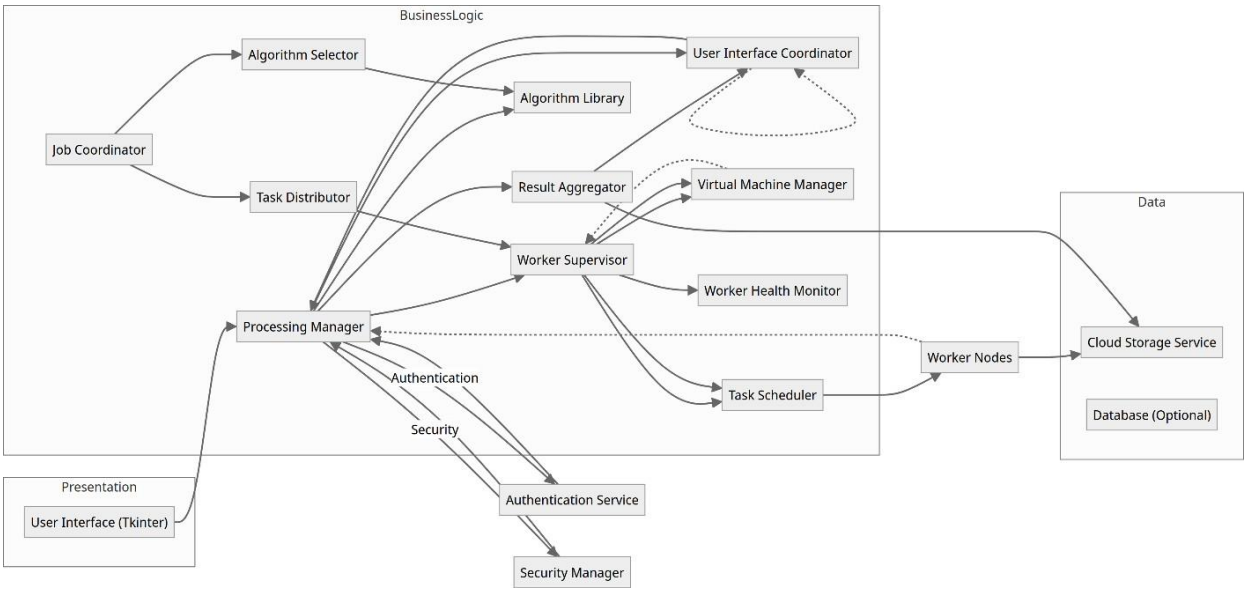
Sequence Diagram: -



Network Diagram: -



Infrastructure Diagram: -



7. Project Plan and timelines

Phase 1: Two to three weeks of project planning and design

Describe the project's goals, requirements, and scope (Week 1)

Tasks:

Have a launch meeting to go over the objectives and scope of the work.
Establish the goals and success standards for the project.
Collect and record specific needs from interested parties.

Week 1: Design System Architecture and Choose Technologies

Tasks:

System Architect:

Create the general architecture of the system, including its parts and functions.
Select MPI or OpenCL for parallel processing.

Engineers in Software:

Investigate and choose a cloud computing platform (such as GCP, AWS, or Azure).
Ascertain picture data storage options.

Establish a thorough project schedule (Week 2)

Tasks:

Manager of the Project:

Divide the project into manageable stages, tasks, and subtasks.
Establish deadlines and assign roles.
Keep track of the duties, deadlines, and tasks for every stage.
Specify deadlines and checkpoints.

Acceptance Criteria and User Stories (Week 2)

Tasks:

Product Owner:

Refine user stories based on gathered requirements.
Define acceptance criteria for each user story.

Phase 2: Development of Basic Functionality (2-3 weeks)

Implement Basic Image Processing Operations (Week 3)

Tasks:

Software Engineers:

Develop functionality to upload images.
Implement basic image processing algorithms such as filtering and color manipulation.

Set Up Cloud Environment and Virtual Machines (Week 3)

Tasks:

System Administrator:

Provision cloud-based virtual machines.
Configure networking and security settings**.**

Develop Worker Thread for Processing Tasks (Week 4)

Tasks:

Software Engineers:

Create worker thread for executing image processing tasks.
Integrate worker thread with cloud environment for distributed computing.

Phase 3: Development of Advanced Functionality (2-3 weeks)

Implement Advanced Image Processing Operations (Week 5)

Tasks:

Software Engineers:

Implement advanced image processing algorithms such as edge detection.
Test and optimize algorithms for performance.

Develop Distributed Processing Functionality (Week 5-6)

Tasks:

Software Engineers:

Implement distributed processing logic using selected technology (MPI or OpenCL).
Handle task distribution and coordination among virtual machines.

Implement Scalability and Fault Tolerance Features (Week 6-7)

Tasks:

System Architect:

Design mechanisms for scaling the system dynamically.
Implement fault tolerance features such as task reassignment.

Phase 4: Deployment, Documentation, and Testing (two to three weeks)

Perform Extensive Testing (Week 8)

Tasks:

Quality Control Group:

Evaluate systems, integrate systems, and test units.
Determine and fix errors and problems.

Record User Instructions, Code, and System Design (Week 9)

Technical Writers' Tasks:

Record the codebase, algorithms, and architecture of the system.
Write troubleshooting manuals and user instructions.

Ensure Operational and Deploy System to Cloud (Week 10)

System Administrator tasks:

Install the completed system in a cloud environment.
Make one last check to make sure the system is working.

Milestones:

Phase 1: Project Planning and Design Completed
Phase 2: Development of Basic Functionality Completed
Phase 3: Development of Advanced Functionality Completed
Phase 4: Testing, Documentation, and Deployment Completed

8. User Stories

1-As a user, I want to upload an image to the system for processing, so that I can apply various image processing operations.
Acceptance Criteria:
The system should provide a user interface or API endpoint for uploading images.
Supported image formats should include common formats such as JPEG, PNG, and GIF.
Users should receive confirmation once the image has been successfully uploaded.

2-As a user, I want to select the type of image processing operation to be performed, so that I can customize the processing according to my requirements.
Acceptance Criteria:
The system should present users with a list of available image processing algorithms or operations.
Users should be able to choose one or more operations to be applied to their uploaded image.
The selected operations should be clearly displayed or indicated to the user for confirmation.

3-As a user, I want to download the processed image once the operation is complete, so that I can use the processed image for further purposes.
Acceptance Criteria:
Once the image processing task is complete, users should be provided with a download link or option.
The processed image should be available in a commonly supported image format.
Users should receive confirmation that the download is successful.

4-As a user, I want to monitor the progress of the image processing task, so that I can track the status and estimated completion time.
Acceptance Criteria:
The system should provide a progress indicator or status update for ongoing image processing tasks.

Users should be able to view the percentage completion, estimated time remaining, or any other relevant progress information.
If there are any errors or issues during processing, users should be notified appropriately.

Ain Shams University
Faculty of Engineering

University of East London

CSE354: Distributed Computing

Project Title:

Distributed Image Processing System using Cloud Computing

Phase 2: Development of Basic Functionality

Group(46)

Submitted to:

Prof. Ayman M. Bahaa-Eldin

The code

```
import tkinter as tk
from tkinter import filedialog
import threading
import queue
import cv2  # OpenCV for image processing

class WorkerThread(threading.Thread):
    def __init__(self, task_queue):
        super()._init_()
        self.task_queue = task_queue

    def run(self):
        while True:
            task = self.task_queue.get()
            if task is None:
                break
            image, operation, filter_type = task
            result = self.process_image(image, operation, filter_type)
            self.display_result(result)

    def process_image(self, image, operation, filter_type):
        # Load the image
        img = cv2.imread(image, cv2.IMREAD_COLOR)
        # Perform the specified operation
        if operation == 'edge_detection':
            result = cv2.Canny(img, 100, 200)
        elif operation == 'color_inversion':
            result = cv2.bitwise_not(img)
        # Apply the selected filter
        if filter_type == 'grayscale':
            result = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        elif filter_type == 'blur':
            result = cv2.GaussianBlur(img, (5, 5), 0)
        # Add more filters as needed...
        return result

    def display_result(self, result):
        # Display the processed image
        cv2.imshow('Processed Image', result)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

def select_image():
    file_path = filedialog.askopenfilename()
```

```
    entry.delete(0, tk.END)
    entry.insert(0, file_path)

def process_image():
    image_path = entry.get()
    operation = var.get()
    filter_type = filter_var.get()
    task_queue.put((image_path, operation, filter_type))
    start_worker_thread()

def start_worker_thread():
    WorkerThread(task_queue).start()

root = tk.Tk()
root.title("Image Processing")

frame = tk.Frame(root)
frame.pack(padx=10, pady=10)

label1 = tk.Label(frame, text="Select Image:")
label1.grid(row=0, column=0, sticky="w")

entry = tk.Entry(frame, width=50)
entry.grid(row=0, column=1, padx=5, pady=5)

button = tk.Button(frame, text="Browse", command=select_image)
button.grid(row=0, column=2, padx=5, pady=5)

label2 = tk.Label(frame, text="Select Operation:")
label2.grid(row=1, column=0, sticky="w")

var = tk.StringVar()
var.set("edge_detection")

option1 = tk.Radiobutton(frame, text="Edge Detection", variable=var, value="edge_detection")
option1.grid(row=1, column=1, sticky="w")

option2 = tk.Radiobutton(frame, text="Color Inversion", variable=var, value="color_inversion")
option2.grid(row=2, column=1, sticky="w")

label3 = tk.Label(frame, text="Select Filter:")
label3.grid(row=3, column=0, sticky="w")

filter_var = tk.StringVar()
filter_var.set("grayscale")

filter_option1 = tk.Radiobutton(frame, text="Grayscale", variable=filter_var,
value="grayscale")
```

```
filter_option1.grid(row=3, column=1, sticky="w")

filter_option2 = tk.Radiobutton(frame, text="Blur", variable=filter_var, value="blur")
filter_option2.grid(row=4, column=1, sticky="w")

process_button = tk.Button(frame, text="Process Image", command=process_image)
process_button.grid(row=5, column=1, padx=5, pady=10)

task_queue = queue.Queue()

root.mainloop()
```

how we implemented the image processing

Let's break down the image processing implementation:

1. Selecting an Image: When the user clicks the "Browse" button, a file dialog is displayed using filedialog.askopenfilename(). This allows the user to select an image file from their file system. The selected file path is then inserted into the entry widget (entry) to display the selected file path.

2. Processing Image: When the user clicks the "Process Image" button, the process_image() function is called. This function retrieves the selected image file path from the entry widget (entry.get()), along with the selected operation (edge detection or color inversion) and filter type (grayscale or blur).

3. Worker Thread: The start_worker_thread() function is then called to start a worker thread (WorkerThread). This worker thread handles the actual image processing. The selected image file path, operation, and filter type are put into a queue (task_queue) and processed by the worker thread.

4. Worker Thread Execution: The WorkerThread class inherits from threading.Thread and overrides the run() method. In the run() method, the thread continuously retrieves tasks from the queue (task_queue) and processes them until it receives a None task, indicating that there are no more tasks to process. For each task, the selected image is loaded using OpenCV (cv2.imread()), and the specified operation and filter are applied to the image using OpenCV functions (cv2.Canny(), cv2.bitwise_not(), cv2.cvtColor(), cv2.GaussianBlur()).

5. Displaying the Processed Image: The processed image is displayed using cv2.imshow(). The GUI remains blocked until the user closes the displayed image window (cv2.waitKey(0)), after which the processed image window is closed (cv2.destroyAllWindows()).

How the worker thread worked

Let's break down how the worker thread works in this script:

1. **Initialization**: The **WorkerThread** class inherits from **threading.Thread**. In its **_init_** method, it initializes the task queue (**task_queue**) passed as an argument.

2. **Run Method**: The **run()** method of the **WorkerThread** class overrides the **run()** method of the **threading.Thread** class. This method defines what the thread will do when it starts running.

3. **Processing Tasks**: Within the **run()** method, there is an infinite loop (**while True:**) that continuously checks for tasks in the task queue (**self.task_queue**). It retrieves tasks using **self.task_queue.get()**.

4. **Handling Task Completion**: If the retrieved task is **None**, it means there are no more tasks to process, so the loop breaks (**if task is None: break**). Otherwise, it proceeds to process the task.

5. **Image Processing**: For each task, which consists of an image file path (**image**), an operation (**operation**), and a filter type (**filter_type**), the **process_image()** method is called. This method loads the image using OpenCV (**cv2.imread()**), performs the specified operation and filter using OpenCV functions (**cv2.Canny()**, **cv2.bitwise_not()**, **cv2.cvtColor()**, **cv2.GaussianBlur()**), and returns the processed image.

6. **Displaying the Result**: The processed image (**result**) is then displayed using OpenCV's **cv2.imshow()** function.

7. **Loop Continuation**: After processing a task, the thread continues to check for more tasks in the task queue. This loop continues until a **None** task is encountered, at which point the thread exits.

screenshot of the output

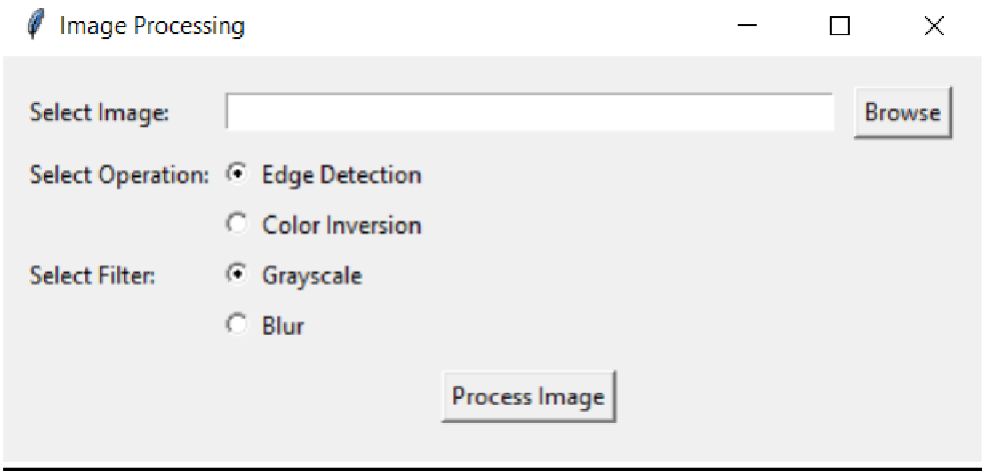


Figure 1:The Gui

Figure 2:The image that we will edit but before editing

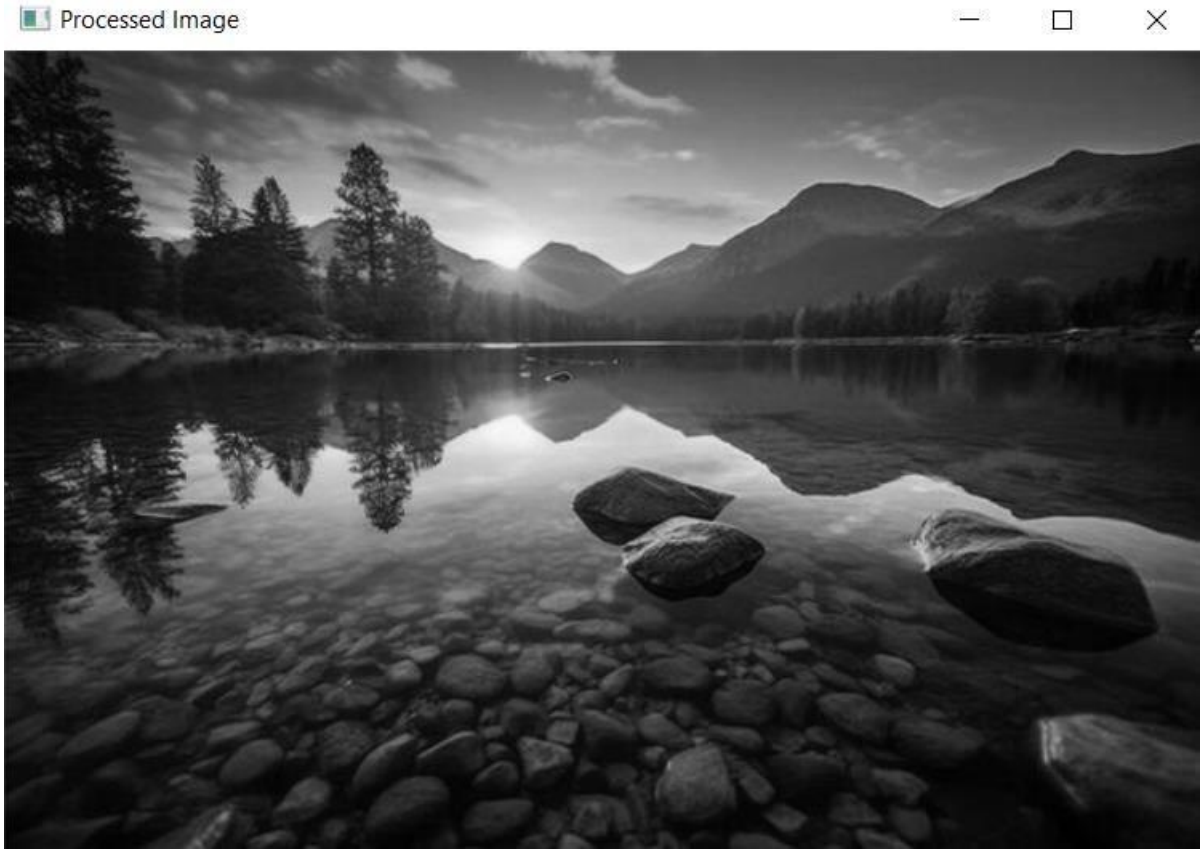


Figure 3:Processed image after processing

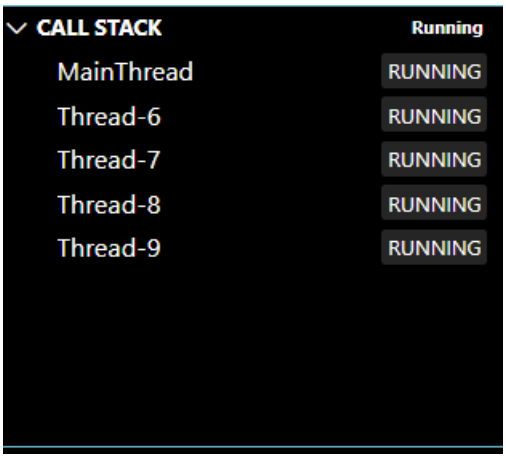


Figure 4:Many threads have been created

https://github.com/ZeyadCESS/Distributed-phase2

Phase 3

Distributed

We modified the image processing code to detect lines and rotate image by degrees and down we will see the code and screenshot of the output

```
import tkinter as tk
from tkinter import filedialog
import threading
import queue
import cv2
import numpy as np

class WorkerThread(threading.Thread):
    def __init__(self, task_queue):
        super().__init__()
        self.task_queue = task_queue

    def run(self):
        while True:
            task = self.task_queue.get()
            if task is None:
                break
            image, operation, filter_type, rotation_angle, resize_scale = task
            result = self.process_image(image, operation, filter_type, rotation_angle,
resize_scale)
            self.display_result(result)

    def process_image(self, image, operation, filter_type, rotation_angle, resize_scale):
        img = cv2.imread(image, cv2.IMREAD_COLOR)

        # Apply specified operations
        if operation == 'edge_detection':
            result = cv2.Canny(img, 100, 200)
        elif operation == 'color_inversion':
            result = cv2.bitwise_not(img)
        else:
            result = img  # Default to original image if no specific operation is selected

        # Apply rotation
        if rotation_angle != 0:
            rows, cols = result.shape[:2]
            rotation_matrix = cv2.getRotationMatrix2D((cols / 2, rows / 2), rotation_angle, 1)
            result = cv2.warpAffine(result, rotation_matrix, (cols, rows))
```

```
        # Apply resizing
        if resize_scale != 1.0:
            result = cv2.resize(result, None, fx=resize_scale, fy=resize_scale)

        # Apply the selected filter
        if filter_type == 'grayscale':
            if len(result.shape) == 3:
                result = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
        elif filter_type == 'blur':
            result = cv2.GaussianBlur(result, (5, 5), 0)
        elif filter_type == 'custom':
            # Define a custom filter kernel
            custom_filter = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]], dtype=np.float32)
            # Apply the custom filter
            result = cv2.filter2D(result, -1, custom_filter)
        # Add more filters as needed...
        return result

    def display_result(self, result):
        cv2.imshow('Processed Image', result)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

def select_image():
    file_path = filedialog.askopenfilename()
    entry.delete(0, tk.END)
    entry.insert(0, file_path)

def process_image():
    image_path = entry.get()
    operation = var.get()
    filter_type = filter_var.get()
    rotation_angle = rotation_scale.get()
    resize_scale = resize_scale_var.get()
    task_queue.put((image_path, operation, filter_type, rotation_angle, resize_scale))
    start_worker_thread()

def start_worker_thread():
    WorkerThread(task_queue).start()

root = tk.Tk()
root.title("Advanced Image Processing")

frame = tk.Frame(root)
frame.pack(padx=10, pady=10)

label1 = tk.Label(frame, text="Select Image:")
```

```
label1.grid(row=0, column=0, sticky="w")

entry = tk.Entry(frame, width=50)
entry.grid(row=0, column=1, padx=5, pady=5)

button = tk.Button(frame, text="Browse", command=select_image)
button.grid(row=0, column=2, padx=5, pady=5)

label2 = tk.Label(frame, text="Select Operation:")
label2.grid(row=1, column=0, sticky="w")

var = tk.StringVar()
var.set("edge_detection")

option1 = tk.Radiobutton(frame, text="Edge Detection", variable=var, value="edge_detection")
option1.grid(row=1, column=1, sticky="w")

option2 = tk.Radiobutton(frame, text="Color Inversion", variable=var, value="color_inversion")
option2.grid(row=2, column=1, sticky="w")

label3 = tk.Label(frame, text="Select Filter:")
label3.grid(row=3, column=0, sticky="w")

filter_var = tk.StringVar()
filter_var.set("grayscale")

filter_option1 = tk.Radiobutton(frame, text="Grayscale", variable=filter_var, value="grayscale")
filter_option1.grid(row=3, column=1, sticky="W")

filter_option2 = tk.Radiobutton(frame, text="Blur", variable=filter_var, value="blur")
filter_option2.grid(row=4, column=1, sticky="W")

filter_option3 = tk.Radiobutton(frame, text="Custom Filter", variable=filter_var,
value="custom")
filter_option3.grid(row=5, column=1, sticky="W")

label4 = tk.Label(frame, text="Rotation Angle (degrees):")
label4.grid(row=6, column=0, sticky="w")

rotation_scale = tk.Scale(frame, from_=0, to=360, orient=tk.HORIZONTAL)
rotation_scale.grid(row=6, column=1, padx=5, pady=5)

label5 = tk.Label(frame, text="Resize Scale:")
label5.grid(row=7, column=0, sticky="w")

resize_scale_var = tk.DoubleVar()
resize_entry = tk.Entry(frame, textvariable=resize_scale_var, width=10)
resize_entry.grid(row=7, column=1, padx=5, pady=5)
```

```
resize_scale_var.set(1.0)

process_button = tk.Button(frame, text="Process Image", command=process_image)
process_button.grid(row=8, column=1, padx=5, pady=10)

task_queue = queue.Queue()

root.mainloop()
```

Screenshots

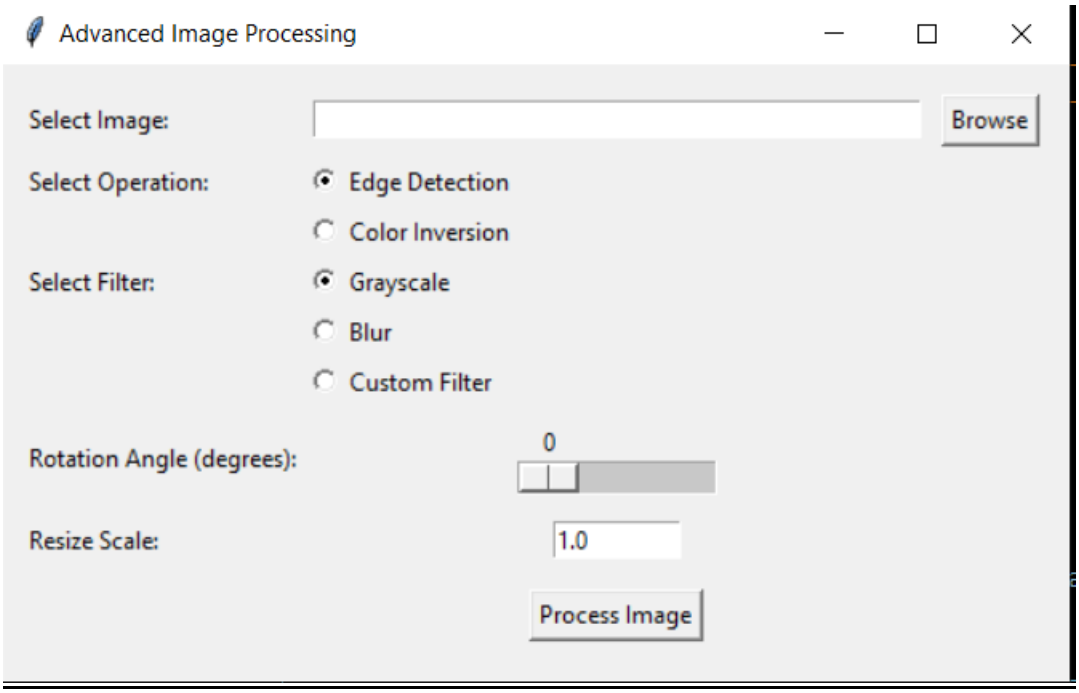


Figure 1:updated Gui showing the updates

Figure 2:image before editing

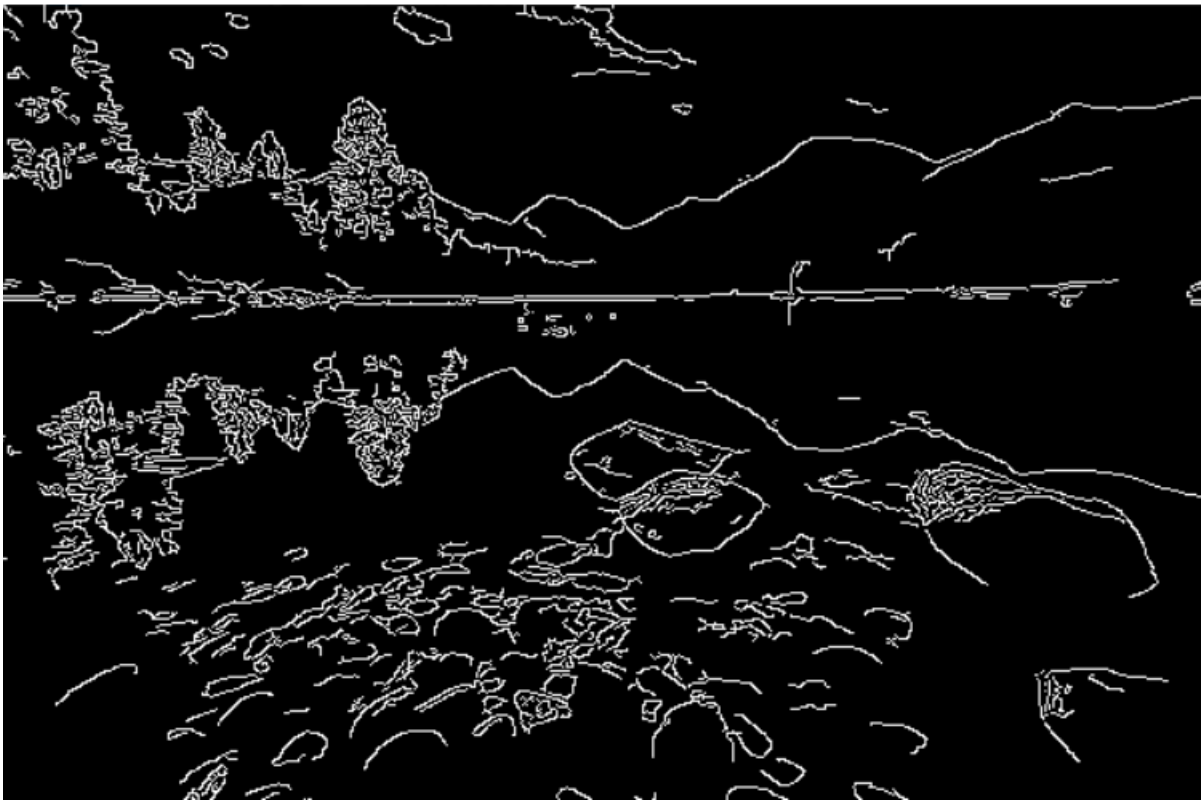


Figure 3:image after editing

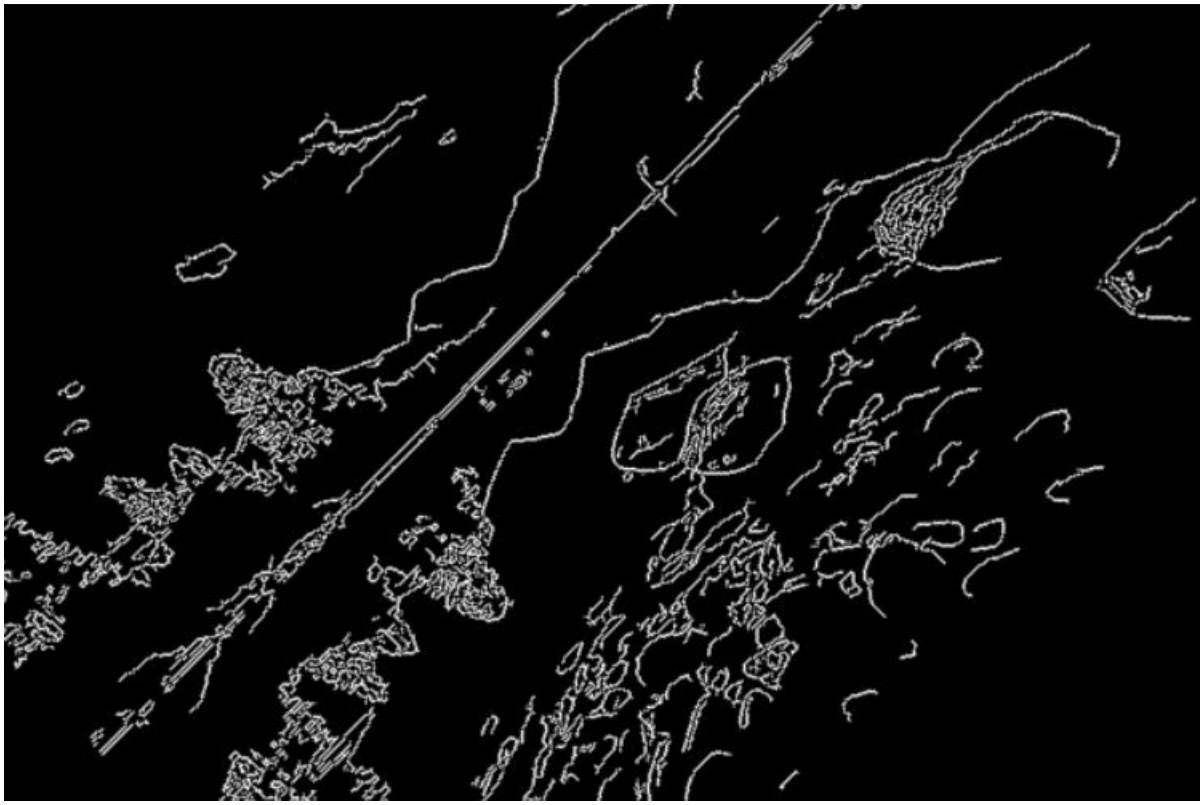


Figure 4:rotating image using rotation angle

After this we implemented two virtual machines and there is there codes

```
import pickle
import select
import socket
import threading
import image_processing_helper as img_helper
import pyopencl as cl

IP_ADDRESS = "0.0.0.0"
PORT = 5030

class VMHandler(threading.Thread):
    def __init__(self, ip, port, socket) -> None:
        super(VMHandler, self).__init__()
        self.ip = ip
        self.port = port
        self.socket = socket
        self.socket.setblocking(0)
```

```
        self.gpu_context = alt_lib.create_context()
        self.gpu_queue = alt_lib.create_queue(self.gpu_context)
        self.kernels = {
            'brighten': """
                __kernel void brighten(__global float* V) {
                    int i = get_global_id(0);
                    if (V[i] + 60.0f <= 255.0f)
                        V[i] = V[i] + 60.0f;
                    else
                        V[i] = 255.0f;
                }
            """,
            'darken': """
                __kernel void darken(__global float* V) {
                    int i = get_global_id(0);
                    if (V[i] - 60.0f >= 0.0f)
                        V[i] = V[i] - 60.0f;
                    else
                        V[i] = 0.0f;
                }
            """,
            'threshold': """
                __kernel void threshold(__global float* V) {
                    int i = get_global_id(0);
                    V[i] = V[i] >= 127.0f ? 255.0f : 0.0f;
                }
            """,
            'greyscale': """
                __kernel void greyscale(__global float* channel, __global float* result) {
                    int i = get_global_id(0);
                    result[i] = channel[i];
                }
            """
        }

    def run(self) -> None:
        while True:
            try:
                received_data = self.receive_data()
                if len(received_data) == 0:
                    continue
                print(f"Received: {received_data[0]} : {received_data[1]} : {received_data[3]} :
")
                operation = received_data[0]
                channel = received_data[1]
                image_channel = received_data[2]
                value = received_data[3]
                height = received_data[4]
```

```
                width = received_data[5]
                processed = self.process_image(self.gpu_context, self.gpu_queue, operation,
image_channel, value, height, width)
                print(f"Sending response {operation} ")
                self.send_data([operation, channel, processed])
            except OSError as o_err:
                print("OSError: {0}".format(o_err))
            except Exception as e:
                print("Exception: {0}".format(e))
                break

    def receive_data(self):
        timeout = 5
        data = b""
        ready_to_read, _, _ = select.select([self.socket], [], [])
        if ready_to_read:
            print(f"Start receiving data")
            self.socket.settimeout(timeout)
            while True:
                try:
                    chunk = self.socket.recv(4096)
                    if not chunk:
                        break
                    data += chunk
                except socket.timeout:
                    break
        array = pickle.loads(data)
        print("Returning received array")
        return array

    def send_data(self, data):
        serialized_data = pickle.dumps(data)
        print("Sending serialized data")
        self.socket.sendall(serialized_data)

    def process_image(self, ctx, queue, op, img, value, height=0, width=0):
        print(f"Starting processing {op}")
        if op == 'brighten':
            processed_channel = gpu_helper.apply_brightness(ctx, queue, img, value,
self.kernels['brighten'], self.kernels['darken'])
            return processed_channel
        elif op == "greyscale":
            processed_channel = img_helper.convert_to_greyscale(ctx, queue, img, height, width,
self.kernels['greyscale'])
            return processed_channel
        elif op == "threshold":
            processed_channel = img_helper.apply_threshold(ctx, queue, img,
self.kernels['threshold'], height, width)
```

```
            return processed_channel

tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_socket.bind((IP_ADDRESS, PORT))
print(f"First virtual machine Start listening on IP:{IP_ADDRESS}, PORT:{PORT}")
tcp_socket.listen(1)

while tcp_socket:
    if not tcp_socket:
        continue
    readable, _, _ = select.select([tcp_socket], [], [])
    server_socket, server_address = tcp_socket.accept()
    new_thread = VMHandler(server_address[0], server_address[1], server_socket)
    print(f"Starting Thread with:{server_address[0]} : {server_address[1]} ")
    new_thread.start()
    break
```

screenshot of running the virtual machine

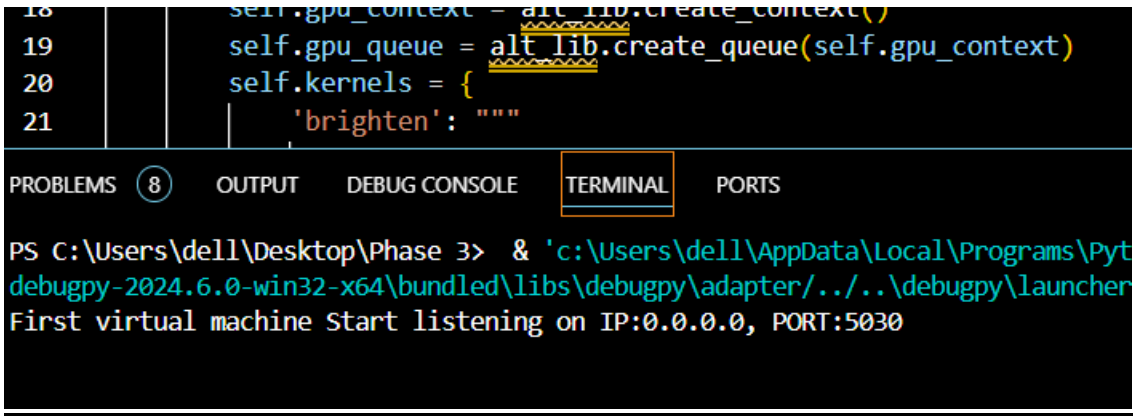


Figure 5:first vm listening

and here is description of what happens

1. **Imports and Constants**:
    - The code imports necessary modules like **pickle**, **select**, **socket**, **threading**, **image_processing_helper**, and **pyopencl**.

- It defines constants **IP_ADDRESS** and **PORT** for the server's IP address and port number, respectively.

2. **VMHandler Class**:
   - This class represents a thread that handles communication and image processing tasks for a client connection.
   - It initializes with the client's IP address, port, and socket.
   - GPU context and queue are created using PyOpenCL.
   - OpenCL kernels for various image processing operations like brighten, darken, threshold, and greyscale are defined.

3. **run Method**:
   - This method is executed when the thread starts.
   - It continuously receives data from the client, processes it, and sends back the result.
   - Error handling is implemented to catch exceptions and continue execution.
4. **receive_data Method**:
   - This method is responsible for receiving data from the client.
   - It waits for data to be available for reading using **select**.
   - Data is received in chunks and deserialized using **pickle**.

5. **send_data Method**:
   - This method serializes and sends data back to the client over the socket connection.

6. **process_image Method**:
   - This method processes the received image data based on the specified operation.
   - It uses PyOpenCL to execute the image processing kernels on the GPU.
   - Depending on the operation (e.g., brighten, greyscale, threshold), the corresponding kernel is selected and applied.

7. **Main Loop**:
    - The main loop of the server listens for incoming connections.
    - When a client connects, it accepts the connection and creates a new **VMHandler** thread to handle communication and processing for that client.

We associated helper functions in a separate file and here is the code

```
import pyopencl as cl
import numpy as np

def convert_to_greyscale(ctx, queue, channel, height, width, greyscale_kernel):
    """
    Convert a single channel to greyscale using OpenCL
    """
    channel_flat = channel.reshape(-1)
    empty_array = np.empty_like(channel_flat)

    channel_buffer = cl.Buffer(ctx, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR,
hostbuf=channel_flat)
    result_buffer = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, size=empty_array.nbytes)

    program = cl.Program(ctx, greyscale_kernel).build()
    program.greyscale(queue, (height * width,), None, channel_buffer, result_buffer)

    result = np.empty_like(channel_flat)
    cl.enqueue_copy(queue, result, result_buffer).wait()

    return result.reshape(height, width)

def apply_intensity_kernel(ctx, queue, channel, value, bright_kernel, dark_kernel):
    """
    Apply brightness/darkness adjustment to a single channel using OpenCL kernel
    """
    channel_flat = channel.reshape(-1)
    empty_array = np.empty_like(channel_flat)

    channel_buffer = cl.Buffer(ctx, cl.mem_flags.READ_WRITE | cl.mem_flags.COPY_HOST_PTR,
hostbuf=channel_flat)
    result_buffer = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, size=empty_array.nbytes)
```

```
    if value == 1:
        program = cl.Program(ctx, bright_kernel).build()
        program.bright(queue, channel_flat.shape, None, channel_buffer)
    else:
        program = cl.Program(ctx, dark_kernel).build()
        program.dark(queue, channel_flat.shape, None, channel_buffer)

    result = np.empty_like(channel_flat)
    cl.enqueue_copy(queue, result, channel_buffer).wait()

    return result.reshape(channel.shape)

def threshold_helper(ctx, queue, data, threshold_kernel, original_height, original_width):
    """
    Apply thresholding operation using OpenCL
    """
    buffer_data = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=data.nbytes)
    cl.enqueue_copy(queue, buffer_data, data)

    program = cl.Program(ctx, threshold_kernel).build()
    program.Thresh(queue, data.shape, (1,), buffer_data)

    cl.enqueue_copy(queue, data, buffer_data)
    thresholded_image = data.reshape((original_height, original_width))
    thresholded_image = thresholded_image.astype(np.uint8)

    return thresholded_image
```

and here is the description of what it does

1. **convert_to_greyscale**:
    - This function takes as input the OpenCL context (**ctx**), command queue (**queue**), a single image channel (**channel**), its height and width, and a kernel for converting the channel to grayscale.
    - It reshapes the input channel into a 1D array (**channel_flat**).
    - Empty arrays are created for storing the result.
    - OpenCL buffers are created for the input channel and the result.
    - The provided kernel is built and executed on the GPU using the **greyscale** program.
    - The result is copied back from the GPU to the host memory and

reshaped to the original height and width before being returned.

2. **apply_intensity_kernel**:
   - This function applies brightness or darkness adjustment to a single channel.
   - It takes similar inputs as the previous function along with a value (**value**) indicating whether to apply brightness (1) or darkness (0).
   - The channel is reshaped into a 1D array, and empty arrays are created for the result.
   - OpenCL buffers are created for the channel and the result.
   - Depending on the value, either the brightness or darkness kernel is built and executed on the GPU.
   - The result is copied back from the GPU to the host memory and reshaped to the original shape before being returned.

3. **threshold_helper**:
   - This function applies thresholding operation to an image.
   - It takes inputs including the OpenCL context, command queue, image data, kernel for thresholding, and the original height and width of the image.
   - The image data is copied to an OpenCL buffer.
   - The provided kernel is built and executed on the GPU to perform thresholding.
   - The thresholded image data is copied back from the GPU to the host memory, reshaped to the original dimensions, and converted to **uint8** data type before being returned.

And here is the code of the second virtual machine and the helper

```
import pickle
import select
import socket
import threading
import image_processing_helper as img_helper
import pyopencl as cl

IP_ADDRESS = "0.0.0.0"
PORT = 5031

class VMHandler(threading.Thread):
    def __init__(self, ip, port, socket) -> None:
        super(VMHandler, self).__init__()
        self.ip = ip
        self.port = port
        self.socket = socket
        self.socket.setblocking(0)
        self.gpu_context = alt_lib.create_context()
        self.gpu_queue = alt_lib.create_queue(self.gpu_context)
        self.kernels = {
            'brighten': """
                __kernel void brighten(__global float* V) {
                    int i = get_global_id(0);
                    if (V[i] + 60.0f <= 255.0f)
                        V[i] = V[i] + 60.0f;
                    else
                        V[i] = 255.0f;
                }
            """,
            'darken': """
                __kernel void darken(__global float* V) {
                    int i = get_global_id(0);
                    if (V[i] - 60.0f >= 0.0f)
                        V[i] = V[i] - 60.0f;
                    else
                        V[i] = 0.0f;
                }
            """,
            'threshold': """
                __kernel void threshold(__global float* V) {
                    int i = get_global_id(0);
                    V[i] = V[i] >= 127.0f ? 255.0f : 0.0f;
                }
            """,
            'greyscale': """
                __kernel void greyscale(__global float* channel, __global float* result) {
                    int i = get_global_id(0);
                    result[i] = channel[i];
```

```
                }
            """
        }

    def run(self) -> None:
        while True:
            try:
                received_data = self.receive_data()
                if len(received_data) == 0:
                    continue
                print(f"Received: {received_data[0]} : {received_data[1]} : {received_data[3]} :
")
                operation = received_data[0]
                channel = received_data[1]
                image_channel = received_data[2]
                value = received_data[3]
                height = received_data[4]
                width = received_data[5]
                processed = self.process_image(self.gpu_context, self.gpu_queue, operation,
image_channel, value, height, width)
                print(f"Sending response {operation} ")
                self.send_data([operation, channel, processed])
            except OSError as o_err:
                print("OSError: {0}".format(o_err))
            except Exception as e:
                print("Exception: {0}".format(e))
                break

    def receive_data(self):
        timeout = 5
        data = b""
        ready_to_read, _, _ = select.select([self.socket], [], [])
        if ready_to_read:
            print(f"Start receiving data")
            self.socket.settimeout(timeout)
            while True:
                try:
                    chunk = self.socket.recv(4096)
                    if not chunk:
                        break
                    data += chunk
                except socket.timeout:
                    break
        array = pickle.loads(data)
        print("Returning received array")
        return array

    def send_data(self, data):
```

```
        serialized_data = pickle.dumps(data)
        print("Sending serialized data")
        self.socket.sendall(serialized_data)

    def process_image(self, ctx, queue, op, img, value, height=0, width=0):
        print(f"Starting processing {op}")
        if op == 'brighten':
            processed_channel = gpu_helper.apply_brightness(ctx, queue, img, value,
self.kernels['brighten'], self.kernels['darken'])
            return processed_channel
        elif op == "greyscale":
            processed_channel = img_helper.convert_to_greyscale(ctx, queue, img, height, width,
self.kernels['greyscale'])
            return processed_channel
        elif op == "threshold":
            processed_channel = img_helper.apply_threshold(ctx, queue, img,
self.kernels['threshold'], height, width)
            return processed_channel

tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_socket.bind((IP_ADDRESS, PORT))
print(f"second virtual machine Start listening on IP:{IP_ADDRESS}, PORT:{PORT}")
tcp_socket.listen(1)

while tcp_socket:
    if not tcp_socket:
        continue
    readable, _, _ = select.select([tcp_socket], [], [])
    server_socket, server_address = tcp_socket.accept()
    new_thread = VMHandler(server_address[0], server_address[1], server_socket)
    print(f"Starting Thread with:{server_address[0]} : {server_address[1]} ")
    new_thread.start()
    break
```

Screenshot of the second virtual machine

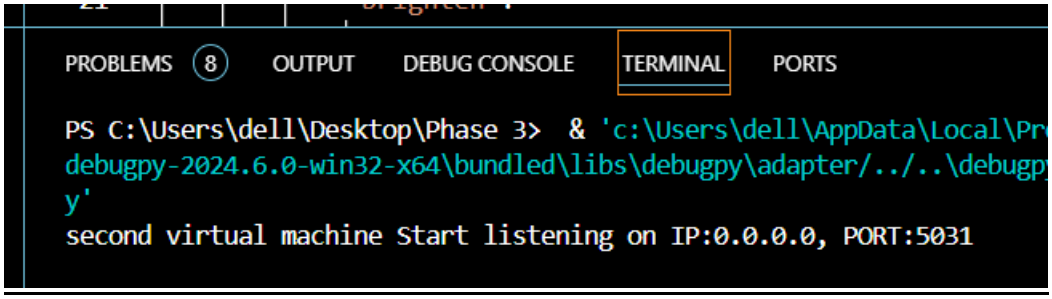


Figure 6:second vm running

here is the code of the helper functions

```
import pyopencl as cl
import numpy as np

def convert_to_greyscale(ctx, queue, channel, height, width, greyscale_kernel):
    """
    Convert a single channel to greyscale using OpenCL
    """
    channel_flat = channel.reshape(-1)
    empty_array = np.empty_like(channel_flat)

    channel_buffer = cl.Buffer(ctx, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR,
hostbuf=channel_flat)
    result_buffer = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, size=empty_array.nbytes)

    program = cl.Program(ctx, greyscale_kernel).build()
    program.greyscale(queue, (height * width,), None, channel_buffer, result_buffer)

    result = np.empty_like(channel_flat)
    cl.enqueue_copy(queue, result, result_buffer).wait()

    return result.reshape(height, width)

def apply_intensity_kernel(ctx, queue, channel, value, bright_kernel, dark_kernel):
    """
    Apply brightness/darkness adjustment to a single channel using OpenCL kernel
    """
    channel_flat = channel.reshape(-1)
    empty_array = np.empty_like(channel_flat)
```

```
    channel_buffer = cl.Buffer(ctx, cl.mem_flags.READ_WRITE | cl.mem_flags.COPY_HOST_PTR,
hostbuf=channel_flat)
    result_buffer = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, size=empty_array.nbytes)

    program = cl.Program(ctx, bright_kernel if value == 1 else dark_kernel).build()
    kernel_function = program.bright if value == 1 else program.dark
    kernel_function(queue, channel_flat.shape, None, channel_buffer)

    result = np.empty_like(channel_flat)
    cl.enqueue_copy(queue, result, channel_buffer).wait()

    return result.reshape(channel.shape)

def apply_threshold(ctx, queue, data, threshold_kernel, original_height, original_width):
    """
    Apply thresholding operation using OpenCL
    """
    buffer_data = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=data.nbytes)
    cl.enqueue_copy(queue, buffer_data, data)

    program = cl.Program(ctx, threshold_kernel).build()
    program.Thresh(queue, data.shape, (1,), buffer_data)

    cl.enqueue_copy(queue, data, buffer_data)
    thresholded_image = data.reshape((original_height, original_width))
    thresholded_image = thresholded_image.astype(np.uint8)

    return thresholded_image
```

https://github.com/ZeyadCESS/phase-3-distributed